

Carsten Knoll  
Institut für Regelungs- und Steuerungstheorie, TU Dresden

# Python für Ingenieur:innen – Kompaktworkshop

Chemnitzer Linux-Tage, 2022-03-13

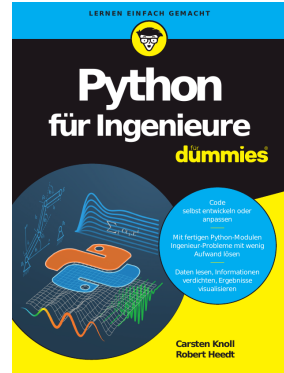
# Vorstellung

- PostDoc (Regelungstheorie, Wissensrepräsentation)
- Seit 2010: <https://tu-dresden.de/pythonkurs>
- Seit 2014: <https://fsfw-dresden.de>  
(Freie Software Freies Wissen)
- Seit 2019: <https://dresden.bits-und-baeume.org>  
(Digitalisierung und Nachhaltigkeit)



# Vorstellung

- PostDoc (Regelungstheorie, Wissensrepräsentation)
- Seit 2010: <https://tu-dresden.de/pythonkurs>
- Seit 2014: <https://fsfw-dresden.de>  
(Freie Software Freies Wissen)
- Seit 2019: <https://dresden.bits-und-baeume.org>  
(Digitalisierung und Nachhaltigkeit)
- 2021: <https://python-fuer-ingenieure.de>



# Motivation

Menschheit: schon viele Probleme gelöst ...  
... aber es bleiben noch viele Herausforderungen

# Motivation

Menschheit: schon viele Probleme gelöst ...  
... aber es bleiben noch viele Herausforderungen

**Organisation der Gesellschaft** (Konfliktreduktion)

**Wissenschaftlich/technischer Fortschritt** (objektive Probleme lösen)

# Motivation

Menschheit: schon viele Probleme gelöst ...  
... aber es bleiben noch viele Herausforderungen

**Organisation der Gesellschaft** (Konfliktreduktion)

**Wissenschaftlich/technischer Fortschritt** (objektive Probleme lösen)

Überzeugung:

→ *Freie Software* und speziell Python hilfreich in beiden Problemfeldern

# Motivation

Menschheit: schon viele Probleme gelöst ...  
... aber es bleiben noch viele Herausforderungen

## Organisation der Gesellschaft (Konfliktreduktion)

- `moodpoll`, `sober-arguments.net`, ...

## Wissenschaftlich/technischer Fortschritt (objektive Probleme lösen)

- `numpy`, `scipy`, `pandas`, ...

Überzeugung:

→ *Freie Software* und speziell Python hilfreich in beiden Problemfeldern

# Motivation

Menschheit: schon viele Probleme gelöst ...  
... aber es bleiben noch viele Herausforderungen

## Organisation der Gesellschaft (Konfliktreduktion)

- `moodpoll`, `sober-arguments.net`, ...

## Wissenschaftlich/technischer Fortschritt (objektive Probleme lösen)

- `numpy`, `scipy`, `pandas`, ...

Überzeugung:

- *Freie Software* und speziell Python hilfreich in beiden Problemfeldern
- Ethische Rahmung wichtig!    Vorschlag: „Gutes Leben für alle!“



# Warum Python? (1)

Python als Programmiersprache

- Klare, lesbare Syntax (wenig „Ballast“)
- Paradigmen: prozedural | objektorientiert | funktional
- Nützliche eingebaute Datentypen (`list`, `tuple`, `dict`, `set`, ...)
- Einfache Modularisierung (`import this`)
- Gute Fehlerverwaltung (Exceptions)
- Umfangreiche Standardbibliothek
- Einfache Einbindung von externem Code (C, C++, Fortran)

# Warum Python? (1)

## Python als Programmiersprache

- Klare, lesbare Syntax (wenig „Ballast“)
- Paradigmen: prozedural | objektorientiert | funktional
- Nützliche eingebaute Datentypen (`list`, `tuple`, `dict`, `set`, ...)
- Einfache Modularisierung (`import this`)
- Gute Fehlerverwaltung (Exceptions)
- Umfangreiche Standardbibliothek
- Einfache Einbindung von externem Code (C, C++, Fortran)

⇒

- Leicht zu lernen
- Problemorientiert (mächtig und flexibel)
- Motivationspotenzial ↗, Frustrationspotenzial ↘

Außerdem: Plattformübergreifend / frei und quelloffen / große u. aktive Community

# Warum Python? (2)

Python als Werkzeug für Ingenieur:innen:

- Numerisches Rechnen (lin. Algebra, DGLn, Optimierung, ...)
- Symbolisches Rechnen (Ableiten, Integrieren, Gl. lösen, ...)
- Visualisieren (2D, 3D)
- Grafische Benutzerschnittstelle (GUI)
- Kommunikation mit externen Geräten
- Parallelisierung
- Datenwissenschaft
- Maschinelles Lernen
- Symbolische KI (Wissensgraphen, Ontologien, ...)

# Warum Python? (2)

Python als Werkzeug für Ingenieur:innen:

- Numerisches Rechnen (lin. Algebra, DGLn, Optimierung, ...)
- Symbolisches Rechnen (Ableiten, Integrieren, Gl. lösen, ...)
- Visualisieren (2D, 3D)
- Grafische Benutzerschnittstelle (GUI)
- Kommunikation mit externen Geräten
- Parallelisierung
- Datenwissenschaft
- Maschinelles Lernen
- Symbolische KI (Wissensgraphen, Ontologien, ...)

⇒ „Problemlöse- und Forschungskompetenz“

# Warum Python? (2)

Python als Werkzeug für Ingenieur:innen:

- Numerisches Rechnen (lin. Algebra, DGLn, Optimierung, ...)
  - Symbolisches Rechnen (Ableiten, Integrieren, Gl. lösen, ...)
  - Visualisieren (2D, 3D)
  - Grafische Benutzerschnittstelle (GUI)
  - Kommunikation mit externen Geräten
  - Parallelisierung
  - Datenwissenschaft
  - Maschinelles Lernen
  - Symbolische KI (Wissensgraphen, Ontologien, ...)
- } heute  
(ansatzweise)

⇒ „Problemlöse- und Forschungskompetenz“

# Fahrplan

## Lernziele:

- Erste Schritte in Python (Datentypen, Kontrollstrukturen)
- Werkzeuge und Bibliotheken kennenlernen: (Jupyter Notebook, Numpy, Scipy, Sympy, Matplotlib)
- Ingenieur-Probleme lösen (andeutungsweise)
- Befähigung zum Selbststudium

# Fahrplan

## Lernziele:

- Erste Schritte in Python (Datentypen, Kontrollstrukturen)
- Werkzeuge und Bibliotheken kennenlernen: (Jupyter Notebook, Numpy, Scipy, Sympy, Matplotlib)
- Ingenieur-Probleme lösen (andeutungsweise)
- Befähigung zum Selbststudium

## Herausforderungen:

- Heterogenes Vorwissen ( → Umfrage)
- Begrenzte Zeit
- Suboptimales Betreuungsverhältnis (ca. 30:1)

# Fahrplan

## Lernziele:

- Erste Schritte in Python (Datentypen, Kontrollstrukturen)
- Werkzeuge und Bibliotheken kennenlernen: (Jupyter Notebook, Numpy, Scipy, Sympy, Matplotlib)
- Ingenieur-Probleme lösen (andeutungsweise)
- Befähigung zum Selbststudium

## Herausforderungen:

- Heterogenes Vorwissen ( → Umfrage)
  - Begrenzte Zeit
  - Suboptimales Betreuungsverhältnis (ca. 30:1)
- ⇒ Selbstorganisierte Gruppenbildung

→ **1. Frontalphase, 2. Selbstlernphase mit Feedback. 3. Abschlussrunde**



# Vorbereitungen

<https://python-fuer-ingenieure.de/workshop>

# Jupyter Notebooks

- Backend: Webserver mit Python-Kernel (lokal oder nicht-lokal)
- Frontend: interaktives Dokument im Browser → „Notebook“
- Notebooks kombinieren Quellcode, Programm-Ausgaben und Dokumentation (inkl.  $\text{\LaTeX}$ -Formeln)
- (Andere Kernel möglich; hier nicht relevant)

# Jupyter

## Important keyboard shortcuts



### Command Mode (press `Esc` to enable)

- `Shift-Return` - execute cell, activate next
- `h` - show keyboard shortcuts
- `m` - change cell type to markdown
- `y` - change cell type to code
- `a` - new cell above

### Edit Mode (press `Return` to enable)

- `Shift-Return` - execute cell, activate next
- `Tab` - code-completion or indent
- `Shift-Tab` - tooltip
- `Ctrl-Z` - undo

# Jupyter

## Important keyboard shortcuts



### Command Mode (press `Esc` to enable)

- `Shift-Return` - execute cell, activate next
- `h` - show keyboard shortcuts
- `m` - change cell type to markdown
- `y` - change cell type to code
- `a` - new cell above

### Edit Mode (press `Return` to enable)

- `Shift-Return` - execute cell, activate next
- `Tab` - code-completion or indent
- `Shift-Tab` - tooltip
- `Ctrl-Z` - undo

→ Now play around with `00_demo1.ipynb` (5min)



Es folgt: Hastiger Überblick über Python-Syntax und Datentypen

# Numerical Data Types

- Integer

```
>>> type(1)
<type 'int'>
```

- floating point number

```
>>> type(1.0)
<type 'float'>
```

- complex number

```
>>> type(1 + 2j)
<type 'complex'>
```

- Operations:

Addition	+
Subtraction	-
Division	/
Integer division	//
Multiplication	*
Taking powers	**
Modulo	%

- Built-in functions

- `round`, `pow`, etc.
- see `dir(__builtins__)`

- Module `math`

- see `help(math)`

# NoneType, Boolean Values, Boolean Operators

- None (univ. value for “undefined”)

```
>>> type(None)
<type 'NoneType'>
```

- Boolean values:  
True and False

```
>>> type(True)
<type 'bool'>
```

- Boolean operators:

```
True and False # -> False
True or False  # -> True
not True       # -> False
```

Data Type	False-Value
NoneType	None
int	0
float	0.0
complex	0 + 0j
str	""
list	[]
tuple	()
dict	{}
set	set()

# Operations

Operation	Shortcut
$x = x + y$	$x += y$
$x = x - y$	$x -= y$
$x = x * y$	$x *= y$
$x = x / y$	$x /= y$
$x = x \% y$	$x \% = y$
$x = x ** y$	$x ** = y$
$x = x // y$	$x //= y$

Hint:

$x = x \% y$  is **modulo operation**  
(remainder of division)

Example:  $15 \% 6 = 3$

(because  $15 = 6 \cdot 2 + 3$ )

Comparison operations
$x == y$
$x != y$
$x < y$
$x <= y$
$x > y$
$x >= y$



# Strings (objects of type str)

```
str1 = "abc"  
str2 = 'xyzabcefgghi'  
str3 = """  
    multi  
    line  
    string  
    """
```

Escape Sequence	Meaning
\n	newline
\r	carriage return
\"	escaping "
\'	escaping '
\\	escaping \

```
>>> str2[0] # indexing starts at 0  
'x'  
>>> str2[1:4]  
'yza'  
>>> str2[-3:]  
'ghi'
```

# String Formatting

- General Syntax

```
"value of x={} and y={}".format(x, y)
```

- Examples

```
>>> a = 'H'
>>> b = 'ello World'
>>> "{}{}! {}".format(a, b, 5)
'Hello World! 5H'
```

- Extension (see also: [reference](#))

```
>>> "a={:06.2f} and b={:05.2f}".format(3.007, 42.1)
'a=003.01 and b=42.10'
```

- important methods of class str:

```
index, replace, split, join,
format, startswith, endswith, ...
```

# String Formatting (2)

- New syntax (since Python3.6): "f-strings"  
→ use Python expressions *inside* the string

```
a = "World"
f"Hello {a}"      # use variables
f"value of x={x} and y={y}"

f"the sum is {x + y}"    # make calculations
f"the result is {call_func(x, y, 'z')}"  # call functions

f"Use {{double braces}} to render braces literals! {a}"
```

- More info: <https://docs.python.org/3/tutorial/inputoutput.html>

# Lists

- Syntax  
`[value_1, ..., value_n]`
  - Can contain values of any type
  - Can be changed
  - Can be sorted
  - Important methods  
`append`, `count`, `index`, `insert`,  
`pop`, `remove`, `reverse`, `sort`
- ⚠ `sort` and `reverse` work „in place“  
(return-value: `None`)

- Examples

```
>>> m = [7, 8, 9]
>>> n = ['a', 'z', 1, False]
>>> m.append('x')
>>> m[0]
7
>>> m[-1]
'x'

>>> m[:] # start to end
[7, 8, 9, 'x']
>>> m.pop(0)
7
>>> m.reverse(); print(m)
['x', 9, 8]
```

# Tuple

- Syntax  
(value\_1, ..., value\_n)
- Can **not** be changed
- → Access much faster than to list
- Can contain elements of any type
- important methods  
index

- Examples

```
>>> t = (7,8,9)
>>> t[0]
7
>>> t[-1]
9
>>> t[:] # start to end
(7,8,9)
>>> z = ('a', 'z', 1, False)
>>> t.index(8)
1
>>> z.index('a')
0
```

# Sequential data types

`str, tuple, list, (numpy.array)`

Operation	Meaning
<code>s in x</code>	tests, whether s is element of x
<code>s not in x</code>	tests, whether s is not element of x
<code>x + y</code>	concatenation of x and y
<code>x * n</code>	concatenation, such that n copies of x exist
<code>x[n]</code>	return the n-th element of x
<code>x[n:m]</code>	return the subs-sequence from index n til m (excluding m)
<code>x[n:m:k]</code>	same with step-size k
<code>len(x)</code>	number of elements
<code>min(x)</code>	minimum
<code>max(x)</code>	maximum

# Dictionaries (Associative Arrays)

- Key-value-pairs
  - Keys must be immutable objects
  - Each key can occur only once
- Syntax

```
{ Key_1: Value_1,
  Key_2: Value_2,
  ... }
```
- Access via
  - `d.get(key, default)`  
or
  - `d[key]`
- Important methods
  - `keys`, `values`, `items`

## Examples

```
>>> d = {"Germany": "Berlin", "Peru": "Lima"}
```

```
>>> type(d)
<type 'dict'>
```

```
>>> e = {1: "a", 2: "b", 400: "c", 1.3: d}
>>> e[1]
'a'
```

```
>>> d.get("Germany")
'Berlin'
```

```
# no entry -> None (no output)
>>> d.get("Bavarya") # -> None
```

```
# with default value
>>> d.get("Bavarya", "unknown capital")
'unknown capital'
```

```
>>> d["Bavaria"]
KeyError: 'Bavaria'
```

# Sets

- Syntax  
`set([element_1, ..., element_n])`
- Every element is contained only once
- Has no specified order
- Can be changed (`frozenset` is immutable)
- Important methods:  
add, remove, union, difference,  
issubset, issuperset

## Examples

```
>>> engineers = set(['Jane', 'John',  
... 'Jack', 'Janice'])  
>>> programmers = set(['Jack', 'Sam',  
... 'Susan', 'Janice'])  
>>> managers = set(['Jane', 'Jack',  
... 'Susan', 'Zack'])  
>>> s1 = engineers.union(programmers)  
>>> s2 = engineers.intersection(managers)  
>>> s3 = managers.difference(engineers)  
>>> engineers.add('Marvin')  
>>> print(engineers)  
set(['Jane', 'Marvin',  
'Janice', 'John', 'Jack'])
```



# Data Types - Final Remarks

- In Python **everything is an object** (even functions, classes, modules)  
→ Everything has a type: `type(object)`
- Type checking (→ True or False):
  - Exact matching: `type("abc") == type("xyz")`
  - Better: respecting inheritance `isinstance(x, str)`
  - Allow multiple types: `isinstance(x, (int, float, complex))`
- Useful construction: `assert isinstance(x, int) and x > 0`

# Distinction of Cases: if, elif, else

- Syntax

```
# note the indention
if <condition1>:
    ...
elif <condition2>:
    ...
else:
    ...
```

- Examples

```
>>> x = 1
>>> if x == 1:
...     print("x is 1")
...
x is 1
>>> x = 4
>>> if x == 1:
...     print("x is 1")
... elif x == 3:
...     print("x is 3")
... else:
...
print("x is neither 1 nor 3")
x is neither 1 nor 3
```

# Iterate over a Sequence: for-loop

- Syntax:

```
for <variable> in <sequence>:  
    ...
```

- easily construct sequences:

`range`-function → iterator

```
range(stop)  
range(start, stop)  
range(start, stop, step)
```

```
# (conversion to list  
#     only for printing)
```

```
>>> list(range(4))  
[0, 1, 2, 3]
```

```
>>> list(range(1, 10, 2))  
[1, 3, 5, 7, 9]
```

- Examples:

```
>>> seq = ['a', 'b', 42]  
>>> for elt in seq:  
...     print(elt*2)  
aa  
bb  
84
```

```
>>> for i in range(3):  
...     print(2**i)  
1  
2  
4  
>>> for x, y in [(1, 2), "AB", [0, -3]]:  
...     print(f"{y} + {x} = {y + x}")  
2 + 1 = 3  
B + A = BA  
-3 + 0 = -3
```

# Loop while condition is true

- Syntax

```
while <condition>:  
    ...
```

- `break` # terminates the loop

```
while <condition1>:  
    if <condition2>:  
        break
```

- `continue` # start next cycle

```
while <condition1>:  
    if <condition2>:  
        continue
```

- Examples

```
>>> x = 4  
>>> while x > 1:  
...     print(x)  
...     x -= 1  
... print("finished")  
4  
3  
2  
finished
```

# Functions

- Syntax

```
def func_name(Param_1, ..., Param_n):  
    ...  
    return <result>
```

- No explicit return-value → None
- Empty function with keyword pass:

```
def empty():  
    pass
```

- default values for optional parameters

```
def test(x=23):  
    print(param)
```

- Arbitrary number of arguments

```
def func(*args, **kwargs):  
    print(type(args)) # -> tuple  
    print(type(kwargs)) # -> dict
```

- Examples

```
>>> def print_sum(a, b):  
...     print(a + b)  
>>> print_sum(1, 2)  
3  
>>> def print_prod(a, b, c=0):  
...     print(a*b + c)  
>>> print_prod(2, 4)  
8  
  
# better readable  
>>> print_prod(a=2, b=4)  
8  
>>> print_prod(2, 4, 1)  
9  
>>> print_prod(c=2, a=4, b=1)  
6
```

# Local Variables (Scopes)

Listing: local-variables.py

```
def square(z):  
    x = z**2 # x: local variable  
    print(x)  
    return x  
  
x, a = 5, 3 # "unpacking" a tuple  
  
square(a) # -> 9  
square(x) # -> 25  
print(x) # -> 5 (not changed)  
  
def square2(z):  
    print(x) # here: x is taken from global scope  
    return z**2  
  
def square3(z):  
    print(x) # Error (local variable not yet known)  
    x = z**2 # x is local variable due to write access  
    return x
```

# General Syntax

- Semantic blocks are defined by **indentation level** (in place of, e.g., { ... })
  - defacto-standard: 4 spaces per level (do not use TABs)
  - every good text editor can be configured adequately (spyder: TAB indention, SHIFT+TAB dedetion of highlighted lines)

- Comments and docstrings:

```
# single line comments begin with a hash
```

```
def my_function(x, y):  
    """
```

```
This is a docstring.  
It can span multiple lines  
    """
```

```
    """  
unassigned multi-line strings can  
be abused as multi-line comment  
    """
```

- Recommended maximum line length 80 (or 100) characters (readability)
- If you need more:
  - Check possibility to split up into two commands (readability)
  - Within braces newlines are ignored
  - Backslash (\) allows line continuation in expression

# Keywords (Reserved words)

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

They cannot be used as variable name or similar.



# File Access

Listing: file-access.py

```
# write in text mode
content_lines = ['some\n', 'more', 'content']
with open('text.txt', 'w') as myfile:
    myfile.write('Hello World.')
    myfile.writelines(content_lines)
    # myfile.close() is called automatically
    # when leaving this block

# read in text mode
with open('text.txt', 'r') as myfile:
    header = myfile.read(10) # first 10 byte
    lines = myfile.readlines() # list of lines
    # (starting from file cursor)
```

# File Access

Listing: file-access.py

```
# write in text mode
content_lines = ['some\n', 'more', 'content']
with open('text.txt', 'w') as myfile:
    myfile.write('Hello World.')
    myfile.writelines(content_lines)
    # myfile.close() is called automatically
    # when leaving this block

# read in text mode
with open('text.txt', 'r') as myfile:
    header = myfile.read(10) # first 10 byte
    lines = myfile.readlines() # list of lines
    # (starting from file cursor)
```

Read/write binary data: use 'rb' and 'wb'

Appending text or binary data: use 'a' or 'ab'

# Some “specialities” of Python

- Indexing starts with 0
- Unpacking of sequential data types:

```
>>> x, y, z = range(3)
>>> y
1
```

```
>>> mapping = [('green', 560), ('red', 700)]
>>> for color, wavelength in mapping:
...     pass
...     # do stuff
```

- ∃ extensive standard library („batteries included“)
  - <http://docs.python.org/3/library/>
  - “Don’t reinvent the wheel!”
  - Important modules: `pickle`, `sys`, `os`, `itertools`, `unittest`, ...

# Umgang mit Fehlern

- Fehler gehören zum Programmieren  
→ Kein Grund für Frustration!
- Fehler in Python: `Exceptions`
- Erzeugen typischerweise einen umfangreichen `Traceback`

# Umgang mit Fehlern

- Fehler gehören zum Programmieren  
→ Kein Grund für Frustration!
- Fehler in Python: `Exceptions`
- Erzeugen typischerweise einen umfangreichen `Traceback`  
→ Gewöhnungsbedürftig aber **sehr hilfreich!**
- Zusätzlich noch konkrete *Fehlermeldung*, z. B.

```
TypeError: can only concatenate tuple (not "list") to tuple
```

# Umgang mit Fehlern

- Fehler gehören zum Programmieren  
→ Kein Grund für Frustration!
- Fehler in Python: `Exceptions`
- Erzeugen typischerweise einen umfangreichen `Traceback`  
→ Gewöhnungsbedürftig aber **sehr hilfreich!**
- Zusätzlich noch konkrete *Fehlermeldung*, z. B.

```
TypeError: can only concatenate tuple (not "list") to tuple
```

- Fehlertypen:
  - `SyntaxError`
  - `TypeError`
  - `ValueError`
  - `IndexError`
  - `KeyError`
  - `ZeroDivisionError`
  - ...

# Quellen und Links (Python allgemein)

- Pythonkurs für Ingenieur:innen: <http://www.tu-dresden.de/pythonkurs>(Kurs 1 + 2)  
(Folien, Screencasts, Quiz-Fragen, Übungsaufgaben, Lösungen)
- Material zum Buch: <https://python-fuer-ingenieure.de/> (Kap. 1-3)
- Offizielles Tutorial: <http://docs.python.org/3/tutorial/>
- Interaktives Tutorial: <http://www.learnpython.org/>
- Ausführlicher gut strukturierter Kurs: <http://www.diveintopython3.net/>
- Kurzübersicht 1: <https://quickref.me/python>
- Kurzübersicht 2: <https://learnxinyminutes.com/docs/python/>

# Numpy

- Ziel: grober Überblick über die Möglichkeiten
- Aufbau:
  - Numpy Arrays
  - Numpy (grundlegende Numerik)
  - Scipy (anwendungsorientierte Numerik)



# Numpy arrays (I)

- Bisher folgende Container-Klassen („Sequenzen“) vorgestellt:  
Liste: [1, 2, 3], Tupel: (1, 2, 3), String: "1, 2, 3"
- Ungeeignet um damit zu rechnen

```
1  # sinnvoll bei Strings:
2  linie = "-." * 10 # -> "-.-.-.-.-.-.-.-.-.-"
3
4  # zum Rechnen nicht geeignet:
5  zahlen = [3, 4, 5]
6  res1 = zahlen*2 # -> [3, 4, 5, 3, 4, 5]
7
8  # geht gar nicht:
9  res2 = zahlen*1.5
10 res3 = zahlen**2
```

# Numpy arrays (II)

- Bei Numpy-Arrays: Berechnungen *elementweise*

```
1 import numpy as np
2
3 zahlen = [3.0, 4.0, 5.0] # Liste mit Gleitkommazahlen
4
5 x = np.array(zahlen)
6 res1 = x*1.5 # -> array([ 4.5,  6. ,  7.5])
7 res2 = x**2 # -> array([ 9., 16., 25.])
8 res3 = res1 - res2 # -> array([ -4.5, -10., -18.5])
```

- Arrays können  $n$  Dimensionen haben

```
1 arr_2d = np.array( [[1., 2, 3], [4, 5, 6]] )*1.0 # ->
2 # array( [[1., 2., 3.],
3 #         [4., 5., 6.]] )
4
5 print(arr_2d.shape) # -> (2, 3)
```

# Numpy arrays (III)

## Weitere Möglichkeiten, array-Objekte zu erstellen:

Listing: 01\_arrays\_erzeugen.py

```
import numpy as np
x0 = np.arange(10) # wie range(...) nur mit arrays
x1 = np.linspace(-10, 10, 200)
    # 200 Werte: array([-10., -9.899497, ..., 10])
x2 = np.logspace(1,100, 1e4) # 10000 Werte, immer gleicher Quotient

x3 = np.zeros(10) # np.ones analog
x4 = np.zeros( (3, 5) ) # Achtung: nur ein Argument! (=shape)

x5 = np.eye(3)
x6 = np.diag( (1, 2, 3) ) # 3x3-Diagonalmatrix

x7 = np.random.rand(5) # array mit 5 Zufallszahlen
x8 = np.random.rand(4, 2) # array mit 8 Zufallszahlen (shape=(4, 2))

from numpy import r_, c_ # index-Tricks für rows und columns
x9 = r_[6, 5, 4.] # array([ 6.,  5.,  4.])
x10 = r_[x9, -84, x3[:2]] # array([ 6.,  5.,  4., -84,  0.,  1.])
x11 = c_[x9, x6, x5] # in Spalten-Richtung stapeln -> 7x3 array
```

# Slicing und Broadcasting

- Slicing: Werte in einem Array adressieren
- Analog wie bei anderen Sequenzen: `x[start:stop:step]`
- Dimensionen durch Kommata getrennt; negative Indizes zählen von hinten

Listing: numpy\_scipy/02\_slicing.py

```
import numpy as np
a = np.arange(18)*2.0
A = np.array( [ [0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11] ] )

x1 = a[3] # Element Nr. "3" (-> 6.0)
x2 = a[3:6] # Elemente 3 bis 5 -> array([ 6., 8., 10.])
x3 = a[-3:] # Vom 3.-letzten bis Ende -> array([30.,32.,34.])
# Achtung a, x2 und x3 teilen sich die Daten!
a[-2:]*= -1
print(x3) # -> [-30.,-32.,-34.]

y1 = A[:, 0] # erste Spalte von A
y2 = A[1, :3] # ersten drei Elemente der zweiten Zeile
```

- ☐ „Broadcasting“: Automatisches vergrößern (z.B. Array + Zahl)

# „Broadcasting“

- Numpy's Umgang mit Arrays mit unterschiedlichen Abmessungen („shapes“) (bei elementweise ausgeführte Berechnungen)
- Trivialbeispiel:  $x$  (2d-array) +  $y$  (float) →  $y$  wird auf Shape von  $x$  „aufgeblasen“
- Anderes Beispiel: 2d-array + 1d-array
- Regel: Die Größe entlang der letzten *Achsen* beider Operanden müssen übereinstimmen oder eine von beiden muss eins sein.

- Zwei Beispiele:

3d-array\*1d-array = 3d-array

img.shape	(256,	256,	3)
scale.shape			(3,)
(img*scale).shape	(256,	256,	3)

4d-array+3d-array = 4d-array

A.shape	(8,	1,	6,	1)
B.shape		(7,	1,	5)
(A+B).shape	(8,	7,	6,	5)

- Führt manchmal zu Verwirrung/Problemen:

ValueError: shape mismatch: objects cannot be broadcast to a single shape

→ Im Zweifel **Doku lesen** uder<sup>1</sup> (interaktiv) ausprobieren

- siehe auch `numpy_scipy/03_broadcast_beispiel.py`

<sup>1</sup>: <https://cknoll.github.io/uder.html>

# Broadcasting-Beispiel

Listing: numpy\_scipy/03\_broadcast\_beispiel.py

```
import numpy as np
import time

E = np.ones((4, 3)) # -> shape=(4, 3)
b = np.array([-1, 2, 7]) # -> shape=(3,)
print(E*b) # -> shape=(4, 3)

b_13 = b.reshape((1, 3))
print(E*b_13) # -> shape=(4, 3)

print("\n"*2, "Achtung, die nächste Anweisung erzeugt einen Fehler.")
time.sleep(2)

b_31 = b_13.T # Transponieren -> shape=(3,1)
print(E*b_31) # broadcasting error
```

Erinnerung:  $E*b_{13}$  ist **keine** Matrix-Vektormultiplikation (siehe Folie 43)

# Numpy-Funktionen

```
1 import numpy as np
2 from numpy import sin, pi # Tipparbeit sparen
3
4 t = np.linspace(0, 2*pi, 1000)
5
6 x = sin(t) # analog: cos, exp, sqrt, log, log2, ...
7 xd = np.diff(x) # numerisch differenzieren
8 # Achtung: xd hat einen Eintrag weniger!
9 X = np.cumsum(x) # num. "integrieren" (kummulativ summieren)
```

- Keine python-Schleifen notwendig → Numpy-Funktionen sind schnell wie C-Code

- Vergleichsoperationen:

```
1 # Elementweise:
2 y1 = np.arange(3) >= 2
3     # -> array([False, False, True], dtype=bool)
4 # Array-weit:
5 y2 = np.all( np.arange(3) >= 0) # -> True
6 y3 = np.any( np.arange(3) < 0) # -> False
```

# Weitere Numpy Funktionen

- `min`, `max`, `argmin`, `argmax`, `sum` (→ Skalare)
- `abs`, `real`, `imag` (→ Arrays)
- Shape ändern: `.T` (transponieren), `reshape`, `flatten`, `vstack`, `hstack`

## Lineare Algebra:

- Matrix-Multiplikation:
  - `dot(a, b)` (empfohlen)
  - `a@b` (@-Operator in Python 3.5 eingeführt)
  - `np.matrix(a)*np.matrix(b)` (vom Kursleiter nicht empfohlen)
- Submodul: `numpy.linalg`:
  - `det`, `inv`, `solve` (LGS lösen), `eig` (Eigenwerte u. -vektoren),
  - `pinv` (Pseudoinverse), `svd` (Singulärwertzerlegung), ...



# Scipy

- Paket, das auf Numpy aufsetzt
- Bietet Funktionalität für
  - Daten-Ein- u. Ausgabe (z.B. mat-Format (Matlab))
  - Physikalische Konstanten
  - Noch mehr lineare Algebra
  - Signalverarbeitung (Fouriertransformation, Filter, ...)
  - Statistik
  - Optimierung
  - Interpolation
  - Numerische Integration („Simulation“)

# scipy.optimize (1)

- Besonders nützlich: `fsolve` (Doku) und `minimize` (Doku)
- `fsolve`: findet Nullstelle einer skalaren Funktion  $f : \mathbb{R} \rightarrow \mathbb{R}$  oder eines (nichtlinearen) Gleichungssystems  $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$
- Bei beiden: Startschätzung wichtig
- Beispiel: Näherungslösung der nichtlinearen Gleichung

$$x + 2.3 \cdot \cos(x) \stackrel{!}{=} 1 \quad \Leftrightarrow \quad x + 2.3 \cdot \cos(x) - 1 \stackrel{!}{=} 0$$

Listing: numpy\_scipy/06\_fsolve\_beispiel.py

```
import numpy as np
from scipy import optimize

def fnc1(x):
    return x + 2.3*np.cos(x) - 1

sol = optimize.fsolve(fnc1, 0) # -> array([-0.723632])
# Probe:
print(sol, sol + 2.3*np.cos(sol)) # -> [-0.72363261] [1.]
```

# scipy.optimize (2)

- `minimize`: findet Minimum einer Funktion  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  (Doku)
- Schnittstelle zu verschiedenen Minimierungsalgorithmen
- Erlaubt viele optionale Argumente  
(z. B. Angabe von Schranken oder (Un)gleichungs-Nebenbedingungen)
- Kann auch Gleichungen lösen, durch Minimierung des quadratischen **Gleichungsfehlers**:

Listing: numpy\_scipy/07\_minimize\_beispiel.py

```
import numpy as np
from scipy import optimize

def fnc2(x):
    return (x + 2.3*np.cos(x) - 1)**2 # quadratischer Gleichungsfehler

res = optimize.minimize(fnc2, 0) # Optimierung mit Startschätzung 0
# Probe:
print(res.x, res.x + 2.3*np.cos(res.x)) # -> [-0.7236326] [1.00000004]

# mit Grenzen und veränderter Startschätzung -> andere Lösung
res = optimize.minimize(fnc2, 0.5, bounds=[(0, 3)])
# Probe:
print(res.x, res.x + 2.3*np.cos(res.x)) # -> [2.03999505] [1.00000003]
```

# Num. Integration von DGLn (Theorie)

- „Simulation“ = numerisches Lösen von Differentialgleichungen
- DGL-Systeme (engl. **O**rdinary **D**ifferential **E**quations) in Zustandsdarstellung:  
 $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
- Zeitableitung des Zustands  $\mathbf{z}$  hängt vom Zustand selber ab (und von  $t$ )
- Lösung der DGL: Zeitverlauf  $\mathbf{z}(t)$  (hängt vom Anfangszustand  $\mathbf{z}(0)$  ab)  
→ „Anfangswertproblem“ (engl. „**I**nitial **V**alue **P**roblem“ (IVP))

# Num. Integration von DGLn (Theorie)

- „Simulation“ = numerisches Lösen von Differentialgleichungen
- DGL-Systeme (engl. **O**rdinary **D**ifferential **E**quations) in Zustandsdarstellung:  
 $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
- Zeitableitung des Zustands  $\mathbf{z}$  hängt vom Zustand selber ab (und von  $t$ )
- Lösung der DGL: Zeitverlauf  $\mathbf{z}(t)$  (hängt vom Anfangszustand  $\mathbf{z}(0)$  ab)  
→ „Anfangswertproblem“ (engl. „**I**nitial **V**alue **P**roblem“ (IVP))
- Bsp. harmonischer Oszillator mit DGL:  $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- Vorbereitung überführung in „Zustandsraumdarstellung“  
(eine DGLn 2. Ordnung → zwei DGLn 1. Ordnung):  
Zustand:  $\mathbf{z} = (z_1, z_2)^T$  mit  $z_1 := y, z_2 := \dot{y} \rightarrow$  zwei DGLn:  
 $\dot{z}_1 = z_2$  („definitorische Gleichung“)  
 $\dot{z}_2 = -2\delta z_2 - \omega^2 z_1$  ( $= \ddot{y}$ )
- $\exists$  verschiedene Integrationsalgorithmen (Euler, Runge-Kutta, ...)
- Nutzbar in Python über Universal-Funktion: `scipy.integrate.solve_ivp` (Doku)

# Num. Integration von DGLn (Theorie)

- „Simulation“ = numerisches Lösen von Differentialgleichungen
- DGL-Systeme (engl. **O**rdinary **D**ifferential **E**quations) in Zustandsdarstellung:  
 $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
- Zeitableitung des Zustands  $\mathbf{z}$  hängt vom Zustand selber ab (und von  $t$ )
- Lösung der DGL: Zeitverlauf  $\mathbf{z}(t)$  (hängt vom Anfangszustand  $\mathbf{z}(0)$  ab)  
→ „Anfangswertproblem“ (engl. „**I**nitial **V**alue **P**roblem“ (IVP))
- Bsp. harmonischer Oszillator mit DGL:  $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- Vorbereitung überführung in „Zustandsraumdarstellung“  
(eine DGLn 2. Ordnung → zwei DGLn 1. Ordnung):  
Zustand:  $\mathbf{z} = (z_1, z_2)^T$  mit  $z_1 := y, z_2 := \dot{y} \rightarrow$  zwei DGLn:  
 $\dot{z}_1 = z_2$  („definitorische Gleichung“)  
 $\dot{z}_2 = -2\delta z_2 - \omega^2 z_1$  ( $= \ddot{y}$ )
- $\exists$  verschiedene Integrationsalgorithmen (Euler, Runge-Kutta, ...)
- Nutzbar in Python über Universal-Funktion: `scipy.integrate.solve_ivp` (Doku)

**Ausführliche Erläuterung in Notebook:** `Simulation_dynamischer_Systeme.ipynb`

# Num. Integration (Umsetzung)

Listing: numpy\_scipy/04\_solve\_ivp\_beispiel.py

```
import numpy as np
from scipy.integrate import solve_ivp

delta = .1
omega_2 = 2**2
def rhs(t, z):
    """ rhs heißt 'right hand side [function]' """
    # Argument t muss in Signatur vorhanden sein, kann aber ignoriert werden
    z1, z2 = z # Entpacken des Zustandsvektors in seine zwei Komponenten
    z1_dot = z2
    z2_dot = -(2*delta*z2 + omega_2*z1)
    return [z1_dot, z2_dot]

tt = np.arange(0, 100, .01) # unabhängige Variable (Zeit)
z0 = [10, 0] # Anfangszustand für z1 und z2 (=y, und y_dot)
res = solve_ivp(rhs, (tt[0], tt[-1]), z0, t_eval=tt) # Aufruf des Integrators
zz = res.y # Zustandsverlaufs-Array (Zeilen: Zustandskomponenten; Spalten: Zeitschritte)

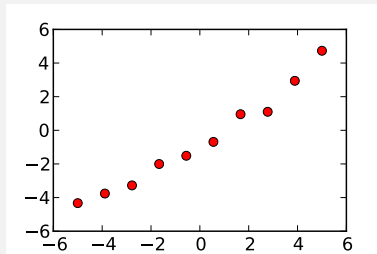
from matplotlib import pyplot as plt
plt.plot(tt, zz[0, :]) # Verlauf von z1 plotten
plt.show()
```

- Die Funktion `rhs` ist „ganz normales“ Objekt
- Kann als Argument an eine andere Funktion (hier: `solve_ivp`) übergeben werden
- Hinweis: Funktion `odeint` ist Vorgänger von `solve_ivp`.

Hauptunterschied: Argumentenreihenfolge und Rückgabeobjekt

# Regression, Interpolation

Regression mit `numpy.polyfit` (Doku):

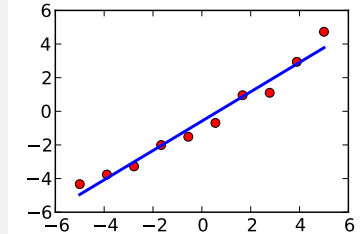




# Regression, Interpolation

Regression mit `numpy.polyfit` (Doku):

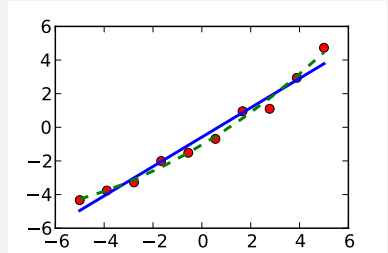
- Regressionsgerade (blau)



# Regression, Interpolation

Regression mit `numpy.polyfit` (Doku):

- Regressionsgerade (blau)
- oder höherer Ordnung



# Regression, Interpolation

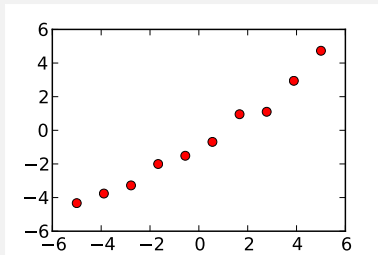
Regression mit `numpy.polyfit` (Doku):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation mit

`scipy.interpolation.interp1d` (Doku):

- Stückweise polynomial (→ „Spline“)
- beliebige Ordnung  
(hier: 0., 1., 2. Ordnung)



# Regression, Interpolation

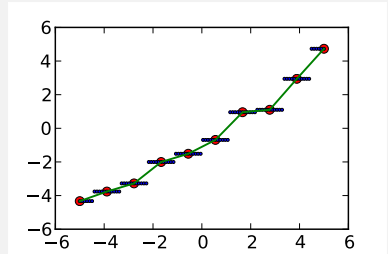
Regression mit `numpy.polyfit` (Doku):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation mit

`scipy.interpolation.interp1d` (Doku):

- Stückweise polynomial (→ „Spline“)
- beliebige Ordnung  
(hier: 0., 1., 2. Ordnung)



# Regression, Interpolation

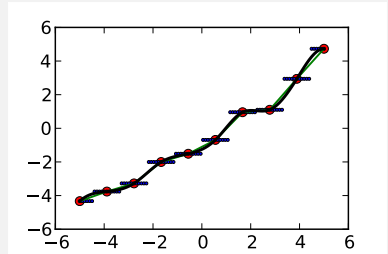
Regression mit `numpy.polyfit` (Doku):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation mit

`scipy.interpolation.interp1d` (Doku):

- Stückweise polynomial (→ „Spline“)
- beliebige Ordnung  
(hier: 0., 1., 2. Ordnung)



# Regression, Interpolation

Regression mit `numpy.polyfit` (Doku):

- Regressionsgerade (blau)
- oder höherer Ordnung

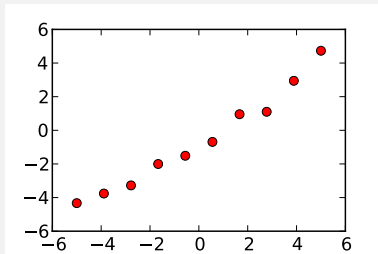
Interpolation mit

`scipy.interpolation.interp1d` (Doku):

- Stückweise polynomial (→ „Spline“)
- beliebige Ordnung  
(hier: 0., 1., 2. Ordnung)

Mischform mit `scipy.interpolation.splrep` (Doku):

- „geglätteter Spline“ (Glattheit über Koeffizient einstellbar)



# Regression, Interpolation

Regression mit `numpy.polyfit` (Doku):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation mit

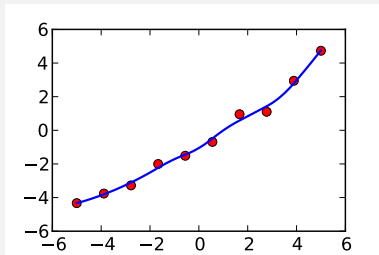
`scipy.interpolation.interp1d` (Doku):

- Stückweise polynomial (→ „Spline“)
- beliebige Ordnung  
(hier: 0., 1., 2. Ordnung)

Mischform mit `scipy.interpolation.splrep` (Doku):

- „geglätteter Spline“ (Glattheit über Koeffizient einstellbar)

Siehe auch `numpy_scipy/05_interp_beispiel.py`



# Regression, Interpolation

Regression mit `numpy.polyfit` (Doku):

- Regressionsgerade (blau)
- oder höherer Ordnung

Interpolation mit

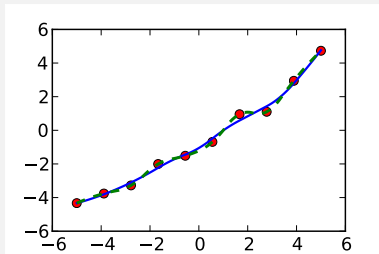
`scipy.interpolation.interp1d` (Doku):

- Stückweise polynomial (→ „Spline“)
- beliebige Ordnung  
(hier: 0., 1., 2. Ordnung)

Mischform mit `scipy.interpolation.splrep` (Doku):

- „geglätteter Spline“ (Glattheit über Koeffizient einstellbar)

Siehe auch `numpy_scipy/05_interp_beispiel.py`





# Zusammenfassung (Numpy/Scipy)

- `numpy`-Arrays
- Weitere `numpy`-Funktionen
- `scipy`-Funktionen (Numerische Integration, Interpolation + Regression)

# Das Paket sympy

- Python-Bibliothek für symbolische Berechnungen („mit Buchstaben rechnen“)  
→ Backend eines Computer Algebra Systems (CAS)
- Mögliche Frontends: eigenes Python-Skript, IPython-Shell, IPython-Notebook (im Browser)
- Vorteil: CAS-Funktionalität zusammen mit richtiger Programmiersprache

# sympy Überblick

- Varianten das Paket bzw. Objekte daraus zu importieren:
  1. `import sympy as sp`
  2. `from sympy import sin, cos, pi`
  3. `from sympy import *` ⚠ Vorisicht: „namespace pollution“

# sympy Überblick

- Varianten das Paket bzw. Objekte daraus zu importieren:
  1. `import sympy as sp`
  2. `from sympy import sin, cos, pi`
  3. `from sympy import *` ⚠ Vorisicht: „namespace pollution“
- `sp.symbols, sp.Symbol` : Symbole erzeugen ( $\neq$  Variable)
- `sp.sin(x), sp.cos(2*pi*t), sp.exp(x), ...` : mathematische Funktionen
- `sp.Function('f')(x)` : eigene Funktion anlegen (und auswerten)
- `sp.diff(<expr>, <var>)` oder `<expr>.diff(<var>)` : ableiten
- `sp.simplify(<expr>)` : vereinfachen
- `<expr>.expand()` : ausmultiplizieren
- `<expr>.subs(...)` : substituieren
- `sp.pprint(<expr>)` : „pretty printing“

# Rechnen mit sympy

Listing: sympy1.py

```
import sympy as sp
x = sp.Symbol('x')
a, b, c = sp.symbols('a b c') # verschiedene Wege Symbole zu erzeugen

z = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b*2))
print(z) # -> -b*c*(-1/(2*b) + 2*a*b*x/c) + 2*a*x*b**2
print(z.expand()) # -> c/2 (Ausmultiplizieren)

# Funktionen anwenden:
y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a)
print(y) # -> a**(1/2)*exp(x)*sin(x)

# Eigene Funktionen definieren
f1 = sp.Function('f') # -> sympy.core.function.f (nicht ausgewertet)
g1 = sp.Function('g')(x) # -> g(x) (Funktion ausgewertet bei x)

# Differenzieren
print(y.diff(x)) # -> 3*sqrt(a)*exp(3*x)*sin(x) + sqrt(a)*exp(3*x)*cos(x)
print(g1.diff(x)) # -> Derivative(g(x), x)

# Vereinfachungen:
print(sp.trigsimp(sp.sin(x)**2+sp.cos(x)**2)) # -> 1
```

# Substituieren: `<expr>.subs(...)`

- Vergleichbar mit `<str>.replace(alt, neu)`
- Nützlich für: manuelle Vereinfachungen, (partielle) Funktionsauswertungen, Koordinatentransformationen

```
term1 = a*b*sp.exp(c*x)
term2 = term1.subs(a, 1/b)
print(term2) # -> exp(c*x)
```

# Substituieren: `<expr>.subs(...)`

- Vergleichbar mit `<str>.replace(alt, neu)`
- Nützlich für: manuelle Vereinfachungen, (partielle) Funktionsauswertungen, Koordinatentransformationen

```
term1 = a*b*sp.exp(c*x)
term2 = term1.subs(a, 1/b)
print(term2) # -> exp(c*x)
```

- Aufrufmöglichkeiten: 1. direkt, 2. Liste mit Tupeln, 3. dict (nicht empfohlen)
  1. `<expr>.subs(alt, neu)`
  2. `<expr>.subs([(alt1, neu1), (alt2, neu2), ...])`
    - Reihenfolge der Liste → Substitutionsreihenfolge
    - Relevant beim Substituieren von Ableitungen (siehe [Beispiel-Notebook](#))
- Wichtig: `subs(...)` liefert Rückgabewert (original-Ausdruck bleibt unverändert)

# Die wichtigen Methoden / Funktionen / Typen

- `sp.Matrix([[x, a+b], [c*x, sp.sin(x)]])` : Matrizen
- `<mtrx>.jacobian(xx)` : Jacobi-Matrix eines Vektors
- `sp.solve(x**2 + x - a, x)` : Gleichungen und Gleichungssysteme lösen
- `<expr>.atoms(), <expr>.atoms(sp.sin)` : „Atome“ (bestimmten Typs)
- `<expr>.args` : Argumente der jeweiligen Klasse (Summanden, Faktoren, ...)
- `sp.simplify(...)` : Datentypen-Anpassung
- `sp.integrate(<expr>, <var>)` : Integration
- `sp.series(...)` : Reihenentwicklung
- `sp.limit(<var>, <value>)` : Grenzwert
- `<expr>.as_num_denom()` : Zähler-Nenner-Aufspaltung
- `sp.Polynomial(x**7+a*x**3+b*x+c, x, domain='EX')` : Polynome
- `sp.Piecewise(...)` : Stückweise definierte Funktionen



# Numerische Formelauswertung

- Gegeben: Formel und Werte der einzelnen Variablen
- Gesucht: numerisches Ergebnis

# Numerische Formelauswertung

- Gegeben: Formel und Werte der einzelnen Variablen
- Gesucht: numerisches Ergebnis
- Prinzipiell möglich: `expr.subs(num_werte).evalf()`

# Numerische Formelauswertung

- Gegeben: Formel und Werte der einzelnen Variablen
- Gesucht: numerisches Ergebnis
- Prinzipiell möglich: `expr.subs(num_werte).evalf()`
- Besser (bzgl. Geschwindigkeit): `lambdify` (Namensherkunft: Pythons `lambda`-Funktionen)
- Erzeugt eine Python-Funktion, die man dann mit den Argumenten aufrufen kann

```
f = a*sp.sin(b*x)
df_xa = f.diff(x)
```

```
# Funktion erzeugen
```

```
df_xa_fnc = sp.lambdify((a, b, x), df_xa, modules='numpy')
```

```
# Funktion auswerten
```

```
print( f_xa_fnc(1.2, 0.5, 3.14) )
```

# Sympy-Unterstützung in Jupyter

Another trick: let sympy expressions be nicely rendered by  $\LaTeX \Rightarrow$  readability ↑.

Siehe Beispiel-Notebook

[../notebooks/sympy-notebook1.html](#)

```
In [7]: from sympy.interactive import printing
printing.init_printing()
```

```
%load_ext ipydx.displaytools
```

```
In [8]: # same code with special-comments (`##:`)
```

```
x = sp.Symbol("x")
a, b, c, z = sp.symbols("a b c z") # create several symbols at once

some_formula = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b*2)) ##:

# some calculus
y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a) ##:

# derive
yd = y.diff(x) ##:
```

$$\text{some\_formula} := 2ab^2x - bc \left( \frac{2a}{c}bx - \frac{1}{2b} \right)$$

---

$$y := \sqrt{ae^{3x}} \sin(x)$$

---

$$yd := 3\sqrt{ae^{3x}} \sin(x) + \sqrt{ae^{3x}} \cos(x)$$

---

# Sympy-Unterstützung in Jupyter

Another trick: let sympy expressions be nicely rendered by  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} \Rightarrow$  readability ↑.

Siehe Beispiel-Notebook

[../notebooks/sympy-notebook1.html](#)

In [7]: `from sympy.interactive import printing`  
`printing.init_printing()` ←

`%load_ext ipydx.displaytools`

Für  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -Ausgabe

In [8]: `# same code with special-comments (`##:`)`

```
x = sp.Symbol("x")
a, b, c, z = sp.symbols("a b c z") # create several symbols at once

some_formula = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b*2)) ##:

# some calculus
y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a) ##:

# derive
yd = y.diff(x) ##:
```

$$\text{some\_formula} := 2ab^2x - bc \left( \frac{2a}{c}bx - \frac{1}{2b} \right)$$

---

$$y := \sqrt{a}e^{3x} \sin(x)$$

---

$$yd := 3\sqrt{a}e^{3x} \sin(x) + \sqrt{a}e^{3x} \cos(x)$$

---

# Sympy-Unterstützung in Jupyter

Another trick: let sympy expressions be nicely rendered by  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} \Rightarrow$  readability ↑.

Siehe Beispiel-Notebook

[../notebooks/sympy-notebook1.html](#)

In [7]: `from sympy.interactive import printing`  
`printing.init_printing()` ←

`%load_ext ipydx.displaytools` ←

Für  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -Ausgabe

In [8]: `# same code with special-comments (`##:`)`

```
x = sp.Symbol("x")
a, b, c, z = sp.symbols("a b c z") # create several symbols at once

some_formula = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b*2)) ##:

# some calculus
y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a) ##:

# derive
yd = y.diff(x) ##:
```

Aktiviert speziellen Kommentar:

`##:`

$$\text{some\_formula} := 2ab^2x - bc \left( \frac{2a}{c}bx - \frac{1}{2b} \right)$$

---

$$y := \sqrt{ae^{3x}} \sin(x)$$

---

$$yd := 3\sqrt{ae^{3x}} \sin(x) + \sqrt{ae^{3x}} \cos(x)$$

---

# Sympy-Unterstützung in Jupyter

Another trick: let sympy expressions be nicely rendered by  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} \Rightarrow$  readability 1.

Siehe Beispiel-Notebook

[../notebooks/sympy-notebook1.html](#)

In [7]: `from sympy.interactive import printing`  
`printing.init_printing()`

`%load_ext ipydx.displaytools`

Für  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -Ausgabe

In [8]: `# same code with special-comments (`##:`)`

`x = sp.Symbol("x")`  
`a, b, c, z = sp.symbols("a b c z") # create several symbols at once`

`some_formula = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b*2)) ##:`

Aktiviert speziellen Kommentar:

`# some calculus`

`y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a) ##:`

##:

`# derive`

`yd = y.diff(x) ##:`

Dieser zeigt Ergebnis von  
Zuweisungen an

`some_formula :=  $2ab^2x - bc \left( \frac{2a}{c}bx - \frac{1}{2b} \right)$`

---

`y :=  $\sqrt{ae^{3x}} \sin(x)$`

---

`yd :=  $3\sqrt{ae^{3x}} \sin(x) + \sqrt{ae^{3x}} \cos(x)$`

---

# Sympy-Unterstützung in Jupyter

Another trick: let sympy expressions be nicely rendered by  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} \Rightarrow$  readability ↑.

Siehe Beispiel-Notebook

[../notebooks/sympy-notebook1.html](#)

In [7]: `from sympy.interactive import printing`  
`printing.init_printing()`

`%load_ext ipydx.displaytools`

Für  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -Ausgabe

In [8]: `# same code with special-comments (`##:`)`

`x = sp.Symbol("x")`  
`a, b, c, z = sp.symbols("a b c z")` # create several symbols at once

`some_formula = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b*2))` ##:

Aktiviert speziellen Kommentar:

`# some calculus`

`y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a)` ##:

##:

`# derive`

`yd = y.diff(x)` ##:

Dieser zeigt Ergebnis von  
Zuweisungen an

`some_formula :=`  $2ab^2x - bc \left( \frac{2a}{c}bx - \frac{1}{2b} \right)$

---

`y :=`  $\sqrt{ae^{3x}} \sin(x)$

---

`yd :=`  $3\sqrt{ae^{3x}} \sin(x) + \sqrt{ae^{3x}} \cos(x)$

---



# Sympy-Unterstützung in Jupyter

Another trick: let sympy expressions be nicely rendered by  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X} \Rightarrow$  readability ↑.

Siehe Beispiel-Notebook

[../notebooks/sympy-notebook1.html](#)

In [7]: `from sympy.interactive import printing`  
`printing.init_printing()`

`%load_ext ipydx.displaytools`

Für  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ -Ausgabe

In [8]: `# same code with special-comments (`##:`)`

`x = sp.Symbol("x")`  
`a, b, c, z = sp.symbols("a b c z")` # create several symbols at once

`some_formula = a*b*x*b + b**2*a*x - c*b*(2*a/c*x*b-1/(b*2))` `##:`

`# some calculus`

`y = sp.sin(x)*sp.exp(3*x)*sp.sqrt(a)` `##:`

`# derive`

`yd = y.diff(x)` `##:`

Aktiviert speziellen Kommentar:

`##:`

`some_formula :=`  $2ab^2x - bc \left( \frac{2a}{c}bx - \frac{1}{2b} \right)$

---

`y :=`  $\sqrt{ae^{3x}} \sin(x)$

---

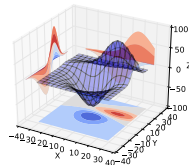
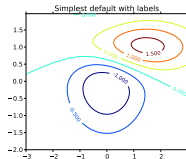
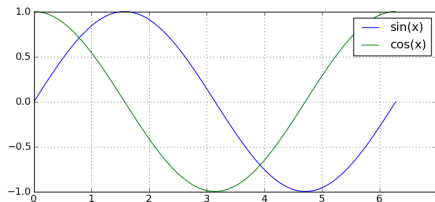
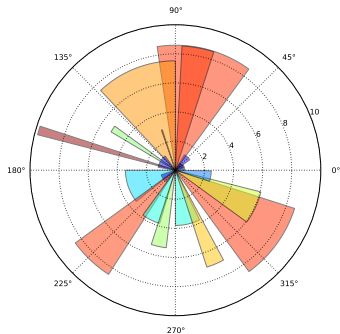
`yd :=`  $3\sqrt{ae^{3x}} \sin(x) + \sqrt{ae^{3x}} \cos(x)$

---

Dieser zeigt Ergebnis von  
Zuweisungen an

# 2D-Visualisierung mit matplotlib

- Visualisierung von Messdaten, Simulationsergebnissen etc.
- Publikationsfertige Grafiken für Studien- und Diplomarbeiten, Dissertationen, wissenschaftliche Artikel und Beiträge



# Das Paket Matplotlib



- Quasi-Standard für 2D-Plotten mit Python (kann auch 3D)
- Syntax an Matlab angelehnt → einfach für Umsteiger
- Sehr viele Plotfunktionen und -arten
- Interaktiv (zooming, panning)
- Einbinden von Formeln und Symbolen
- $\text{\LaTeX}$ -Schnittstelle für Text in Diagrammen
- Speichern von Plots in als Rastergrafik (z. B. png) oder Vektorgrafik (z. B. pdf)
- Umfangreiche Dokumentation
- Performance: akzeptabel, aber nicht ideal für Echtzeit-Visualisierung

<http://matplotlib.org/gallery.html>

# Matplotlib - Einstieg

Zwei Möglichkeiten der Verwendung

1. interaktiv in der Konsole (pylab) → IPython
2. objektorientiert in Skripten und Anwendungen (pyplot)

```
1 import matplotlib
2 from matplotlib.pylab import * # Matlab-artige Umgebung
3
4 ion()                          # interaktiv on/off
5 ioff()
```

# Grundlagen 1

## Einführungsbeispiel interaktiv

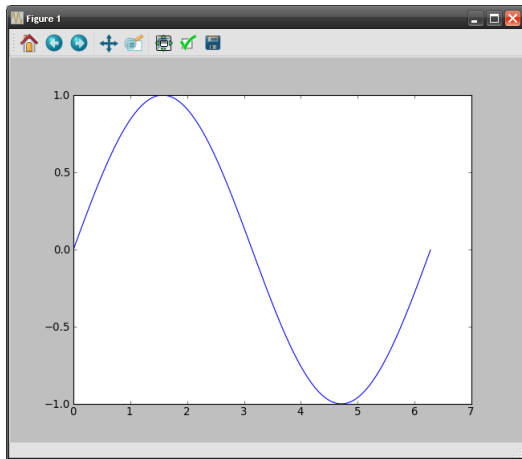
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0,6.28,100)
5 y = np.sin(x)
6
7 plt.plot(x, y)
```

## Plot individualisieren

```
1 plt.title('Sinusfunktion')
2 plt.grid()
3 plt.xlabel('x')
4 plt.ylabel('y')
```

## Schließen (nur in interaktivem Modus sinnvoll)

```
plt.close()
```



# Grundlagen 2

## Plot mit Optionen (kurz)

```
plt.plot(x, y, 'ro:') → Farbe, Marker und Linientyp
```

## Plot mit Optionen (lang)

```
1 plt.plot(x, y, color='#FF0000', ls=':', lw=2, marker='o')
```

## Mehrere Kurven mit einem Befehl

```
1 plt.plot(x, sin(x), 'r--', x, cos(x), 'g:')
```

## Legende

```
1 plt.plot(x, sin(x), label='sin(x)')  
2 plt.plot(x, cos(x), label='cos(x)')  
3 plt.legend()
```

# Grundlagen 3

## Sonderzeichen / Formeln $\rightarrow$ $\text{\LaTeX}$ Support

```
1 plot(x, y/x, label=(r'$\frac{\sin(\phi)}{\phi}$'))  
2 leg = plt.legend()
```

## Schriftgröße Legende

```
1 plt.setp(leg.texts, fontsize=20)
```

## Schriftgröße Label

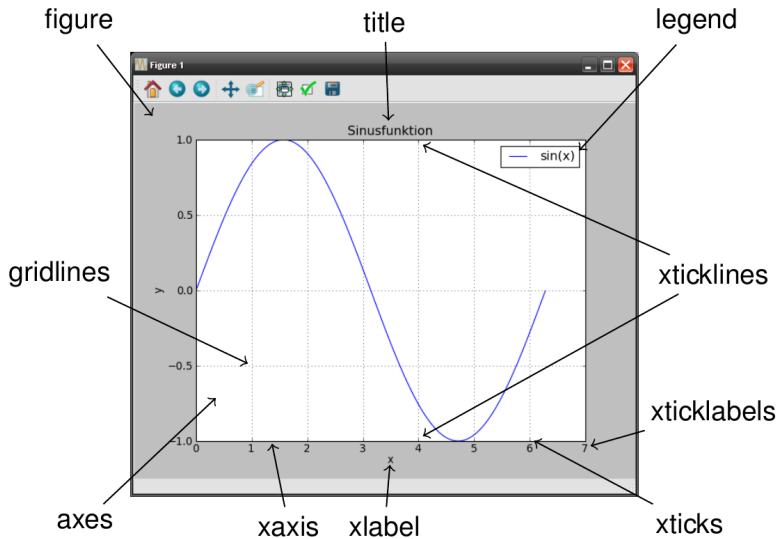
```
1 plt.xlabel('Zeit[s]', fontsize=14)
```

## Aussehen nachträglich ändern

```
1 p = plt.plot(x,y)
```

```
plt.setp(p, linewidth=3)  $\rightarrow$  beliebige Anzahl von Optionen
```

# Begriffe





# Beispiel-Skript 1

Listing: matplotlib1.py

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 alpha = np.linspace(0, 6.28, 100)
5 y = np.sin(alpha)
6
7 mm = 1./25.4 # Umrechnung mm -> zoll
8 fig = plt.figure(figsize=(250*mm, 180*mm)) # Abmessungen explizit vorgeben
9
10 plt.plot(alpha, y, label=r'$\sin(\alpha)$')
11 plt.xlabel(r'$\alpha$ in rad')
12 plt.ylabel('$y$')
13 plt.title('Sinusfunktion')
14 plt.legend() # Legende einblenden
15 plt.grid() # Gitterlinien einblenden
16
17 plt.savefig('test.pdf') # Dateiformat wird aus Endung erkannt
18 plt.show() # Anzeige des Bildes
```

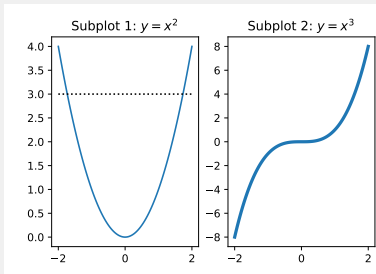
## Empfehlungen:

- Daten-Erzeugung (z. B. Simulation) und Visualisierung getrennt implementieren
- Visualisierung komplett automatisieren (möglichst keine manuelle Interaktion)
- Grund: später oft noch Anpassungsbedarf an Visualisierungs-Details  
→ nicht jedes Mal neue Simulation notwendig

# Beispiel-Skript 2 (Subplots)

Listing: matplotlib2.py

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 xx = np.linspace(-2, 2, 100)
5
6 mm = 1./25.4 # Umrechnung mm -> zoll
7 scale = 0.5 # Für proportionale Skalierung
8 fs = np.array([250*mm, 180*mm])*scale
9
10 # kleiner rechter Rand
11 plt.rcParams['figure.subplot.right'] = .98
12
13 # subplots: 1 Zeile, 2 Spalten
14 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=fs);
15
16 ax1.plot(xx, xx**2)
17 ax1.set_title("Subplot 1:  $y=x^2$ ")
18
19 ax2.plot(xx, xx**3, lw=3)
20 ax2.set_title("Subplot 2:  $y=x^3$ ")
21
22 ax1.plot(xx, xx*0+3, ":k") # gepunktete schwarze Linie bei y=3
23
24 plt.savefig('subplots.pdf') # Dateiformat wird aus Endung erkannt
25 plt.show() # Anzeige des Bildes
```



# Wo bekomme ich Hilfe?

- Matplotlib ist sehr umfangreich und komplex → Doku unter <http://matplotlib.org/contents.html>
- Tipp 1: Gallery (<http://matplotlib.org/gallery.html>)  
Sehr viele Beispiele (Bilder und Code)
- Tipp 2: Dokumentationen der Axes-Klasse  
[http://matplotlib.org/api/axes\\_api.html?matplotlib.axes.Axes](http://matplotlib.org/api/axes_api.html?matplotlib.axes.Axes)
- Alle Plot- und Zeichenfunktionen sind über die `axes`-Klasse zu erreichen!
  - `ax.plot()` , `ax.bar()` , `ax.scatter()` , `ax.arrow()` , ...
  - Besonders wichtig: Keyword-Argumente
- Siehe auch Cheatsheet im Anhang

# Häufig benötigte Einstellungen

- `ax.set_aspect('equal')` → Seitenverhältnis 1:1
- `ax.set_xlim(0, 10)` → Wertebereich der X-Achse
- `ax.set_xticklabels(['a', 'b'])` → eigene Beschriftungen
- `ax.legend(loc=1)` → Position der Legende
- `ax.tick_params(**kwargs)` → Optik der Achsenbeschriftung

Grundsätzlich auch:

- `setp()` ermöglicht das setzen von Eigenschaften von Objekten
- `getp()` liefert die Objekte und deren `kwargs` (Introspection)

```
1 setp(getp(leg, "texts"), fontsize=10)
```

# L<sup>A</sup>T<sub>E</sub>X und matplotlib

- Matplotlib bringt einen (reduzierten) L<sup>A</sup>T<sub>E</sub>X Compiler mit
- Ausprobieren:

```
- ax.plot(x, y, label='$\sin(x)$')  
- ax.set_ylabel('$\mu_2$')
```

- Optionen können global in rc gesetzt werden:

```
1 import matplotlib as mpl  
2  
3 mpl.rc('font', family = serif,  
4       serif = ['Computer Modern Roman'],  
5       size = 11)  
6  
7 mpl.rc('text', usetex=True)
```

→ Damit werden alle Strings mit L<sup>A</sup>T<sub>E</sub>X kompiliert

- Alternativ: direkter Zugriff auf das dict -Objekt `mpl.rcParams`
  - Entdecken mit:

```
1 import ipydex # ggf. vorher pip install ipydex ausführen  
2 ipydex.dirsearch("size", mpl.rcParams)
```

# Matplotlib Zusammenfassung

- Bibliothek für 2D-Visualisierung: `matplotlib`
- Sehr umfangreiche Optionen
- Bewährtes Vorgehen:
  - Beispiele unter [matplotlib.org/gallery.html](https://matplotlib.org/gallery.html) nachschauen und anpassen

# Matplotlib Cheatsheet: Linienstile

Zwei Möglichkeiten. Kurz: als Format-String

```
plot(x, y, '-.')
```

Lang: als Keyword-Argument (Kurz- und Langform zulässig)

```
plot(x, y, linestyle='dashed') oder plot(x, y, ls='--')
```

Kurzform	Langform	Beschreibung	Output
' ' oder ' '	None	ohne Linie	
'-'	'solid'	durchgehende Linie ( <i>Vorgabe</i> )	_____
'--'	'dashed'	gestrichelte Linie	- - - - -
'-.'	'dashdot'	Strichpunktlinie	- . - . - .
':'	'dotted'	gepunktete Linie	. . . . .

# Matplotlib Cheatsheet: Marker

Wiederum zwei Möglichkeiten, kurz als Format-String

```
plot(x, y, 'o')
```

Lang als Keyword-Argument

```
plot(x, y, marker='x')
```

Größe mit `markersize=10` oder `ms=10` einstellbar

Auswahl:

Kurzform	Beschreibung	Output
' ' oder <code>None</code>	ohne Marker ( <i>Vorgabe</i> )	
'.'	Punkt	●
'o'	Kreis	○
'D'	Diamant	◇
'H'	Hexagon	⬡
'+'	Plus	+
's'	Quadrat	■
'x'	Kreuz	×



# Matplotlib Cheatsheet: Farben

Im Format-String, oder als Keyword-Agrument

```
plot(x, y, 'g') oder plot(x, y, color='green')
```

Grau: Zahlenwert zwischen 0 ( $\hat{=}$ schwarz) und 1 ( $\hat{=}$ weiß) als `str`-Objekt: `plot(x, y, color='0.3')`

Für anspruchsvollere Fälle

- Hexadezimale Notation (wie in HTML/CSS): `color='#e3e6f7'`
- 3-Tupel ( $\hat{=}$  **R**ot, **G**rün, **B**lau): `color=(0.3, 0.8, 0.1)`
- 4-Tupel ( $\hat{=}$  RGB + **A**lpha (Deckungskraft)): `color=(0.3, 0.8, 0.1, 0.7) # 30% Transparenz`

Grundlegende Farben sind vordefiniert:

Kurzform	Langform	Farbe
'b'	'blue'	Blau
'g'	'green'	Grün
'r'	'red'	Rot
'c'	'cyan'	Cyan
'm'	'magenta'	Magenta
'y'	'yellow'	Gelb
'k'	'black'	Schwarz

# Das Pandas Paket

## → „Tabellenkalkulation mit Python“

- Wichtigstes Paket für „Data Science“
- basiert auf Numpy
- Wichtigster Datentyp: `pandas.DataFrame`
  - Modelliert Tabelle
  - Spalten haben Namen
  - Spalten können unterschiedliche Datentypen haben
- Zweitwichtigster Datentyp: `pandas.Series`
  - Modelliert Zeile oder Spalte

Alles weitere, siehe <https://pandas.pydata.org/docs/>

# Quellen und Links (wissenschaftliche Pakete)

- <https://numpy.org/doc/>
- <https://docs.scipy.org/doc/scipy/reference/>
- [http://www.scipy.org/Tentative\\_NumPy\\_Tutorial](http://www.scipy.org/Tentative_NumPy_Tutorial)
- [http://www.scipy.org/NumPy\\_for\\_Matlab\\_Users](http://www.scipy.org/NumPy_for_Matlab_Users)
- <https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>
- [http://scipy.org/Numpy\\_Example\\_List\\_With\\_Doc](http://scipy.org/Numpy_Example_List_With_Doc) (umfangreich)
- <http://docs.scipy.org/doc/scipy/reference/> (Tutorial + Referenz)
- <http://www.scipy.org/Cookbook>
- <http://docs.sympy.org/latest/modules/>
- <https://matplotlib.org/stable/users/index.html>

Auch hilfreich: google, stackoverflow, ...

# Schlussbemerkungen Phase 1

## Das war zu viel Kram in zu kurzer Zeit!

- Wissen ist wissen,
  - **was** es gibt und
  - **wo** es dokumentiert ist.
- Anwendungsbereitschaft erfordert Übung (→ Phase 2)

## Möglichkeiten für Anschlusskommunikation

- Kontakt: `carsten.knoll@posteo.de`
- Mailingliste `py4ing@lists.python-fuer-ingenieure.de` (Abbo-Link)
- Chat Raum: `#py4ing:matrix.org`

# Phase 2: Übung

- Bearbeitung von Übungsaufgaben (vorzugsweise in 3er-Gruppen)
  - Alle TN bleiben im Hauptraum eingewählt; dieser bleibt still; Chatten OK!
  - Es werden nach Bedarf zusätzliche Gruppenräume geöffnet
- Eine Person teilt Bildschirm, Gruppe arbeitet gemeinsam an Aufgaben
- Bei unterschiedlichen Interessen löst sich Gruppe auf → ggf. Haupt-Chat

# Phase 2: Übung

- Bearbeitung von Übungsaufgaben (vorzugsweise in 3er-Gruppen)
- Alle TN bleiben im Hauptraum eingewählt; dieser bleibt still; Chatten OK!
- Es werden nach Bedarf zusätzliche Gruppenräume geöffnet
- Eine Person teilt Bildschirm, Gruppe arbeitet gemeinsam an Aufgaben
- Bei unterschiedlichen Interessen löst sich Gruppe auf → ggf. Haupt-Chat
- Gruppenaufteilung:
  - TN durchnummerieren
  - $\text{Gruppe}_1 = \{ \text{TN}_1 + \text{TN}_2 + \text{TN}_3 \}$ ,  $\text{Gruppe}_2 = \{ \text{TN}_4 + \text{TN}_5 + \text{TN}_6 \}$  usw.
  - Fragen: Warteliste mit Link zum Gruppenraum
  - Empfehlung: Während Wartezeit Weiterüben (andere Aufgaben)

# Phase 2: Übung

- Bearbeitung von Übungsaufgaben (vorzugsweise in 3er-Gruppen)
- Alle TN bleiben im Hauptraum eingewählt; dieser bleibt still; Chatten OK!
- Es werden nach Bedarf zusätzliche Gruppenräume geöffnet  
→ Eine Person teilt Bildschirm, Gruppe arbeitet gemeinsam an Aufgaben
- Bei unterschiedlichen Interessen löst sich Gruppe auf → ggf. Haupt-Chat
- Gruppenaufteilung:
  - TN durchnummerieren
  - $\text{Gruppe}_1 = \{ \text{TN}_1 + \text{TN}_2 + \text{TN}_3 \}$ ,  $\text{Gruppe}_2 = \{ \text{TN}_4 + \text{TN}_5 + \text{TN}_6 \}$  usw.
  - Fragen: Warteliste mit Link zum Gruppenraum
  - Empfehlung: Während Wartezeit Weiterüben (andere Aufgaben)
- Koordination von Gruppen und Fragen: <https://yopad.eu/p/py4ing-365days>

# Phase 2: Übung (2)

01\_Uebung.ipynb (10 Aufgaben)



# Abschlussrunde

Ingenieur-Python nicht an einem Tag lernbar, aber

- Möglichkeiten andeuten
- Erste Ergebnisse ermöglichen
- Zum weiteren Selbststudium anregen

## Möglichkeiten für Anschlusskommunikation

- Kontakt: `carsten.knoll@posteo.de`
- Mailingliste `py4ing@lists.python-fuer-ingenieure.de` (Abbo-Link)
- Chat Raum: `#py4ing:matrix.org`

# Werbung



<https://python-fuer-ingenieure.de>