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.
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.
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...
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.
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* by Eric Matthes,
- *Python Data Science Handbook* by Jake VanderPlas,
- *Python for Data Analysis, 2nd Edition* by Wes McKinney,
- *Python for Finance* by Yves Hilpisch.
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.

```bash
python3 --version
```

## Python 3.6.7

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 analytical purposes in order to obtain fast results when performing simple applications.
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.
In this course – in a numerical and analytical 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.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.
A Jupyter notebook is divided into individual, vertically arranged cells, which can be executed separately:

```
In [2]: a = 10
   b = 15

In [4]: a
Out[4]: 10

In [5]: a + b
Out[5]: 25
```

The notebook approach is not novel and comes from the field of computer algebra software.
Actually, an interactive Python interpreter called IPython is started “in the core”.

```
IPython running [command line]

ipython3 --version
```

# 6.5.0

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.
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*
Table of contents

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

© 2019 PyEcon.org
Essential concepts

1.1 Getting started
1.2 Procedural programming
1.3 Object-orientation
Essential concepts

Getting started
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.
... 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**

```python
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.
Overview of the Python development by versions and dates:

- **Python’s birthday**
- **Python 2.0**
- **Python 3.0**
- **Python 2.7 lives forever**
- **Python 2.7 will die**
- **Python 3.6**
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.
In comparison

**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:

Source: https://stackoverflow.blog/2017/09/06/incredible-growth-python/
So, you’re on the right track – because who wants to bet on the wrong horse?

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

Procedural programming
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:

```python
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.
Let’s add a user input to the program:

```python
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 `+`. 
Determining weekdays

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

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

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

```python
## 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?

```python
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"!
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.
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

<table>
<thead>
<tr>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
</tr>
<tr>
<td>-</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>**</td>
</tr>
<tr>
<td>//</td>
</tr>
<tr>
<td>%</td>
</tr>
<tr>
<td>@</td>
</tr>
<tr>
<td>&lt;&lt;</td>
</tr>
<tr>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>&amp;=</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>^=</td>
</tr>
<tr>
<td>~</td>
</tr>
<tr>
<td>~</td>
</tr>
<tr>
<td>==</td>
</tr>
<tr>
<td>!=</td>
</tr>
<tr>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;=</td>
</tr>
<tr>
<td>&gt;=</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>not</td>
</tr>
<tr>
<td>in</td>
</tr>
<tr>
<td>not in</td>
</tr>
<tr>
<td>is</td>
</tr>
<tr>
<td>is not</td>
</tr>
</tbody>
</table>

An overview of the delimiters follows:

### Overview of delimiters

<table>
<thead>
<tr>
<th>Delimiters</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>[</td>
</tr>
<tr>
<td>]</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>,</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>=</td>
</tr>
<tr>
<td>;</td>
</tr>
<tr>
<td>-&gt;</td>
</tr>
<tr>
<td>+=</td>
</tr>
<tr>
<td>-=</td>
</tr>
<tr>
<td>*=</td>
</tr>
<tr>
<td>/=</td>
</tr>
<tr>
<td>**=</td>
</tr>
<tr>
<td>//=</td>
</tr>
<tr>
<td>%=</td>
</tr>
<tr>
<td>@=</td>
</tr>
<tr>
<td>&lt;&lt;</td>
</tr>
<tr>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
</tr>
<tr>
<td>&amp;=</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>^=</td>
</tr>
<tr>
<td>~</td>
</tr>
<tr>
<td>''</td>
</tr>
<tr>
<td>\</td>
</tr>
<tr>
<td>@</td>
</tr>
<tr>
<td>SPACE</td>
</tr>
</tbody>
</table>
All regular arithmetic operations involving numbers are possible:

<table>
<thead>
<tr>
<th>Pocket calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 + 5</td>
</tr>
<tr>
<td>100 - 20</td>
</tr>
<tr>
<td>8 / 2</td>
</tr>
<tr>
<td>4 * (10 + 20)</td>
</tr>
<tr>
<td>2**3</td>
</tr>
<tr>
<td># 15</td>
</tr>
<tr>
<td># 80</td>
</tr>
<tr>
<td># 4.0</td>
</tr>
<tr>
<td># 120</td>
</tr>
<tr>
<td># 8</td>
</tr>
</tbody>
</table>

- 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.
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

```python
# 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:3} {a and b:6} {a or b:8} {not a:7}"
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a and b</th>
<th>a or b</th>
<th>not a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```
The programmer explains the structure of his/her program to the interpreter via a restricted set of short commands, the *keywords*:

<table>
<thead>
<tr>
<th>Overview of keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>## and</td>
</tr>
<tr>
<td>## def</td>
</tr>
<tr>
<td>## finally</td>
</tr>
<tr>
<td>## in</td>
</tr>
<tr>
<td>## or</td>
</tr>
<tr>
<td>## while</td>
</tr>
</tbody>
</table>

There are two ways to make *comments*:

<table>
<thead>
<tr>
<th>Provide some comments</th>
</tr>
</thead>
<tbody>
<tr>
<td># Set variable to something - or nothing?</td>
</tr>
<tr>
<td>something = None</td>
</tr>
</tbody>
</table>

""
I am a docstring!
A multiline string comment hybrid.
I will be useful for describing classes and methods.
"""
Python offers the following *basic data types*, which we will use in this course:

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

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

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

```python
## ['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**, and
- The data type `list()` possesses its own methods.
**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:

<table>
<thead>
<tr>
<th>Selecting elements in sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>lottery = (1, 8, 9, 12, 24, 28)</td>
</tr>
<tr>
<td>len(lottery)</td>
</tr>
<tr>
<td>lottery[1:3]</td>
</tr>
<tr>
<td>lottery[:4]</td>
</tr>
<tr>
<td>lottery[-1]</td>
</tr>
<tr>
<td>lottery[-2:]</td>
</tr>
</tbody>
</table>

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

```python
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()
```

```python
## {imho: in my humble opinion, lol: laughing out loud, tl;dr: too long; didn’t read}
## 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:

<table>
<thead>
<tr>
<th>Set operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = {&quot;o&quot;, &quot;n&quot;, &quot;y&quot;, &quot;t&quot;}</td>
</tr>
<tr>
<td>y = {&quot;p&quot;, &quot;h&quot;, &quot;o&quot;, &quot;n&quot;}</td>
</tr>
<tr>
<td>x &amp; y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>x - y</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Op.</th>
<th>True, only if the value of the left operand is</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>less than the value of the right operand</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>less than or equal to the value of the right operand</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than the value of the right operand</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>greater than or equal to the value of the right operand</td>
</tr>
<tr>
<td><code>==</code></td>
<td>equal to the right operand</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>not equal to the right operand</td>
</tr>
</tbody>
</table>

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.
Comparing examples

```python
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.
In Python, comparison operators can also be chained.

<table>
<thead>
<tr>
<th>Chaining comparison examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 5</code></td>
</tr>
<tr>
<td><code>5 &gt;= x &gt; 4</code></td>
</tr>
<tr>
<td><code>## True</code></td>
</tr>
<tr>
<td><code>12 &lt; x &lt; 20</code></td>
</tr>
<tr>
<td><code>## False</code></td>
</tr>
<tr>
<td><code>2 &lt; x &lt; 10</code></td>
</tr>
<tr>
<td><code>## True</code></td>
</tr>
<tr>
<td><code>2 &lt; x and x &lt; 10</code></td>
</tr>
<tr>
<td><code>## True</code></td>
</tr>
</tbody>
</table>

The comparison is performed for both sides and combined by `and`. 
There are three logical operators: `not`, `and`, `or`.

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>not</code> x</td>
<td>Returns <code>True</code> only if <code>x</code> is <code>False</code></td>
</tr>
<tr>
<td>x and y</td>
<td>Returns <code>True</code> only if <code>x</code> and <code>y</code> are <code>True</code></td>
</tr>
<tr>
<td>x or y</td>
<td>Returns <code>True</code> only if <code>x</code> or <code>y</code> or both are <code>True</code></td>
</tr>
</tbody>
</table>

**Logical operators examples**

```python
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 `!=`.

```
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.
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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
</tr>
</tbody>
</table>
How to convert binary numbers to integers (the unknown keywords and language structures will be introduced soon):

```python
def bintoint(binary):
    binary = binary[::-1]
    num = 0
    for i in range(len(binary)):
        num += int(binary[i]) * 2**i
    return num

bintoint("1101001")
```

```python
# 105
```

```python
int("1101001", 2)  # compare with built-in function
```

```python
# 105
```
How to convert integers to binary numbers:

```python
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'
```
Python offers distinct bitwise operators. Some of them will be redefined entirely different by extensions, such as, e.g., vectorization.

<table>
<thead>
<tr>
<th>Bit. op.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x &gt;&gt; y</code></td>
<td>Returns <code>x</code> with the bits shifted to the left by <code>y</code> places</td>
</tr>
<tr>
<td><code>x &lt;&lt; y</code></td>
<td>Returns <code>x</code> with the bits shifted to the right by <code>y</code> places</td>
</tr>
<tr>
<td><code>x &amp; y</code></td>
<td>Does a bitwise <strong>and</strong></td>
</tr>
<tr>
<td>`x</td>
<td>y`</td>
</tr>
<tr>
<td><code>~ x</code></td>
<td>Returns the complement of <code>x</code></td>
</tr>
<tr>
<td><code>x ^ y</code></td>
<td>Does a bitwise exclusive or</td>
</tr>
</tbody>
</table>

```python
a, b = 5, 7

c = a & b  # bitwise and

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

print(c)

# 5
```
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

```python
bytes = 1000000000 / 8  # e.g. DSL 100000
if bytes >= 1e9:
    print(f"{bytes/1e9:.2f} GByte")
elif bytes >= 1e6:
    print(f"{bytes/1e6:.2f} MByte")
elif bytes >= 1e3:
    print(f"{bytes/1e3:.2f} KByte")
else:
    print(f"{bytes:.2f} Byte")
```

## 12.50 MByte

Control flow structures may be nested in any order:

### Nestings

```python
if a > 1:
    if b > 2:
        pass  # special keyword for empty blocks
```
In Python there exist two conventional *program loops* – *for-in-else*:

### Total sum

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

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

Lists or other collections can also be created dynamically:

### Powers of 2

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

```python
# [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
# {'***': 1, '**': 2, '*': 3}
```
Loops can skip iterations (**continue**):

Continue the loop

```python
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

```python
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}\.")
```  
## C
Control flow: The while loop

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`:

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

```python
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)
```

```python
## [2, 3, 4, 16, 23, 28]
## [82, 84, 94, 95, 99, 100]
## [5, 12, 16]
```
Functions are of type `callable()`, defined as closures, and can be created and used like other objects:

```python
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

Object-orientation
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.
When you assign a variable, a *reference* to an object is set:

<table>
<thead>
<tr>
<th>Equal but not identical</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = ['Star', 'Trek']</td>
</tr>
<tr>
<td>b = ['Star', 'Trek']</td>
</tr>
<tr>
<td>c = a</td>
</tr>
<tr>
<td>a == b</td>
</tr>
<tr>
<td>a == c</td>
</tr>
<tr>
<td>a is b</td>
</tr>
<tr>
<td>a is c</td>
</tr>
</tbody>
</table>

```python
# ["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.
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.
In Python, arguments are *passed by assignment*, i.e., call-by-reference:

```python
def last_element(x):
    return x.pop(-1)
```

```python
a = stocks
last_element(a)
a
```

```python
## ['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.
We are able to make an exact copy of the object:

```python
def last_element(x):
    y = x.copy()
    return y.pop(-1)
```

```python
a = stocks
last_element(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.
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:

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

```python
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`:

```python
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:

```python
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()`.
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:

```python
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.
Namespaces

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.
Namespaces

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

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

```python
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 0x7fe838606f28>
## EHEHEHEH
## EHEHEHEHEH
```
In order to provide, maintain and extend modular functionality with Python, its code containing components can be described hierarchically:

```
+------+
| Packages |
+------+
    | Modules |
    +------+
    | Classes |
    +------+
        | Functions |
```

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.
**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:

<table>
<thead>
<tr>
<th>Import statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>import datetime</td>
</tr>
<tr>
<td>import datetime as dt</td>
</tr>
<tr>
<td>from datetime import date, timedelta</td>
</tr>
<tr>
<td>from datetime import *</td>
</tr>
</tbody>
</table>

```
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.
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](https://docs.python.org/) give an overview of all build-in modules.

**Usage of build-in modules**

```python
import math
from random import randint

math.pi
```

```
# 3.141592653589793
```

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

```
# 120
```

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

```
# 18
```
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`
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
e98308214098ef3471d76ab74200f9800efeef15/opencv_python-4.0.0.21-cp36-cp36m-manylinux1_x86_64.whl (25.4MB)
  100% [===============================================] 25.4MB 523kB/s
Requirement already satisfied: numpy>=1.11.3 in /usr/local/lib/python3.6/dist-packages (from opencv-python) (1.15.4)
Installing collected packages: opencv-python
Successfully installed opencv-python-4.0.0.21
```
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 focusing on the Python language.

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

```python
s = "Hello world!"
l = [1, 2, 3, 5, 5]

def add_one(n):
    return n + 1
```

File: `mymodule.py`
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:

```python
import mymodule
mymodule.s
mymodule.l
mymodule.add_one(5)
## Hello world!
## [1, 2, 3, 5, 5]
## 6
```
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:

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

# 5
If a package directory contains a file \texttt{\_\_init\_.py}, its code is invoked when the package gets imported. The directory \texttt{mypackage}, now, contains the two modules and the initialization file:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{mypackage_directory.png}
\caption{Directory structure of the package.}
\end{figure}

The file \texttt{\_\_init\_.py} can be empty but can also be used for package initialization purposes.
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.
## ...
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.
Numerical programming

2.1 NumPy package
2.2 Array basics
2.3 Linear algebra
Numerical programming

NumPy package
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**

```python
import numpy as np
```
Element-wise addition

```python
cvec1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
cvec2 = np.array(cvec1)
cvec1 + cvec1
```

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

```python
cvec2 + cvec2
```

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

```python
for i in range(len(cvec1)):
    cvec1[i] += cvec1[i]
```

```python
cvec1
```

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

```python
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.]])
```
Motivation

Time comparison

```python
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 17.29180230837526 times faster!
```
Numerical programming

Array basics
Creating NumPy arrays

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.

<table>
<thead>
<tr>
<th>Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr1 = [4, 8, 2]</td>
</tr>
<tr>
<td>arr1 = np.array(arr1)</td>
</tr>
<tr>
<td>arr2 = np.array([24.3, 0., 8.9, 4.4, 1.65, 45])</td>
</tr>
<tr>
<td>arr3 = np.array([[4, 8, 5], [9, 3, 4], [1, 0, 6]])</td>
</tr>
<tr>
<td>arr1.ndim</td>
</tr>
<tr>
<td>## 1</td>
</tr>
<tr>
<td>arr3.shape</td>
</tr>
<tr>
<td>## (3, 3)</td>
</tr>
</tbody>
</table>

From now on, the name array refers to an np.array().
Array creation functions

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))

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

np.arange(6)

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

np.identity(3)

```python
## array([[1., 0., 0.],
##        [0., 1., 0.],
##        [0., 0., 1.]])
```
array creation functions

\[
\text{\texttt{np.linspace}(\text{\texttt{start}}, \text{\texttt{stop}}, \text{\texttt{n}}): Creates vector of } n \text{ evenly divided values from } \text{\texttt{start}} \text{ to } \text{\texttt{stop}.}\\
\text{\texttt{np.full}((\text{\texttt{row}}, \text{\texttt{column}}), \text{\texttt{k}}): Creates array with all values set to } k.
\]

### Array creation

\[
\text{\texttt{np.linspace}(0, 80, 5)}
\]

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

\[
\text{\texttt{np.full}((5, 4), 7)}
\]

```python
## array([[7, 7, 7, 7],
## [7, 7, 7, 7],
## [7, 7, 7, 7],
## [7, 7, 7, 7],
## [7, 7, 7, 7]])
```
Array creation functions

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)

```python
## 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))

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

Reference

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

<table>
<thead>
<tr>
<th>Copy</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr3</td>
<td>arr3</td>
</tr>
</tbody>
</table>
| ## array([[4, 8, 5],
| ## [9, 3, 4],
| ## [1, 0, 6]])
| arr = arr3.copy()
| arr[1, 1] = 777
| arr3          | arr3                 |
| ## array([[4, 8, 5],
| ## [9, 3, 4],
| ## [1, 0, 6]])
| arr3[1, 1] = 3 |
### Overview: Array creation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>Convert input array in NumPy array</td>
</tr>
<tr>
<td>arange(start,stop,step)</td>
<td>Creates array from given input</td>
</tr>
<tr>
<td>ones</td>
<td>Creates array containing only ones</td>
</tr>
<tr>
<td>zeros</td>
<td>Creates array containing only zeros</td>
</tr>
<tr>
<td>empty</td>
<td>Allocating memory without specific values</td>
</tr>
<tr>
<td>eye, identity</td>
<td>Creates $N \times N$ identity matrix</td>
</tr>
<tr>
<td>linspace</td>
<td>Creates array of evenly divided values</td>
</tr>
<tr>
<td>full</td>
<td>Creates array with values set to one number</td>
</tr>
<tr>
<td>random.rand</td>
<td>Creates array of random floats</td>
</tr>
<tr>
<td>random.randint</td>
<td>Creates array of random int</td>
</tr>
</tbody>
</table>
array.dtype: Returns the type of array.
array.astype(np.type): Conducts a manual typecast.

<table>
<thead>
<tr>
<th>Data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr1.dtype</td>
</tr>
<tr>
<td># dtype('int64')</td>
</tr>
<tr>
<td>arr2.dtype</td>
</tr>
<tr>
<td># dtype('float64')</td>
</tr>
<tr>
<td>arr1 = arr1 * 2.5</td>
</tr>
<tr>
<td>arr1.dtype</td>
</tr>
<tr>
<td># dtype('float64')</td>
</tr>
<tr>
<td>arr1 = (arr1 / 2.5).astype(np.int64)</td>
</tr>
<tr>
<td>arr1.dtype</td>
</tr>
<tr>
<td># dtype('int64')</td>
</tr>
</tbody>
</table>
**Element-wise operations**

Calculation operators on NumPy arrays operate element-wise.

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

**Indexing with an integer**

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

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

```python
arr[4]
```

```markdown
## 4
```

```python
arr[-1]
```

```markdown
## 9
```
array[start : stop : step]: Selects a subset of the data.

### Slicing in one dimension

```python
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])
```
### 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])
```
Slicing in higher dimensions

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

Indexing rows

```python
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])
```
Slicing

Slicing in two dimensions

```python
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]])
```

© 2019 PyEcon.org
Figure: Python for Data Analysis (2017) on page 99
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.
A view is created automatically when you do basic indexing such as slicing:

```python
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]])
```
### Create a view by slicing

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

```python
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
```
Obtaining views explicitly

Obtain a view

```python
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.
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:

```
arr3

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

arr = arr3[[0, 2], [0, 2]]

arr

# array([[0, 2], [0, 2]])

arr.base
```
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.
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.

```python
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**.
Boolean arrays

Logical operations on NumPy arrays work in a similar way compared to bitwise operators.
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.

```python
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])
```
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:

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

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

```python
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.

```python
Best practice Step 1a

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

```mat
```n
cmat

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

```mat
```n
mat

```mat

```mat

mat[2, 2]

```mat
```n
10

mat[0, -1]

```mat
```n
3

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

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

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

```python
mat[2]
```

```
[[ 8,  9, 10, 11]]
```

```python
mat[0]
```

```
[[0, 1, 2, 3]]
```

By specifying the *row index* only, we create arrays which are views.
Best practice: Indexing arrays

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]])
```
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

```python
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.
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.

```python
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.
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

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

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

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

```
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]])
```
### Adding and removing elements of arrays

**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

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

```python
## [ 0 1 2 77 3 4 8]
```

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

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

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

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

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

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

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
```
**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]])
```
## Overview: Element-wise array functions

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

```python
x = np.array([3, -6, 8, 4, 3, 5])
y = np.array([3, 5, 7, 3, 5, 9])
np.maximum(x, y)
```

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

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

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

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

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

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

```python
# array([0, 4, 1, 1, 3, 5])
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add elements of arrays</td>
</tr>
<tr>
<td>subtract</td>
<td>Subtract elements in the second from the first array</td>
</tr>
<tr>
<td>multiply</td>
<td>Multiply elements</td>
</tr>
<tr>
<td>divide</td>
<td>Divide elements</td>
</tr>
<tr>
<td>power</td>
<td>Raise elements in first array to powers in second</td>
</tr>
<tr>
<td>maximum</td>
<td>Element-wise maximum</td>
</tr>
<tr>
<td>minimum</td>
<td>Element-wise minimum</td>
</tr>
<tr>
<td>mod</td>
<td>Element-wise modulus</td>
</tr>
<tr>
<td>greater, less, equal gives boolean</td>
<td></td>
</tr>
</tbody>
</table>
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

```python
p = np.arange(-5, 5, 0.01)
x, y = np.meshgrid(p, p)
x
```

```python
## array([[ 5.        ,  4.99777778, ...,  4.97038462,  4.98031467,  4.99024472],
##        [ 5.        ,  4.99777778, ...,  4.97038462,  4.98031467,  4.99024472],
##        [ 5.        ,  4.99777778, ...,  4.97038462,  4.98031467,  4.99024472],
##        ..., 
##        [ 5.        ,  4.99777778, ...,  4.97038462,  4.98031467,  4.99024472],
##        [ 5.        ,  4.99777778, ...,  4.97038462,  4.98031467,  4.99024472],
##        [ 5.        ,  4.99777778, ...,  4.97038462,  4.98031467,  4.99024472]])
```
Evaluate the function $f(x, y) = \sqrt{x^2 + y^2}$ on a 10 x 10 grid.

```python
import matplotlib.pyplot as plt
val = np.sqrt(x**2 + y**2)
plt.figure(figsize=(2, 2))
plt.imshow(val, cmap="hot")
plt.colorbar()
```
Evaluate the function $f(x, y) = \sqrt{x^2 + y^2}$ on a 10 x 10 grid.

```python
plt.show()
```
### Conditional logic

np.

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

#### Conditional logic

```python
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
```

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

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

```python
# array([4, 9, 8, 3, 9, 3])
```
Conditional logic, examples

```python
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]])
```
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()

## 4.444444444444445

arr3.sum()

## 40

arr3.argmin()

## 7
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>Sum of all array elements</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of all array elements</td>
</tr>
<tr>
<td>std, var</td>
<td>Standard deviation, variance</td>
</tr>
<tr>
<td>min, max</td>
<td>Minimum and Maximum value in array</td>
</tr>
<tr>
<td>argmin, argmax</td>
<td>Indices of Minimum and Maximum value</td>
</tr>
</tbody>
</table>
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 (\texttt{axis=0}), the second one running horizontally across the columns (\texttt{axis=1}).

```python
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])
```
**array.sort(axis):** Sorts array by an axis.

### Sorting one-dimensional arrays

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

Linear algebra
**Inverse matrix**

**Import numpy.linalg**

```python
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**

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

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

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

```python
## True
```
nplin.det(array): Computes the determinant.
np.trace(array): Computes the trace.
np.diag(array): Returns the diagonal elements as an array.

```
Linear algebra functions

nplin.det(arr3)

## -1.0

np.trace(arr3)

## 13

np.diag(arr3)

## array([0, 4, 9])
```
nplin.eig(array): Returns the array of eigenvalues and the array of eigenvectors as a list.

Get eigenvalues and eigenvectors

```python
A = np.array([[3, -1, 0], [2, 0, 0], [-2, 2, -1]])
eigenval, eigenvec = nplin.eig(A)
eigenval
## array([-1., 1., 2.])
eigenvec
## array([[ 0. , -0.40824829, -0.70710678],
##        [ 0. , -0.81649658, -0.70710678],
##        [ 1. , -0.40824829,  0. ]])
```
### Check eigenvalues and eigenvectors

**eigenval** * **eigenvec**

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

```python
np.dot(A, eigenvvec)
```

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

\[
\begin{pmatrix}
3 & -1 & 0 \\
2 & 0 & 0 \\
-2 & 2 & -1
\end{pmatrix} \cdot \begin{pmatrix}
0 \\
0 \\
1
\end{pmatrix} = (-1) \cdot \begin{pmatrix}
0 \\
0 \\
1
\end{pmatrix} = \begin{pmatrix}
0 \\
0 \\
-1
\end{pmatrix}
\]
nplin.qr(array): Conducts a QR decomposition and returns Q and R as lists.

```
Q, R = nplin.qr(arr3)
```

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

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

```python
np.allclose(arr3, np.dot(Q, R))
## True
```
nplin.solve(A, b): Returns the solution of the linearsystem $Ax = b$.

Solve linearsystems

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

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

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

```python
## True
```

\[
\begin{align*}
3x_1 - 1x_2 + 0x_3 & = 7 \\
2x_1 - 0x_2 + 0x_3 & = 4 \\
-2x_1 + 2x_2 - 1x_3 & = 8
\end{align*}
\]

\[
\begin{pmatrix}
3 & -1 & 0 \\
2 & 0 & 0 \\
-2 & 2 & -1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
7 \\
4 \\
8
\end{pmatrix}
\]

\[
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
2 \\
-1 \\
-14
\end{pmatrix}
\]
Overview: Linear algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>np.dot</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>np.trace</td>
<td>Sum of the diagonal elements</td>
</tr>
<tr>
<td>np.diag</td>
<td>Diagonal elements as an array</td>
</tr>
<tr>
<td>nplin.det</td>
<td>Matrix determinant</td>
</tr>
<tr>
<td>nplin.eig</td>
<td>Eigenvalues and eigenvectors</td>
</tr>
<tr>
<td>nplin.inv</td>
<td>Inverse matrix</td>
</tr>
<tr>
<td>nplin.qr</td>
<td>QR decomposition</td>
</tr>
<tr>
<td>nplin.solve</td>
<td>Solve linearsystem</td>
</tr>
</tbody>
</table>
Chapter 3

Data formats and handling

3.1 Pandas package
3.2 Series
3.3 DataFrame
3.4 Import/Export data
Data formats and handling

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

```python
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")
```
Data formats and handling

Series
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.

<table>
<thead>
<tr>
<th>#</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>#</td>
<td>2</td>
<td>-8</td>
</tr>
<tr>
<td>#</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>#</td>
<td>4</td>
<td>26</td>
</tr>
</tbody>
</table>

# dtype: int64
pd.Series(): Creates one-dimensional array-like object including values and an index.

Importing Pandas and creating a Series

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

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

obj
```

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

- Simple Series formed only from a list,
- An index is added automatically.
### Series indexing vs. Numpy indexing

```python
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
```

```python
obj2["b"]
```
```
## -5
```

```python
npobj[1]
```
```
## -5
```

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

- Lists,
- NumPy arrays,
- Dicts.

Series creation from Numpy arrays

```python
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
```
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.
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.
### Series properties

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

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

```python
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.
**Series manipulation**

```
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.
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.
## Series manipulation

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

```
## Hamburg  1900000.0
## Göttingen 117665.0
## Berlin  3574830.0
## Hannover  532163.0
## dtype: float64
```

```py
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.
Detect missing data

\[
\text{pd.} \texttt{isnull}() \texttt{: True} \text{ if data is missing.}
\]

\[
\text{pd.} \texttt{notnull}() \texttt{: False} \text{ if data is missing.}
\]

### NaN

\[
\text{pd.} \texttt{isnull(obj4)}
\]

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

\[
\text{pd.} \texttt{notnull(obj4)}
\]

```python
## Hamburg True
## Göttingen True
## Berlin True
## Hannover True
## dtype: bool
```
Align differently indexed data

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

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj3</td>
<td>obj4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td># Gottingen</td>
<td>117665</td>
</tr>
<tr>
<td># Northeim</td>
<td>28920</td>
</tr>
<tr>
<td># Hannover</td>
<td>532163</td>
</tr>
<tr>
<td># Berlin</td>
<td>3574830</td>
</tr>
<tr>
<td># dtype: int64</td>
<td># dtype: float64</td>
</tr>
</tbody>
</table>

Align data:

```python
obj3 + obj4
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># Berlin</td>
<td>7174830.0</td>
</tr>
<tr>
<td># Gottingen</td>
<td>235330.0</td>
</tr>
<tr>
<td># Hamburg</td>
<td>NaN</td>
</tr>
<tr>
<td># Hannover</td>
<td>1632163.0</td>
</tr>
<tr>
<td># Northeim</td>
<td>NaN</td>
</tr>
<tr>
<td># dtype: float64</td>
<td># dtype: float64</td>
</tr>
</tbody>
</table>

© 2019 PyEcon.org
Naming Series

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

### Naming

```python
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.
- 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.
Data formats and handling

DataFrame
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

<table>
<thead>
<tr>
<th>#</th>
<th>company</th>
<th>price</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Daimler</td>
<td>69.20</td>
<td>4456290</td>
</tr>
<tr>
<td>1</td>
<td>E.ON</td>
<td>8.11</td>
<td>3667975</td>
</tr>
<tr>
<td>2</td>
<td>Siemens</td>
<td>110.92</td>
<td>3669487</td>
</tr>
<tr>
<td>3</td>
<td>BASF</td>
<td>87.28</td>
<td>1778058</td>
</tr>
<tr>
<td>4</td>
<td>BMW</td>
<td>87.81</td>
<td>1824582</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th></th>
<th>company</th>
<th>price</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Daimler</td>
<td>69.20</td>
<td>4456290</td>
</tr>
<tr>
<td>1</td>
<td>E.ON</td>
<td>8.11</td>
<td>3667975</td>
</tr>
<tr>
<td>2</td>
<td>Siemens</td>
<td>110.92</td>
<td>3669487</td>
</tr>
<tr>
<td>3</td>
<td>BASF</td>
<td>87.28</td>
<td>1778058</td>
</tr>
<tr>
<td>4</td>
<td>BMW</td>
<td>87.81</td>
<td>1824582</td>
</tr>
</tbody>
</table>

- 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.
Print DataFrame

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

<table>
<thead>
<tr>
<th></th>
<th>company</th>
<th>volume</th>
<th>price</th>
<th>change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Daimler</td>
<td>4456290</td>
<td>69.20</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>E.ON</td>
<td>3667975</td>
<td>8.11</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>Siemens</td>
<td>3669487</td>
<td>110.92</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>BASF</td>
<td>1778058</td>
<td>87.28</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>BMW</td>
<td>1824582</td>
<td>87.81</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- 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.
## Inputs to DataFrame constructor

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D NumPy arrays</td>
<td>A matrix of data</td>
</tr>
<tr>
<td>dict of arrays, lists, or tuples</td>
<td>Each sequence becomes a column</td>
</tr>
<tr>
<td>dict of Series</td>
<td>Each value becomes a column</td>
</tr>
<tr>
<td>dict of dicts</td>
<td>Each inner dict becomes a column</td>
</tr>
<tr>
<td>List of dicts or Series</td>
<td>Each item becomes a row</td>
</tr>
<tr>
<td>List of lists or tuples</td>
<td>Treated as the 2D NumPy arrays</td>
</tr>
<tr>
<td>Another DataFrame</td>
<td>Same indexes</td>
</tr>
</tbody>
</table>
### Add data to DataFrame

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

<table>
<thead>
<tr>
<th>#</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.20</td>
</tr>
<tr>
<td>1</td>
<td>-3.20</td>
</tr>
<tr>
<td>2</td>
<td>0.40</td>
</tr>
<tr>
<td>3</td>
<td>-0.12</td>
</tr>
<tr>
<td>4</td>
<td>2.40</td>
</tr>
</tbody>
</table>

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.
Indexing DataFrames

```python
frame2[['company', 'change']]
```

<table>
<thead>
<tr>
<th></th>
<th>company</th>
<th>change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Daimler</td>
<td>1.20</td>
</tr>
<tr>
<td>1</td>
<td>E.ON</td>
<td>-3.20</td>
</tr>
<tr>
<td>2</td>
<td>Siemens</td>
<td>0.40</td>
</tr>
<tr>
<td>3</td>
<td>BASF</td>
<td>-0.12</td>
</tr>
<tr>
<td>4</td>
<td>BMW</td>
<td>2.40</td>
</tr>
</tbody>
</table>

- Using a list of multiple columns while indexing, the result is a DataFrame,
- The returned DataFrame has the same index as the initial one.
\texttt{del DataFrame[column]}: Deletes column from DataFrame.

---

### DataFrame delete column

\begin{verbatim}
\texttt{del frame2["volume"]}
\end{verbatim}

```
frame2
```

```python
## 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')
```
Naming DataFrames

<table>
<thead>
<tr>
<th>feature: company</th>
<th>price</th>
<th>change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daimler</td>
<td>69.20</td>
<td>1.20</td>
</tr>
<tr>
<td>E.ON</td>
<td>8.11</td>
<td>-3.20</td>
</tr>
<tr>
<td>Siemens</td>
<td>110.92</td>
<td>0.40</td>
</tr>
<tr>
<td>BASF</td>
<td>87.28</td>
<td>-0.12</td>
</tr>
<tr>
<td>BMW</td>
<td>87.81</td>
<td>2.40</td>
</tr>
</tbody>
</table>

In DataFrames there is no default name for the index or the columns.
DataFrame.reindex(): Creates new DataFrame with data conformed to a new index, while the initial DataFrame will not be changed.

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

<table>
<thead>
<tr>
<th>#</th>
<th>company</th>
<th>price</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Daimler</td>
<td>69.20</td>
<td>4456290</td>
</tr>
<tr>
<td>2</td>
<td>Siemens</td>
<td>110.92</td>
<td>3669487</td>
</tr>
<tr>
<td>3</td>
<td>BASF</td>
<td>87.28</td>
<td>1778058</td>
</tr>
<tr>
<td>4</td>
<td>BMW</td>
<td>87.81</td>
<td>1824582</td>
</tr>
</tbody>
</table>

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

Filling missing values

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

```plaintext
## 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        0.00     0
```

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

```plaintext
## company   price  market cap
## 0  Daimler  69.20   NaN
## 2  Siemens  110.92  NaN
## 3  BASF     87.28   NaN
## 4  BMW      87.81   NaN
```
DataFrame.fillna(value): Fills NaNs with value.

Filling NaN

```py
frame4[:3]
```

```none
##    company  price  market  cap
## 0  Daimler    69.2  NaN    NaN
## 2  Siemens   110.9  NaN    NaN
## 3    BASF    87.2  NaN    NaN
```

```py
frame4.fillna(1000000, inplace=True)
```

```py
frame4[:3]
```

```none
##    company  price  market  cap
## 0  Daimler    69.2  1000000.0
## 2  Siemens   110.9  1000000.0
## 3    BASF    87.2  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.
DataFrame\texttt{.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
```
Dropping entries

Dropping column

```python
frame5[:2]
```

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

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

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

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

```python
import pandas as pd

data = {'company': ['Daimler', 'E.ON', 'Siemens', 'BASF', 'BMW'],
        'price': [69.20, 8.11, 110.92, 87.28, 87.81],
        'volume': [4456290, 3667975, 3669487, 1778058, 1824582]}

frame = pd.DataFrame(data)

# Indexing

frame
```

<table>
<thead>
<tr>
<th>company</th>
<th>price</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daimler</td>
<td>69.20</td>
<td>4456290</td>
</tr>
<tr>
<td>E.ON</td>
<td>8.11</td>
<td>3667975</td>
</tr>
<tr>
<td>Siemens</td>
<td>110.92</td>
<td>3669487</td>
</tr>
<tr>
<td>BASF</td>
<td>87.28</td>
<td>1778058</td>
</tr>
<tr>
<td>BMW</td>
<td>87.81</td>
<td>1824582</td>
</tr>
</tbody>
</table>

```python
frame[2:]
```

<table>
<thead>
<tr>
<th>company</th>
<th>price</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Siemens</td>
<td>110.92</td>
<td>3669487</td>
</tr>
<tr>
<td>BASF</td>
<td>87.28</td>
<td>1778058</td>
</tr>
<tr>
<td>BMW</td>
<td>87.81</td>
<td>1824582</td>
</tr>
</tbody>
</table>
Indexing, selecting and filtering

```python
frame6 = pd.DataFrame(data, index=["a", "b", "c", "d", "e"], columns=["company", "price", "volume"])
print(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.
Data\texttt{Frame}.\texttt{loc}(): Selects a subset of rows and columns from a DataFrame using axis labels.

Data\texttt{Frame}.\texttt{iloc}(): Selects a subset of rows and columns from a DataFrame using integers.

### Selection with \texttt{loc} and \texttt{iloc}

\texttt{frame6.loc["c", ["company", "price"]]}  

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

\texttt{frame6.iloc[2, [0, 1]]}  

```python  
## company Siemens  
## price 110.92  
## Name: c, dtype: object  
```
Indexing, selecting and filtering

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.
## DataFrame indexing options

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

© 2019 PyEcon.org
Hierarchical indexing enables you to have multiple index levels.

```
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(levels=[['a', 'b'], [1, 2, 3]],
             labels=[[0, 0, 0, 1, 1], [0, 1, 2, 0, 1]],
             names=['index1', 'index2'])
```
## Selecting of a multiindex

```python
frame6.loc["a"]
```

### index2

<table>
<thead>
<tr>
<th></th>
<th>first</th>
<th>second</th>
<th>third</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

```python
frame6.loc["b", 1]
```

### Name: (b, 1), dtype: int64

<table>
<thead>
<tr>
<th></th>
<th>first</th>
<th>second</th>
<th>third</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Operations between DataFrame and Series

```python
frame7 = frame[['price', 'volume']]
frame7.index = ['Daimler', 'E.ON', 'Siemens', 'BASF', 'BMW']
series = frame7.iloc[2]
frame7
```

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daimler</td>
<td>69.20</td>
<td>4456290</td>
</tr>
<tr>
<td>E.ON</td>
<td>8.11</td>
<td>3667975</td>
</tr>
<tr>
<td>Siemens</td>
<td>110.92</td>
<td>3669487</td>
</tr>
<tr>
<td>BASF</td>
<td>87.28</td>
<td>1778058</td>
</tr>
<tr>
<td>BMW</td>
<td>87.81</td>
<td>1824582</td>
</tr>
</tbody>
</table>

```python
series
```

```python
## price     110.92
## volume    3669487.00
## Name: Siemens, dtype: float64
```

Here the Series was generated from the first row of the DataFrame.
Operations between Series and DataFrames down the rows

frame7 + series

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daimler</td>
<td>180.12</td>
<td>8125777.0</td>
</tr>
<tr>
<td>E.ON</td>
<td>119.03</td>
<td>7337462.0</td>
</tr>
<tr>
<td>Siemens</td>
<td>221.84</td>
<td>7338974.0</td>
</tr>
<tr>
<td>BASF</td>
<td>198.20</td>
<td>5447545.0</td>
</tr>
<tr>
<td>BMW</td>
<td>198.73</td>
<td>5494069.0</td>
</tr>
</tbody>
</table>

- 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.
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`).
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.
DataFrame.apply(np.function, axis): Applies a NumPy function on the DataFrame axis. See also statistical and mathematical NumPy functions.

### Numpy functions on DataFrames

```python
frame7[:2]
```

```ini
## price  volume
## Daimler 69.20 4456290
## E.ON 8.11 3667975
```

```python
frame7.apply(np.mean)
```

```ini
## price  volume
## Daimler 72.664 3079278.400
## E.ON 8.11 3667975
```

```
frame7.apply(np.sqrt)[:2]
```

```ini
## price  volume
## Daimler 8.318654 2110.992657
## E.ON 2.847806 1915.195812
```
**Grouping DataFrames**

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.

```python
vote = pd.read_csv("data/vote.csv")[["Party", "Member", "Vote"]]
vote.head()
```

<table>
<thead>
<tr>
<th></th>
<th>Party</th>
<th>Member</th>
<th>Vote</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CDU/CSU</td>
<td>Abercron</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>CDU/CSU</td>
<td>Albani</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>CDU/CSU</td>
<td>Altenkamp</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>CDU/CSU</td>
<td>Altmaier</td>
<td>absent</td>
</tr>
<tr>
<td>4</td>
<td>CDU/CSU</td>
<td>Amthor</td>
<td>yes</td>
</tr>
</tbody>
</table>

Adding the functions `count()` or `mean()` to `groupby()` returns the sum or the mean of the grouped columns.
## Groupby

```python
res = vote.groupby(["Party", "Vote"]).count()
res
```

<table>
<thead>
<tr>
<th>Party</th>
<th>Vote</th>
<th>Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>AfD</td>
<td>absent</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>86</td>
</tr>
<tr>
<td>BÜ90/GR</td>
<td>absent</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>58</td>
</tr>
<tr>
<td>CDU/CSU</td>
<td>absent</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>239</td>
</tr>
<tr>
<td>DIE LINKE.</td>
<td>absent</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>62</td>
</tr>
<tr>
<td>FDP</td>
<td>absent</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>75</td>
</tr>
<tr>
<td>Fraktionslos</td>
<td>absent</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>1</td>
</tr>
<tr>
<td>SPD</td>
<td>absent</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>yes</td>
<td>147</td>
</tr>
</tbody>
</table>
Data formats and handling

Import/Export data
Reading data in text format

**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
```
tab.txt

```
a| b| c| d| hello
1| 2| 3| 4| world
5| 6| 7| 8| python
2| 3| 5| 7| pandas
```

\[ \text{pd.read_table}("file", \text{sep}) \]: Reads table with any separators into DataFrame.

```
\begin{verbatim}
\textbf{Read table values}

\textbf{df} = \textbf{pd.read_table}("data/tab.txt", \text{sep}"|")
\textbf{df}
\end{verbatim}
```

```
## a  b  c  d hello
## 0  1  2  3  4 world
## 1  5  6  7  8 python
## 2  2  3  5  7 pandas
```
Reading data in text format

**CSV file without header row:**

```python
ex2.csv

1, 2, 3, 4, world
5, 6, 7, 8, python
2, 3, 5, 7, pandas

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
```
Reading data in text format

```
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
```
ex2.csv

1, 2, 3, 4, world
5, 6, 7, 8, python
2, 3, 5, 7, pandas

Use `hello`-column as the index:

```python
import pandas as pd

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
```
Reading data in text format

```
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
```
DataFrame.to_csv("filename"): Writes DataFrame to CSV.

Write to CSV

```python
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).
Writing data to text file

```
Write to CSV and settings

```

```
def = 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
```
Writing data to text file

```python
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
```
Reading Excel files

pd.read_excel("file.xls"): Reads .xls files.

<table>
<thead>
<tr>
<th>Date</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Adj Close</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018-01-31</td>
<td>1170.569946</td>
<td>1173</td>
<td>1159.130005</td>
<td>1169.939941</td>
<td>1169.939941</td>
<td>1538700</td>
</tr>
<tr>
<td>2018-02-01</td>
<td>1162.609985</td>
<td>1174</td>
<td>1157.52002</td>
<td>1167.699951</td>
<td>1167.699951</td>
<td>2412100</td>
</tr>
<tr>
<td>2018-02-02</td>
<td>1122</td>
<td>1123.069946</td>
<td>1107.277954</td>
<td>1111.900024</td>
<td>1111.900024</td>
<td>4857900</td>
</tr>
<tr>
<td>2018-02-05</td>
<td>1090.599976</td>
<td>1110</td>
<td>1052.030029</td>
<td>1055.800049</td>
<td>1055.800049</td>
<td>3798300</td>
</tr>
<tr>
<td>2018-02-06</td>
<td>1027.180054</td>
<td>1081.709961</td>
<td>1023.137024</td>
<td>1080.599976</td>
<td>1080.599976</td>
<td>3448000</td>
</tr>
</tbody>
</table>
| 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")
Reading Excel files

### Excel as a DataFrame

```python
xls_frame[["Adj Close", "Volume", "High"]
```

<table>
<thead>
<tr>
<th></th>
<th>Adj Close</th>
<th>Volume</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1169.939941</td>
<td>1538700</td>
<td>1173.000000</td>
</tr>
<tr>
<td>1</td>
<td>1167.699951</td>
<td>2412100</td>
<td>1174.000000</td>
</tr>
<tr>
<td>2</td>
<td>1111.900024</td>
<td>4857900</td>
<td>1123.069946</td>
</tr>
<tr>
<td>3</td>
<td>1055.800049</td>
<td>3798300</td>
<td>1110.000000</td>
</tr>
<tr>
<td>4</td>
<td>1080.599976</td>
<td>3448000</td>
<td>1081.709961</td>
</tr>
<tr>
<td>5</td>
<td>1048.579956</td>
<td>2341700</td>
<td>1081.780029</td>
</tr>
</tbody>
</table>
Remote data access

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:

```
import pandas-datareader
```

```
from pandas_datareader import data
```
data.DataReader("symbol", "source", "start", "end"): Returns financial data of a stock in a certain time period.

### IEX get data

```python
ford = data.DataReader("F", "iex", "2017-01-01", "2018-01-31")
ford.head()[["close", "volume"]]
```

<table>
<thead>
<tr>
<th>date</th>
<th>close</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-01-03</td>
<td>10.7619</td>
<td>40510821</td>
</tr>
<tr>
<td>2017-01-04</td>
<td>11.2577</td>
<td>77638075</td>
</tr>
<tr>
<td>2017-01-05</td>
<td>10.9158</td>
<td>75628443</td>
</tr>
<tr>
<td>2017-01-06</td>
<td>10.9072</td>
<td>40315887</td>
</tr>
<tr>
<td>2017-01-09</td>
<td>10.7961</td>
<td>39438393</td>
</tr>
</tbody>
</table>
**IEX handle data**

```python
ford.index

## Index(['2017-01-03', '2017-01-04',...]
## dtype='object', name='date',...)

ford.loc['2018-01-26']

## open 1.046130e+01
## high 1.056060e+01
## low 1.038010e+01
## close 1.051550e+01
## volume 5.249600e+07
## Name: 2018-01-26, dtype: float64
```

**DataFrame index**

Index of the DataFrame is different at different sources. Always check `DataFrame.index`!
Data access: IEX

```
sap = data.DataReader("SAP", "iex", "2017-01-01", "2018-01-31")
sap[25:27]
```

```
## open    high    low    close    volume
## date
## 2017-02-08 89.5382 90.0263 89.4405 89.6065 653804
## 2017-02-09 89.7139 89.9738 89.5284 89.5284 548787
```

```
sap.loc["2017-02-08"]
```

```
## open    89.5382
## high    90.0263
## low     89.4405
## close   89.6065
## volume  653804.0000
## Name: 2017-02-08, dtype: float64
```
Data access: Eurostat

```python
population = data.DataReader("tps00001", "eurostat", "2007-01-01", "2018-01-01")

population.columns

## MultiIndex(levels=[[Population on 1 January - total], [Albania, Andorra, Armenia, Austria, Azerbaijan, Belarus, Belgium, ...])

population["Population on 1 January - total", "France"][0:5]

## FREQ Annual
## TIME_PERIOD
## 2007-01-01 63645065.0
## 2008-01-01 64007193.0
## 2009-01-01 64350226.0
## 2010-01-01 64658856.0
## 2011-01-01 64978721.0
```

Eurostat Database

© 2019 PyEcon.org
Website used for the example:

```python
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.
Bollinger

```python
sap = data.DataReader("SAP", "iex", "2017-01-01", "2018-08-31")
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()
anx = fig.add_subplot(1, 1, 1)
boll.plot(ax=axn, 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")
```
Motivation

Essential concepts
- Getting started
- Procedural programming
- Object-orientation

Numerical programming
- NumPy package
- Array basics
- Linear algebra

Data formats and handling
- Pandas package
- Series
- DataFrame

Import/Export data

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

Applications
- Time series
- Moving window
- Financial applications

© 2019 PyEcon.org
Visual illustrations

4.1 Matplotlib package
4.2 Figures and subplots
4.3 Plot types and styles
4.4 Pandas layers
Visual illustrations

Matplotlib package
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.
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
Simple plot

plt.plot(array): Plots the values of a list, the X-axis has by default the range \([0, 1, \ldots, n-1]\).

Import matplotlib and simple example

```python
import matplotlib.pyplot as plt
import numpy as np
plt.plot(np.arange(10))
plt.savefig("out/list.pdf")
```

0 2 4 6 8
0 2 4 6 8
Visual illustrations

Figures and subplots
Plots in `matplotlib` reside in a *Figure object*:

`plt.figure(...)`: Creates new Figure object allowing for multiple parameters.

`plt.gca()`: Returns the reference of the active figure.

### Create Figures

```python
fig = plt.figure(figsize=(16, 8))
print(plt.gcf())
```

```python
## 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.
plt.savefig("filename"): Saves active figure to file. Available file formats are among others:

<table>
<thead>
<tr>
<th>Filename extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.png</td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>.pdf</td>
<td>Portable Document Format</td>
</tr>
<tr>
<td>.svg</td>
<td>Scalable Vector Graphics</td>
</tr>
<tr>
<td>.jpeg</td>
<td>JPEG File Interchange Format</td>
</tr>
<tr>
<td>.jpg</td>
<td>JPEG File Interchange Format</td>
</tr>
<tr>
<td>.ps</td>
<td>PostScript</td>
</tr>
<tr>
<td>.raw</td>
<td>Raw Image Format</td>
</tr>
</tbody>
</table>
Subplots

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.
Filling subplots with content

```python
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.
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

© 2019 PyEcon.org
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

```python
def 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")
```

Standard creation of plots
Visual illustrations

Plot types and styles
Plot types

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.

A vast number of plot types can be found in the examples gallery.
Plot types
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.

```python
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")
```
Adjusting the spacing around subplots
Colors, markers and line styles

\[ \text{ax.plot(data, linestyle, color, marker): Sets data and styles of subplot ax.} \]

**Styles**

\[
\text{fig, ax = plt.subplots(1, figsize=(15, 6))}
\]
\[
\text{ax.plot(randn(10), linestyle="--", color="darkcyan", marker="p")}
\]
\[
\text{fig.savefig("out/style.pdf")}
\]
Plot line styles

line styles

Plot line styles
## Plot markers

<table>
<thead>
<tr>
<th>Marker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>point</td>
</tr>
<tr>
<td>,</td>
<td>pixel</td>
</tr>
<tr>
<td>o</td>
<td>circle</td>
</tr>
<tr>
<td>v</td>
<td>triangle_down</td>
</tr>
<tr>
<td>8</td>
<td>octagon</td>
</tr>
<tr>
<td>s</td>
<td>square</td>
</tr>
<tr>
<td>p</td>
<td>pentagon</td>
</tr>
<tr>
<td>P</td>
<td>plus (filled)</td>
</tr>
<tr>
<td>*</td>
<td>star</td>
</tr>
<tr>
<td>h</td>
<td>hexagon1</td>
</tr>
<tr>
<td>H</td>
<td>hexagon2</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td>x (filled)</td>
</tr>
<tr>
<td>D</td>
<td>diamond</td>
</tr>
</tbody>
</table>

© 2019 PyEcon.org
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

```python
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.
Ticks and labels

Essential concepts
   Getting started
   Procedural programming
   Object-orientation

Numerical programming
   NumPy package
   Array basics
   Linear algebra

Data formats and handling
   Pandas package
   Series
   DataFrame
   Import/Export data

Visual illustrations
   Matplotlib package
   Figures and subplots

Plot types and styles

Applications
   Time series
   Moving window
   Financial applications

© 2019 PyEcon.org
Ticks and labels

Set ticks and labels

```python
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`.

© 2019 PyEcon.org
Using multiple plots in one subplot one needs a legend.

```python
ax.legend(loc): Shows the legend at location loc.
```

Some options: "best", "upper right", "center left", ...

```python
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 placed in a corner where is does not interfere 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

© 2019 PyEcon.org
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.
Annotation Lehman

```python
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")
```
Dow Jones Industrial Average

Lehman Bankruptcy
plt.Rectangle((x, y), width, height, angle): Creates a rectangle
plt.Circle((x, y), radius): Creates a circle.

```python
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](https://matplotlib.org/api/patches_api.html)
Step 1
Create a Figure object and subplots

Best practice Step 1

```python
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

```python
x = np.arange(0, 10, 0.1)
y = np.sin(x)
ax.scatter(x, y)
```
Best practice: Visual illustrations

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)
```
Sine wave
Best practice: Visual illustrations

Step 5
Set labels

Best practice Step 5

```python
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

```python
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

```python
fig.savefig("out/sinewave.pdf")
```
Visual illustrations

Pandas layers
DataFrame/Series.plot(): Plots a DataFrame or a Series.

Simple line plot

```python
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

© 2019 PyEcon.org
Line plots

def = pd.DataFrame(np.random.randn(10, 3), index=np.arange(10), columns=['a', 'b', 'c'])

df

# Line plots

df.plot(figsize=(15, 12))
plt.savefig("out/line2.pdf")
Line plots
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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>&quot;line&quot;, &quot;bar&quot;, etc</td>
</tr>
<tr>
<td>logy</td>
<td>logarithmic scale on Y-axis</td>
</tr>
<tr>
<td>use_index</td>
<td>If True, use index for tick labels</td>
</tr>
<tr>
<td>rot</td>
<td>Rotation of tick labels</td>
</tr>
<tr>
<td>xticks</td>
<td>Values for x ticks</td>
</tr>
<tr>
<td>yticks</td>
<td>Values for y ticks</td>
</tr>
<tr>
<td>grid</td>
<td>Set grid True or False</td>
</tr>
<tr>
<td>xlim</td>
<td>X-axis limits</td>
</tr>
<tr>
<td>ylim</td>
<td>Y-axis limits</td>
</tr>
<tr>
<td>subplots</td>
<td>Plot each DataFrame column in a new subplot</td>
</tr>
</tbody>
</table>

Table: Pandas plot arguments
Separated line plots

df.plot(grid=True, rot=45, subplots=True, title="Example", figsize=(15, 10))

plt.savefig("out/pandas.pdf")
Standard creation of plots and pandas

```
dataframe.plot(ax=subplot): Plots a dataframe into subplot.
```

### Standard creation

```python
define = 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
```
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.
Bar plot

Canteen use in Göttingen

- Zentral
- Turm

Bar chart showing canteen use in Göttingen for different days of the week, with Zentral and Turm locations.
### Bar plot - stacked

```python
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")
```
Canteen use in Göttingen

<table>
<thead>
<tr>
<th>Day</th>
<th>Zentral</th>
<th>Turm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>1250</td>
<td>500</td>
</tr>
<tr>
<td>Tue</td>
<td>1000</td>
<td>750</td>
</tr>
<tr>
<td>Wed</td>
<td>1500</td>
<td>750</td>
</tr>
<tr>
<td>Thu</td>
<td>1750</td>
<td>750</td>
</tr>
<tr>
<td>Fri</td>
<td>1500</td>
<td>750</td>
</tr>
<tr>
<td>Sat</td>
<td>750</td>
<td>750</td>
</tr>
</tbody>
</table>

© 2019 PyEcon.org
Plot financial data

BTC chart

```python
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")
```
Plot financial data

BTC-EUR

Date

Price


0 2500 5000 7500 10000 12500 15000

© 2019 PyEcon.org
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

© 2019 PyEcon.org
```python
amazon = amazon/amazon[0] * 100
siemens = siemens/siemens[0] * 100
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(1, 1, 1)
anx.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

Plot financial data
Applications

5.1 Time series
5.2 Moving window
5.3 Financial applications
Applications

▶ Time series
Data types for date and time are included in the Python standard library.

### Datetime creation

```python
from datetime import datetime
now = datetime.now()
```

```python
## datetime.datetime(2019, 4, 28, 16, 26, 48, 256113)
```

```python
now.day
```

```python
## 28
```

```python
now.hour
```

```python
## 16
```

From `datetime` you can get the attributes `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`. 
**Set datetime**

```
datetime(year, month, day, ..., microsecond): Sets date and time.
```

### Datetime representation

```python
datetime(2018, 12, 24, 8, 30)
holiday

## datetime.datetime(2018, 12, 24, 8, 30)
```

```python
datetime(2018, 11, 9, 10)
exam
```

```python
print("The exam will be on the " + ":%Y-%m-%d".format(exam))
```

## The exam will be on the 2018-11-09
**timedelta**(days, seconds, microseconds): Represents difference between two datetime objects.

```python
from datetime import timedelta

delta = exam - now
print(delta)

## datetime.timedelta(-171, 63191, 743887)

print("The exam will take place in " + str(delta.days) + " days.")

## The exam will take place in -171 days.
```

```python
now
now + timedelta(10, 120)
```

## The exam will take place in -171 days.
datetime.strftime("format"): Converts datetime object into string.
datetime.strptime(datestring, "format"): Converts date as a string into a datetime object.

```
Convert Datetime

stamp = datetime(2018, 4, 12)

## datetime.datetime(2018, 4, 12, 0, 0)

print("German date format: " + stamp.strftime("%d.%m.%Y"))

## German date format: 12.04.2018

val = "2018-5-5"

d = datetime.strptime(val, "%Y-%m-%d")

## datetime.datetime(2018, 5, 5, 0, 0)
```
Converting examples

```python
val = "31.01.2012"
d = datetime.strptime(val, "%d.%m.%Y")

## datetime.datetime(2012, 1, 31, 0, 0)

now.strptime("Today is %A and we are in week %W of the year %Y.")

## 'Today is Sunday and we are in week 16 of the year 2019.'

now.strptime("%c")

## 'Sun 28 Apr 2019 04:26:48 PM '
```
## Overview: Datetime formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Y</td>
<td>4-digit year</td>
</tr>
<tr>
<td>%m</td>
<td>2-digit month [01, 12]</td>
</tr>
<tr>
<td>%d</td>
<td>2-digit day [01, 31]</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) [00, 23]</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) [01, 12]</td>
</tr>
<tr>
<td>%M</td>
<td>2-digit minute [00, 59]</td>
</tr>
<tr>
<td>%S</td>
<td>Second [00, 61]</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year [00, 53]</td>
</tr>
<tr>
<td>%F</td>
<td>Shortcut for %Y-%m-%d</td>
</tr>
</tbody>
</table>
### Overview: Datetime formats

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%a</code></td>
<td>Abbreviated weekday name</td>
</tr>
<tr>
<td><code>%A</code></td>
<td>Full weekday name</td>
</tr>
<tr>
<td><code>%b</code></td>
<td>Abbreviated month name</td>
</tr>
<tr>
<td><code>%B</code></td>
<td>Full month name</td>
</tr>
<tr>
<td><code>%c</code></td>
<td>Full date and time</td>
</tr>
<tr>
<td><code>%x</code></td>
<td>Locale-appropriate formatted date</td>
</tr>
</tbody>
</table>
Generating date ranges with pandas

```
pd.date_range(start, end, freq): Generates a date range.
```

### Date ranges

```python
import pandas as pd
index = pd.date_range("2018-01-01", now)
index[0:2]
index[15:16]
index = pd.date_range("2018-01-01", now, freq="M")
index[0:2]
```

```
## DatetimeIndex(["2018-01-01", '2018-01-02'], dtype='datetime64[ns]', freq='D')
## DatetimeIndex(['2018-01-16'], dtype='datetime64[ns]', freq='D')
## DatetimeIndex(['2018-01-31', '2018-02-01'], dtype='datetime64[ns]', freq='M')
```
<table>
<thead>
<tr>
<th>Alias</th>
<th>Offset type</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day</td>
</tr>
<tr>
<td>B</td>
<td>Business day</td>
</tr>
<tr>
<td>H</td>
<td>Hour</td>
</tr>
<tr>
<td>T</td>
<td>Minute</td>
</tr>
<tr>
<td>S</td>
<td>Second</td>
</tr>
<tr>
<td>M</td>
<td>Month end</td>
</tr>
<tr>
<td>BM</td>
<td>Business month end</td>
</tr>
<tr>
<td>Q-JAN, Q-FEB, ...</td>
<td>Quarter end</td>
</tr>
<tr>
<td>A-JAN, A-FEB, ...</td>
<td>Year end</td>
</tr>
<tr>
<td>AS-JAN, AS-FEB, ...</td>
<td>Year begin</td>
</tr>
<tr>
<td>BA-JAN, BA-FEB, ...</td>
<td>Business year end</td>
</tr>
<tr>
<td>BAS-JAN, BAS-FEB, ...</td>
<td>Business year begin</td>
</tr>
</tbody>
</table>
**DataFrame**.resample("frequency"): Resamples time series by a specified frequency.

```python
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
```

```python
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
```
Applications

Moving window
DataFrame.**rolling**(window): Conducts rolling window computations.

### Rolling mean

```python
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()`.
Moving window functions

Amazon price and rolling mean

Price

Date

900
1000
1100
1200
1300
1400
1500

Amazon
Rolling mean

© 2019 PyEcon.org
Moving window functions

Standard deviation

```python
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")
```
Moving window functions

Essential concepts
- Getting started
- Procedural programming
- Object-orientation

Numerical programming
- NumPy package
- Array basics
- Linear algebra

Data formats and handling
- Pandas package
- Series
- DataFrame
- Import/Export data

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

Applications
- Time series
- Moving window
- Financial applications

© 2019 PyEcon.org
Logarithmic standard deviation

```python
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")
```
Moving window functions

Logarithmic standard deviation

Date

10
100
101
102

amazon
pfizer
pg


© 2019 PyEcon.org
DataFrame.ewm(span): Computes exponentially weighted rolling window functions.

```python
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")
```
Exponentially weighted functions

Rolling mean
Exp mean
Amazon price
DataFrame.pct_change(): Computes the percentage changes per period.

```python
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")
```
**DataFrame.rolling().corr(benchmark):** Computes correlation between two time series.

```python
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")
```
Binary moving window functions

20 days correlation

Date
-0.4
-0.2
0.0
0.2
0.4
0.6
0.8

amazon
pfizer
pg
Applications

▶ Financial applications
Cumulative returns

```python
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")
```
Cumulative returns

- Amazon
- Pfizer
- PG

Date

Cumulative returns

- 2017-03
- 2017-05
- 2017-07
- 2017-09
- 2017-11
- 2018-01
- 2018-03
Cumulative returns

Monthly returns

```python
returns_m = ret_index.resample("BM").last().pct_change()
returns_m.head()
```

<table>
<thead>
<tr>
<th>Date</th>
<th>amazon</th>
<th>pfizer</th>
<th>pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-02-28</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-03-31</td>
<td>0.049110</td>
<td>0.002638</td>
<td>-0.013396</td>
</tr>
<tr>
<td>2017-04-28</td>
<td>0.043371</td>
<td>-0.008477</td>
<td>-0.020604</td>
</tr>
<tr>
<td>2017-05-31</td>
<td>0.075276</td>
<td>-0.028124</td>
<td>0.008703</td>
</tr>
<tr>
<td>2017-06-30</td>
<td>-0.026764</td>
<td>0.028790</td>
<td>-0.010671</td>
</tr>
</tbody>
</table>
Volatility

```python
fig = plt.figure(figsize=(16, 8))
x = fig.add_subplot(1, 1, 1)
volatility = returns.rolling(window=20).std() * np.sqrt(20)
volatility.plot(ax=x)
x.set_title("Volatility", fontsize=25)
fig.savefig("out/vola.pdf")
```
Volatility calculation

Essential concepts
- Getting started
- Procedural programming
- Object-orientation

Numerical programming
- NumPy package
- Array basics
- Linear algebra

Data formats and handling
- Pandas package
- Series
- DataFrame
- Import/Export data

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

Applications
- Time series
- Moving window

Financial applications

© 2019 PyEcon.org
DataFrame.describe(): Shows a statistical summary.

<table>
<thead>
<tr>
<th></th>
<th>amazon</th>
<th>pfizer</th>
<th>pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>252.000000</td>
<td>251.000000</td>
<td>252.000000</td>
</tr>
<tr>
<td>mean</td>
<td>1044.521903</td>
<td>33.892665</td>
<td>87.934304</td>
</tr>
<tr>
<td>std</td>
<td>158.041844</td>
<td>1.694680</td>
<td>2.728659</td>
</tr>
<tr>
<td>min</td>
<td>843.200012</td>
<td>30.872143</td>
<td>79.919998</td>
</tr>
<tr>
<td>25%</td>
<td>953.567474</td>
<td>32.593733</td>
<td>86.241475</td>
</tr>
<tr>
<td>50%</td>
<td>988.680023</td>
<td>33.147469</td>
<td>87.863598</td>
</tr>
<tr>
<td>75%</td>
<td>1136.952484</td>
<td>35.331834</td>
<td>90.363035</td>
</tr>
<tr>
<td>max</td>
<td>1485.339966</td>
<td>38.661823</td>
<td>92.988976</td>
</tr>
</tbody>
</table>
```python
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")
```
Return analysis

Essential concepts
- Getting started
- Procedural programming
- Object-orientation

Numerical programming
- NumPy package
- Array basics
- Linear algebra

Data formats and handling
- Pandas package
- Series
- DataFrame
- Import/Export data

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

Applications
- Time series
- Moving window

Financial applications

© 2019 PyEcon.org
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**

```python
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")
```
Ordinary Least Squares
Regression

```python
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()`.

<table>
<thead>
<tr>
<th>Dep. Variable:</th>
<th>y</th>
<th>R-squared:</th>
<th>0.965</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model:</td>
<td>OLS</td>
<td>Adj. R-squared:</td>
<td>0.959</td>
</tr>
<tr>
<td>Method:</td>
<td>Least Squares</td>
<td>F-statistic:</td>
<td>190.2</td>
</tr>
<tr>
<td>Date:</td>
<td>Mo, 19 Mär 2018</td>
<td>Prob (F-statistic):</td>
<td>2.49e-06</td>
</tr>
<tr>
<td>Time:</td>
<td>15:21:30</td>
<td>Log-Likelihood:</td>
<td>-29.706</td>
</tr>
<tr>
<td>No. Observations:</td>
<td>9</td>
<td>AIC:</td>
<td>63.41</td>
</tr>
<tr>
<td>Df Residuals:</td>
<td>7</td>
<td>BIC:</td>
<td>63.81</td>
</tr>
<tr>
<td>Df Model:</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Covariance Type:</td>
<td>nonrobust</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|            | 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   |

<table>
<thead>
<tr>
<th>Omnibus:</th>
<th>0.788</th>
<th>Durbin-Watson:</th>
<th>1.627</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prob(Omnibus):</td>
<td>0.674</td>
<td>Jarque-Bera (JB):</td>
<td>0.117</td>
</tr>
<tr>
<td>Skew:</td>
<td>-0.268</td>
<td>Prob(JB):</td>
<td>0.943</td>
</tr>
<tr>
<td>Kurtosis:</td>
<td>2.841</td>
<td>Cond. No.</td>
<td>9.06</td>
</tr>
</tbody>
</table>
The Newton-Raphson method is an algorithm for finding successively better approximations to the roots of real-valued functions.

Let $F : \mathbb{R}^k \to \mathbb{R}^k$ 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:

$$x_{n+1} := x_n - (J(x_n)^{-1}F(x_n))$$

with an initial guess $x_0$.

For $f : \mathbb{R} \to \mathbb{R}$ 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 $f' = df/dx$. 
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 \( x_0 = -1 \).
The first step involves the definition of the function $f(x)$ and its derivation $f'(x)$ in Python:

```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 $x_0, x_1, \ldots, x_N$. 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.
Newton-Raphson implementation

```python
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
```
Newton-Raphson implementation

Essential concepts
Getting started
Procedural programming
Object-orientation

Numerical programming
NumPy package
Array basics
Linear algebra

Data formats and handling
Pandas package
Series
DataFrame
Import/Export data

Visual illustrations
Matplotlib package
Figures and subplots
Plot types and styles
Pandas layers

Applications
Time series
Moving window

Financial applications

© 2019 PyEcon.org
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:

```python
# 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
The End... but not finally