

# Python *for* Econometrics

Lecturer: Fabian H. C. Raters

Institute: Econometrics, University of Goettingen

Version: October 31, 2020





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Welcome to this course and to the world of Python!

Learning objectives of this course:

- Python: The course is about Python programming.
- *for*: You will learn tools and methods.
- Econometrics:
  - Statistics: Numerical programming in Python.
  - applied to: We will use it on examples.
  - Economics: In an economic context.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Knowledge after completing this course:

- You have acquired a basic understanding of programming in general with Python and a special knowledge of working with standard numerical packages.
- You are able to study Python in depth and absorb new knowledge for your scientific work with Python.
- You know the capabilities and further possibilities to use Python in econometrics.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

What you should not expect from this course:

- A guide how to install or maintain an application.
- An introduction to programming for beginners.
- An introduction to professional development tools.
- Non-scientific, general purpose programming (beyond the language essentials).
- Few content and less effort...



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

This course can be seen as an applied lecture:

### Lecture:

We try to explain the partly theoretical knowledge on Python by simple, easy to understand examples. You can learn the programming language's subtleties by reading [literature](#).

### Exercises:

Digital work sheets in the form of Jupyter notebooks with applied tasks are available for each chapter. For all exercises there are sample solutions available in separate notebooks.

### Self-tests:

At the end of each of the five chapters there are typical exam questions.

### Written exam:

There will be a final exam. This will be a pure multiple choice exam: 60 questions, 90 minutes.

After the successful participation in the exam you will receive 6 ECTS.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

The programming language Python is already established and very well in trend for numerical applications. Some keywords:

- Data science,
- Data wrangling,
- Machine learning,
- Numerical statistics,
- ...

Recommended literature while following this course:

- *Learning Python, 5th Edition* by Mark Lutz,
- *Python Crash Course, 2nd Edition* by Eric Matthes,
- *Python Data Science Handbook* by Jake VanderPlas,
- *Python for Data Analysis, 2nd Edition* by Wes McKinney,
- *Python for Finance, 2nd Edition* by Yves Hilpisch.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

We are using *Python 3*. There was a big revision in the migration from Python 2 to version 3 and the new version is no longer backwards compatible to the old version.

## Python 3 running [command line]

```
python --version
```

```
## Python 3.8.5
```

The normal execution mode is that the Python interpreter processes the instructions in the background – in other numeric programming languages such as *R* this is known as *batch mode*. It executes program code that is usually located in a source code file.

The interpreter can also be started in an *interactive mode*. It is used for testing and analytic purposes in order to obtain fast results when performing simple applications.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

For everyday work with Python it would be extremely tedious to make all edits in interactive mode.

There are a number of excellent integrated development environments (IDEs) for Python, with three being emphasized here:

- *Jupyter* (and *IPython*)
- *Spyder* (scientific IDE)
- *PyCharm* (by *IntelliJ*)

Of course, you can also use a simple text editor. However, you would probably miss the comfort of an IDE.

Installing, adding and maintaining Python is not trivial at the beginning. Therefore, as a beginner, you are well advised to [download and install the Python distribution \*Anaconda\*](#). Bonus: Many standard packages are supplied directly or you can post-install them conveniently.





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

In this course – in a numeric and analytic context – we use only **Jupyter with the IPython kernel**.

That is why we have combined

- 1 all the code from the slides, and
- 2 all the exercises and solutions

into interactive Jupyter notebooks that you can use online without having to install software locally on your computer. The GWDG has set up a **cloud-based Jupyter-Hub** for you.

You can access the working environment with your university credentials at

<https://jupyter-cloud.gwdg.de/>

create a profile and get started right away – even using your smart devices. However, so far you are still asked to upload the course notebooks by yourself or rewrite the code from scratch.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

A Jupyter notebook is divided into individual, vertically arranged cells, which can be executed separately:

The screenshot shows the Jupyter Notebook interface for a notebook titled 'first\_notebook'. The top bar includes the Jupyter logo, the notebook title, and the Python 3 kernel. Below the top bar is a menu bar with options: File, Edit, View, Insert, Cell, Kernel, Widgets, Help. A toolbar below the menu bar contains icons for saving, adding, deleting, and duplicating cells, as well as navigation and execution controls. The main area displays three code cells. The first cell, labeled 'In [2]:', contains the code 'a = 10' and 'b = 15'. The second cell, labeled 'In [4]:', contains the code 'a', and its output, 'Out[4]: 10', is displayed below it. The third cell, labeled 'In [5]:', contains the code 'a + b', and its output, 'Out[5]: 25', is displayed below it.

The notebook approach is not novel and comes from the field of computer algebra software.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Actually, an interactive Python interpreter called IPython is started “in the core”.

### IPython running [command line]

```
ipython --version
```

```
## 7.19.0
```

Roughly speaking, this is a greatly enhanced version of the Python 3 interpreter, which has numerous, convenient advantages over the “normal” interpreter in interactive mode, such as, e.g.,

- printing of return values,
- color highlighting, and
- magic commands.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Finally, we wish you a lot of fun and success with and in this course!

*Practice makes perfect!*

## Contribution and credits:

Fabian H. C. Raters

Eike Manßen

GWDG for the Jupyter-Hub



## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## 1 Essential concepts

### 1.1 Getting started

### 1.2 Procedural programming

### 1.3 Object-orientation

## 2 Numerical programming

### 2.1 NumPy package

### 2.2 Array basics

### 2.3 Linear algebra

## 3 Data formats and handling

### 3.1 Pandas package

### 3.2 Series

### 3.3 DataFrame

### 3.4 Import/Export data

## 4 Visual illustrations

### 4.1 Matplotlib package

### 4.2 Figures and subplots

### 4.3 Plot types and styles

### 4.4 Pandas layers

## 5 Applications

### 5.1 Time series

### 5.2 Moving window

### 5.3 Financial applications

### 5.4 Optimization



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

# Essential concepts

## 1.1 Getting started

## 1.2 Procedural programming

## 1.3 Object-orientation



## Essential concepts

### Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

# Essential concepts

## ▶ Getting started



## Essential concepts

### Getting started

#### Procedural programming

#### Object-orientation

## Numerical programming

#### NumPy package

#### Array basics

#### Linear algebra

## Data formats and handling

#### Pandas package

#### Series

#### DataFrame

#### Import/Export data

## Visual illustrations

#### Matplotlib package

#### Figures and subplots

#### Plot types and styles

#### Pandas layers

## Applications

#### Time series

#### Moving window

#### Financial applications

#### Optimization

Python can be described as

- a dynamic, strongly typed, multi-paradigm and object-oriented programming language,
- for versatile, powerful, elegant and clear programming,
- with a general, high-level, multi-platform application scope,
- which is being used very successfully in the data science sector and very much in trend.

Moreover, Python is relatively easy to learn and its successful language design supports novices to professional developers. Much of Python's success is due to a *high degree of standardization* and a huge community that elaborates and collectively recognizes *conventions and paradigms*.





... of the Python era:

The language was originally developed in 1991 by Guido van Rossum. Its name was based on Monty Python's Flying Circus. Its main identification feature is the novel markup of code blocks – by indentation:

## Indentation example

```
password = input("I am your bank. Password please: ")  
  
## I am your bank. Password please: sparkasse  
  
if password == "sparkasse":  
    print("You successfully logged in!")  
else:  
    print("Fail. Will call the police!")  
  
## You successfully logged in!
```

This increases the readability of code and should at the same time encourage the programmer in programming neatly. Since the source code can be written more compactly with Python, an increased efficiency in daily work can be expected.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

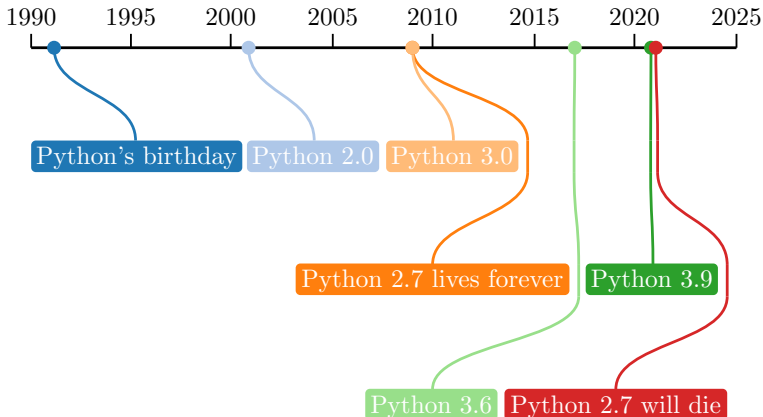
Time series

Moving window

Financial applications

Optimization

## Overview of the Python development by versions and dates:





Comparing the way Python works with common programming languages, we briefly discuss a selection of popular competitors:

## C/C++:

- CPython is interpreted, not compiled.
- C/C++ are strongly static, complex languages.

## Java:

- CPython is not compiled just-in-time.
- Java has a C-type syntax.

## MATLAB

- In Python you primarily follow a scalar way of thinking, while in *MATLAB* you write matrix-based programs.
- In the numerical context, the matrix view and syntax are very similar to those of *MATLAB*.
- *MATLAB* is partially compiled just-in-time.

Where *CPython* is the reference implementation – the “Original Python”, which is implemented in C itself.



## R

- In Python you primarily follow a scalar way of thinking, while in *R* you write vector-based programs.
- *R* has a C-type syntax including additions to novel language concepts.

## Stata

- Any comparison would inadequately describe the differences.

## Reference semantics

An extremely important difference between the first two languages, C/C++ and Java, as well as Python itself, and the last three languages is that they follow a call-by-reference semantic, while MATLAB, R and Stata are call-by-copy.

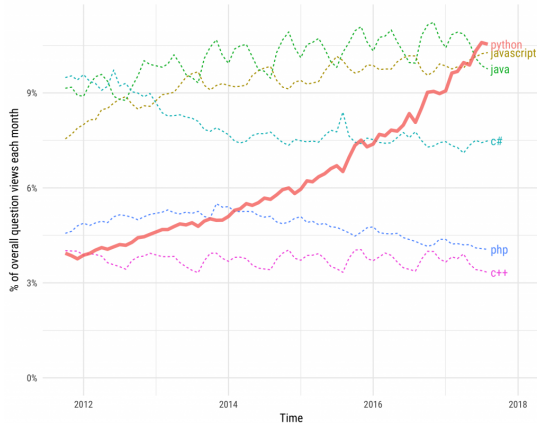
Further specific differences and similarities to MATLAB and R will be addressed in other parts of this course.



## Python has become extremely popular:

### Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries



Source: <https://stackoverflow.blog/2017/09/06/incredible-growth-python/>



## Essential concepts

### Getting started

#### Procedural programming

#### Object-orientation

### Numerical programming

#### NumPy package

#### Array basics

#### Linear algebra

### Data formats and handling

#### Pandas package

#### Series

#### DataFrame

#### Import/Export data

### Visual illustrations

#### Matplotlib package

#### Figures and subplots

#### Plot types and styles

#### Pandas layers

### Applications

#### Time series

#### Moving window

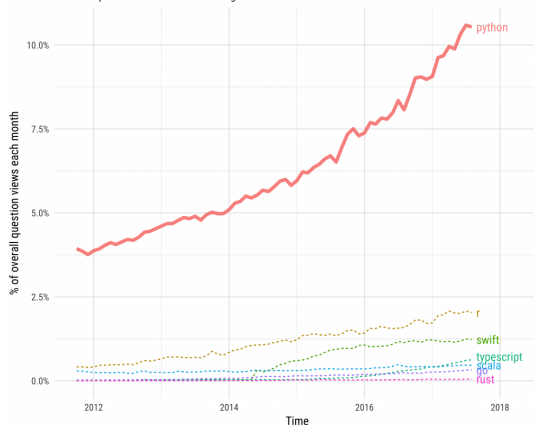
#### Financial applications

#### Optimization

So, you're on the right track – because who wants to bet on the wrong *hoRse*?

### Python compared to smaller, growing technologies

Based on question traffic in World Bank high-income countries



Source: <https://stackoverflow.blog/2017/09/06/incredible-growth-python/>



## Essential concepts

### Getting started

#### Procedural programming

#### Object-orientation

## Numerical programming

#### NumPy package

#### Array basics

#### Linear algebra

## Data formats and handling

#### Pandas package

#### Series

#### DataFrame

#### Import/Export data

## Visual illustrations

#### Matplotlib package

#### Figures and subplots

#### Plot types and styles

#### Pandas layers

## Applications

#### Time series

#### Moving window

#### Financial applications

#### Optimization

Areas in which Python is used with great success:

- Scripts,
- Console applications,
- GUI applications,
- Game development,
- Website development, and
- **Numerical programming.**

Places where Python is used:





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

In this course we will successively gain the following insights:

- 1 General basics of the language.
- 2 Numerical programming and handling of data sets.
- 3 Application to economic and analytical questions.





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Essential concepts

# ▶ Procedural programming



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Programs can be implemented very quickly – this is a pretty minimal example. You can write this command to a text file of your choice and run it directly on your system:

```
Hello there
```

```
print("Hello there!")
```

```
## Hello there!
```

- Only one *function* `print()` (shown here as a *keyword*),
- Function displays *argument* (a string) on screen,
- Arguments are passed to the function in parentheses,
- A string must be wrapped in " " or ' ',
- No semicolon at the end.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

Let's add a user input to the program:

## Hello you

```
name = input("Please enter your name: ")  
## Please enter your name: Angela Merkel  
print("Hello " + name + "!")  
## Hello Angela Merkel!
```

- The function `input()` is used for interactive text input,
- You can use the equal sign `=` to assign variables (here: `name`),
- Strings can be joined by the (overloaded) Operator `+`.



We are now trying to find out on which weekday a person was born (Merkel's birthday is 17-07-1954):

## Weekday of birth

```
from datetime import datetime

answer = input("Your birthday (DD-MM-YYYY): ")

## Your birthday (DD-MM-YYYY): 17-07-1954

birthday = datetime.strptime(answer, "%d-%m-%Y")
print("Your birthday was on a " + birthday.strftime("%A") + "!")

## Your birthday was on a Saturday!
```

- It is really easy to import functionality from other *modules*,
- Function `strptime()` is a *method* of *class* `datetime`,
- Both methods, `strptime()` and `strftime()`, are used to convert between strings and date time specifications.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

And how many days have passed since then (until Merkel's 4th swearing-in as Federal Chancellor)?

### Age in days

```
someday = datetime.strptime("14-03-2018", "%d-%m-%Y")
print("You are " + str((someday - birthday).days) + " days old!")

## You are 23251 days old!
```

- You can create time differences, i.e., the operator `-` is overloaded,
- The difference represents a new *object*, with its own *attributes*, such as `days`,
- When using the overloaded operator `+`, you have to explicitly convert the number of days by means of `str()` into a string.



How many years, weeks and days do you think that is?

## Human readable age

```
from dateutil.relativedelta import relativedelta
delta = relativedelta(someday, birthday)
print(f"That's {delta.years} years, {delta.months} months "
      f"and {delta.days} days!!")

## That's 63 years, 7 months and 25 days!!
```

- You don't have to keep reinventing the wheel – a wealth of packages and individual modules are freely available,
- A lowercase `f` before `"..."` provides convenient *formatting* – there are other options as well,
- Two strings in sequence are implicitly joined together – `"That's nice"`



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

When working with the interactive interpreter, i.e., in a notebook, you can quickly get useful information about Python objects:

## Help system

```
help(len)
```

```
## Help on built-in function len in module builtins:
##
## len(obj, /)
##     Return the number of items in a container.
```

Alternatively, e.g., for more complex problems, it is best to search directly with your preferred internet search engine.

You can find neat solutions to conventional challenges in [literature](#).



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

As with natural language, programming languages have a lexical structure. Source code consists of the smallest possible, indivisible elements, the tokens. In Python you can find the following groups of elements:

- Literals
- Variables
- Operators
- Delimiters
- Keywords
- Comments

These terms give us a rock-solid foundation for exploring the heart of a programming language.





Basically, we distinguish between *literals* and *variables*:

## Assigning variables with literals

```
myint = 7
myfloat = 4.0
myboat = "nice"
mybool = True
myfloat = myboat
```

- In this course, we will work with four different literals: integer (7), float (4.0), string ("nice") and boolean (True),
- Literals are assigned to variables at runtime,
- In Python the data type is derived from the literal and does not have to be described explicitly,
- It is allowed to assign values of different data types to the same variable (name) sequentially,
- If we don't assign a literal to any variables, we forfeit it.



Most *operators* and *delimiters* will be introduced to you during this course. Here is an overview of the operators:

## Overview of operators

## +	-	*	/	**	//
## %	@	<<	>>	&	
## ^	~	==	!=	<	>
## <=	>=	and	or	not	in
## not in	is	is not			

An overview of the delimiters follows:

## Overview of delimiters

## (	)	[	]	{	}
## ,	:	.	=	;	->
## +=	-=	*=	/=	**=	//=
## %=	@=	<<=	>>=	&=	=
## ^=	'	"	\	@	SPACE



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

All regular *arithmetic operations* involving numbers are possible:

### Pocket calculator

10 + 5

100 - 20

8 / 2

4 \* (10 + 20)

2\*\*3

## 15

## 80

## 4.0

## 120

## 8

- The result of dividing two integers is a floating point number,
- The conventional rules apply: Parentheses first, then multiplication and division, etc.,
- The operator **\*\*** is used for exponentiation.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

In order to demonstrate the use of *logical operators* (and formatted strings and `for`-loops), we create a handy table summarizing some important results from *boolean algebra*:

## Logical table

```
# Create table head
print("a    b    a and b    a or b    not a\n"
      "-----")

# Loop through the rows
for a in [False, True]:
    for b in [False, True]:
        print(f"{a:1} {b:1} {a and b:1} {a or b:1} {not a:1}")
```

##	a	b	a and b	a or b	not a
##	0	0	0	0	1
##	0	1	0	1	1
##	1	0	0	1	0
##	1	1	1	1	0



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

The programmer explains the structure of his/her program to the interpreter via a restricted set of short commands, the *keywords*:

## Overview of keywords

##	and	as	assert	break	class	continue
##	def	del	elif	else	except	False
##	finally	for	from	global	if	import
##	in	is	lambda	None	nonlocal	not
##	or	pass	raise	return	True	try
##	while	with	yield			

There are two ways to make *comments*:

## Provide some comments

```
# Set variable to something - or nothing?  
something = None
```

```
"""  
I am a docstring!  
A multiline string comment hybrid.  
I will be useful for describing classes and methods.  
"""
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

Python offers the following *basic data types*, which we will use in this course:

Data type	Description
<code>int()</code>	Integers
<code>float()</code>	Floating point numbers
<code>str()</code>	Strings, i.e., unicode (UTF-8) texts
<code>bool()</code>	Boolean, i.e., <code>True</code> or <code>False</code>
<code>list()</code>	List, an ordered array of objects
<code>tuple()</code>	Tuple, an ordered, unmutable array of objects
<code>dict()</code>	Dictionary, an unordered, associative array of objects
<code>set()</code>	Set, an unordered array/set of objects
<code>None()</code>	Nothing, emptiness, the void..

Each data type has its own methods, that is, functions that are applicable specifically to an object of this type.

You will gradually get to know new and more complex data types or object classes.



A *list* is an ordered array of objects, accessible via an *index*:

## Listing tech companies

```
stocks = ["Google", "Amazon", "Facebook", "Apple"]
stocks[1]
stocks.append("Twitter")
stocks.insert(2, "Microsoft")
stocks.sort()

## ['Google', 'Amazon', 'Facebook', 'Apple']
## Amazon
## ['Google', 'Amazon', 'Facebook', 'Apple', 'Twitter']
## ['Google', 'Amazon', 'Microsoft', 'Facebook', 'Apple', 'Twitter']
## ['Amazon', 'Apple', 'Facebook', 'Google', 'Microsoft', 'Twitter']
```

- The constructor for new lists is `[ ]`,
- The **first element has the index 0**,
- The data type `list()` possesses its own methods.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

*Tuples* are immutable sequences related to lists that cannot be extended, for example. The drawbacks in flexibility are compensated by the advantages in speed and memory usage:

## Selecting elements in sequences

```
lottery = (1, 8, 9, 12, 24, 28)
len(lottery)
lottery[1:3]
lottery[:4]
lottery[-1]
lottery[-2:]

## (1, 8, 9, 12, 24, 28)
## 6
## (8, 9)
## (1, 8, 9, 12)
## 28
## (24, 28)
```

The same operations are also supported when using lists.





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

*Dictionaries* are associative collections of *key-value pairs*. The *key* must be immutable and unique:

## Internet slang dictionary

```
slang = {"imho": "in my humble opinion",
        "lol": "laughing out loud",
        "tl;dr": "too long; didn't read"}

slang["lol"]
slang["gl&hl"] = "good luck & have fun"
slang.keys()
slang.values()

## {'imho': 'in...ion', 'lol': 'la...oud', 'tl;dr': 'to...ead'}
## laughing out loud
## good luck & have fun
## dict_keys(['imho', 'lol', 'tl;dr', 'gl&hl'])
## dict_values([... & have fun'])
```

- The constructor for `dict()` is `{ }` with `:`,
- The pairs are unordered, iterable sequences.



A *set* is an unordered collection of objects without duplicates:

## Set operations

```
x = {"o", "n", "y", "t"}
y = {"p", "h", "o", "n"}
x & y
x | y
x - y

## {'t', 'n', 'o', 'y'}
## {'o', 'p', 'n', 'h'}
## {'n', 'o'}
## {'p', 'h', 't', 'n', 'o', 'y'}
## {'t', 'y'}
```

- The constructor for `set()` is `{ }`,
- Defines its own operators that overload existing ones.
- Empty set via `set()`, because `{}` already creates `dict()`.



The `<`, `<=`, `>`, `>=`, `==`, `!=` operators compare the values of two objects and return `True` or `False`.

Op.	<code>True</code> , only if the value of the left operand is
-----	--

<code>&lt;</code>	less than the value of the right operand
-------------------	--

<code>&lt;=</code>	less than or equal to the value of the right operand
--------------------	--

<code>&gt;</code>	greater than the value of the right operand
-------------------	---

<code>&gt;=</code>	greater than or equal to the value of the right operand
--------------------	---

<code>==</code>	equal to the right operand
-----------------	----------------------------

<code>!=</code>	not equal to the right operand
-----------------	--------------------------------

The comparison depends on the `datatype` of the objects. For example `"7" == 7` will return `False`, while `7.0 == 7` will return `True`.

- Numbers are compared arithmetically.
- Strings are compared lexicographically.
- Tuples and lists are compared lexicographically using comparison of corresponding elements. This behaviour can be altered.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Comparing examples

```
x, y = 5, 8
print("x < y is", x < y)

## x < y is True

print("x > y is", x > y)

## x > y is False

print("x == y is", x == y)

## x == y is False

print("x != y is", x != y)

## x != y is True

print("This is", "Name" == "Name", "and not", "Name" == "name")

## This is True and not False
```

Comparing strings, the case has to be considered.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

In Python, comparison operators can also be chained.

## Chaining comparison examples

```
x = 5
```

```
5 >= x > 4
```

```
## True
```

```
12 < x < 20
```

```
## False
```

```
2 < x < 10
```

```
## True
```

```
2 < x and x < 10 # unchained expression
```

```
## True
```

The comparison is performed for both sides and combined by `and`.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

There are three logical operators: `not`, `and`, `or`.

Op.	Description
<code>not x</code>	Returns <code>True</code> only if <code>x</code> is <code>False</code>
<code>x and y</code>	Returns <code>True</code> only if <code>x</code> and <code>y</code> are <code>True</code>
<code>x or y</code>	Returns <code>True</code> only if <code>x</code> or <code>y</code> or both are <code>True</code>

## Logical operators examples

```
x, y = 5, 8
```

```
(x == 5) and (y == 9)
```

```
## False
```

```
(x == 5) or (y == 8)
```

```
## True
```

```
not(x == 4) or (y == 9)
```

```
## True
```



In some situations, you need a logical operation that is **True** only when the operands differ (one is **True**, the other is **False**). This task can be solved by using the logical operators **not**, **and**, **or** or simply **!=**.

## Exclusive or

```
x, y = 5, 8
```

```
((x == 5) and not (y == 8)) or (not (x == 5) and (y == 8))
```

```
## False
```

```
x = 4
```

```
((x == 5) and not (y == 8)) or (not (x == 5) and (y == 8))
```

```
## True
```

```
(x == 5) != (y == 8)
```

```
## True
```

In many other programming languages, an operation “exclusive or” or *xor* is explicitly part of the language, but not in Python.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

**Bitwise operators** operate on numbers, but instead of treating that number as if it were a single (decimal) value, they operate on the string of bits representation, written in binary. A binary number is a number expressed in the base-2 numeral system, also called binary numeral system, which consists of only two distinct symbols: typically 0 (zero) and 1 (one).

## Binary numbers

## Decimal: Binary:

## 0: 0

## 1: 1

## 2: 10

## 3: 11

## 4: 100

## 5: 101

## 6: 110

## 7: 111

## 8: 1000

## 9: 1001

## 10: 1010





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

How to convert binary numbers to integers (the unknown keywords and language structures will be introduced soon):

## Binary to integer

```
def bintoint(binary):  
    binary = binary[::-1]  
    num = 0  
    for i in range(len(binary)):  
        num += int(binary[i]) * 2**i  
    return num  
  
bintoint("1101001")  
  
## 105  
  
int("1101001", 2)  # compare with built-in function  
  
## 105
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## How to convert integers to binary numbers:

### Integers to binary

```
def inttobin(num):  
    binary = ""  
    if num != 0:  
        while num >= 1:  
            if num % 2 == 0:  
                binary += "0"  
                num = num / 2  
            else:  
                binary += "1"  
                num = (num - 1) / 2  
    else:  
        binary = "0"  
    return binary[::-1]  
inttobin(105)  
  
## '1101001'  
  
bin(105)[2:] # compare with built-in function  
  
## '1101001'
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numeral  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

Python offers distinct bitwise operators. Some of them will be redefined entirely different by extensions, such as, e. g., [vectorization](#).

Bit. op.	Description
$x \gg y$	Returns $x$ with the bits shifted to the left by $y$ places
$x \ll y$	Returns $x$ with the bits shifted to the right by $y$ places
$x \& y$	Does a bitwise <a href="#">and</a>
$x   y$	Does a bitwise <a href="#">or</a>
$\sim x$	Returns the complement of $x$
$x \wedge y$	Does a bitwise exclusive or

## Bitwise operators

```
a, b = 5, 7
c = a & b  # bitwise and

## a: 101
## b: 111
## c: 101

print(c)

## 5
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Bitwise operators

```
a, b = 5, 7
c = a | b  # bitwise or

## a: 101
## b: 111
## c: 111

print(c)

## 7

a = 13
b = a << 2  # bitwise shift

## a: 1101
## b: 110100

a, b = 35, 37
c = a ^ b  # bitwise exclusive or

## a: 100011
## b: 100101
## c: 000110
```



Python has only one kind of conditional statement – `if-elif-else`:

## Computer data sizes

```
bytes = 100000000 / 8 # e.g. DSL 100000
if bytes >= 1e9:
    print(f"{bytes/1e9:6.2f} GByte")
elif bytes >= 1e6:
    print(f"{bytes/1e6:6.2f} MByte")
elif bytes >= 1e3:
    print(f"{bytes/1e3:6.2f} KByte")
else:
    print(f"{bytes:6.2f} Byte")

## 12.50 MByte
```

Control flow structures may be nested in any order:

## Nestings

```
if a > 1:
    if b > 2:
        pass # a special keyword for empty blocks
```



In Python there exist two conventional *program loops* – `for-in-else`:

## Total sum

```
numbers = [7, 3, 4, 5, 6, 15]
y = 0
for i in numbers:
    y += i
print(f"The sum of 'numbers' is {y}.")

## The sum of 'numbers' is 40.
```

Lists or other collections can also be created dynamically:

## Powers of 2

```
powers = [2 ** i for i in range(11)]
teacher = ["***", "***", "*"]
grades = {star: len(teacher) - len(star) + 1 for star in teacher}

## [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
## {'***': 1, '**': 2, '*': 3}
```



Loops can skip iterations (`continue`):

## Continue the loop

```
for x in ["a", "b", "c"]:  
    a = x.upper()  
    continue  
    print(x)  
print(a)  
  
## C
```

Or a loop can be aborted instantly (`break`):

## Breaking the habit

```
y = 0  
for i in [7, 3, 4, "x", 6, 15]:  
    if not isinstance(i, int):  
        break  
    y += i  
print(f"The total sum is {y}.")  
  
## The total sum is 14.
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

For loops where the number of iterations is not known at the beginning, you use `while-else`.

Have you already noticed the keyword `else`? Python only executes the branch if it was **not terminated** by `break`:

## Favorite lottery number

```
import random
n = 0
favorite = 7
while n < 100:
    n += 1
    draw = random.randint(1, 49) # e.g. German lottery
    if draw == favorite:
        print("Got my number! :)")
        break
else:
    print("My favorite did not show up! :)")
print(f"I tried {n} times!")

## Got my number! :)
## I tried 10 times!
```





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

*Functions* are defined using the keyword `def`. The structure of *function signature* and *body* is specified by indentation, too:

## Drawing lottery numbers

```
def draw_sample(n, first=1, last=49):
    numbers = list(range(first, last + 1))
    sample = []
    for i in range(n):
        ind = random.randint(0, len(numbers) - 1)
        sample.append(numbers.pop(ind))
    sample.sort()
    return sample

draw_sample(6)
draw_sample(6, 80, 100)
draw_sample(3, first=5)

## [2, 3, 4, 16, 23, 28]
## [82, 84, 94, 95, 99, 100]
## [5, 12, 16]
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

Functions are of type `callable()`, defined as `closures`, and can be created and used like other objects:

## Prime numbers

```
def primes(n):
    numbers = [2]

    def is_prime(num):
        for i in numbers:
            if num % i == 0:
                return False
        return True

    if n == 2:
        return numbers
    for i in range(3, n + 1):
        if is_prime(i):
            numbers.append(i)
    return numbers
primes(50)

## [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Seems weird? We discuss `namespaces` in the next section.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Essential concepts

# ▶ Object-orientation



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

There are three widely known programming paradigms: *procedural*, *functional* and *object-oriented programming (OOP)*. Python supports them all.

You have learned how to handle predefined data types in Python. Actually, we have already encountered classes and instances, take for example `dict()`.

In this section you will learn the basics of dealing with (your own) classes:

- 1 References
- 2 Classes
- 3 Instances
- 4 Main principles
- 5 Garbage collection

OOP is a wide field and challenging for beginners. Don't get discouraged and, if you find deficits in yourself, read the [literature](#).



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

When you assign a variable, a *reference* to an object is set:

### Equal but not identical

```
a = ["Star", "Trek"]
b = ["Star", "Trek"]
c = a
a == b
a == c
a is b
a is c

## ['Star', 'Trek']
## ['Star', 'Trek']
## ['Star', 'Trek']
## True
## True
## False
## True
```

- Two equal but **not identical** objects are created,
- Variables `a` and `c` link to the same object.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

When we introduced [lists](#), we initially did not mention that they are a first-class example of *mutable* objects:

### Collecting grades

```
grades = [1.7, 1.3, 2.7, 2.0]
result = grades.append(1.0)
result
grades
finals = grades
finals.remove(2.7)
finals
grades
## None
## [1.7, 1.3, 2.7, 2.0, 1.0]
## [1.7, 1.3, 2.0, 1.0]
## [1.7, 1.3, 2.0, 1.0]
```

- Modifications can be *in-place* – the object itself is modified.
- Changing an object that is referenced several times could cause (un)intended consequences.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

In Python, arguments are *passed by assignment*, i.e., call-by-reference:

## Side effects

```
def last_element(x):  
    return x.pop(-1)  
  
a = stocks  
last_element(a)  
a  
  
## ['Amazon', 'Apple', 'Facebook', 'Google', 'Microsoft', 'Twitter']  
## Twitter  
## ['Amazon', 'Apple', 'Facebook', 'Google', 'Microsoft']
```

- There are **side effects**,
- Referenced *mutable* objects might be modified,
- Referenced *immutable* objects might be copied.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

We are able to make an exact copy of the object:

## Copying

```
def last_element(x):  
    y = x.copy()  
    return y.pop(-1)  
  
a = stocks  
last_element(a)  
a  
## ['Amazon', 'Apple', 'Facebook', 'Google', 'Microsoft']  
## Microsoft  
## ['Amazon', 'Apple', 'Facebook', 'Google', 'Microsoft']
```

- We receive a **new object**,
- The new object is **not identical** to the old one.





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

However, keep in mind that, in most cases, a method `copy()` will create *shallow* copies while only *deep copying* will duplicate also the contents of a mutable object with a complex structure:

### Cloning fast food

```
fastfood = [ ["burgers", "hot dogs"], ["pizza", "pasta"] ]
italian = fastfood.copy()
italian.pop(0)
american = list(fastfood)
american.pop(1)
american[0] = american[0].copy()
fastfood[0][1] = "chicken wings"
fastfood[1][0] = "risotto"
italian
american

## [['risotto', 'pasta']]
## [['burgers', 'hot dogs']]
```

Both approaches, `copy()` and `list()`, create new `list` objects containing new references to the original sub-lists. But for a *deep copy*, you have to **recursively** create duplicates of all its objects.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numeral  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

In Python everything is an object and more complex objects consist of several other objects.

In the OOP, we create objects according to patterns. These kinds of blueprints are called *classes* and are characterized by two categories of elements:

## Attributes:

Variables that represent the properties of

- an object, *object attributes*, or
- a class, named *class attributes*.

## Methods:

Functions that are defined within a class:

- (*non-static*) *methods* can access all attributes, while
- *static methods* can only access class attributes.

Every generated object is an *instance* of such a construction plan.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

Specifically, we want to create “rectangle object” and define a separate Rectangle class for it:

## Rectangle class

```
class Rectangle:
    width = 0
    height = 0

    def area(self):
        return self.width * self.height

myrectangle = Rectangle()
myrectangle.width = 10
myrectangle.height = 20
myrectangle.area()

## 200
```

- New classes are defined using the keyword `class`,
- The variable `self` always refers to the instance itself.



We add a *constructor* (method) `__init__()`, that is called to *initialize* an object of `Rectangle`:

## Rectangle class with constructor

```
class Rectangle:
    width = 0
    height = 0

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

myrectangle = Rectangle(15, 30)
myrectangle.area()

## 450
```

In our example, we use the constructor to set the attributes. Methods with names matching `__fun__()` have a special, standardized meaning in Python.



One of the most important concepts of OOP is *inheritance*. A class inherits all attributes and methods of its *parent class* and can *add new* or *overwrite* existing ones:

## Square inherits Rectangle

```
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

    def diagonal(self):
        return (self.width**2 + self.height**2)**0.5

mysquare = Square(15)

print(f"Area: {mysquare.area()}")
print(f"Diagonal length: {mysquare.diagonal():7.4f}")

## Area: 225
## Diagonal length: 21.2132
```

The methods of the parent class, including the constructor, may be referenced by `super()`.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numeral  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

You do not have to worry about memory management in Python. The *garbage collector* will tidy up for you.

If there are no more references to an object, it is automatically disposed of by the garbage collector:

## Garbage collection in action

```
class Dog:
    def __del__(self):
        print("Woof! The dogcatcher got me! Entering the void.. :(")
        # My old dog on a leash
mydog = Dog()
        # A new dog is born
newdog = Dog()
        # Using my leash for the new dog
mydog = newdog

## Woof! The dogcatcher got me! Entering the void.. :(
```

The *destructor* `__del__()` is executed as the last act before an object gets deleted.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Everyone involved in programming will encounter errors of various types. These errors can be stressful and annoying but being aware of the basic types of errors that can occur will give you the chance to handle them.

Seeing the line **SyntaxError** may let you think "oh no, I've done everything wrong", but errors are normal and even experienced programmers face them frequently. Hints on error handling:

- Dissect the error: Find the line in the error message that is specified. Many errors have messages that are not important to the actual error. In Python you often find the important information at the end of the error message.
- Errors are often oversights: In most cases the error message will give you the line in your code where the error occurred.
- Search the web: If you are not able to fix the errors on your own, copy the error message into a search engine and read through the results. Probably someone else also had this problem and the community already found a solution.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Data formats and handling

NumPy package

Array basics

Linear algebra

## Visual illustrations

Pandas package

Series

DataFrame

Import/Export data

## Applications

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Time series

Moving window

Financial applications

Optimization

A Python program terminates immediately as it encounters an error. In Python, errors can be either *syntax errors* or *exceptions*. Syntax errors occur when the parser detects a wrong sequence in the Python code. An arrow indicates the exact position of the syntax error:

## Syntax Error

```
## print("Hello Word"))  
## File "<stdin>", line 1  
##     print("Hello World"))  
##                          ^  
##     SyntaxError: invalid syntax
```

An exception occurs whenever a syntactically correct Python code results in an error:

## Exception

```
a = 0 / 0  
## <stdin> in <module>()  
## ----> 1 a = 0 / 0  
## ZeroDivisionError: division by zero
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Exceptions appear in different types and the type is printed as a part of the error message. The next example shows three common built-in exceptions:

### Frequent exception

```
0 / 0
```

```
## <stdin> in <module>()
## ----> 1 0 / 0
## ZeroDivisionError: division by zero
```

```
3 + a
```

```
## <stdin> in <module>()
## ----> 1 3 + a
## NameError: name 'a' is not defined
```

```
3 + "2"
```

```
## <stdin> in <module>()
## ----> 1 3 + "2"
## TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

A list of all exception classes of the standard library can be found [here](#).



## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

When an exception occurs, the Python interpreter throws an error message and exits. But in most situations, you do not want your whole program to stop.

The `try` block can test a block of code for errors.  
The `except` block lets you handle the error.

## Try and except

```
try:
    print(abc)
except:
    print("An exception occurred")

## An exception occurred
```

The statement above will raise an error, because the variable `abc` is not defined.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

You can define multiple exception blocks. For example, if you want to execute code when you expect a special kind of error to occur:

### Multiple exception blocks

```
try:
    print(abc)
except NameError:
    print("Variable abc is not defined")
except:
    print("Something else went wrong")

## Variable abc is not defined

try:
    0 / 0
except NameError:
    print("Variable abc is not defined")
except:
    print("Something else went wrong")

## Something else went wrong
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Complementary, like for `if-else`, the `else` keyword defines a block of code to be executed if no errors were thrown:

### Else exception

```
try:
    print("Hello World")
except:
    print("Something went wrong")
else:
    print("Everything is okay")

## Hello World
## Everything is okay
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

The `finally` block will be executed regardless if the `try` block raises an error or not. Hence, you can make sure the code is run:

### Finally exception

```
try:
    print(abc)
except:
    print("Something went wrong")
finally:
    print("This will always be displayed")

## Something went wrong
## This will always be displayed

try:
    print("Hello World")
except:
    print("Something went wrong")
finally:
    print("This will always be displayed")

## Hello World
## This will always be displayed
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Built-in exceptions are raised whenever pre-defined interpreter errors occur. In some situations you might want to raise exceptions on your own:

The `raise` keyword is used to raise an exception.

In the following, the interpreter raises an error if the variable `x` is lower than 0:

## Raise exception

```
x = -3
if x < 0:
    raise Exception("Sorry, 'x' is lower than 0.")

## <stdin> in <module>()
## ----> 3 raise Exception(Sorry, 'x' is lower than 0.)
## Exception: Sorry, 'x' is lower than 0.
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

**LBYL:** *Look before you leap.*

**EAFP:** *It is easier to ask forgiveness than it is to get permission.*

LBYL and EAFP are two techniques to deal (i.e., avoid) with exceptions. In short, in LBYL you first check whether something will succeed and only proceed if it does. EAFP means that you do what you expect and if an exception might occur, you deal with it:

## LBYL

```
if x != 0:
    print(10 / x)
```

## EAFP

```
try:
    print(10 / x)
except ZeroDivisionError:
    pass
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

So, why use EAFP although it needs more lines of code?

- Often, the code is more readable and straight.
- Explicit is better than implicit (Zen of Python, see [below](#)).
- Best performance in case no exception is raised.
- Detailed exception handling. You can not only consider errors, but also different kinds of errors and then proceed differently.

## EAFP

```
try:
    print(10 / x)
except ZeroDivisionError:
    print("Zero division")
except NameError:
    print("Variable 'x' is not defined")
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Python has multiple built-in exceptions which terminate your program when something goes wrong. But you can also create custom exceptions that serve specific purposes.

Your own exception can be implemented by defining a new class which derives from the `Exception` class or a subclass:

### User-defined exception

```
class ValueTooLargeError(Exception):
    """Raised when the input value is too large"""
    pass

x = 3
try:
    if x > 2:
        raise ValueTooLargeError
except ValueTooLargeError:
    print("The number is too large.")

## The number is too large.
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

We have already come into contact with *namespaces* in Python many times. These are hierarchically linked layers in which the references to objects are defined. A rough distinction is made between

- the *global* namespace, and
- the *local* namespace.

The global namespace is the *outermost environment* whose references are known by all objects.

On the other hand, locally defined references are only known in a local, i.e., *internal environment*.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Reference names from the local namespace mask the same names in an outer or in the global namespace:

## Namespaces

```
def multiplier(x):  
    x = 4 * x  
    return x  
x = "OH"  
multiplier("AH")  
multiplier(x)  
x  
## OH  
## AHAHAHAH  
## OHOH OH  
## OH
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

In fact, **functions** defined in Python are themselves objects that remember and can access their own context where they were created. This concept comes from functional programming and is called *closure*:

## Closures

```
def gen_multiplier(a):  
    def fun(x):  
        return a * x  
    return fun  
  
multi1 = gen_multiplier(4)  
multi2 = gen_multiplier(5)  
multi1  
multi1("EH")  
multi2("EH")  
  
## <function gen_multiplier.<locals>.fun at 0x7f654804cd30>  
## EHEHEHEH  
## EHEHEHEHEH
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

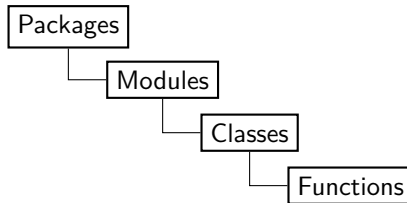
Time series

Moving window

Financial applications

Optimization

In order to provide, maintain and extend modular functionality with Python, its code containing components can be described hierarchically:



The organization in Python is very straightforward and is based on the local namespaces mentioned before.

When you download and use new *packages*, such as *NumPy* for numerical programming in the next chapter, the packages are loaded and the namespaces initialized.

The development of custom packages is an advanced topic and not essential for a reasonable code structure of small projects, as it is in other programming languages.



## Essential concepts

Getting started

Procedural programming

Object-orientation

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

*Modules* provide classes and functions via namespaces. It is Python code that is executed in a local namespace and whose classes and functions you can import. Basically, there are the following alternatives how to *import* from an module:

## Import statements

```
import datetime
import datetime as dt
from datetime import date, timedelta
from datetime import *
```

```
dt.date.today()
dt.timedelta.days
```

```
date.today()
timedelta.days
```

```
datetime.now()
```

In the latter case, all classes and functions, but no instances, are imported from the `datetime` namespace.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Data formats and handling

NumPy package

Array basics

Linear algebra

## Visual illustrations

Pandas package

Series

DataFrame

Import/Export data

## Applications

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Time series

Moving window

Financial applications

Optimization

A Python installation ships with a *standard library* consisting of *built-in modules*. These modules provide standardized solutions for many problems that occur in everyday programming - “batteries included”. For example, they provide access to system functionality such as file management. The [Python Docs](#) give an overview of all build-in modules.

## Usage of build-in modules

```
import math
from random import randint
```

```
math.pi
```

```
## 3.141592653589793
```

```
math.factorial(5)
```

```
## 120
```

```
randint(10, 20)
```

```
## 18
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Often you might want to use extended functionality. Python has a large and active community of users who make their developments publicly available under open source license terms. Packages are containers of modules which can be imported and used within your Python code.

These third-party packages can be installed comfortably by using the (command line) package manager *pip*. The [Python Package Index](#) provides an overview of the thousands of packages available. Basic commands for maintaining, for example, the installation of the package “numpy”:

- Installing the package: `pip install numpy`
- Upgrading the package: `pip install --upgrade numpy`
- Installing the package locally for the current user:  
`pip install --user numpy`
- Uninstalling the package: `pip uninstall numpy`





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Example: *OpenCV* is a package for image processing in Python. Here you can see how the installation proceeds in a Unix terminal.

```
~$ pip install opencv-python
```

```
Collecting opencv-python
  Downloading https://files.pythonhosted.org/packages/37/49/874d119948a5a084a7eb
e98308214098ef3471d76ab74200f9800efeeef15/opencv_python-4.0.0.21-cp36-cp36m-manyl
inux1_x86_64.whl (25.4MB)
    100% |████████████████████████████████████████| 25.4MB 523kB/s
Requirement already satisfied: numpy>=1.11.3 in /usr/local/lib/python3.6/dist-pa
ckages (from opencv-python) (1.15.4)
Installing collected packages: opencv-python
Successfully installed opencv-python-4.0.0.21
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Your Python projects will become complex and you will need to maintain the codes properly. Therefore, one can break a large, unwieldy programming task into separate, more manageable modules. Modules can be written in Python itself or in C, but here we keep focussing on the Python language.

Creating modules in Python is very straightforward - a Python module is a file containing Python code, for example:

```
s = "Hello world!"  
l = [1, 2, 3, 5, 5]  
  
def add_one(n):  
    return n + 1
```

File: **mymodule.py**



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

If you import the module **mymodule**, the interpreter looks in the current working directory for a file **mymodule.py**, reads and interprets its contents and makes its namespace available:

### Usage of own modules

```
import mymodule
mymodule.s
mymodule.l
mymodule.add_one(5)

## Hello world!
## [1, 2, 3, 5, 5]
## 6
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

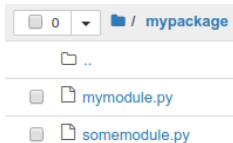
Time series

Moving window

Financial applications

Optimization

Large projects could require more than one module. Packages allow to structure the modules and their namespaces hierarchically by using the *dot notation*. They are simple folders containing modules and (sub-)packages. Consider the following structure:



The directory **mypackage** contains two modules which we can import separately:

## Usage of own package

```
import mypackage.mymodule
import mypackage.somemodule
mypackage.mymodule.add_one(4)

## 5
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

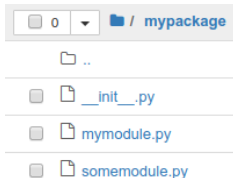
Time series

Moving window

Financial applications

Optimization

If a package directory contains a file `__init__.py`, its code is invoked when the package gets imported. The directory **mypackage**, now, contains the two modules and the initialization file:



The file `__init__.py` can be empty but can also be used for package initialization purposes.



## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## The Zen of Python

```
import this

## The Zen of Python, by Tim Peters
##
##
## Beautiful is better than ugly.
## Explicit is better than implicit.
## Simple is better than complex.
## Complex is better than complicated.
## Flat is better than nested.
## Sparse is better than dense.
## Readability counts.
## Special cases aren't special enough to break the rules.
## Although practicality beats purity.
## Errors should never pass silently.
## Unless explicitly silenced.
## In the face of ambiguity, refuse the temptation to guess.
## ...
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

A selection of exciting topics that are among the advanced basics but are not covered in this lecture:

- Dynamic language concepts, such as duck typing,
- Further, complex type classes, such as `ChainMap` or `OrderedDict`,
- Iterators and generators in detail,
- Exception handling, raising exceptions, catching errors,
- Debugging, introspection and annotations.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Numerical programming

## 2.1 NumPy package

## 2.2 Array basics

## 2.3 Linear algebra





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

# Numerical programming

## ▶ NumPy package



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization



The *Numerical Python* package NumPy provides efficient tools for scientific computing and data analysis:

- `np.array()`: Multidimensional array capable of doing fast and efficient computations,
- Built-in mathematical functions on arrays without writing loops,
- Built-in linear algebra functions.

Import NumPy

```
import numpy as np
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

### NumPy package

- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Element-wise addition

```
vec1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
vec2 = np.array(vec1)
vec1 + vec1

## [1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9]

vec2 + vec2

## array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])

for i in range(len(vec1)):
    vec1[i] += vec1[i]
vec1

## [2, 4, 6, 8, 10, 12, 14, 16, 18]
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Matrix multiplication

```
mat1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
mat2 = np.array(mat1)
np.dot(mat2, mat2)

## array([[ 30,  36,  42],
##        [ 66,  81,  96],
##        [102, 126, 150]])

mat3 = np.zeros([3, 3])
for i in range(3):
    for k in range(3):
        for j in range(3):
            mat3[i][k] = mat3[i][k] + mat1[i][j] * mat1[j][k]

mat3

## array([[ 30.,  36.,  42.],
##        [ 66.,  81.,  96.],
##        [102., 126., 150.]])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Time comparison

```
import time
mat1 = np.random.rand(50, 50)
mat2 = np.array(mat1)
t = time.time()
mat3 = np.dot(mat2, mat2)
nptime = time.time() - t
mat3 = np.zeros([50, 50])
t = time.time()
for i in range(50):
    for k in range(50):
        for j in range(50):
            mat3[i][k] = mat3[i][k] + mat1[i][j] * mat1[j][k]
pytime = time.time() - t
times = str(pytime / nptime)
print("NumPy is " + times + " times faster!")

## NumPy is 41.55256533479793 times faster!
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

# Numerical programming

## ▶ Array basics



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`np.array(list)`: Converts python list into NumPy arrays.  
`array.ndim`: Returns Dimension of the array.  
`array.shape`: Returns shape of the array as a list.

## Creation

```
arr1 = [4, 8, 2]
arr1 = np.array(arr1)
arr2 = np.array([24.3, 0., 8.9, 4.4, 1.65, 45])
arr3 = np.array([[4, 8, 5], [9, 3, 4], [1, 0, 6]])
arr1.ndim

## 1

arr3.shape

## (3, 3)
```

From now on, the name `array` refers to an `np.array()`.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## NumPy package

NumPy package

## Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`np.arange(start, stop, step)`: Creates vector of values from `start` to `stop` with step width `step`.

`np.zeros((rows, columns))`: Creates array with all values set to 0.

`np.identity(n)`: Creates identity matrix of dimension `n`.

## Creation functions

```
np.zeros((4, 3))
```

```
## array([[0., 0., 0.],  
##        [0., 0., 0.],  
##        [0., 0., 0.],  
##        [0., 0., 0.]])
```

```
np.arange(6)
```

```
## array([0, 1, 2, 3, 4, 5])
```

```
np.identity(3)
```

```
## array([[1., 0., 0.],  
##        [0., 1., 0.],  
##        [0., 0., 1.]])
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`np.linspace(start, stop, n)`: Creates vector of `n` evenly divided values from `start` to `stop`.

`np.full((row, column), k)`: Creates array with all values set to `k`.

## Array creation

```
np.linspace(0, 80, 5)
```

```
## array([ 0., 20., 40., 60., 80.])
```

```
np.full((5, 4), 7)
```

```
## array([[7, 7, 7, 7],  
##        [7, 7, 7, 7],  
##        [7, 7, 7, 7],  
##        [7, 7, 7, 7],  
##        [7, 7, 7, 7]])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`np.random.rand(rows, columns)`: Creates array of random floats between zero and one.

`np.random.randint(k, size=(rows, columns))`: Creates array of random integers between 0 and  $k-1$ .

## Array of random numbers

```
np.random.rand(3, 3)
```

```
## array([[0.01014591, 0.55955228, 0.48103055],  
##        [0.30368877, 0.99078572, 0.61537046],  
##        [0.83572553, 0.45976471, 0.63241975]])
```

```
np.random.randint(10, size=(5, 4))
```

```
## array([[7, 9, 7, 8],  
##        [0, 6, 7, 5],  
##        [7, 3, 4, 7],  
##        [9, 4, 4, 8],  
##        [8, 0, 6, 1]])
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package

### Array basics

- Linear algebra

## Data formats and handling

- Pandas package

- Series

- DataFrame

- Import/Export data

## Visual illustrations

- Matplotlib package

- Figures and subplots

- Plot types and styles

- Pandas layers

## Applications

- Time series

- Moving window

- Financial applications

- Optimization

## Reference

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
arr = arr3  
arr[1, 1] = 777  
arr3
```

```
## array([[ 4,    8,    5],  
##        [ 9, 777,    4],  
##        [ 1,    0,    6]])
```

```
arr3[1, 1] = 3
```

## call-by-reference

`arr = arr3` binds `arr` to the existing `arr3`. They both refer to the same object.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`array.copy()`: Copies an array without reference (call-by-value).

### Copy

```
arr3

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])

arr = arr3.copy()
arr[1, 1] = 777
arr3

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])
```

### Reference

```
arr3

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])

arr = arr3
arr[1, 1] = 777
arr3

## array([[ 4,  8,  5],
##        [ 9, 777, 4],
##        [ 1,  0, 6]])

arr3[1, 1] = 3
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Function	Description
<code>array</code>	Convert input array in NumPy array
<code>arange(start,stop,step)</code>	Creates array from given input
<code>ones</code>	Creates array containing only ones
<code>zeros</code>	Creates array containing only zeros
<code>empty</code>	Allocating memory without specific values
<code>eye, identity</code>	Creates $N \times N$ identity matrix
<code>linspace</code>	Creates array of evenly divided values
<code>full</code>	Creates array with values set to one number
<code>random.rand</code>	Creates array of random floats
<code>random.randint</code>	Creates array of random int



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`array.dtype`: Returns the type of array.

`array.astype(np.type)`: Conducts a manual typecast.

## Data types

```
arr1.dtype
```

```
## dtype('int64')
```

```
arr2.dtype
```

```
## dtype('float64')
```

```
arr1 = arr1 * 2.5
```

```
arr1.dtype
```

```
## dtype('float64')
```

```
arr1 = (arr1 / 2.5).astype(np.int64)
```

```
arr1.dtype
```

```
## dtype('int64')
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Element-wise operations

Calculation operators on NumPy arrays operate element-wise.

## Element-wise operations

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
arr3 + arr3
```

```
## array([[ 8, 16, 10],  
##        [18,  6,  8],  
##        [ 2,  0, 12]])
```

```
arr3**2
```

```
## array([[16, 64, 25],  
##        [81,  9, 16],  
##        [ 1,  0, 36]])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Matrix multiplication

Operator `*` applied on arrays does not do the matrix multiplication.

## Element-wise operations

```
arr3 * arr3
```

```
## array([[16, 64, 25],  
##        [81,  9, 16],  
##        [ 1,  0, 36]])
```

```
arr = np.ones((3, 2))  
arr
```

```
## array([[1., 1.],  
##        [1., 1.],  
##        [1., 1.]])
```

```
arr3 * arr  # not defined for element-wise multiplication
```

```
## ValueError: operands could not be broadcast together
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`array[index]`: Selects the value at position `index` from the data.

## Indexing with an integer

```
arr = np.arange(10)
arr

## array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr[4]

## 4

arr[-1]

## 9
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`array[start : stop : step]`: Selects a subset of the data.

## Slicing in one dimension

```
arr = np.arange(10)
arr

## array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

arr[3:7]

## array([3, 4, 5, 6])

arr[1:]

## array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package

### Array basics

- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Slicing in one dimension with steps

```
arr[:7]
```

```
## array([0, 1, 2, 3, 4, 5, 6])
```

```
arr[-3:]
```

```
## array([7, 8, 9])
```

```
arr[::-1]
```

```
## array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

```
arr[::-2]
```

```
## array([0, 2, 4, 6, 8])
```

```
arr[5:-1]
```

```
## array([9, 8, 7, 6])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Slicing in higher dimensions

In  $n$ -dimensional arrays the element at each index is an  $(n - 1)$ -dimensional array.

## Indexing rows

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
vec = arr3[1]
```

```
vec
```

```
## array([9, 3, 4])
```

```
arr3[-1]
```

```
## array([1, 0, 6])
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Slicing in two dimensions

```
arr3
```

```
## array([[4, 8, 5],  
##       [9, 3, 4],  
##       [1, 0, 6]])
```

```
arr3[0:2, 0:2]
```

```
## array([[4, 8],  
##       [9, 3]])
```

```
arr3[2:, :]
```

```
## array([[1, 0, 6]])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization





	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

Figure: Python for Data Analysis (2017) on page 99



## Essential concepts

Getting started

Procedural programming

Object-orientation

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

So far, selecting by **index numbers or slicing** belongs to *basic indexing* in NumPy. With basic indexing you get **NO COPY** of your data but a so-called *view* on the existing data set – a different perspective.

A view on an array can be seen as a reference to a rectangular memory area of its values. The view is intended to

- edit a rectangular part of a matrix, e.g., a sub-matrix, a column, or a single value,
- change the shape of the matrix or the arrangement of its elements, e.g., transpose or reshape a matrix,
- change the visual representation of values, e.g., to cast a `float` array into an `int` array,
- map the values in other program areas.

The crucial point here is that for efficiency reasons data arrays in your working memory do not have to be copied again and again for simple index operations, which would require an excessive additional effort writing to the computer memory.



## Essential concepts

[Getting started](#)[Procedural programming](#)[Object-orientation](#)

## Numerical programming

[NumPy package](#)[Array basics](#)[Linear algebra](#)

## Data formats and handling

[Pandas package](#)[Series](#)[DataFrame](#)[Import/Export data](#)

## Visual illustrations

[Matplotlib package](#)[Figures and subplots](#)[Plot types and styles](#)[Pandas layers](#)

## Applications

[Time series](#)[Moving window](#)[Financial applications](#)[Optimization](#)

A view is created automatically when you do basic indexing such as slicing:

### Create a view by slicing

```
column = arr3[:, 1]
column

## array([8, 3, 0])

column.base

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])

column[1] = 100
arr3

## array([[ 4,    8,    5],
##        [ 9, 100,    4],
##        [ 1,    0,    6]])
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Create a view by slicing

```
elem = column[1:2]
elem.base

## array([[ 4,   8,   5],
##        [ 9, 100,   4],
##        [ 1,   0,   6]])

elem[0] = 3
arr3

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])
```

- The middle column is a view of the base array referenced by `arr3`,
- Any changes to the values of a view directly affect the base data,
- A view of a view is another view on the same base matrix.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

In addition, an array contains methods and attributes that return a view of its data:

## Obtain a view

```
arr3_t = arr3.T
arr3_t

## array([[4, 9, 1],
##        [8, 3, 0],
##        [5, 4, 6]])

arr3_t.flags.owndata

## False

arr3_r = arr3.reshape(1, 9)
arr3_r

## array([[4, 8, 5, 9, 3, 4, 1, 0, 6]])

arr3_t.flags.owndata

## False
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Obtain a view

```
arr3_v = arr3.view()
arr3_v.flags.owndata

## False
```

- The transposed matrix is a predefined view that is available as an attribute,
- Reshaping is also just another way of looking at the same set of data,
- By means of the method `view()` you create a view with an identical representation.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

The behavior described above changes with *advanced indexing*, i. e., if at least one component of the index tuple is not a scalar index number or slice. The case of *fancy indexing* is described below:

## Advanced and basic indexing

```
arr3
## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])

arr = arr3[[0, 2], [0, 2]]
arr
## array([4, 6])

arr.base
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Advanced and basic indexing

```
arr = arr3[0:3:2, 0:3:2]
```

```
arr
```

```
## array([[4, 5],  
##       [1, 6]])
```

```
arr.base
```

```
## array([[4, 8, 5],  
##       [9, 3, 4],  
##       [1, 0, 6]])
```

- Contrary to intuition, fancy indexing does not return a  $(2 \times 2)$ -matrix, but a vector of the matrix elements  $(0,0)$  and  $(2,2)$ . This is a complete copy – a new object and not a view to the original matrix.
- A submatrix (view) with the corner elements of the initial matrix can be obtained with slicing.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

A boolean array is a NumPy array with boolean `True` and `False` values. Such an array can be created by applying a `comparison operator` on NumPy arrays.

### Boolean arrays

```
bool_arr = (arr3 < 5)
bool_arr

## array([[ True, False, False],
##        [False,  True,  True],
##        [ True,  True, False]])

bool_arr1 = (arr3 == 0)
bool_arr1

## array([[False, False, False],
##        [False, False, False],
##        [False,  True, False]])
```

The comparison operators on arrays can be combined by means of NumPy redefined `bitwise operators`.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Boolean arrays and bitwise operators

```
a = np.array([3, 8, 4, 1, 9, 5, 2])
b = np.array([2, 3, 5, 6, 11, 15, 17])
c = (a % 2 == 0) | (b % 3 == 0) # or
c

## array([False,  True,  True,  True, False,  True,  True])

d = (a > b) ^ (a % 2 == 1) # exclusive or
d

## array([False,  True, False,  True,  True,  True, False])

c ^ d # exclusive or

## array([False, False,  True, False,  True, False,  True])
```

## Boolean arrays

Logical operations on NumPy arrays work in a similar way compared to [bitwise operators](#).



## Essential concepts

[Getting started](#)[Procedural programming](#)[Object-orientation](#)

## NumPy package

[NumPy package](#)[Array basics](#)[Linear algebra](#)

## Data formats and handling

[Pandas package](#)[Series](#)[DataFrame](#)[Import/Export data](#)

## Visual illustrations

[Matplotlib package](#)[Figures and subplots](#)[Plot types and styles](#)[Pandas layers](#)

## Applications

[Time series](#)[Moving window](#)[Financial applications](#)[Optimization](#)

Boolean arrays can be used to select elements of other NumPy arrays. If `x` is an array and `y` is a boolean array of the same dimension, then `a[b]` selects all the elements of `x`, for which the corresponding value (at the same position) of `y` is `True`.

## Indexing with boolean arrays

```
arr3

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])

y = arr3 % 2 == 0
y

## array([[ True,  True, False],
##        [False, False,  True],
##        [False,  True,  True]])

arr3[y]

## array([4, 8, 4, 0, 6])
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## NumPy package

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Conditional indexing allows you using boolean arrays to select subsets of values and to avoid loops. Applying [comparison operator](#) on arrays, every element of the array is tested, if it corresponds to the logical condition. Consider an application setting all even numbers to 5:

### Find and replace values in arrays

```
a, b = arr3.copy(), arr3.copy()
for i in range(a.shape[0]):
    for j in range(a.shape[1]):
        if a[i, j] % 2 == 0:
            a[i, j] = 5
```

```
b[b % 2 == 0] = 5
b
```

```
## array([[5, 5, 5],
##        [9, 3, 5],
##        [1, 5, 5]])
```

```
np.allclose(a, b)
```

```
## True
```



## Essential concepts

Getting started  
Procedural programming  
Object-orientation

## Numerical programming

NumPy package  
Array basics  
Linear algebra

## Data formats and handling

Pandas package  
Series  
DataFrame  
Import/Export data

## Visual illustrations

Matplotlib package  
Figures and subplots  
Plot types and styles  
Pandas layers

## Applications

Time series  
Moving window  
Financial applications  
Optimization

### Find and replace values in arrays, condition: equal

```
arr3

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])

arr = arr3.copy()
arr[arr == 4] = 100
arr

## array([[100,  8,  5],
##        [ 9,  3, 100],
##        [ 1,  0,  6]])
```

- In this example, `arr == 4` creates a boolean array as described [before](#) which is then used to index the array `arr`.
- Finally, every element of `arr` which is *marked True* according to the boolean index array will be set to `100`.



## Step 1a

**Integer indexing** `array[row index, column index]`: Indexing an  $n$ -dimensional array with  $n$  integer indices returns the single value at this position.

### Best practice Step 1a

```
mat = np.arange(12).reshape((3, 4))  
mat
```

```
## array([[ 0,  1,  2,  3],  
##        [ 4,  5,  6,  7],  
##        [ 8,  9, 10, 11]])
```

```
mat[2, 2]
```

```
## 10
```

```
mat[0, -1]
```

```
## 3
```

Keep in mind that, in this case only, the results are not arrays but values!



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Step 1b

**Integer indexing** `array[row index]`: In  $n$ -dimensional arrays, the element at each index is an  $(n - 1)$ -dimensional array.

### Best practice Step 1b

```
mat = np.arange(12).reshape((3, 4))
mat

## array([[ 0,  1,  2,  3],
##        [ 4,  5,  6,  7],
##        [ 8,  9, 10, 11]])

mat[2]

## array([ 8,  9, 10, 11])

mat[0]

## array([0, 1, 2, 3])
```

By specifying the `row index` only, we create arrays which are views.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Step 2a

**Slicing** array[start : stop : step]: Slicing can be used separately for rows and columns.

### Best practice Step 2a

```
mat = np.arange(12).reshape((3, 4))
```

```
mat
```

```
## array([[ 0,  1,  2,  3],
##        [ 4,  5,  6,  7],
##        [ 8,  9, 10, 11]])
```

```
mat[0:2]
```

```
## array([[0, 1, 2, 3],
##        [4, 5, 6, 7]])
```

```
mat[0:2, ::2]
```

```
## array([[0, 2],
##        [4, 6]])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Step 2b

A frequent task is to get a specific row or column of an array. This can be done easily by slicing.

### Best practice Step 2b

```
mat

## array([[ 0,  1,  2,  3],
##        [ 4,  5,  6,  7],
##        [ 8,  9, 10, 11]])

row = mat[1]  # get second row
column = mat[:, 2]  # get third column

row

## array([4, 5, 6, 7])

column

## array([ 2,  6, 10])
```

Slicing with `[:]` means to take every element from the first to the last.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Step 3

**Fancy indexing** `array[rows list, columns list]`: Return a one dimensional array with the values at the index tuples specified elementwise by the index lists.

### Best practice Step 3

```
mat = np.arange(12).reshape((3, 4))
mat
```

```
## array([[ 0,  1,  2,  3],
##        [ 4,  5,  6,  7],
##        [ 8,  9, 10, 11]])
```

```
mat[[1, 2], [1, 2]]
```

```
## array([ 5, 10])
```

```
mat[[0, -1], [-1]]
```

```
## array([ 3, 11])
```

The index lists might also contain just a single element.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Step 4

**Conditional indexing:** Applying **comparison operators** to arrays, the boolean operations are evaluated elementwise in a vectorized fashion.

### Best practice Step 4

```
bool_mat = mat > 0
bool_mat

## array([[False,  True,  True,  True],
##        [ True,  True,  True,  True],
##        [ True,  True,  True,  True]])

mat[bool_mat] = 111 # equivalent to mat[mat > 0] = 111
mat

## array([[ 0, 111, 111, 111],
##        [111, 111, 111, 111],
##        [111, 111, 111, 111]])
```





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming  
NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Step 5

Replacing values in arrays. Assigning a slice of an array to new values, the shape of slice must be considered.

### Best practice Step 5

```
mat[0] = np.array([3, 2, 1]) # Fails because the shapes do not fit
## Error: could not broadcast array from shape (3) into shape (4)

mat[2, 3] = 100
mat[:, 0] = np.array([3, 3, 3])
mat

## array([[ 3, 111, 111, 111],
##        [ 3, 111, 111, 111],
##        [ 3, 111, 111, 100]])

mat[1:3, 1:3] = np.array([[0, 0], [0, 0]])
mat

## array([[ 3, 111, 111, 111],
##        [ 3,   0,   0, 111],
##        [ 3,   0,   0, 100]])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming  
NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`array.reshape((rows, columns))`: Reshapes an existing array.

`array.resize((rows, columns))`: Changes array shape to `rows` x `columns` and fills new values with `0`.

## Reshape

```
arr = np.arange(15)
arr.reshape((3, 5))
```

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

```
arr = np.arange(15)
arr.resize((3, 7))
arr
```

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



## Essential concepts

[Getting started](#)[Procedural programming](#)[Object-orientation](#)

## Numerical programming

[NumPy package](#)[Array basics](#)[Linear algebra](#)

## Data formats and handling

[Pandas package](#)[Series](#)[DataFrame](#)[Import/Export data](#)

## Visual illustrations

[Matplotlib package](#)[Figures and subplots](#)[Plot types and styles](#)[Pandas layers](#)

## Applications

[Time series](#)[Moving window](#)[Financial applications](#)[Optimization](#)

`np.append(array, value)`: Appends value to the end of array.  
`np.insert(array, index, value)`: Inserts values before index.  
`np.delete(array, index, axis)`: Deletes row or column on index.

## Naming

```
a = np.arange(5)
a = np.append(a, 8)
a = np.insert(a, 3, 77)
print(a)

## [ 0  1  2 77  3  4  8]

a.resize((3, 3))
np.delete(a, 1, axis=0)

## array([[0, 1, 2],
##        [8, 0, 0]])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`np.concatenate((arr1, arr2), axis):` Joins a sequence of arrays along an existing axis.

`np.split(array, n):` Splits an array into multiple sub-arrays.

`np.hsplit(array, n):` Splits an array into multiple sub-arrays horizontally.

## Naming

```
np.concatenate((a, np.arange(6).reshape(2, 3)), axis=0)
```

```
## array([[ 0,  1,  2],
##        [77,  3,  4],
##        [ 8,  0,  0],
##        [ 0,  1,  2],
##        [ 3,  4,  5]])
```

```
np.split(np.arange(8), 4)
```

```
## [array([0, 1]), array([2, 3]), array([4, 5]), array([6, 7])]
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`array.T`: Returns the transposed array (as a view).

## Transpose

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
arr3.T
```

```
## array([[4, 9, 1],  
##        [8, 3, 0],  
##        [5, 4, 6]])
```

```
np.eye(3).T
```

```
## array([[1., 0., 0.],  
##        [0., 1., 0.],  
##        [0., 0., 1.]])
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`np.dot(arr1, arr2)`: Conducts a matrix multiplication of `arr1` and `arr2`. The `@` operator can be used instead of the `np.dot()` function.

## Matrix multiplication

```
res = np.dot(arr3, np.arange(18).reshape((3, 6)))
```

```
res
```

```
## array([[108, 125, 142, 159, 176, 193],
##        [ 66,  82,  98, 114, 130, 146],
##        [ 72,  79,  86,  93, 100, 107]])
```

```
res2 = arr3 @ np.arange(18).reshape((3, 6))
```

```
res2
```

```
## array([[108, 125, 142, 159, 176, 193],
##        [ 66,  82,  98, 114, 130, 146],
##        [ 72,  79,  86,  93, 100, 107]])
```

```
np.allclose(res, res2)
```

```
## True
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package

### Array basics

- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Element-wise functions

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
np.sqrt(arr3)
```

```
## array([[2.          , 2.82842712, 2.23606798],  
##        [3.          , 1.73205081, 2.          ],  
##        [1.          , 0.          , 2.44948974]])
```

```
np.exp(arr3)
```

```
## array([[5.45981500e+01, 2.98095799e+03, 1.48413159e+02],  
##        [8.10308393e+03, 2.00855369e+01, 5.45981500e+01],  
##        [2.71828183e+00, 1.00000000e+00, 4.03428793e+02]])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Function	Description
abs	Absolute value of integer and floating point
sqrt	Square root
exp	Exponential function
log, log10, log2	Natural logarithm, log base 10, log base 2
sign	Sign (1 : positiv, 0: zero, -1 : negative)
ceil	Rounding up to integer
floor	Round down to integer
rint	Round to nearest integer
modf	Returns fractional parts
sin, cos, tan, sinh, cosh, tanh, arcsin, ...	





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Binary

```
x = np.array([3, -6, 8, 4, 3, 5])
```

```
y = np.array([3, 5, 7, 3, 5, 9])
```

```
np.maximum(x, y)
```

```
## array([3, 5, 8, 4, 5, 9])
```

```
np.greater_equal(x, y)
```

```
## array([ True, False,  True,  True, False, False])
```

```
np.add(x, y)
```

```
## array([ 6, -1, 15,  7,  8, 14])
```

```
np.mod(x, y)
```

```
## array([0, 4, 1, 1, 3, 5])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Function	Description
add	Add elements of arrays
subtract	Subtract elements in the second from the first array
multiply	Multiply elements
divide	Divide elements
power	Raise elements in first array to powers in second
maximum	Element-wise maximum
minimum	Element-wise minimum
mod	Element-wise modulus
greater, less, equal gives boolean	



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`np.meshgrid(array1, array2)`: Returns coordinate matrices from coordinate arrays.

Evaluate the function  $f(x, y) = \sqrt{x^2 + y^2}$  on a 10 x 10 grid

```
p = np.arange(-5, 5, 0.01)
x, y = np.meshgrid(p, p)
x
## array([[ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
##        [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
##        [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
##        ...,
##        [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
##        [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
##        [ -5.   , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

### Array basics

#### Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Evaluate the function  $f(x, y) = \sqrt{x^2 + y^2}$  on a 10 x 10 grid.

```
import matplotlib.pyplot as plt
val = np.sqrt(x**2 + y**2)
plt.figure(figsize=(2, 2))
plt.imshow(val, cmap="hot")
plt.colorbar()

## <matplotlib.colorbar.Colorbar object at 0x7f6545c7f5b0>
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

NumPy package

### Array basics

Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

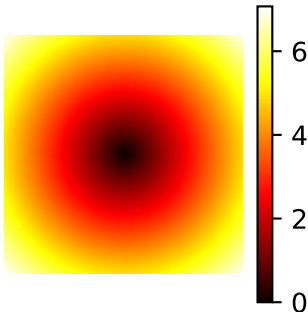
- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Evaluate the function  $f(x, y) = \sqrt{x^2 + y^2}$  on a  $10 \times 10$  grid.

```
plt.show()
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`np.where(condition, a, b)`: If condition is `True`, returns value `a`, otherwise returns `b`.

## Conditional logic

```
a = np.array([4, 7, 5, -7, 9, 0])
b = np.array([-1, 9, 8, 3, 3, 3])
cond = np.array([True, True, False, True, False, False])
res = np.where(cond, a, b)
res

## array([ 4,  7,  8, -7,  3,  3])

res = np.where(a <= b, b, a)
res

## array([4, 9, 8, 3, 9, 3])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Conditional logic, examples

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
res = np.where(arr3 < 5, 0, arr3)  
res
```

```
## array([[0, 8, 5],  
##        [9, 0, 0],  
##        [0, 0, 6]])
```

```
even = np.where(arr3 % 2 == 0, arr3, arr3 + 1)  
even
```

```
## array([[ 4,  8,  6],  
##        [10,  4,  4],  
##        [ 2,  0,  6]])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`array.mean()`: Computes the mean of all array elements.

`array.sum()`: Computes the sum of all array elements.

## Statistical methods

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
arr3.mean()
```

```
## 4.4444444444444445
```

```
arr3.sum()
```

```
## 40
```

```
arr3.argmin()
```

```
## 7
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Method	Description
sum	Sum of all array elements
mean	Mean of all array elements
std, var	Standard deviation, variance
min, max	Minimum and Maximum value in array
argmin, argmax	Indices of Minimum and Maximum value

Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

## Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Axes are defined for arrays with more than one dimension. A two-dimensional array has two axes. The first one is running vertically downwards across the rows (`axis=0`), the second one running horizontally across the columns (`axis=1`).

## Axis

```
arr3

## array([[4, 8, 5],
##        [9, 3, 4],
##        [1, 0, 6]])

arr3.sum(axis=0)

## array([14, 11, 15])

arr3.sum(axis=1)

## array([17, 16, 7])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`array.sort(axis)`: Sorts array by an `axis`.

## Sorting one-dimensional arrays

```
arr2
```

```
## array([24.3 ,  0.   ,  8.9 ,  4.4 ,  1.65, 45.  ])
```

```
arr2.sort()  
arr2
```

```
## array([ 0.   ,  1.65,  4.4 ,  8.9 , 24.3 , 45.  ])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Sorting two-dimensional arrays

```
arr3
```

```
## array([[4, 8, 5],  
##        [9, 3, 4],  
##        [1, 0, 6]])
```

```
arr3.sort()
```

```
arr3
```

```
## array([[4, 5, 8],  
##        [3, 4, 9],  
##        [0, 1, 6]])
```

```
arr3.sort(axis=0)
```

```
arr3
```

```
## array([[0, 1, 6],  
##        [3, 4, 8],  
##        [4, 5, 9]])
```

The default axis using `sort()` is `-1`, which means to sort along the last axis (in this case axis 1).



### Essential concepts

- Getting started
- Procedural programming
- Object-orientation

### Numerical programming

- NumPy package
- Array basics

### Linear algebra

### Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

### Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

### Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Numerical programming

## ▶ Linear algebra



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Import numpy.linalg

```
import numpy.linalg as nplin
```

`nplin.inv(array)`: Computes the inverse matrix.

`np.allclose(array1, array2)`: Returns `True` if two arrays are element-wise equal within a tolerance.

## Inverse

```
inv = nplin.inv(arr3)
inv
```

```
## array([[ 4., -21., 16.],
##         [-5., 24., -18.],
##         [ 1., -4., 3.]])
```

```
np.allclose(np.identity(3), np.dot(inv, arr3))
```

```
## True
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Linear programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`npln.det(array)`: Computes the determinant.

`np.trace(array)`: Computes the trace.

`np.diag(array)`: Returns the diagonal elements as an array.

## Linear algebra functions

```
npln.det(arr3)
```

```
## -1.0
```

```
np.trace(arr3)
```

```
## 13
```

```
np.diag(arr3)
```

```
## array([0, 4, 9])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`npln.eig(array)`: Returns the array of eigenvalues and the array of eigenvectors as a list.

### Get eigenvalues and eigenvectors

```
A = np.array([[3, -1, 0], [2, 0, 0], [-2, 2, -1]])
eigenval, eigenvec = npln.eig(A)
eigenval

## array([-1.,  1.,  2.])

eigenvec

## array([[ 0.          , -0.40824829, -0.70710678],
##        [ 0.          , -0.81649658, -0.70710678],
##        [ 1.          , -0.40824829,  0.          ]])
```





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual

illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Check eigenvalues and eigenvectors

```
eigenval * eigenvec
```

```
## array([[ -0.          , -0.40824829, -1.41421356],  
##        [ -0.          , -0.81649658, -1.41421356],  
##        [ -1.          , -0.40824829,  0.          ]])
```

```
np.dot(A, eigenvec)
```

```
## array([[ 0.          , -0.40824829, -1.41421356],  
##        [ 0.          , -0.81649658, -1.41421356],  
##        [ -1.          , -0.40824829,  0.          ]])
```

$$\begin{pmatrix} 3 & -1 & 0 \\ 2 & 0 & 0 \\ -2 & 2 & -1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = (-1) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics

## Linear algebra

## Data formats and handling

- Pandas package
- Series

## DataFrame

- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`np.linalg.qr(array)`: Conducts a QR decomposition and returns Q and R as lists.

## QR decomposition

```
Q, R = np.linalg.qr(arr3)
```

Q

```
## array([[ 0.          ,  0.98058068,  0.19611614],
##        [-0.6        ,  0.15689291, -0.78446454],
##        [-0.8        , -0.11766968,  0.58834841]])
```

R

```
## array([[ -5.          , -6.4          , -12.          ],
##        [  0.          ,  1.0198039   ,  6.07960019],
##        [  0.          ,  0.          ,  0.19611614]])
```

```
np.allclose(arr3, np.dot(Q, R))
```

```
## True
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`nplin.solve(A, b)`: Returns the solution of the linearsystem  $Ax = b$ .

## Solve linearsystems

```
b = np.array([7, 4, 8])
x = nplin.solve(A, b)
x

## array([ 2., -1., -14.])

np.allclose(np.dot(A, x), b)

## True
```

$$\begin{array}{rcl} 3x_1 - 1x_2 + 0x_3 & = & 7 \\ 2x_1 - 0x_2 + 0x_3 & = & 4 \\ -2x_1 + 2x_2 - 1x_3 & = & 8 \end{array} \rightarrow \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ -14 \end{pmatrix}$$



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

## Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Function	Description
<code>np.dot</code>	Matrix multiplication
<code>np.trace</code>	Sum of the diagonal elements
<code>np.diag</code>	Diagonal elements as an array
<code>nplin.det</code>	Matrix determinant
<code>nplin.eig</code>	Eigenvalues and eigenvectors
<code>nplin.inv</code>	Inverse matrix
<code>nplin.qr</code>	QR decomposition
<code>nplin.solve</code>	Solve linearsystem



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Data formats and handling

## 3.1 Pandas package

## 3.2 Series

## 3.3 DataFrame

## 3.4 Import/Export data



### Essential concepts

- Getting started
- Procedural programming
- Object-orientation

### Numerical programming

- NumPy package
- Array basics
- Linear algebra

### Data formats and handling

- Pandas package**

- Series
- DataFrame
- Import/Export data

### Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

### Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Data formats and handling

## ▶ Pandas package



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization



The package `pandas` is a free software library for Python including the following features:

- Data manipulation and analysis,
- DataFrame objects and Series,
- Export and import data from files and web,
- Handling of missing data.

→ Provides high-performance data structures and data analysis tools.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package**
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

With `pandas` you can import and visualize financial data in only a few lines of code.

### Motivation

```
import pandas as pd
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
dow = pd.read_csv("data/dji.csv", index_col=0, parse_dates=True)
close = dow["Close"]
close.plot(ax=ax)
ax.set_xlabel("Date")
ax.set_ylabel("Price")
ax.set_title("DJI")
fig.savefig("out/dji.pdf", format="pdf")
```





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

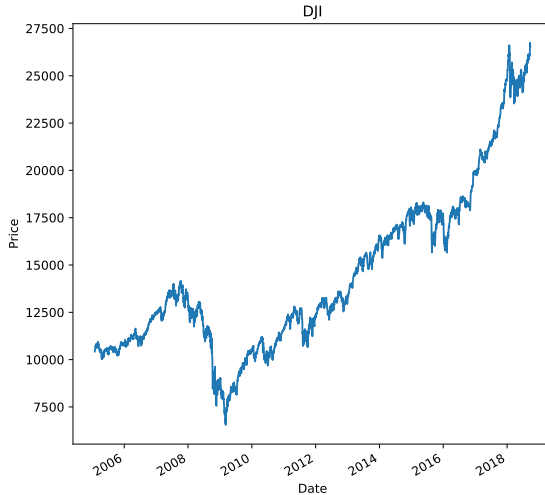
- Pandas package**
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series**
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Data formats and handling

## ▶ Series



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

**Series**

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

*Series* are a data structure in pandas.

- One-dimensional array-like object,
- Containing a sequence of values and a corresponding array of labels, called the index,
- The string representation of a Series displays the index on the left and the values on the right,
- The default index consists of the integers 0 through N-1.

### String representation of a Series

```
## 0      3
## 1      7
## 2     -8
## 3      4
## 4     26
## dtype: int64
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package

### Series

- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`pd.Series()`: Creates one-dimensional array-like object including values and an index.

## Importing Pandas and creating a Series

```
import numpy as np
import pandas as pd
```

```
obj = pd.Series([2, -5, 9, 4])
obj
```

```
## 0      2
## 1     -5
## 2      9
## 3      4
## dtype: int64
```

- Simple Series formed only from a list,
- An index is added automatically.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

**Series**

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Series indexing vs. Numpy indexing

```
obj2 = pd.Series([2, -5, 9, 4], index=["a", "b", "c", "d"])
```

```
npobj = np.array([2, -5, 9, 4])
```

```
obj2
```

```
## a    2
```

```
## b   -5
```

```
## c    9
```

```
## d    4
```

```
## dtype: int64
```

```
obj2["b"]
```

```
## -5
```

```
npobj[1]
```

```
## -5
```

- NumPy arrays can only be indexed by integers while Series can be indexed by the manually set index.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package

### Series

- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Pandas Series can be created from:

- Lists,
- NumPy arrays,
- Dicts.

### Series creation from Numpy arrays

```
npobj = np.array([2, -5, 9, 4])
obj2 = pd.Series(npobj, index=["a", "b", "c", "d"])
obj2
```

```
## a      2
## b     -5
## c      9
## d      4
## dtype: int64
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series**
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### Series from dicts

```
dictdata = {"Göttingen": 117665, "Northeim": 28920,  
            "Hannover": 532163, "Berlin": 3574830}  
obj3 = pd.Series(dictdata)  
obj3
```

```
## Göttingen      117665  
## Northeim       28920  
## Hannover       532163  
## Berlin         3574830  
## dtype: int64
```

- The index of the Series can be set manually,
- Compared to NumPy array you can use the set index to select single values,
- Data contained in a dict can be passed to a Series. The index of the resulting Series consists of the dict's keys.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Dict to Series with manual index

```
cities = ["Hamburg", "Göttingen", "Berlin", "Hannover"]  
obj4 = pd.Series(dictdata, index=cities)  
obj4
```

```
## Hamburg          NaN  
## Göttingen       117665.0  
## Berlin          3574830.0  
## Hannover        532163.0  
## dtype: float64
```

- Passing a dict to a Series, the index can be set manually,
- NaN (not a number) marks missing values where the index and the dict do not match.





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`Series.values`: Returns the values of a Series.

`Series.index`: Returns the index of a Series.

## Series properties

```
obj.values
```

```
## array([ 2, -5,  9,  4])
```

```
obj.index
```

```
## RangeIndex(start=0, stop=4, step=1)
```

```
obj2.index
```

```
## Index(['a', 'b', 'c', 'd'], dtype='object')
```

- The values and the index of a Series can be printed separately.
- The default index, if none was explicitly specified, is a `RangeIndex`.
- `RangeIndex` inherits from `Index` class.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Series manipulation

```
obj2[["c", "d", "a"]]
```

```
## c      9
```

```
## d      4
```

```
## a      2
```

```
## dtype: int64
```

```
obj2[obj2 < 0]
```

```
## b     -5
```

```
## dtype: int64
```

NumPy-like functions can be applied on Series

- For filtering data,
- To do scalar multiplications or applying math functions,
- The index-value link will be preserved.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Series functions

```
obj2 * 2

## a      4
## b     -10
## c      18
## d       8
## dtype: int64

np.exp(obj2)["a":"c"]

## a      7.389056
## b      0.006738
## c     8103.083928
## dtype: float64

"c" in obj2

## True
```

- Mathematical functions applied to a Series will only be applied on its *values* – **not** on its index.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Series manipulation

```
obj4["Hamburg"] = 1900000
```

```
obj4
```

```
## Hamburg      1900000.0
```

```
## Göttingen    117665.0
```

```
## Berlin       3574830.0
```

```
## Hannover     532163.0
```

```
## dtype: float64
```

```
obj4[["Berlin", "Hannover"]] = [3600000, 1100000]
```

```
obj4
```

```
## Hamburg      1900000.0
```

```
## Göttingen    117665.0
```

```
## Berlin       3600000.0
```

```
## Hannover     1100000.0
```

```
## dtype: float64
```

- Values can be manipulated by using the labels in the index,
- Sets of values can be set in one line.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`pd.isnull()`: **True** if data is missing.

`pd.notnull()`: **False** if data is missing.

## NaN

```
pd.isnull(obj4)
```

```
## Hamburg      False
## Göttingen     False
## Berlin        False
## Hannover      False
## dtype: bool
```

```
pd.notnull(obj4)
```

```
## Hamburg      True
## Göttingen     True
## Berlin        True
## Hannover      True
## dtype: bool
```



## Essential concepts

[Getting started](#)[Procedural programming](#)[Object-orientation](#)

## Numerical programming

[NumPy package](#)[Array basics](#)[Linear algebra](#)

## Data formats and handling

[Pandas package](#)[Series](#)[DataFrame](#)[Import/Export data](#)

## Visual illustrations

[Matplotlib package](#)[Figures and subplots](#)[Plot types and styles](#)[Pandas layers](#)

## Applications

[Time series](#)[Moving window](#)[Financial applications](#)[Optimization](#)

There are not two values to align for `Hamburg` and `Northeim` – so they are marked with `NaN` (not a number).

### Data 1

```
obj3
```

```
## Göttingen      117665
## Northeim        28920
## Hannover        532163
## Berlin          3574830
## dtype: int64
```

### Data 2

```
obj4
```

```
## Hamburg        1900000.0
## Göttingen       117665.0
## Berlin          3600000.0
## Hannover        1100000.0
## dtype: float64
```

## Align data

```
obj3 + obj4
```

```
## Berlin          7174830.0
## Göttingen       235330.0
## Hamburg          NaN
## Hannover        1632163.0
## Northeim         NaN
## dtype: float64
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`Series.name`: Returns name of the Series.

`Series.index.name`: Returns name of the Series' index.

## Naming

```
obj4.name = "population"
obj4.index.name = "city"
obj4

## city
## Hamburg      1900000.0
## Göttingen    117665.0
## Berlin        3600000.0
## Hannover      1100000.0
## Name: population, dtype: float64
```

- The attribute `name` will change the name of the existing Series,
- There is no default name of the Series or the index.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

- NumPy arrays are accessed by their integer positions,
- Series can be accessed by a user defined index, including letters and numbers,
- Different Series can be aligned efficiently by the index,
- Series can work with missing values, so operations do not automatically fail.





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series

### DataFrame

- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Data formats and handling

## ▶ DataFrame



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

- *DataFrames* are the primary structure of pandas,
- It represents a table of data with an ordered collection of columns,
- Each column can have a different data type,
- A DataFrame can be thought of as a dict of Series sharing the same index,
- Physically a DataFrame is two-dimensional but by using hierarchical indexing it can represent higher dimensional data.

## String representation of a DataFrame

##	company	price	volume
## 0	Daimler	69.20	4456290
## 1	E.ON	8.11	3667975
## 2	Siemens	110.92	3669487
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`pd.DataFrame()`: Creates a DataFrame which is a two-dimensional tabular-like structure with labeled axis (rows and columns).

### Creating a DataFrame

```
data = {"company": ["Daimler", "E.ON", "Siemens", "BASF", "BMW"],
        "price": [69.2, 8.11, 110.92, 87.28, 87.81],
        "volume": [4456290, 3667975, 3669487, 1778058, 1824582]}
frame = pd.DataFrame(data)
frame
```

##	company	price	volume
## 0	Daimler	69.20	4456290
## 1	E.ON	8.11	3667975
## 2	Siemens	110.92	3669487
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582

- In this example the construction of the DataFrame `frame` is done by passing a dict of equal-length lists,
- Instead of passing a dict of lists, it is also possible to pass a dict of NumPy arrays.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Print DataFrame

```
frame2 = pd.DataFrame(data, columns=["company", "volume",  
                                     "price", "change"])
```

```
frame2
```

##	company	volume	price	change
## 0	Daimler	4456290	69.20	NaN
## 1	E.ON	3667975	8.11	NaN
## 2	Siemens	3669487	110.92	NaN
## 3	BASF	1778058	87.28	NaN
## 4	BMW	1824582	87.81	NaN

- Passing a column that is not contained in the dict, it will be marked with NaN,
- The default index will be assigned automatically as with Series.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Type	Description
2D NumPy arrays	A matrix of data
dict of arrays, lists, or tuples	Each sequence becomes a column
dict of Series	Each value becomes a column
dict of dicts	Each inner dict becomes a column
List of dicts or Series	Each item becomes a row
List of lists or tuples	Treated as the 2D NumPy arrays
Another DataFrame	Same indexes



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Add data to DataFrame

```
frame2["change"] = [1.2, -3.2, 0.4, -0.12, 2.4]
frame2["change"]

## 0      1.20
## 1     -3.20
## 2      0.40
## 3     -0.12
## 4      2.40
## Name: change, dtype: float64
```

- Selecting the column of DataFrame, a Series is returned,
- A attribute-like access, e.g., `frame2.change`, is also possible,
- The returned Series has the same index as the initial DataFrame.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Indexing DataFrames

```
frame2[["company", "change"]]
```

```
##      company  change
## 0  Daimler    1.20
## 1    E.ON   -3.20
## 2  Siemens    0.40
## 3   BASF   -0.12
## 4    BMW    2.40
```

- Using a list of multiple columns while indexing, the result is a DataFrame,
- The returned DataFrame has the same index as the initial one.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual

illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`del DataFrame[column]`: Deletes column from DataFrame.

### DataFrame delete column

```
del frame2["volume"]
frame2
```

##	company	price	change
## 0	Daimler	69.20	1.20
## 1	E.ON	8.11	-3.20
## 2	Siemens	110.92	0.40
## 3	BASF	87.28	-0.12
## 4	BMW	87.81	2.40

```
frame2.columns
```

```
## Index(['company', 'price', 'change'], dtype='object')
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Naming properties

```
frame2.index.name = "number:"  
frame2.columns.name = "feature:"  
frame2
```

```
## feature:  company    price  change  
## number:  
## 0          Daimler    69.20    1.20  
## 1           E.ON     8.11   -3.20  
## 2        Siemens  110.92    0.40  
## 3          BASF    87.28   -0.12  
## 4           BMW    87.81    2.40
```

- In DataFrames there is no default name for the index or the columns.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.reindex()`: Creates new DataFrame with data conformed to a new index, while the initial DataFrame will not be changed.

## Reindexing

```
frame3 = frame.reindex([0, 2, 3, 4])  
frame3
```

##	company	price	volume
## 0	Daimler	69.20	4456290
## 2	Siemens	110.92	3669487
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582

- Index values that are not already present will be filled with `NaN` by default,
- There are many options for filling missing values.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Filling missing values

```
frame4 = frame.reindex(index=[0, 2, 3, 4, 5], fill_value=0,  
                        columns=["company", "price", "market cap"])
```

frame4

##	company	price	market cap
## 0	Daimler	69.20	0
## 2	Siemens	110.92	0
## 3	BASF	87.28	0
## 4	BMW	87.81	0
## 5	0	0.00	0

```
frame4 = frame.reindex(index=[0, 2, 3, 4], fill_value=np.nan,  
                        columns=["company", "price", "market cap"])
```

frame4

##	company	price	market cap
## 0	Daimler	69.20	NaN
## 2	Siemens	110.92	NaN
## 3	BASF	87.28	NaN
## 4	BMW	87.81	NaN



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.fillna(value)`: Fills NaNs with value.

## Filling NaN

```
frame4[:3]
```

##	company	price	market cap
## 0	Daimler	69.20	NaN
## 2	Siemens	110.92	NaN
## 3	BASF	87.28	NaN

```
frame4.fillna(1000000, inplace=True)  
frame4[:3]
```

##	company	price	market cap
## 0	Daimler	69.20	1000000.0
## 2	Siemens	110.92	1000000.0
## 3	BASF	87.28	1000000.0

- The option `inplace=True` fills the current DataFrame (here `frame4`). Without using `inplace` a new DataFrame will be created, filled with NaN values.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.drop(index, axis)`: Returns a new object with labels in requested axis removed.

### Dropping index

```
frame5 = frame
frame5
```

##	company	price	volume
## 0	Daimler	69.20	4456290
## 1	E.ON	8.11	3667975
## 2	Siemens	110.92	3669487
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582

```
frame5.drop([1, 2])
```

##	company	price	volume
## 0	Daimler	69.20	4456290
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame**
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Dropping column

```
frame5[:2]
```

##	company	price	volume
## 0	Daimler	69.20	4456290
## 1	E.ON	8.11	3667975

```
frame5.drop("price", axis=1)[:3]
```

##	company	volume
## 0	Daimler	4456290
## 1	E.ON	3667975
## 2	Siemens	3669487

```
frame5.drop(2, axis=0)
```

##	company	price	volume
## 0	Daimler	69.20	4456290
## 1	E.ON	8.11	3667975
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582



## Essential concepts

[Getting started](#)[Procedural programming](#)[Object-orientation](#)

## Numerical programming

[NumPy package](#)[Array basics](#)[Linear algebra](#)

## Data formats and handling

[Pandas package](#)[Series](#)[DataFrame](#)[Import/Export data](#)

## Visual illustrations

[Matplotlib package](#)[Figures and subplots](#)[Plot types and styles](#)[Pandas layers](#)

## Applications

[Time series](#)[Moving window](#)[Financial applications](#)[Optimization](#)

Indexing of DataFrames works like indexing an `numpy` array, you can use the default index values and a manually set index.

## Indexing

```
frame
```

##	company	price	volume
## 0	Daimler	69.20	4456290
## 1	E.ON	8.11	3667975
## 2	Siemens	110.92	3669487
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582

```
frame[2:]
```

##	company	price	volume
## 2	Siemens	110.92	3669487
## 3	BASF	87.28	1778058
## 4	BMW	87.81	1824582



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Indexing

```
frame6 = pd.DataFrame(data, index=["a", "b", "c", "d", "e"])  
frame6
```

	company	price	volume
a	Daimler	69.20	4456290
b	E.ON	8.11	3667975
c	Siemens	110.92	3669487
d	BASF	87.28	1778058
e	BMW	87.81	1824582

```
frame6["b":"d"]
```

	company	price	volume
b	E.ON	8.11	3667975
c	Siemens	110.92	3669487
d	BASF	87.28	1778058

- When *slicing with labels* the **end element is inclusive**.





## Essential concepts

Getting started

Procedural programming

Object-orientation

## NumPy package

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.loc()`: Selects a subset of rows and columns from a DataFrame using axis labels.

`DataFrame.iloc()`: Selects a subset of rows and columns from a DataFrame using integers.

### Selection with loc and iloc

```
frame6.loc["c", ["company", "price"]]
```

```
## company      Siemens
## price        110.92
## Name: c, dtype: object
```

```
frame6.iloc[2, [0, 1]]
```

```
## company      Siemens
## price        110.92
## Name: c, dtype: object
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Selection with loc and iloc

```
frame6.loc[["c", "d", "e"], ["volume", "price", "company"]]
```

##		volume	price	company
##	c	3669487	110.92	Siemens
##	d	1778058	87.28	BASF
##	e	1824582	87.81	BMW

```
frame6.iloc[2:, :-1]
```

##		volume	price	company
##	c	3669487	110.92	Siemens
##	d	1778058	87.28	BASF
##	e	1824582	87.81	BMW

- Both of the indexing functions work with slices or lists of labels,
- Many ways to select and rearrange pandas objects.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual

### illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Type	Description
<code>df[val]</code>	Select single column or set of columns
<code>df.loc[val]</code>	Select single row or set of rows
<code>df.loc[:, val]</code>	Select single column or set of columns
<code>df.loc[val1, val2]</code>	Select row and column by label
<code>df.iloc[where]</code>	Select row or set of rows by integer position
<code>df.iloc[:, where]</code>	Select column or set of columns by integer pos.
<code>df.iloc[w1, w2]</code>	Select row and column by integer position



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numeral programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Hierarchical indexing enables you to have multiple index levels.

## Multiindex

```
ind = [["a", "a", "a", "b", "b"], [1, 2, 3, 1, 2]]
frame6 = pd.DataFrame(np.arange(15).reshape((5, 3)), index=ind,
                      columns=["first", "second", "third"])
```

frame6

##		first	second	third
##	a 1	0	1	2
##	2	3	4	5
##	3	6	7	8
##	b 1	9	10	11
##	2	12	13	14

```
frame6.index.names = ["index1", "index2"]
frame6.index
```

```
## MultiIndex([('a', 1),
##             ('a', 2),
##             ('a', 3),
##             ('b', 1),
##             ('b', 2)],
##            names=['index1', 'index2'])
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame**

Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Selecting of a multiindex

```
frame6.loc["a"]
```

```
##           first  second  third
## index2
## 1             0        1      2
## 2             3        4      5
## 3             6        7      8
```

```
frame6.loc["b", 1]
```

```
## first      9
## second    10
## third     11
## Name: (b, 1), dtype: int64
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Series and DataFrames

```
frame7 = frame[["price", "volume"]]  
frame7.index = ["Daimler", "E.ON", "Siemens", "BASF", "BMW"]  
series = frame7.iloc[2]  
frame7
```

```
##           price  volume  
## Daimler    69.20  4456290  
## E.ON       8.11  3667975  
## Siemens   110.92  3669487  
## BASF      87.28  1778058  
## BMW       87.81  1824582
```

```
series  
  
## price           110.92  
## volume        3669487.00  
## Name: Siemens, dtype: float64
```

- Here the Series was generated from the first row of the DataFrame.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Operations between Series and DataFrames down the rows

```
frame7 + series
```

##	price	volume
## Daimler	180.12	8125777.0
## E.ON	119.03	7337462.0
## Siemens	221.84	7338974.0
## BASF	198.20	5447545.0
## BMW	198.73	5494069.0

- By default arithmetic operations between DataFrames and Series match the index of the Series on the DataFrame's columns,
- The operations will be broadcasted along the rows.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Operations between Series and DataFrames down the columns

```
series2 = frame7["price"]  
frame7.add(series2, axis=0)  
  
##           price      volume  
## Daimler  138.40  4456359.20  
## E.ON     16.22   3667983.11  
## Siemens 221.84  3669597.92  
## BASF     174.56  1778145.28  
## BMW      175.62  1824669.81
```

- Here, the Series was generated from the `price` column,
- The arithmetic operation will be broadcasted along a column matching the DataFrame's row index (`axis=0`).





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

**DataFrame**

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Pandas vs Numpy

```
nparr = np.arange(12.).reshape((3, 4))
row = nparr[0]
nparr - row

## array([[0., 0., 0., 0.],
##        [4., 4., 4., 4.],
##        [8., 8., 8., 8.]])
```

- Operations between DataFrames are similar to operations between one- and two-dimensional Numpy arrays,
- As in DataFrames and Series the arithmetic operations will be broadcasted along the rows.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame**

- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`DataFrame.apply(np.function, axis)`: Applies a NumPy function on the DataFrame axis. See also [statistical](#) and [mathematical NumPy functions](#).

## Numpy functions on DataFrames

```
frame7[:2]
```

```
##           price    volume
## Daimler    69.20  4456290
## E.ON       8.11   3667975
```

```
frame7.apply(np.mean)
```

```
## price           72.664
## volume        3079278.400
## dtype: float64
```

```
frame7.apply(np.sqrt)[:2]
```

```
##           price           volume
## Daimler    8.318654  2110.992657
## E.ON       2.847806  1915.195812
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.groupby(col1, col2)`: Groups DataFrame by columns (grouping by one or more than two columns is also possible). See also [how to import data from CSV files](#).

## Groupby

```
vote = pd.read_csv("data/vote.csv")[["Party", "Member", "Vote"]]  
vote.head()
```

##		Party	Member	Vote
## 0		CDU/CSU	Abercron	yes
## 1		CDU/CSU	Albani	yes
## 2		CDU/CSU	Altenkamp	yes
## 3		CDU/CSU	Altmaier	absent
## 4		CDU/CSU	Amthor	yes

Adding the functions `count()` or `mean()` to `groupby()` returns the sum or the mean of the grouped columns.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual

illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Groupby

```
res = vote.groupby(["Party", "Vote"]).count()
```

```
res
```

	Member
##	
## Party	Vote
## AfD	absent 6
##	no 86
## BÜ90/GR	absent 9
##	no 58
## CDU/CSU	absent 7
##	yes 239
## DIE LINKE.	absent 7
##	no 62
## FDP	absent 5
##	no 75
## Fraktionslos	absent 1
##	no 1
## SPD	absent 6
##	yes 147



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame

### Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Data formats and handling

## ▶ Import/Export data



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

ex1.csv

```
a, b, c, d, hello
1, 2, 3, 4, world
5, 6, 7, 8, python
2, 3, 5, 7, pandas
```

`pd.read_csv("file")`: Reads CSV into DataFrame.

## Read comma-separated values

```
df = pd.read_csv("data/ex1.csv")
df
```

```
##      a      b      c      d      hello
## 0     1     2     3     4     world
## 1     5     6     7     8     python
## 2     2     3     5     7     pandas
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

tab.txt

```
a| b| c| d| hello
1| 2| 3| 4| world
5| 6| 7| 8| python
2| 3| 5| 7| pandas
```

`pd.read_table("file", sep=|)`: Reads table with any separators into DataFrame.

## Read table values

```
df = pd.read_table("data/tab.txt", sep="|")
```

df

```
##      a      b      c      d      hello
## 0      1      2      3      4      world
## 1      5      6      7      8      python
## 2      2      3      5      7      pandas
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

ex2.csv

```
1, 2, 3, 4, world
5, 6, 7, 8, python
2, 3, 5, 7, pandas
```

CSV file without header row:

## Read CSV and header settings

```
df = pd.read_csv("data/ex2.csv", header=None)
df
```

```
##      0  1  2  3      4
## 0  1  2  3  4    world
## 1  5  6  7  8    python
## 2  2  3  5  7    pandas
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

ex2.csv

```
1, 2, 3, 4, world
5, 6, 7, 8, python
2, 3, 5, 7, pandas
```

Specify header:

## Read CSV and header names

```
df = pd.read_csv("data/ex2.csv",
                  names=["a", "b", "c", "d", "hello"])
```

df

##	a	b	c	d	hello
## 0	1	2	3	4	world
## 1	5	6	7	8	python
## 2	2	3	5	7	pandas



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

ex2.csv

```
1, 2, 3, 4, world
5, 6, 7, 8, python
2, 3, 5, 7, pandas
```

Use [hello](#)-column as the index:

## Read CSV and specify index

```
df = pd.read_csv("data/ex2.csv",
                  names=["a", "b", "c", "d", "hello"],
                  index_col="hello")
```

df

```
##          a  b  c  d
## hello
## world    1  2  3  4
## python   5  6  7  8
## pandas   2  3  5  7
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

ex3.csv

```
1, 2, 3, 4, world
##-.,.-'*'-. ,
5, 6, 7, 8, python
87646756754456978
2, 3, 5, 7, pandas
```

Skip rows while reading:

## Read CSV and choose rows

```
df = pd.read_csv("data/ex3.csv", skiprows=[1, 3])
df
```

```
##      1      2      3      4      world
## 0      5      6      7      8      python
## 1      2      3      5      7      pandas
```



## Essential concepts

[Getting started](#)[Procedural programming](#)[Object-orientation](#)

## NumPy package

[Array basics](#)[Linear algebra](#)

## Data formats and handling

[Pandas package](#)[Series](#)[DataFrame](#)[Import/Export data](#)

## Visual illustrations

[Matplotlib package](#)[Figures and subplots](#)[Plot types and styles](#)[Pandas layers](#)

## Applications

[Time series](#)[Moving window](#)[Financial applications](#)[Optimization](#)

`DataFrame.to_csv("filename")`: Writes DataFrame to CSV.

## Write to CSV

```
df = pd.read_csv("data/ex3.csv", skiprows=[1, 3])  
df.to_csv("out/out1.csv")
```

out1.csv

```
,1, 2, 3, 4, world  
0,5,6,7,8, python  
1,2,3,5,7, pandas
```

In the .csv file, the index and header is included (reason why ,1).



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame

## Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Write to CSV and settings

```
df = pd.read_csv("data/ex3.csv", skiprows=[1, 3])  
df.to_csv("out/out2.csv", index=False, header=False)
```

out2.csv

```
5,6,7,8, python  
2,3,5,7, pandas
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### Write to CSV and specify header

```
df = pd.read_csv("data/ex3.csv", skiprows=[1, 3, 4])
df.to_csv("out/out3.csv", index=False,
          header=["a", "b", "c", "d", "e"])
```

out3.csv

```
a,b,c,d,e
5,6,7,8, python
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual

### illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`pd.read_excel("file.xls")`: Reads .xls files.

	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Adj Close	Volume
2	2018-01-31	1170.569946	1173	1159.130005	1169.939941	1169.939941	1538700
3	2018-02-01	1162.609985	1174	1157.52002	1167.699951	1167.699951	2412100
4	2018-02-02	1122	1123.069946	1107.277954	1111.900024	1111.900024	4857900
5	2018-02-05	1090.599976	1110	1052.030029	1055.800049	1055.800049	3798300
6	2018-02-06	1027.180054	1081.709961	1023.137024	1080.599976	1080.599976	3448000
7	2018-02-07	1081.540039	1081.780029	1048.26001	1048.579956	1048.579956	2341700

Figure: goog.xls

## Reading Excel

```
xls_frame = pd.read_excel("data/goog.xls")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame

## Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Excel as a DataFrame

```
xls_frame[["Adj Close", "Volume", "High"]]
```

	##	Adj Close	Volume	High
##	0	1169.939941	1538700	1173.000000
##	1	1167.699951	2412100	1174.000000
##	2	1111.900024	4857900	1123.069946
##	3	1055.800049	3798300	1110.000000
##	4	1080.599976	3448000	1081.709961
##	5	1048.579956	2341700	1081.780029





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Extract financial data from Internet sources into a DataFrame. There are different sources offering different kind of data. Some sources are:

- Robinhood
- IEX
- Yahoo Finance
- World Bank
- OECD
- Eurostat

A complete list of the sources and the usage can be found here:

▶ [pandas-datareader](#)

## Import pandas-datareader

```
from pandas_datareader import data
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`data.DataReader("symbol", "source", "start", "end")`: Returns financial data of a stock in a certain time period.

### Get data of Ford

```
ford = data.DataReader("F", "yahoo", "2020-01-01", "2020-01-31")
ford.head()[["Close", "Volume"]]
```

##	Close	Volume
## Date		
## 2020-01-02	9.42	43425700.0
## 2020-01-03	9.21	45040800.0
## 2020-01-06	9.16	43372300.0
## 2020-01-07	9.25	44984100.0
## 2020-01-08	9.25	45994900.0

► [Stock code list](#)



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Explore Ford dataset

```
ford.index
```

```
## DatetimeIndex(['2020-01-02', '2020-01-03', ...  
## ...dtype='datetime64[ns]', name='Date', ...
```

```
ford.loc["2020-01-28"]
```

```
## High          9.000000e+00  
## Low           8.860000e+00  
## Open          8.940000e+00  
## Close         8.970000e+00  
## Volume        8.516340e+07  
## Adj Close     8.820001e+00  
## Name: 2020-01-28 00:00:00, dtype: float64
```

## DataFrame index

Index of the DataFrame is different at different sources. Always check DataFrame.index!



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Download and explore SAP data

```
sap = data.DataReader("SAP", "yahoo", "2020-01-01", "2020-06-30")
sap[25:27]
```

```
##                               High           Low  ...      Volume      Adj Close
## Date                               ...
## 2020-02-07  136.020004  134.639999  ...    511700.0  133.137955
## 2020-02-10  135.369995  134.679993  ...    381200.0  133.305527
##
## [2 rows x 6 columns]
```

```
sap.loc["2020-03-09"]
```

```
## High           1.161900e+02
## Low            1.105500e+02
## Open           1.136100e+02
## Close          1.115000e+02
## Volume         1.571800e+06
## Adj Close      1.099132e+02
## Name: 2020-03-09 00:00:00, dtype: float64
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Eurostat

```
population = data.DataReader("tps00001", "eurostat", "2010-01-01",  
                             "2020-01-01")
```

```
population.columns
```

```
## MultiIndex(levels=[[Population on 1 January - total], [Albania,  
## Andorra, Armenia, Austria, Azerbaijan, Belarus, Belgium, ...
```

```
population["Population on 1 January - total", "France"][-5:]
```

```
## FREQ          Annual  
## TIME_PERIOD  
## 2016-01-01    66638391.0  
## 2017-01-01    66809816.0  
## 2018-01-01    66918941.0  
## 2019-01-01    67012883.0  
## 2020-01-01    67098824.0
```

► Eurostat Database



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Website used for the example: [Econometrics](#)

## Beautiful Soup

```
from bs4 import BeautifulSoup
import requests
url = "www.uni-goettingen.de/de/applied-econometrics/412565.html"
r = requests.get("https://" + url)
d = r.text
soup = BeautifulSoup(d, "lxml")

soup.title

## <title>Applied Econometrics - Georg-August-... ...</title>
```

Reading data from HTML in detail exceeds the content of this course. If you are interested in this kind of importing data, you can find detailed information on *Beautiful Soup* [here](#).



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame

## Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Bollinger

```
sap = data.DataReader("SAP", "yahoo", "2019-01-01", "2020-08-31")
sap.index = pd.to_datetime(sap.index)
boll = sap["Close"].rolling(window=20, center=False).mean()
std = sap["Close"].rolling(window=20, center=False).std()
upp = boll + std * 2
low = boll - std * 2
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
boll.plot(ax=ax, label="20 days Rolling mean")
upp.plot(ax=ax, label="Upper Band")
low.plot(ax=ax, label="Lower Band")
sap["Close"].plot(ax=ax, label="SAP Price")
ax.legend(loc="best")
fig.savefig("out/boll.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

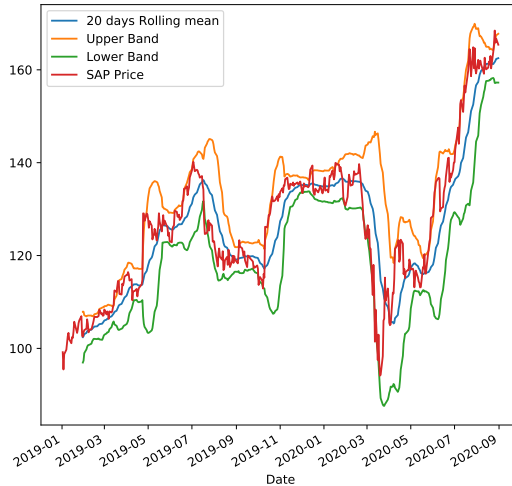
## Applications

Time series

Moving window

Financial applications

Optimization







## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Visual illustrations

- 4.1 Matplotlib package
- 4.2 Figures and subplots
- 4.3 Plot types and styles
- 4.4 Pandas layers



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package**
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Visual illustrations

## ▶ Matplotlib package



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization



The package `matplotlib` is a free software library for python including the following functions:

- Image plots, Contour plots, Scatter plots, Polar plots, Line plots, 3D plots,
- Variety of hardcopy formats,
- Works in Python scripts, the Python and IPython shell and the Jupyter notebook,
- Interactive environments.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numeral  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Usage of matplotlib

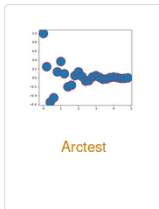
*matplotlib* has a vast number of functions and options, which is hard to remember. But for almost every task there is an example you can take code from. A great source of information is the [examples gallery](#) on the `matplotlib` homepage. Also note the [best practice quick start guide](#).

## Gallery

This gallery contains examples of the many things you can do with Matplotlib. Click on any image to see the full image and source code.

For longer tutorials, see our [tutorials page](#). You can also find [external resources](#) and a [FAQ](#) in our [user guide](#).

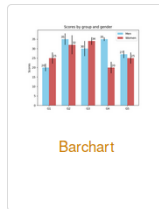
## Lines, bars and markers



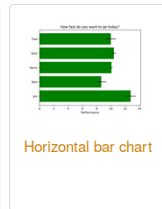
Arctest



Stacked Bar Graph



Barchart



Horizontal bar chart



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package

- Figures and subplots
- Plot types and styles
- Pandas layers

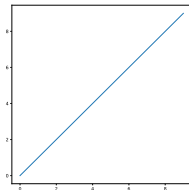
## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`plt.plot(array)`: Plots the values of a list, the X-axis has by default the range `[0, 1, ..., n-1]`.

### Import matplotlib and simple example

```
import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(10))
plt.savefig("out/list.pdf")
```





### Essential concepts

- Getting started
- Procedural programming
- Object-orientation

### Numerical programming

- NumPy package
- Array basics
- Linear algebra

### Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

### Visual illustrations

- Matplotlib package

#### Figures and subplots

- Plot types and styles
- Pandas layers

### Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Visual illustrations

## ► Figures and subplots



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Plots in `matplotlib` reside in a *Figure object*:

`plt.figure(...)`: Creates new Figure object allowing for multiple parameters.

`plt.gcf()`: Returns the reference of the active figure.

## Create Figures

```
fig = plt.figure(figsize=(16, 8))
print(plt.gcf())

## Figure(1600x800)
```

- A Figure object can be considered as an empty window,
- The Figure object has a number of options, such as the size or the aspect ratio,
- You cannot draw a plot in a blank figure. There has to be a `subplot` in the Figure object.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`plt.savefig("filename")`: Saves active figure to file.  
Available file formats are among others:

Filename extension	Description
.png	Portable Network Graphics
.pdf	Portable Document Format
.svg	Scalable Vector Graphics
.jpeg	JPEG File Interchange Format
.jpg	JPEG File Interchange Format
.ps	PostScript
.raw	Raw Image Format





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`fig.add_subplot()`: Adds subplot to the Figure `fig`.

Example: `fig.add_subplot(2, 2, 1)` creates four subplots and selects the first.

### Adding subplots

```
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
ax4 = fig.add_subplot(2, 2, 4)
fig.savefig("out/subplots.pdf")
```

- The Figure object is filled with subplots in which the plots reside,
- Using the `plt.plot()` command **without creating a subplot in advance**, matplotlib will create a Figure object and a subplot automatically,
- The Figure object and its subplots can be **created in one line**.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

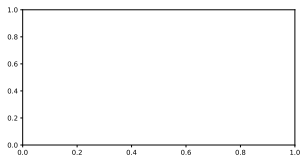
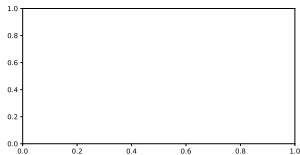
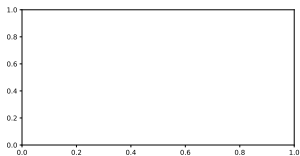
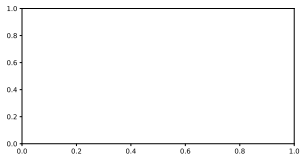
- Matplotlib package

## Figures and subplots

- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Filling subplots with content

```
from numpy.random import randn
ax1.plot([5, 7, 4, 3, 1])
ax2.hist(randn(100), bins=20, color="r")
ax3.scatter(np.arange(30), np.arange(30) * randn(30))
ax4.plot(randn(40), "k--")
fig.savefig("out/content.pdf")
```

- The subplots in one Figure object can be filled with different [plot types](#),
- Using only `plt.plot()` matplotlib draws the plot in the last Figure object and last subplot selected.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

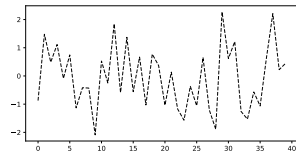
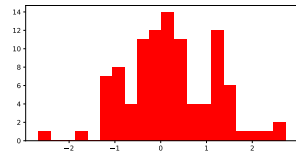
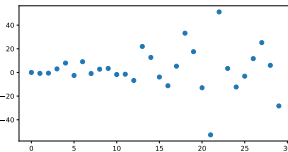
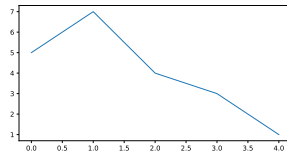
## Visual illustrations

- Matplotlib package
- Figures and subplots**

- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

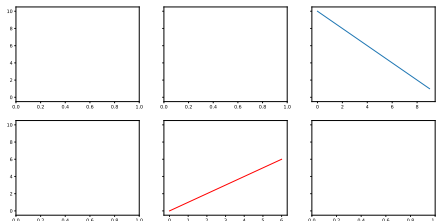
Financial applications

Optimization

`plt.subplots(nrows, ncols, sharex, sharey)`: Creates figure and subplots in one line. If `sharex` or `sharey` are `True`, all subplots share the same X- or Y-ticks.

### Standard creation

```
fig, axes = plt.subplots(2, 3, figsize=(16, 8), sharey=True)
axes[1, 1].plot(np.arange(7), color="r")
axes[0, 2].plot(np.arange(10, 0, -1))
fig.savefig("out/standard.pdf")
```





### Essential concepts

- Getting started
- Procedural programming
- Object-orientation

### Numerical programming

- NumPy package
- Array basics
- Linear algebra

### Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

### Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

### Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Visual illustrations

## ▶ Plot types and styles



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots

## Plot types and styles

- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`ax.scatter(x, y)`: Creates a scatter plot of `x` vs `y`.

`ax.hist(x, bins)`: Creates a histogram.

`ax.fill_between(x, y, a)`: Creates a plot of `x` vs `y` and fills plot between `a` and `y`.

## Types

```
fig, ax = plt.subplots(1, 3, figsize=(16, 8))
ax[0].hist([1, 2, 3, 4, 5, 4, 3, 2, 3, 4, 2, 3, 4, 4],
           bins=5, color="yellow")
x = np.arange(0, 10, 0.1)
y = np.sin(x)
ax[1].fill_between(x, y, 0, color="green")
ax[2].scatter(x, y)
fig.savefig("out/types.pdf")
```

A vast number of plot types can be found in the [examples gallery](#).



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

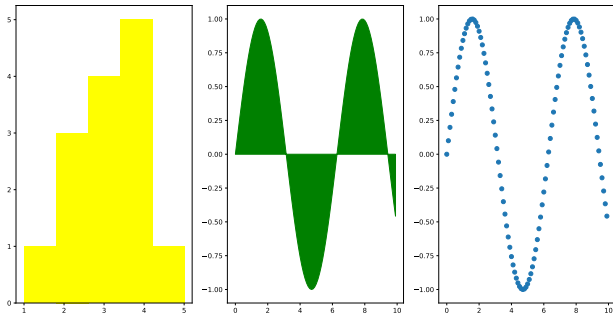
- Matplotlib package
- Figures and subplots

## Plot types and styles

- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization







## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

`plt.subplots_adjust(left, bottom, ..., hspace)`: Sets the space between the subplots. `wspace` and `hspace` control the percentage of the figure width and figure height, respectively, to use as spacing between subplots.

## Adjust spacing

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i][j].plot(randn(10))
plt.subplots_adjust(wspace=0, hspace=0)
fig.savefig("out/spacing.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

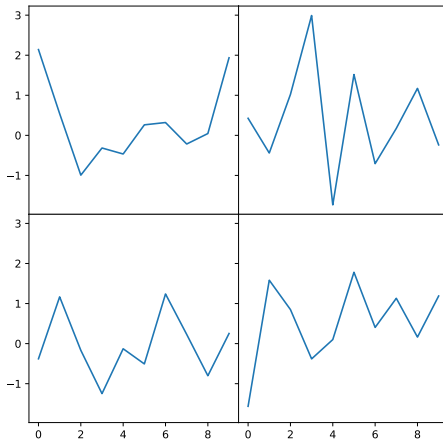
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

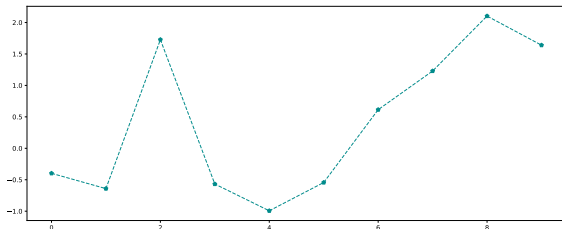
## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`ax.plot(data, linestyle, color, marker)`: Sets data and styles of subplot `ax`.

## Styles

```
fig, ax = plt.subplots(1, figsize=(15, 6))
ax.plot(randn(10), linestyle="--", color="darkcyan", marker="p")
fig.savefig("out/style.pdf")
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

black	black
gray	gray
silver	silver
whitesmoke	whitesmoke
rosybrown	rosybrown
firebrick	firebrick
red	red
darksalmon	darksalmon
sienna	sienna
sandybrown	sandybrown
bisque	bisque
tan	tan
moccasin	moccasin
floralwhite	floralwhite
gold	gold
darkkhaki	darkkhaki
lightgoldenrodyellow	lightgoldenrodyellow
olivedrab	olivedrab
chartreuse	chartreuse
palegreen	palegreen
darkgreen	darkgreen
seagreen	seagreen
mediumspringgreen	mediumspringgreen
lightseagreen	lightseagreen
paleturquoise	paleturquoise
darkcyan	darkcyan
darkturquoise	darkturquoise
deepskyblue	deepskyblue
aliceblue	aliceblue
slategray	slategray
royalblue	royalblue
navy	navy
blue	blue
mediumpurple	mediumpurple
darkorchid	darkorchid
plum	plum
m	m
mediumvioletred	mediumvioletred
palevioletred	palevioletred

k	k
gray	gray
lightgray	lightgray
w	w
lightcoral	lightcoral
maroon	maroon
mistyrose	mistyrose
coral	coral
seashell	seashell
peachpuff	peachpuff
darkorange	darkorange
navajowhite	navajowhite
orange	orange
darkgoldenrod	darkgoldenrod
lemonchiffon	lemonchiffon
ivory	ivory
olive	olive
yellowgreen	yellowgreen
lawngreen	lawngreen
lightgreen	lightgreen
g	g
mediumseagreen	mediumseagreen
mediumaquamarine	mediumaquamarine
mediumturquoise	mediumturquoise
darkslategray	darkslategray
c	c
cadetblue	cadetblue
skyblue	skyblue
dodgerblue	dodgerblue
slategrey	slategrey
ghostwhite	ghostwhite
darkblue	darkblue
slateblue	slateblue
rebeccapurple	rebeccapurple
darkviolet	darkviolet
violet	violet
fuchsia	fuchsia
deeppink	deeppink
crimson	crimson

dimgray	dimgray
darkgray	darkgray
lightgray	lightgray
white	white
indianred	indianred
darkred	darkred
salmon	salmon
orangered	orangered
chocolate	chocolate
peru	peru
burlywood	burlywood
blanchedalmond	blanchedalmond
wheat	wheat
goldenrod	goldenrod
khaki	khaki
beige	beige
y	y
darkolivegreen	darkolivegreen
honeydew	honeydew
forestgreen	forestgreen
green	green
springgreen	springgreen
aquamarine	aquamarine
azure	azure
darkslategrey	darkslategrey
aqua	aqua
powderblue	powderblue
lightskyblue	lightskyblue
lightslategray	lightslategray
lightsteelblue	lightsteelblue
lavender	lavender
mediumblue	mediumblue
darkslateblue	darkslateblue
blueviolet	blueviolet
mediumorchid	mediumorchid
purple	purple
magenta	magenta
hotpink	hotpink
pink	pink

dimgray	dimgray
darkgray	darkgray
gainsboro	gainsboro
snow	snow
brown	brown
r	r
tomato	tomato
lightsalmon	lightsalmon
saddlebrown	saddlebrown
linen	linen
antiquewhite	antiquewhite
papayawhip	papayawhip
oldlace	oldlace
cornsilk	cornsilk
palegoldenrod	palegoldenrod
lightyellow	lightyellow
yellow	yellow
greenyellow	greenyellow
darkseagreen	darkseagreen
limegreen	limegreen
lime	lime
mintcream	mintcream
turquoise	turquoise
lightcyan	lightcyan
teal	teal
cyan	cyan
lightblue	lightblue
steelblue	steelblue
lightslategrey	lightslategrey
cornflowerblue	cornflowerblue
midnightblue	midnightblue
b	b
mediumslateblue	mediumslateblue
indigo	indigo
thistle	thistle
darkmagenta	darkmagenta
orchid	orchid
lavenderblush	lavenderblush
lightpink	lightpink



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### line styles





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots

## Plot types and styles

- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Marker	Description
"."	point
","	pixel
"o"	circle
"v"	triangle_down
"8"	octagon
"s"	square
"p"	pentagon
"P"	plus (filled)
"*"	star
"h"	hexagon1
"H"	hexagon2
"+"	plus
"x"	x
"X"	x (filled)
"D"	diamond



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`ax.set_xticks()`: Sets list of X-ticks, analogously for Y-axis.

`ax.set_xlabel()`: Sets the X-label.

`ax.set_title()`: Sets the subplot title.

## Ticks and labels - default

```
fig, ax = plt.subplots(1, figsize=(15, 10))
ax.plot(randn(1000).cumsum())
fig.savefig("out/withoutlabels.pdf")
```

- Here, we create a Figure object as well as a subplot and fill it with a line plot of a *random walk*,
- By default matplotlib places the ticks evenly distributed along the data range. Individual ticks can be set as follows,
- By default there is no axis label or title.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

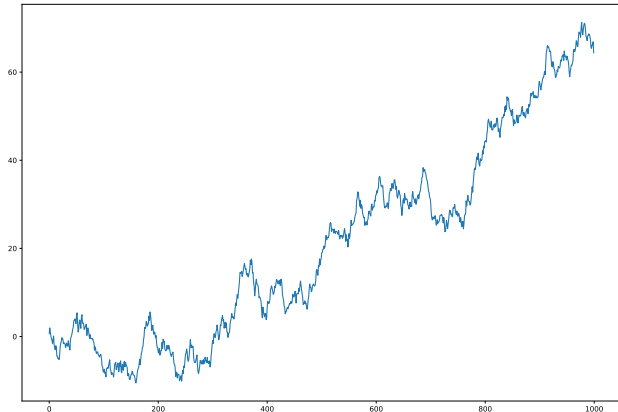
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization







## Essential concepts

Getting started  
Procedural programming  
Object-orientation

## Numerical programming

NumPy package  
Array basics  
Linear algebra

## Data formats and handling

Pandas package  
Series  
DataFrame  
Import/Export data

## Visual illustrations

Matplotlib package  
Figures and subplots  
Plot types and styles  
Pandas layers

## Applications

Time series  
Moving window  
Financial applications  
Optimization

### Set ticks and labels

```
ax.set_xticks([0, 250, 500, 750, 1000])  
ax.set_xlabel("Days", fontsize=20)  
ax.set_ylabel("Change", fontsize=20)  
ax.set_title("Simulation", fontsize=30)  
fig.savefig("out/labels.pdf")
```

- The individual ticks are given as a list to `ax.set_xticks()`,
- The label and title can be set to an individual size using the argument `fontsize`.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

**Plot types and styles**

Pandas layers

## Applications

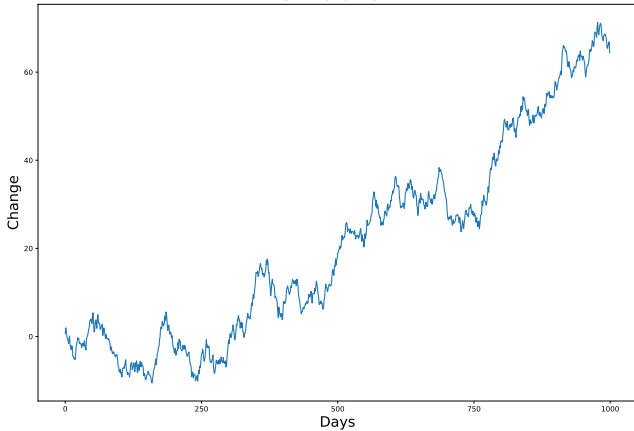
Time series

Moving window

Financial applications

Optimization

### Simulation





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

**Plot types and styles**

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Using multiple plots in one subplot one needs a legend.

`ax.legend(loc)`: Shows the legend at location `loc`.

Some options: `"best"`, `"upper right"`, `"center left"`, ...

## Set legend

```
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(1, 1, 1)
ax.plot(randn(1000).cumsum(), label="first")
ax.plot(randn(1000).cumsum(), label="second")
ax.plot(randn(1000).cumsum(), label="third")
ax.legend(loc="best", fontsize=20)
fig.savefig("out/legend.pdf")
```

- The legend displays the label and the color of the associated plot,
- Using the option `"best"` the legend will be placed in a corner where it does not interfere with the plots.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

**Plot types and styles**

Pandas layers

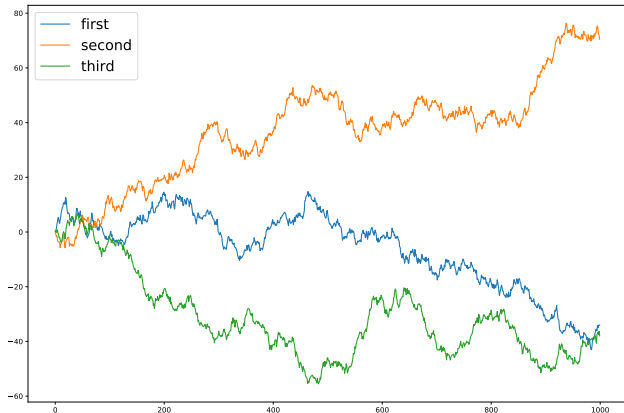
## Applications

Time series

Moving window

Financial applications

Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`ax.text(x, y, "text", fontsize)`: Inserts a text into a subplot.  
`ax.annotate("text", xy, xytext, arrowprops)`: Inserts an arrow with annotations.

## Annotations

```
ax.text(400, -30, "here", fontsize=50)
ax.annotate("there",
            fontsize=40,
            xy=(0, 0),
            xytext=(400, 8),
            arrowprops=dict(facecolor="black",
                            shrink=0.05))
ax.set_yticks([-40, -30, -20, -10, 0, 10, 20, 30, 40])
fig.savefig("out/arrow.pdf")
```

- Using `ax.annotate()` the arrow head points at `xy` and the bottom left corner of the text will be placed at `xytext`.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

**Plot types and styles**

Pandas layers

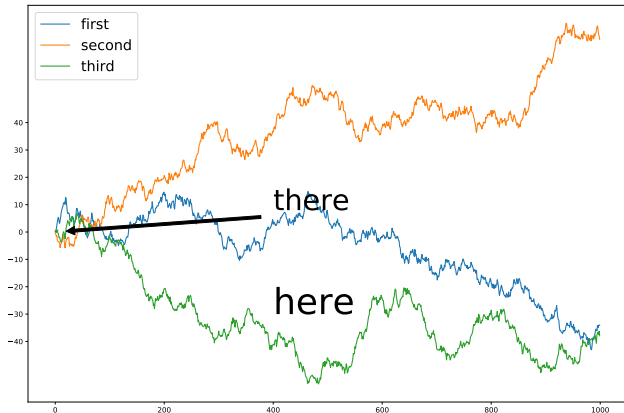
## Applications

Time series

Moving window

Financial applications

Optimization





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Annotation Lehman

```
import pandas as pd
from datetime import datetime

date = datetime(2008, 9, 15)
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
dow = pd.read_csv("data/dji.csv", index_col=0, parse_dates=True)
close = dow["Close"]
close.plot(ax=ax)
ax.annotate("Lehman Bankruptcy",
            fontsize=30,
            xy=(date, close.loc[date] + 400),
            xytext=(date, 22000),
            arrowprops=dict(facecolor="red",
                            shrink=0.03))
ax.set_title("Dow Jones Industrial Average", size=40)
fig.savefig("out/lehman.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

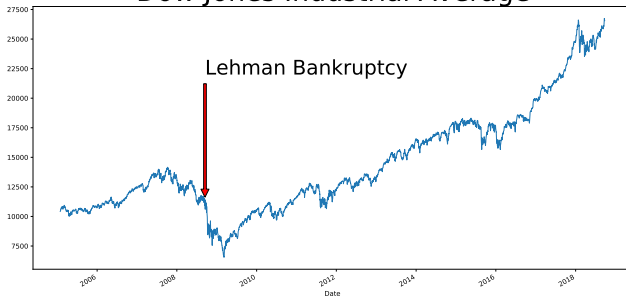
## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Dow Jones Industrial Average







## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`plt.Rectangle((x, y), width, height, angle)`: Creates a rectangle

`plt.Circle((x,y), radius)`: Creates a circle.

## Drawing

```
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(1, 1, 1)
ax.set_xticks([0, 1, 2, 3, 4, 5])
ax.set_yticks([0, 1, 2, 3, 4, 5])
rectangle = plt.Rectangle((1.5, 1),
                           width=0.8, height=2,
                           color="red", angle=30)
circ = plt.Circle((3, 3),
                   radius=1, color="blue")
ax.add_patch(rectangle)
ax.add_patch(circ)
fig.savefig("out/draw.pdf")
```

A list of all available patches can be found here: [matplotlib-patches](#)



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

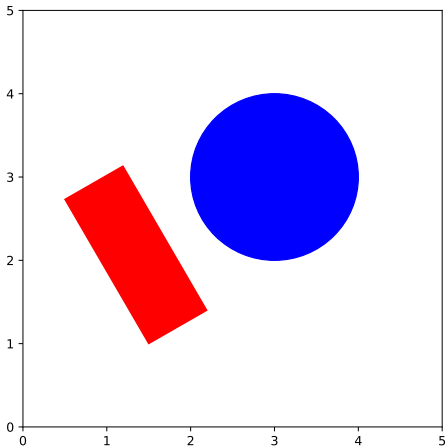
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Step 1

### Create a Figure object and subplots

#### Best practice Step 1

```
fig, ax = plt.subplots(1, 1, figsize=(16, 8))
```

## Step 2

### Plot data using different plot types

An overview of plot types can be found in the [examples gallery](#).

#### Best practice Step 2

```
x = np.arange(0, 10, 0.1)
y = np.sin(x)
ax.scatter(x, y)
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

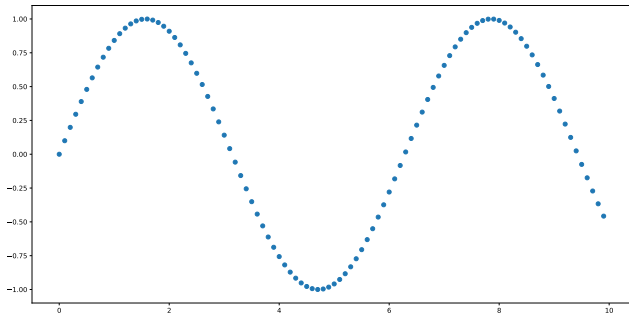
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Step 3

### Set colors, markers and line styles

#### Best practice Step 3

```
ax.scatter(x, y, color="green", marker="s")
```

## Step 4

### Set title, axis labels and ticks

#### Best practice Step 4

```
ax.set_title("Sine wave", fontsize=30)
ax.set_xticks([0, 2.5, 5, 7.5, 10])
ax.set_yticks([-1, 0, 1])
ax.set_ylabel("y-value", fontsize=20)
ax.set_xlabel("x-value", fontsize=20)
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

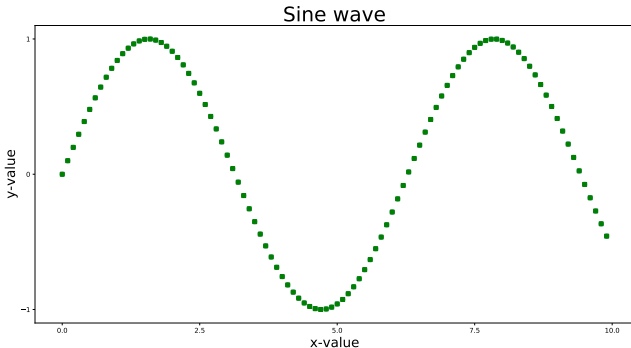
- Matplotlib package
- Figures and subplots

### Plot types and styles

- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles**
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Step 5 Set labels

### Best practice Step 5

```
ax.scatter(x, y, color="green", marker="s", label="Sine")
```

## Step 6 Set legend (if you add another plot to an existing figure)

### Best practice Step 6

```
ax.plot(np.arange(11) / 10, color="blue", linestyle="-",  
        label="Linear")  
ax.legend(fontsize=20)
```

## Step 7 Save plot to file

### Best practice Step 7

```
fig.savefig("out/sinewave.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

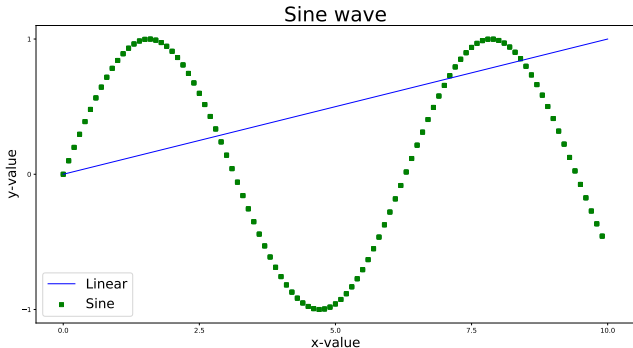
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization







## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Visual illustrations

## ▶ Pandas layers



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Plotting with `matplotlib` is often tedious and requires some research: You need to recall parameter details to create a professional charts. For recurring, everyday tasks, you might prefer another level of abstraction: Layer frameworks, which operate on top of `matplotlib`, produce pretty looking results with short methods and less code. The most popular packages are:



- `pandas` provides a convenient layer with frequently demanded plotting methods for its objects, such as [Series](#) and [DataFrames](#).
- `Seaborn` is a powerful graphics framework that allows you to easily create beautiful, complex graphics using a simple interface.

→ In this section, we will have a look at `pandas`' integrated layer methods. However, `Seaborn` also works very well with `pandas` objects.



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

DataFrame/Series.plot(): Plots a DataFrame or a Series.

## Simple line plot

```
plt.close("all")
p = pd.Series(np.random.rand(10).cumsum(),
              index=np.arange(0, 1000, 100))
```

p

```
## 0      0.669761
## 100    0.989702
## 200    1.655715
## 300    1.966073
## 400    2.151883
## 500    2.776987
## 600    2.839751
## 700    3.188431
## 800    4.169061
## 900    4.923286
## dtype: float64
```

```
p.plot()
plt.savefig("out/line.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

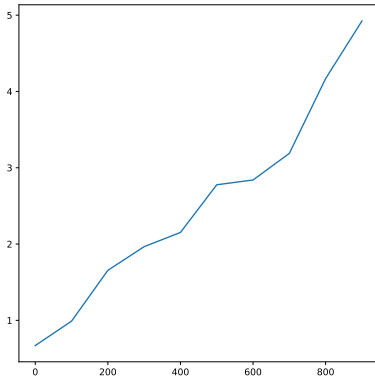
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numeral  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Line plots

```
df = pd.DataFrame(np.random.randn(10, 3), index=np.arange(10),  
                  columns=["a", "b", "c"])
```

df

##		a	b	c
## 0		1.703615	-1.376905	-1.336154
## 1		-1.402924	0.812501	1.739143
## 2		0.593504	0.699582	0.423217
## 3		1.140647	-1.454363	0.250578
## 4		-0.044809	0.438279	-0.821514
## 5		1.897959	-0.254581	0.157704
## 6		0.782639	1.196116	0.763081
## 7		0.577947	1.815039	1.175842
## 8		-0.278585	-0.538956	0.102930
## 9		-0.091891	0.310788	-0.857167

```
df.plot(figsize=(15, 12))  
plt.savefig("out/line2.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

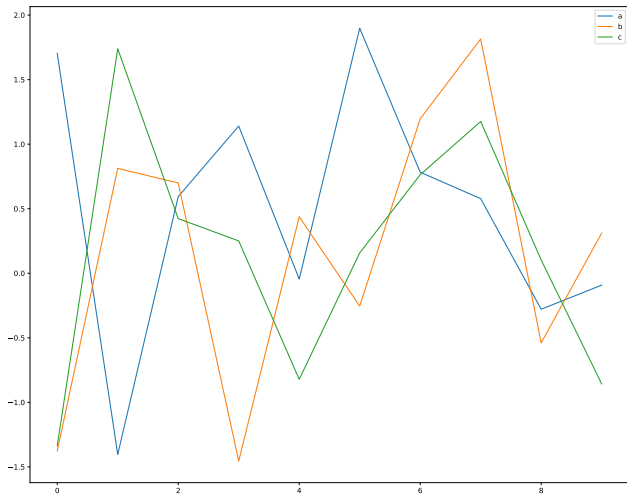
## Applications

Time series

Moving window

Financial applications

Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

The plot method applied to a DataFrame plots each column as a different line and shows the legend automatically. Plotting DataFrames, there are several arguments to change the style of the plot:

Argument	Description
kind	"line", "bar", etc
logy	logarithmic scale on Y-axis
use_index	If True, use index for tick labels
rot	Rotation of tick labels
xticks	Values for x ticks
yticks	Values for y ticks
grid	Set grid True or False
xlim	X-axis limits
ylim	Y-axis limits
subplots	Plot each DataFrame column in a new subplot

Table: Pandas plot arguments



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

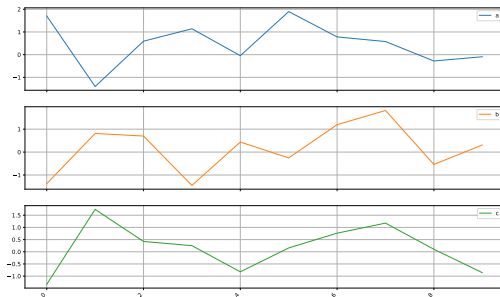
## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Separated line plots

```
df.plot(grid=True, rot=45, subplots=True, title="Example",  
        figsize=(15, 10))  
plt.savefig("out/pandas.pdf")
```

Example







## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual

### illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`dataframe.plot(ax=subplot)`: Plots a dataframe into subplot.

## Standard creation

```
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(1, 1, 1)
guests = np.array([[1334, 456], [1243, 597], [1477, 505],
                    [1502, 404], [854, 512], [682, 0]])
canteen = pd.DataFrame(guests,
                        index=["Mon", "Tue", "Wed",
                              "Thu", "Fri", "Sat"],
                        columns=["Zentral", "Turm"])
```

canteen

##	Zentral	Turm
## Mon	1334	456
## Tue	1243	597
## Wed	1477	505
## Thu	1502	404
## Fri	854	512
## Sat	682	0



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### Bar plot

```
canteen.plot(ax=ax, kind="bar")
ax.set_ylabel("guests", fontsize=20)
ax.set_title("Canteen use in Göttingen", fontsize=20)
fig.savefig("out/canteen.pdf")
```

- The bar plot resides in the subplot `ax`,
- The label and title are set as `shown before` without using pandas.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

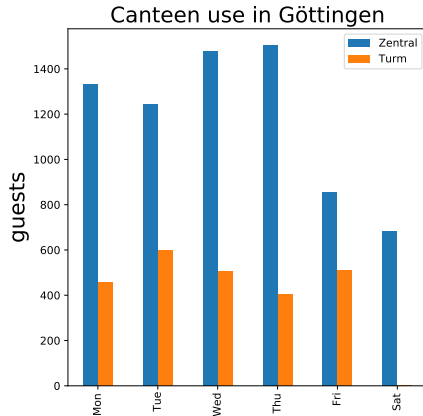
## Applications

Time series

Moving window

Financial applications

Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### Bar plot - stacked

```
canteen.plot(ax=ax, kind="bar", stacked=True)
ax.set_ylabel("guests", fontsize=20)
ax.set_title("Canteen use in Göttingen", fontsize=20)
fig.savefig("out/canteenstacked.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

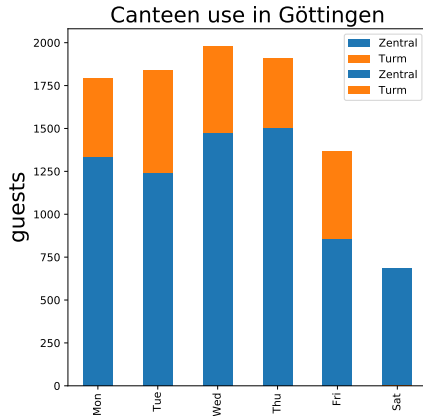
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### BTC chart

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
ax.set_ylabel("price", fontsize=20)
ax.set_xlabel("Date", fontsize=20)
BTC = pd.read_csv("data/btc-eur.csv", index_col=0, parse_dates=True)
BTCclose = BTC["Close"]
BTCclose.plot(ax=ax)
ax.set_title("BTC-EUR", fontsize=20)
fig.savefig("out/btc.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

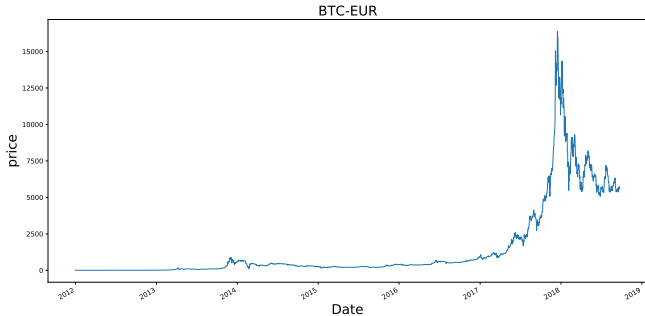
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Compare - bad illustration

```
amazon = pd.read_csv("data/amzn.csv", index_col=0,  
                      parse_dates=True)["Close"]  
siemens = pd.read_csv("data/sie.de.csv", index_col=0,  
                      parse_dates=True)["Close"]  
fig = plt.figure(figsize=(16, 8))  
ax = fig.add_subplot(1, 1, 1)  
ax.set_ylabel("price")  
amazon.plot(ax=ax, label="Amazon")  
siemens.plot(ax=ax, label="Siemens")  
ax.legend(loc="best")  
fig.savefig("out/compare.pdf")
```

- In this illustration you can hardly compare the trend of the two stocks,
- Using pandas you can standardize both dataframes in one line.





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

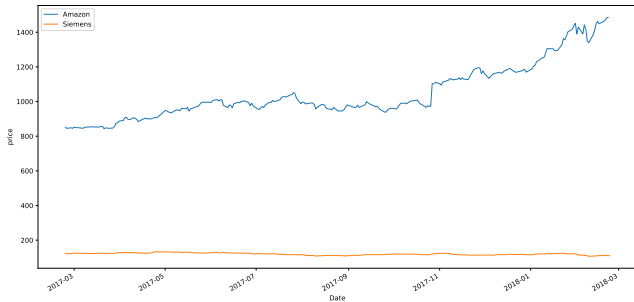
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### Compare - good illustration

```
amazon = amazon / amazon[0] * 100
siemens = siemens / siemens[0] * 100
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
ax.set_ylabel("percentage")
amazon.plot(ax=ax, label="Amazon")
siemens.plot(ax=ax, label="Siemens")
ax.legend(loc="best")
fig.savefig("out/comparenew.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

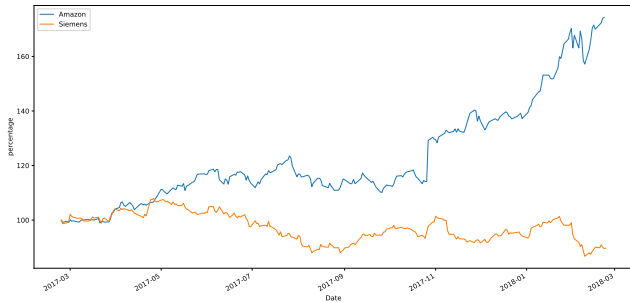
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers**

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Applications

## 5.1 Time series

## 5.2 Moving window

## 5.3 Financial applications

## 5.4 Optimization



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series**
- Moving window
- Financial applications
- Optimization

# Applications

## ▶ Time series



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Data types for date and time are included in the Python standard library.

### Datetime creation

```
from datetime import datetime
now = datetime.now()
now

## datetime.datetime(2020, 10, 31, 14, 12, 39, 24172)

now.day

## 31

now.hour

## 14
```

From `datetime` you can get the attributes `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`datetime`(year, month, day, ..., microsecond): Sets date and time.

### Datetime representation

```
holiday = datetime(2020, 12, 24, 8, 30)
holiday

## datetime.datetime(2020, 12, 24, 8, 30)

exam = datetime(2020, 12, 9, 10)
print("The exam will be on the " + "{:%Y-%m-%d}".format(exam))

## The exam will be on the 2020-12-09
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series**
- Moving window
- Financial applications
- Optimization

`timedelta(days, seconds, microseconds)`: Represents difference between two datetime objects.

### Datetime difference

```
from datetime import timedelta
delta = exam - now
delta

## datetime.timedelta(days=38, seconds=71240, microseconds=975828)

print("The exam will take place in " + str(delta.days) + " days.")

## The exam will take place in 38 days.

now

## datetime.datetime(2020, 10, 31, 14, 12, 39, 24172)

now + timedelta(10, 120)

## datetime.datetime(2020, 11, 10, 14, 14, 39, 24172)
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`datetime.strptime("format")`: Converts datetime object into string.  
`datetime.strptime(datestring, "format")`: Converts date as a string into a datetime object.

## Convert Datetime

```
stamp = datetime(2020, 4, 12)
stamp

## datetime.datetime(2020, 4, 12, 0, 0)

print("German date format: " + stamp.strftime("%d.%m.%Y"))

## German date format: 12.04.2020

val = "2020-5-5"
d = datetime.strptime(val, "%Y-%m-%d")
d

## datetime.datetime(2020, 5, 5, 0, 0)
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Converting examples

```
val = "31.01.2012"
d = datetime.strptime(val, "%d.%m.%Y")
d

## datetime.datetime(2012, 1, 31, 0, 0)

now.strftime("Today is %A and we are in week %W of the year %Y.")

## 'Today is Saturday and we are in week 43 of the year 2020.'

now.strftime("%c")

## 'Sat 31 Oct 2020 02:12:39 PM '
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Type	Description
%Y	4-digit year
%m	2-digit month [01, 12]
%d	2-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	2-digit minute [00, 59]
%S	Second [00, 61]
%W	Week number of the year [00, 53]
%F	Shortcut for %Y-%m-%d



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Type	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Full date and time
%x	Locale-appropriate formatted date



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`pd.date_range(start, end, freq)`: Generates a date range.

### Date ranges

```
import pandas as pd
index = pd.date_range("2020-01-01", now)
index[0:2]
index[15:16]
index = pd.date_range("2020-01-01", now, freq="M")
index[0:2]

## DatetimeIndex(['2020-01-01', '2...ype='datetime64[ns]', freq='D'])
## DatetimeIndex(['2020-01-16'], dtype='datetime64[ns]', freq='D')
## DatetimeIndex(['2020-01-31', '2...ype='datetime64[ns]', freq='M'])
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Alias	Offset type
D	Day
B	Business day
H	Hour
T	Minute
S	Second
M	Month end
BM	Business month end
Q-JAN, Q-FEB, ...	Quarter end
A-JAN, A-FEB, ...	Year end
AS-JAN, AS-FEB, ...	Year begin
BA-JAN, BA-FEB, ...	Business year end
BAS-JAN, BAS-FEB, ...	Business year begin



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`DataFrame.resample("frequency")`: Resamples time series by a specified frequency.

## Resample date ranges

```
import numpy as np
start = datetime(2016, 1, 1)
ind = pd.date_range(start, now)
numbers = np.arange((now - start).days + 1)
df = pd.DataFrame(numbers, index=ind)
```

```
df.head()
```

##		0
##	2016-01-01	0
##	2016-01-02	1
##	2016-01-03	2
##	2016-01-04	3
##	2016-01-05	4

```
df.resample("3BM").sum().head()
```

##		0
##	2016-01-29	406
##	2016-04-29	6734
##	2016-07-29	15015
##	2016-10-31	24205
##	2017-01-31	32246



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window**
- Financial applications
- Optimization

# Applications

## ▶ Moving window





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window**
- Financial applications
- Optimization

`DataFrame.rolling(window)`: Conducts rolling window computations.

### Rolling mean

```
import matplotlib.pyplot as plt
amazon = pd.read_csv("data/amzn.csv", index_col=0,
                    parse_dates=True)["Adj Close"]
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
ax.set_ylabel("price")
amazon.plot(ax=ax, label="Amazon")
amazon.rolling(window=20).mean().plot(ax=ax, label="Rolling mean")
ax.legend(loc="best")
ax.set_title("Amazon price and rolling mean", fontsize=25)
fig.savefig("out/amzn.pdf")
```

Frequently used rolling functions: `mean()`, `median()`, `sum()`, `var()`, `std()`, `min()`, `max()`.



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

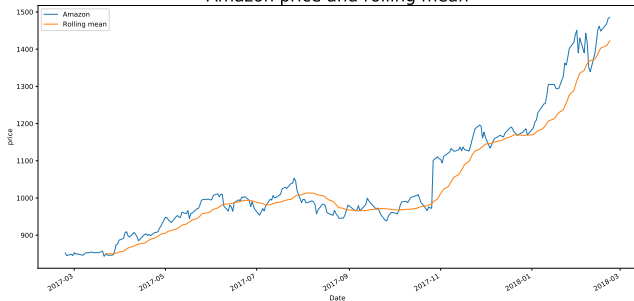
## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window**
- Financial applications
- Optimization

Amazon price and rolling mean





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window**
- Financial applications
- Optimization

## Standard deviation

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
pfizer = pd.read_csv("data/pfe.csv", index_col=0,
                    parse_dates=True)["Adj Close"]
pg = pd.read_csv("data/pg.csv", index_col=0,
                parse_dates=True)["Adj Close"]
prices = pd.DataFrame(index=amazon.index)
prices["amazon"] = pd.DataFrame(amazon)
prices["pfizer"] = pd.DataFrame(pfizer)
prices["pg"] = pd.DataFrame(pg)
prices_std = prices.rolling(window=20).std()
prices_std.plot(ax=ax)
ax.set_title("Standard deviation", fontsize=25)
fig.savefig("out/std.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

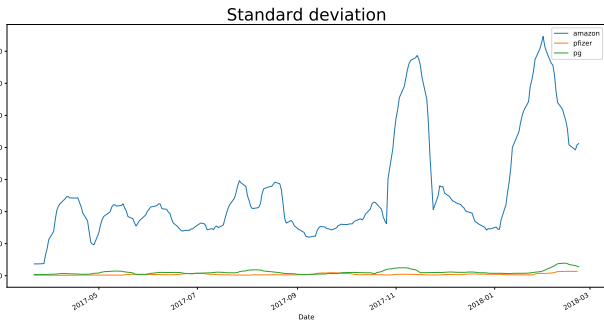
## Applications

Time series

Moving window

Financial applications

Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

### Logarithmic standard deviation

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
prices_std.plot(ax=ax, logy=True)
ax.set_title("Logarithmic standard deviation", fontsize=25)
fig.savefig("out/std_log.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

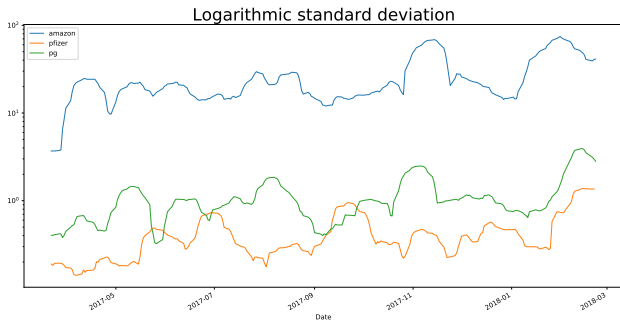
## Applications

Time series

**Moving window**

Financial applications

Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.ewm(span):` Computes exponentially weighted rolling window functions.

## Exponentially weighted functions

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
amazon.rolling(window=40).mean().plot(ax=ax, label="Rolling mean")
amazon.ewm(span=40).mean().plot(ax=ax, label="Exp mean",
                                linestyle="--", color="red")
amazon.plot(ax=ax, label="Amazon price")
ax.legend(loc="best")
ax.set_title("Exponentially weighted functions", fontsize=25)
fig.savefig("out/mean.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

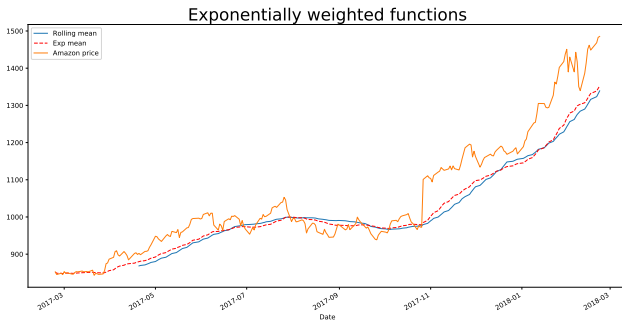
## Applications

Time series

Moving window

Financial applications

Optimization







## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

`DataFrame.pct_change()`: Computes the percentage changes per period.

### Percentage change

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
returns = prices.pct_change()
returns.head()
```

```
##                amazon    pfizer    pg
## Date
## 2017-02-23         NaN         NaN         NaN
## 2017-02-24 -0.008155  0.005872 -0.000878
## 2017-02-27  0.004023  0.000584 -0.001757
## 2017-02-28 -0.004242 -0.004668  0.001980
## 2017-03-01  0.009514  0.008792  0.006479
```

```
returns.plot(ax=ax)
ax.set_title("Returns", fontsize=25)
fig.savefig("out/returns.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.rolling().corr(benchmark)`: Computes correlation between two time series.

### Correlation

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
DJI = pd.read_csv("data/dji.csv", index_col=0,
                  parse_dates=True)["Adj Close"]
DJI_ret = DJI.pct_change()
corr = returns.rolling(window=20).corr(DJI_ret)
corr.plot(ax=ax)
ax.grid()
ax.set_title("20 days correlation", fontsize=25)
fig.savefig("out/corr.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

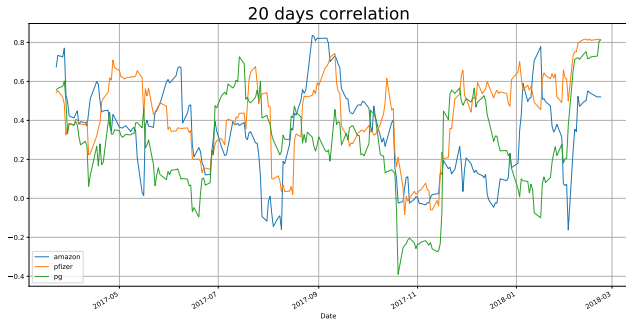
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window**
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications**
- Optimization

# Applications

## ▶ Financial applications



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual

illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Returns

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
ret_index = (1 + returns).cumprod()
stocks = ["amazon", "pfizer", "pg"]
for i in stocks:
    ret_index[i][0] = 1
ret_index.tail()
```

##		amazon	pfizer	pg
##	Date			
##	2018-02-15	1.715298	1.088693	0.932322
##	2018-02-16	1.699961	1.105461	0.934471
##	2018-02-20	1.723031	1.097840	0.920217
##	2018-02-21	1.740128	1.090218	0.907772
##	2018-02-22	1.742968	1.090218	0.914560

```
ret_index.plot(ax=ax)
ax.set_title("Cumulative returns", fontsize=25)
fig.savefig("out/cumret.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

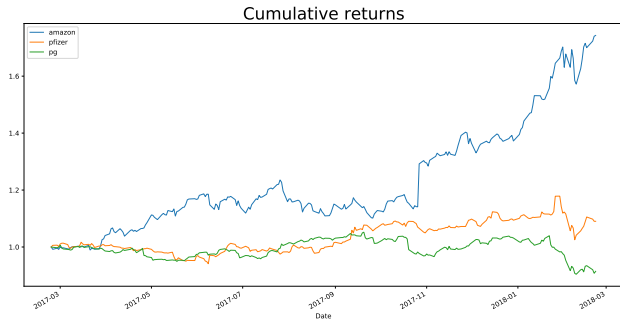
## Applications

Time series

Moving window

**Financial applications**

Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Monthly returns

```
returns_m = ret_index.resample("BM").last().pct_change()  
returns_m.head()
```

##	amazon	pfizer	pg
## Date			
## 2017-02-28	NaN	NaN	NaN
## 2017-03-31	0.049110	0.002638	-0.013396
## 2017-04-28	0.043371	-0.008477	-0.020604
## 2017-05-31	0.075276	-0.028124	0.008703
## 2017-06-30	-0.026764	0.028790	-0.010671





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Volatility

```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
vola = returns.rolling(window=20).std() * np.sqrt(20)
vola.plot(ax=ax)
ax.set_title("Volatility", fontsize=25)
fig.savefig("out/vola.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

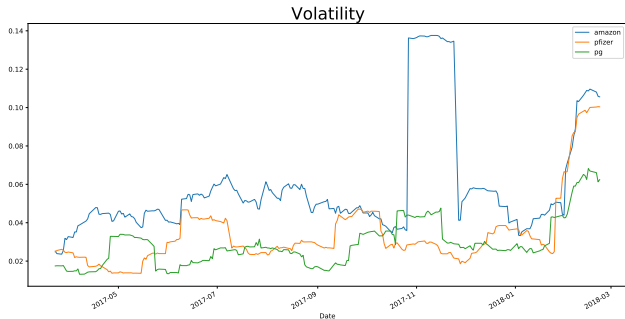
## Applications

Time series

Moving window

Financial applications

Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

`DataFrame.describe()`: Shows a statistical summary.

## Describe

`prices.describe()`

	amazon	pfizer	pg
## count	252.000000	251.000000	252.000000
## mean	1044.521903	33.892665	87.934304
## std	158.041844	1.694680	2.728659
## min	843.200012	30.872143	79.919998
## 25%	953.567474	32.593733	86.241475
## 50%	988.680023	33.147469	87.863598
## 75%	1136.952484	35.331834	90.363035
## max	1485.339966	38.661823	92.988976



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Histogram

```
fig, ax = plt.subplots(3, 1, figsize=(10, 8), sharex=True)
for i in range(3):
    ax[i].set_title(stocks[i])
    returns[stocks[i]].hist(ax=ax[i], bins=50)
fig.savefig("out/return_hist.pdf")
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

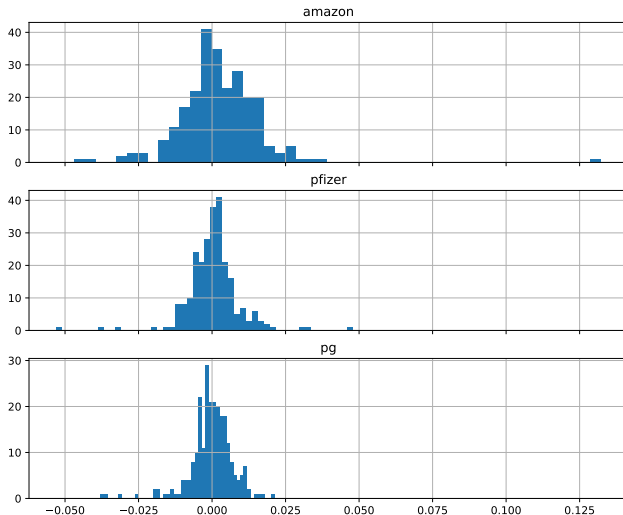
## Applications

Time series

Moving window

Financial applications

Optimization





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Using the statsmodels module to determine regressions:

`Series.tolist()`: Returns a list containing the DataFrame values.

`sm.OLS(Y, X).fit()`: Computes OLS fit of data (X, Y).

## Regression data

```
import statsmodels.api as sm

fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
Y = np.array(amazon.loc["2018-1-1":"2018-1-15"].tolist())
X = np.arange(len(Y))
ax.scatter(x=X, y=Y, marker="o", color="red")
fig.savefig("out/reg_data.pdf")
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

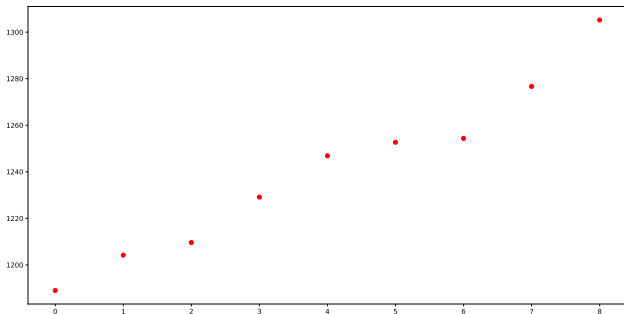
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

## Regression

```
X_reg = sm.add_constant(X)
res = sm.OLS(Y, X_reg).fit()
b, a = res.params
ax.plot(X, a * X + b)
fig.savefig("out/ols.pdf")
```





Summary of OLS regression. To print in python use `res.summary()`.

OLS Regression Results						
=====						
Dep. Variable:	y	R-squared:	0.965			
Model:	OLS	Adj. R-squared:	0.959			
Method:	Least Squares	F-statistic:	190.2			
Date:	Mo, 19 Mär 2018	Prob (F-statistic):	2.49e-06			
Time:	15:21:30	Log-Likelihood:	-29.706			
No. Observations:	9	AIC:	63.41			
Df Residuals:	7	BIC:	63.81			
Df Model:	1					
Covariance Type:	nonrobust					
=====						
	coef	std err	t	P> t	[0.025	0.975]
-----						
const	1187.8418	4.575	259.617	0.000	1177.023	1198.661
x1	13.2540	0.961	13.792	0.000	10.982	15.526
=====						
Omnibus:	0.788	Durbin-Watson:		1.627		
Prob(Omnibus):	0.674	Jarque-Bera (JB):		0.117		
Skew:	-0.268	Prob(JB):		0.943		
Kurtosis:	2.841	Cond. No.		9.06		
=====						



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

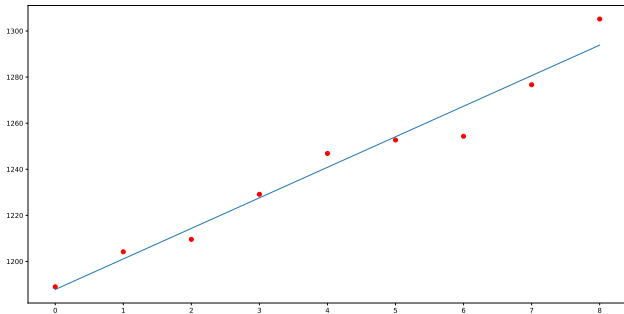
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

# Applications

## ▶ Optimization



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## NumPy package

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

The Newton-Raphson method is an algorithm for finding successively better approximations to the roots of real-valued functions.

Let  $F : \mathbb{R}^k \rightarrow \mathbb{R}^k$  be a continuously differentiable function and  $J_F(x_n)$  the Jacobian matrix of  $F$ . The recursive Newton-Raphson method to find the root of  $F$  is given by:

$$\mathbf{x}_{n+1} := \mathbf{x}_n - (J(\mathbf{x}_n)^{-1}F(\mathbf{x}_n))$$

with an initial guess  $x_0$ .

For  $f : \mathbb{R} \rightarrow \mathbb{R}$  the process is repeated as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Accordingly, we can determine the *optimum* of the function  $f$  by applying the method instead to  $f' = df/dx$ .



## Essential concepts

[Getting started](#)[Procedural programming](#)[Object-orientation](#)

## Numerical programming

[NumPy package](#)[Array basics](#)[Linear algebra](#)

## Data formats and handling

[Pandas package](#)[Series](#)[DataFrame](#)[Import/Export data](#)

## Visual illustrations

[Matplotlib package](#)[Figures and subplots](#)[Plot types and styles](#)[Pandas layers](#)

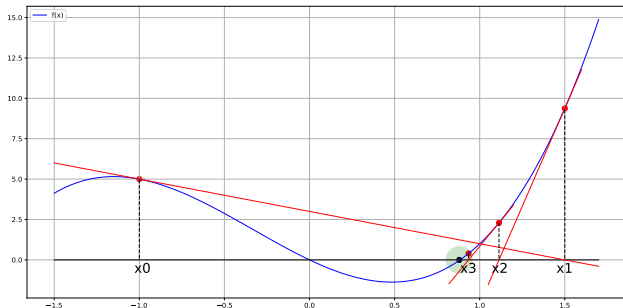
## Applications

[Time series](#)[Moving window](#)[Financial applications](#)[Optimization](#)

As an illustrative application, we consider the function

$$f(x) = 3x^3 + 3x^2 - 5x, \quad x \in \mathbb{R},$$

which is represented by the blue line in the following diagram. The figure depicts the iterative solution path applying the Newton-Raphson method to find the root, e.g.,  $x$  solving  $f(x) = 0$ , by tangent points and tangents starting from the initial guess  $x_0 = -1$ .





## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

The first step involves the definition of the function  $f(x)$  and its derivation  $f'(x)$  in Python:

### Newton-Raphson requirements

```
def f(x):  
    return 3 * x**3 + 3 * x**2 - 5 * x
```

```
def df(x):  
    return 9 * x**2 + 6 * x - 5
```

Finally, we implement the Newton-Raphson algorithm as outlined above. We allow for a (small) absolute deviation between the target function and its target value, i.e., 0. In addition, for a better understanding, we plot the solution path using the tangent points for  $x_0, x_1, \dots, x_N$ . The solution point is colored black. Hence, the lines starting with `ax.scatter()` are not part of the algorithm – they take global variables and are included just for the visual illustration.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

## Newton-Raphson

```
def newton_raphson(fun, dfun, x0, e):
    delta = abs(fun(x0))
    while delta > e:
        ax.scatter(x0, f(x0), color="red", s=80)
        x0 = x0 - fun(x0) / dfun(x0)
        delta = abs(fun(x0))
    ax.scatter(x0, f(x0), color="black", s=80)
    return(x0)

fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
x = np.arange(-1.5, 1.7, 0.001)
ax.plot(x, f(x))
ax.grid()
x_root = newton_raphson(f, df, -1, 0.1)
fig.savefig("out/newton_raphson_root.pdf")
print(f"Root at: {x_root:.4f}")

## Root at: 0.8878
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

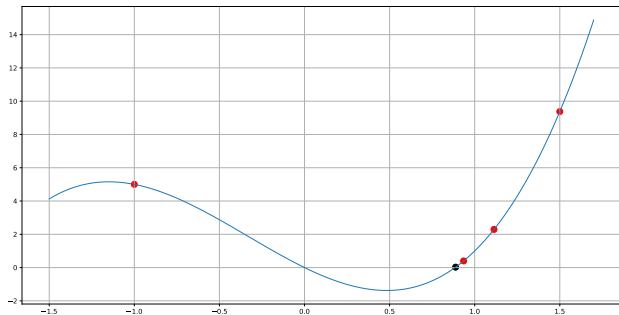
- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization







## Essential concepts

Getting started

Procedural programming

Object-orientation

## NumPy package

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

With the definition of the second derivative  $f''$ , i.e., the derivative of the derivative, we can employ the Newton-Raphson method to obtain an optimum of the target function  $f(x)$  numerically. Hence, the previous example needs only minimal modifications:

## Newton-Raphson

```
def ddf(x):  
    return 18 * x + 6  
  
fig = plt.figure(figsize=(16, 8))  
ax = fig.add_subplot(1, 1, 1)  
x = np.arange(-1.5, 1.7, 0.001)  
ax.plot(x, f(x))  
ax.grid()  
x_opt = newton_raphson(df, ddf, 1, 0.1)  
fig.savefig("out/newton_raphson_optimum.pdf")  
print(f"Minimum at: {x_opt:.4f}")  
  
## Minimum at: 0.4886
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

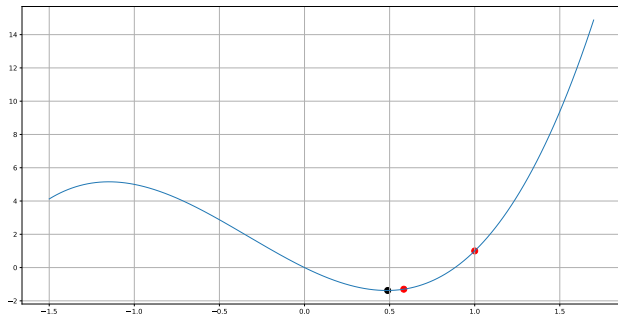
## Applications

Time series

Moving window

Financial applications

Optimization





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization



The `scipy.optimize` package provides several optimization algorithms. A detailed list of the functions is available [here](#).

The module contains:

- Unconstrained and constrained minimization of multivariate scalar functions using a variety of algorithms.
- Global optimization routines.
- Least-squares minimization and curve fitting algorithms.
- Scalar univariate functions minimizers and root finders.
- Multivariate equation system solvers.



## Essential concepts

Getting started  
Procedural programming  
Object-orientation

## Numerical programming

NumPy package  
Array basics  
Linear algebra

## Data formats and handling

Pandas package  
Series  
DataFrame  
Import/Export data

## Visual illustrations

Matplotlib package  
Figures and subplots  
Plot types and styles  
Pandas layers

## Applications

Time series  
Moving window  
Financial applications  
Optimization

`scipy.optimize.minimize()`: Provides minimization algorithms for multivariate scalar functions.

### Import minimize

```
from scipy.optimize import minimize
```

The `minimize()` functions has several parameters, of which the most important are:

- `fun`: The objective function to be minimized.
- `x0`: The initial guess.
- `method`: The solver algorithm, such as: BFGS, Nelder-Mead (Simplex), Newton Conjugate Gradient, and many more.
- `constraints`: The constraints definition.

A detailed list of all parameters can be found [here](#).



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

Let's get started by finding the minimum of the simple scalar function  $f(x) = (x - 4)^2 + 3$  using `minimize()`:

## 1D optimization using minimize

```
def f(x):  
    return (x - 4)**2 + 3  
  
x0 = [1]  # the initial guess  
result = minimize(f, x0)  
result  
  
##          fun: 3.00000000000000036  
##  hess_inv: array([[0.49999999]])  
##         jac: array([-8.94069672e-08])  
##  message: 'Optimization terminated successfully.'  
##         nfev: 6  
##         nit: 2  
##         njev: 3  
##        status: 0  
##    success: True  
##         x: array([3.99999994])
```



## Essential concepts

- Getting started
- Procedural programming
- Object-orientation

## Numerical programming

- NumPy package
- Array basics
- Linear algebra

## Data formats and handling

- Pandas package
- Series
- DataFrame
- Import/Export data

## Visual illustrations

- Matplotlib package
- Figures and subplots
- Plot types and styles
- Pandas layers

## Applications

- Time series
- Moving window
- Financial applications
- Optimization

Let's check if the minimum is correct by plotting the function and marking the minima:

### 1D optimization using minimize

```
min_y = result.fun  # get minimum of the function f
min_x = result.x     # get the x value of the minimum

fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
x = np.arange(1, 7, 0.001)
ax.plot(x, f(x))
ax.scatter(min_x, min_y, color="red", s=120)
fig.savefig("out/minimize_1D.pdf")
```





## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

Now, we consider a simple scalar function of two variables  
 $f(x_1, x_2) = (x_1 - 1)^2 + (x_2 - 2.5)^2$ :

## 2D optimization using minimize

```
def f(x):  
    return (x[0] - 1)**2 + (x[1] - 2.5)**2  
  
x0 = [0, 0]  # the initial guess  
result = minimize(f, x0)  
result  
  
##          fun: 1.968344227868139e-15  
##  hess_inv: array([[ 0.93103448, -0.1724138 ],  
##               [-0.1724138 ,  0.56896552]])  
##          jac: array([-6.95567350e-08,  4.21085256e-08])  
##  message: 'Optimization terminated successfully.'  
##          nfev: 9  
##          nit: 2  
##          njev: 3  
##          status: 0  
##  success: True  
##          x: array([0.99999996, 2.50000001])
```



Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

A famous performance test problem for optimization algorithms is the Rosenbrock function, which is defined by

$$f(x, y) = (a - x)^2 + b(y - x^2)^2.$$

It has a global minimum at  $(x, y) = (a, a^2)$  and the parameters are usually set to  $a = 1$  and  $b = 100$ :

## Comparison

```
def rosen(x):  
    return (1 - x[0])**2 + 100 * (x[1] - x[0]**2)**2  
  
x0 = [1.3, 0.4] # random initial guess  
  
res_1 = minimize(rosen, x0, method="Nelder-Mead")  
res_2 = minimize(rosen, x0, method="Powell")  
res_3 = minimize(rosen, x0, method="CG")  
res_4 = minimize(rosen, x0, method="BFGS")
```





Essential  
concepts

Getting started

Procedural  
programming

Object-orientation

Numerical  
programming

NumPy package

Array basics

Linear algebra

Data formats and  
handling

Pandas package

Series

DataFrame

Import/Export data

Visual  
illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

## Comparison results

*# The perfect solution would be (1, 1)*

res\_1.x

## array([1.00000287, 1.00000496])

res\_2.x

## array([1., 1.])

res\_3.x

## array([0.99999552, 0.99999104])

res\_4.x

## array([0.99999554, 0.99999108])

There is no "best" solver algorithm, it always depends on the problem.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

One key feature of the `minimize()` function is minimization with constraints.

Imagine being a producer of tin cans. Your goal is to create a tin can which has a capacity of 500 ml while consuming as little material as possible. There are two variables which describe the volume  $v$  and the surface  $s$  of a tin can: the radius  $r$  and the height  $h$ .

The volume is given by

$$v(r, h) = \pi \cdot r^2 \cdot h,$$

and the surface can be computed with

$$s(r, h) = 2 \cdot \pi \cdot r \cdot (r + h).$$

Your goal is to minimize the function  $s(r, h)$  with the constraint  $v(r, h) = 500$ .



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

We can easily implement the target and constraining function:

### Tin can optimization

```
def s(x):  
    r = x[0]  
    h = x[1]  
    return 2 * np.pi * r * (r + h)  
  
def v(x):  
    r = x[0]  
    h = x[1]  
    return np.pi * r**2 * h - 500 # as it is compared to zero
```

The constraint is defined in a special dictionary. The type is either `inequal` ("**ineq**") or `equal` ("**eq**") to zero:

### Constraints

```
con = {"type": "eq", "fun": v}
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

Numerical programming

NumPy package

Array basics

Linear algebra

Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

Applications

Time series

Moving window

Financial applications

Optimization

The last step is to set the initial guess and to choose a solver algorithm. Only a few solver algorithms work with constraints, one of which is abbreviated by "SLSQP":

## Tin can optimization

```
x0 = [1, 1]
result = minimize(s, x0, method="SLSQP", constraints=con)
result

##      fun: 348.73420544452335
##      jac: array([108.10271072,  27.02567291])
## message: 'Optimization terminated successfully'
##      nfev: 29
##      nit: 9
##      njev: 9
##      status: 0
##      success: True
##       x: array([4.30126983,  8.60254111])

x = result.x
```



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

We have found a combination of radius and height which gives us a minimal surface of the tin can:

### Tin can optimization result

```
r, h = x
r
## 4.301269826608837
h
## 8.602541108098178
np.pi * r**2 * h
## 499.9999999820085
s(x)
## 348.73420544452335
```

A tin can with  $r = 4.3$  cm and  $h = 8.6$  cm has a volume of 500 ml and a minimal material consumption of 348.7 qcm.



## Essential concepts

Getting started

Procedural programming

Object-orientation

## Numerical programming

NumPy package

Array basics

Linear algebra

## Data formats and handling

Pandas package

Series

DataFrame

Import/Export data

## Visual illustrations

Matplotlib package

Figures and subplots

Plot types and styles

Pandas layers

## Applications

Time series

Moving window

Financial applications

Optimization

