

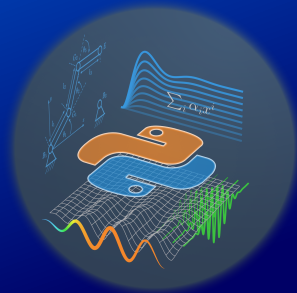
Carsten Knoll¹, Amine Othmane²

TUD, Chair of Fundamentals of Electrical Engineering; ² Univ. d. Saarl., Chair of Modeling and Simulation

PySaar 2024 – Python Course for Engineering

Saarbrücken, 2024-04-02 – 2024-04-04

<https://python-fuer-ingenieure.de/pysaar2024>



Sprache / Language

The material of this course is in English.
Explanations can be given in English or German.

If there is a language barrier: **please ask!**

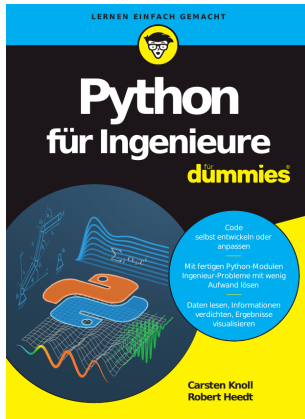
Wenn Sie etwas nicht verstehen: **Bitte fragen!**

Self Introduction: Carsten Knoll

- Postdoc at [Chair of Fundamentals of Electrical Engineering](#)
 - Research: Explainable AI, Semantic Technology, Control Theory
- Co-Founder of:
 - [Hochschulgruppe für Freie Software und Freies Wissen](#)
 - [Bits und Bäume Dresden](#)
 - [Konstruktive Digitale Diskussionskultur](#)
- First experience with Python in 2004, active usage since 2008



Book (currently German only)



<https://python-fuer-ingenieure.de>
(Every code example from the code is available. One Notebook per chapter!)

Why Python? (1)

Python as Programming language

- Clear, readable syntax (little "overhead")
- Object-oriented, procedural, functionally programmable
- Useful built-in data types (`list`, `tuple`, `dict`, `set`, ...)
- Easy modularization (`import this`)
- Good error management (exceptions)
- Extensive standard library
- Easy integration of external code (C, C++, Fortran)

Why Python? (1)

Python as Programming language

- Clear, readable syntax (little "overhead")
- Object-oriented, procedural, functionally programmable
- Useful built-in data types (`list`, `tuple`, `dict`, `set`, ...)
- Easy modularization (`import this`)
- Good error management (exceptions)
- Extensive standard library
- Easy integration of external code (C, C++, Fortran)

⇒

- Easy to learn
- Problem oriented (powerful and flexible)
- Motivation potential ↗, frustration potential ↘

Also: cross-platform / free and open source / large & active community

Why Python? (2)

Python as a tool for engineers:

- Basic algorithms
 - Symbolic calculation (derive, integrate, solve equations, ...)
 - Numerical calculation (lin. algebra, DGLn, optimization, ...)
 - Visualization (2D, 3D, in publication quality)
-
- Machine Learning
 - Communication with external devices (RS232, GPIB, ...)
 - Graphical User Interface (GUI)
 - Web Applications
 - Parallelization
 - Formal Knowledge Representation

Why Python? (2)

Python as a tool for engineers:

- Basic algorithms
- Symbolic calculation (derive, integrate, solve equations, ...)
- Numerical calculation (lin. algebra, DGLn, optimization, ...)
- Visualization (2D, 3D, in publication quality)

} prospective
course
content

- Machine Learning
- Communication with external devices (RS232, GPIB, ...)
- Graphical User Interface (GUI)
- Web Applications
- Parallelization
- Formal Knowledge Representation

⇒ Strengthened "research competence" (bachelor's/master/PhD theses, ...)

Didactic concept: Programming is learned by *actively* reading and writing code

3-day-course

- Learn new content
- Solve Exercises (with given source code fragments and additional help)

Didactic concept: Programming is learned by *actively* reading and writing code

3-day-course

- Learn new content
- Solve Exercises (with given source code fragments and additional help)

Programming project (manageable programming task)

- Completion time until \approx May 2024
- Your own topic suggestions welcome

Didactic concept: Programming is learned by *actively* reading and writing code

3-day-course

- Learn new content
- Solve Exercises (with given source code fragments and additional help)

Programming project (manageable programming task)

- Completion time until \approx May 2024

→ Your own topic suggestions welcome

Online consultations (see also <https://yopad.eu/p/tud-pythonkurs-365days>)

- For asking concrete questions about your project

Preparation and Installation

- **Recommendation** Anaconda distribution, installed via the miniconda installer:
<https://docs.conda.io/en/latest/miniconda.html>
- **Assumption:** You can open a command line window (="Console" = "Terminal") with the Conda environment enabled ("Anaconda prompt"):
- Install relevant packages with

```
# - Installation of auxiliary packages  
pip install numpy scipy matplotlib notebook ipywidgets symbtools ipydx  
pip install spyder
```

- We use Python ≥ 3.8

Preparation and Installation

- **Recommendation** Anaconda distribution, installed via the miniconda installer:
<https://docs.conda.io/en/latest/miniconda.html>
- **Assumption:** You can open a command line window ("Console" = "Terminal") with the Conda environment enabled ("Anaconda prompt"):
- Install relevant packages with

```
# - Installation of auxiliary packages  
pip install numpy scipy matplotlib notebook ipywidgets symbtools ipyindex  
pip install spyder
```

- We use Python ≥ 3.8
- Python 3.x is **not** 100% backward compatible (e. B. `print "hello"` \rightarrow `print("hello")`)
⚠ You can still find a lot of 2.x code on the net, e.g. the old german screencasts of this course

4 Ways to Start Python

- default interpreter (interactive mode)
 - command: `python` or `python3`
 - can execute any Python command; (might be useful for basic testing or as a calculator)

4 Ways to Start Python

- default interpreter (interactive mode)
 - command: `python` or `python3`
 - can execute any Python command; (might be useful for basic testing or as a calculator)
- IPython (“Enhanced Interactive Python” → **clearly** better)
 - command: `ipython`
 - tab completion, intelligent history, dynamic object info (with `?` or `??`), colored error messages

4 Ways to Start Python

- default interpreter (interactive mode)
 - command: `python` or `python3`
 - can execute any Python command; (might be useful for basic testing or as a calculator)
- IPython (“Enhanced Interactive Python” → **clearly** better)
 - command: `ipython`
 - tab completion, intelligent history, dynamic object info (with `?` or `??`), colored error messages
- Integrated Development Environment (e.g. “spyder”, “vs code”, “pycharm”)
 - command `spyder3`
 - customized text editor
 - Significantly more programming relevant functions

4 Ways to Start Python

- default interpreter (interactive mode)
 - command: `python` or `python3`
 - can execute any Python command; (might be useful for basic testing or as a calculator)
- IPython (“Enhanced Interactive Python” → **clearly** better)
 - command: `ipython`
 - tab completion, intelligent history, dynamic object info (with `?` or `??`), colored error messages
- Integrated Development Environment (e.g. “spyder”, “vs code”, “pycharm”)
 - command `spyder3`
 - customized text editor
 - Significantly more programming relevant functions
- Jupyter Notebook (with Python kernel)
 - command: `cd jupyter notebook .` (start the notebook server in current working directory)
 - backend: (local) web server; frontend: Interactive document in browser
 - notebooks combine source code, program output and documentation (incl. \LaTeX formulas)

Jupyter

Key keyboard commands



command mode (Esc to enable)

- Shift-Return - execute cell, activate next one
- h - show keyboard commands
- m - change cell type to "markdown"
- y - change cell type to "code"
- a - new cell above
- b - new cell below

edit mode (enter to activate)

- Shift-Return - execute cell, activate next one
- Tab - Autocomplete or indentation
- Shift-Tab - Remove indentation
- Ctrl-Z - Undo

Jupyter

Key keyboard commands



command mode (Esc to enable)

- Shift-Return - execute cell, activate next one
- h - show keyboard commands
- m - change cell type to "markdown"
- y - change cell type to "code"
- a - new cell above
- b - new cell below

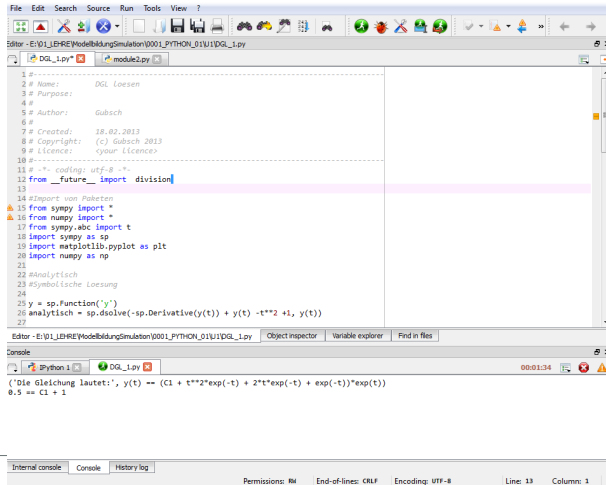
edit mode (enter to activate)

- Shift-Return - execute cell, activate next one
- Tab - Autocomplete or indentation
- Shift-Tab - Remove indentation
- Ctrl-Z - Undo

→ Start the server in the current directory: `jupyter notebook ./`

→ Trying `example-notebook1.ipynb`

Spyder Integrated Development Environment (IDE)



The screenshot displays the Spyder IDE interface. The top menu bar includes File, Edit, Search, Source, Run, Tools, and View. Below the menu is a toolbar with various icons for file operations, editing, and running code. The main editor window shows a Python script with the following content:

```
1 #
2 # Name:      DGL Loesen
3 # Purpose:
4 #
5 # Author:    Gubsch
6 #
7 # Created:   18.02.2013
8 # Copyright: (c) Gubsch 2013
9 # Licence:   <your licence>
10 #-----
11 # -*- coding: utf-8 -*-
12 from __future__ import division
13
14 #Import von Paketen
15 from sympy import *
16 from numpy import *
17 from sympy.abc import t
18 import sympy as sp
19 import matplotlib.pyplot as plt
20 import numpy as np
21
22 #Analytisch
23 #Symbolische Loesung
24
25 y = sp.Function('y')
26 analytisch = sp.solve(-sp.Derivative(y(t)) + y(t) - t**2 + 1, y(t))
27
```

Below the editor is a console window showing the output of the script:

```
('Die Gleichung lautet:', y(t) == (C1 + t**2*exp(-t) + 2*t*exp(-t) + exp(-t))*exp(t))
0.5 == C1 + 1
```

The bottom status bar shows the following information: Internal console, Console, History log, Permissions: RM, End-of-lines: CRLF, Encoding: UTF-8, Line: 13, Column: 1.

Code Example

Listing: 01_a_hello-world.py

```
import math
print("Hello World")
a = 10
b = 20.5
c = a + b + 3**2
print(math.sqrt(c))

while True: # start infinite loop
    x = input("Your name? ") # returns a str-object
    if x == "q":
        break # finish loop
    print("Hello ", x)
```

Code Example

Listing: 01_a_hello-world.py

```
import math
print("Hello World")
a = 10
b = 20.5
c = a + b + 3**2
print(math.sqrt(c))

while True: # start infinite loop
    x = input("Your name? ") # returns a str-object
    if x == "q":
        break # finish loop
    print("Hello ", x)
```

- Indentations have syntactical meaning!
- de facto standard: 4 spaces. (in editors: Blockwise with <TAB>(→) and <SHIFT+TAB> (←)).

Code Example

Listing: 01_a_hello-world.py

```
import math
print("Hello World")
a = 10
b = 20.5
c = a + b + 3**2
print(math.sqrt(c))

while True: # start infinite loop
    x = input("Your name? ") # returns a str-object
    if x == "q":
        break # finish loop
    print("Hello ", x)
```

- Indentations have syntactical meaning!
- de facto standard: 4 spaces. (in editors: Blockwise with <TAB>(→) and <SHIFT+TAB> (←)).

IPython (Interactive Python)

- *Interactive* working very useful when ...
 - “exploring” new modules, features, ...
 - searching for bugs
- IPython = “Interactive Python” = improved Python shell
 - history
 - auto-completion (<TAB>)
 - simple help (<?> and <??>)
 - colors, “magic” commands (e.g. %time), ...

IPython (Interactive Python)

- *Interactive* working very useful when ...
 - “exploring” new modules, features, ...
 - searching for bugs
- IPython = “Interactive Python” = improved Python shell
 - history
 - auto-completion (<TAB>)
 - simple help (<?> and <??>)
 - colors, “magic” commands (e.g. %time), ...
- can also be embedded in your own programs:

```
from ipydx import IPS
...
x = "abcdefg"
IPS()
...
```

IPython (Interactive Python)

- *Interactive* working very useful when ...
 - “exploring” new modules, features, ...
 - searching for bugs
- IPython = “Interactive Python” = improved Python shell
 - history
 - auto-completion (<TAB>)
 - simple help (<?> and <??>)
 - colors, “magic” commands (e.g. %time), ...
- can also be embedded in your own programs:

```
from ipytext import IPS
...
x = "abcdefg"
IPS()
...
```

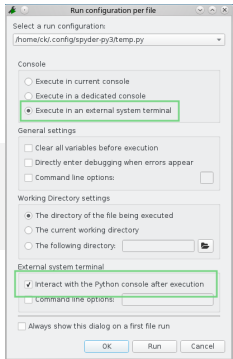
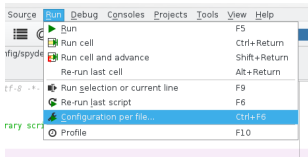
- integration in Spyder not so good

IPython (Interactive Python)

- *Interactive* working very useful when ...
 - “exploring” new modules, features, ...
 - searching for bugs
- IPython = “Interactive Python” = improved Python shell
 - history
 - auto-completion (<TAB>)
 - simple help (<?> and <??>)
 - colors, “magic” commands (e.g. %time), ...
- can also be embedded in your own programs:

```
from ipyindex import IPS
...
x = "abcdefg"
IPS()
...
```

- integration in Spyder not so good
→ Start scripts in external shell



Excercise: Embedded IPython

Listing: 01_b_ipython1.py.py

```
import math

# import embedded shell
from ipydx import IPS

a = 10
b = 20.5
c = a + b + 3**2
d = math.sqrt(c)

# run embedded shell
IPS()
# try: math.sqrt?, math.s<TAB>, history (up, down), %magic
# exit with CTRL-D
```

Intermediate Summary

What we talked about

- Course organization
- Python Shell (command line)
- Jupyter notebook
- Spyder (IDE)



Hasty Overview of Python Syntax and Datatypes

+ some nice features of the standard library

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 $15 = 6 \cdot 2 + \mathbf{3}$)

Comparison operations

$x == y$
 $x != y$
 $x < y$
 $x <= y$
 $x > y$
 $x >= y$

Strings (objects of type str)

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

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

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

String Formating (modern)

- New syntax (since Python3.6): “f-strings” → use code expressions *inside* the string

```
a = "World"
f"Hello {a}"    # use variables

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}"

pi = 3.141592653589793
f"{pi:.4f}"    # round to 4 decimal places

f"value of {some_variable=}"    # insert variable name and value
# useful for debugging (type variable name only once)
```

- More info: <https://docs.python.org/3/tutorial/inputoutput.html>
- Important methods of class `str`:
`a.index`, `.replace`, `.split`, `.find`, `.join`, `.startswith`, `.endswith`, ...

String Formating (old)

- General Syntax

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

- Examples

```
>>> a = 'H'  
>>> b = 'ello World'  
>>> "{}{}! {}{0} ".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'
```

- Even older printf-style string formatting (see [docs](#)):

```
>>> "pi is approximately %.4f." % 3.141592653589793  
'pi is approximately 3.1416.'
```

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

Syntax for unpacking (nested) sequences:

```
1 a, b, c = [10, 20, 30]
2 a, (b, c), d = [10, ["x", "y"], 20]
```

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`

- Important subclasses:

```
from collections import defaultdict
```

```
from collections import Counter
```

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("Bavaria") # -> None
```

```
# with default value  
>>> d.get("Bavaria", "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 in Python (1)

- encapsulate recurring subtasks
- deal with complexity
- common mistakes:
 - forgot colon (syntax error)
 - `return` forgotten (→ `None` is returned)

```
def donothing():  
    pass # dummy statement (for Indentation)  
  
# Call a function without arguments  
print(donothing()) # -> None  
  
def square1(z):  
    """Squaring a number""" # Docstring (=built-in doc)  
    return z**2  
  
square1(7) # -> 49  
square1(7-12) # -> 25
```

Global and Local Variables

```
def square2(z):  
    x = z**2 # local variable (because write access)  
    print(x)  
    return x
```

```
x, a = 5, 3 # "unpacking" of tuple (5, 3)  
square2(a) # -> 9  
square2(x) # -> 25  
print(x) # -> 5
```

```
def square3(z):  
    print(x) # here x is a global variable (only read access)  
    return z**2
```

```
def square4(z):  
    print(x) # Error (local variable not yet known)  
    x = z**2 # write access -> x must be local variable  
    return x
```

≡ keywords `global` and `nonlocal` ([Explanation](#); but in general not recommended)

Default Arguments

```
def square5(z=8):  
    return z**2  
  
square5() # -> 64  
square5(2.5) # -> 6.25
```


Default Arguments

```
def square5(z=8):  
    return z**2
```

```
square5() # -> 64
```

```
square5(2.5) # -> 6.25
```

```
def square_sum(a, b=0):  
    return a**2 + b
```

```
square_sum(10) # -> 100
```

```
square_sum(10, 3) # -> 103
```

Default Arguments

```
def square5(z=8):  
    return z**2
```

```
square5() # -> 64  
square5(2.5) # -> 6.25
```

```
def square_sum(a, b=0):  
    return a**2 + b
```

```
square_sum(10) # -> 100  
square_sum(10, 3) # -> 103
```

```
# explicit naming of the arguments ("keyword args")  
square_sum(b=-3, a=10) # -> 97 (here: order does not matter)
```

explicit naming helpful for functions with many arguments

Arbitrary Number of Arguments (*args, **kwargs)

```
# positional arguments
def mysum(*args):                # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
mysum(5, 20, 3) # -> 28
```

Arbitrary Number of Arguments (*args, **kwargs)

```
# positional arguments
def mysum(*args):                # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
mysum(5, 20, 3) # -> 28

numbers = [7, 4, -3, 15]
mysum(*numbers) # -> 23    # unpacking of the sequence inside the call
```

Arbitrary Number of Arguments (*args, **kwargs)

```
# positional arguments
def mysum(*args):                # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
mysum(5, 20, 3) # -> 28
```

```
numbers = [7, 4, -3, 15]
mysum(*numbers) # -> 23    # unpacking of the sequence inside the call
```

```
# keyword args
def sentence(**kwargs):          # the name `kwargs` is just a convention
    print( type(kwargs) ) # -> dict
    for key, value in kwargs.items():
        print( f"The {key} is {value}." )

sentence(water="cold") # -> The water is cold.
sentence(u=8.2) # -> The u is 8.2.
```

Arbitrary Number of Arguments (*args, **kwargs)

```
# positional arguments
def mysum(*args):                # the name `args` is just a convention
    print( type(args) ) # -> tuple
    s = 0
    for x in args:
        s += x
    return s
mysum(5, 20, 3) # -> 28
```

```
numbers = [7, 4, -3, 15]
mysum(*numbers) # -> 23    # unpacking of the sequence inside the call
```

```
# keyword args
def sentence(**kwargs):          # the name `kwargs` is just a convention
    print( type(kwargs) ) # -> dict
    for key, value in kwargs.items():
        print( f"The {key} is {value}." )

sentence(water="cold") # -> The water is cold.
sentence(u=8.2) # -> The u is 8.2.
```

```
def complex_function(a, b, c, *args, **kwargs):
    ... # combination is possible
```

Functions as Objects → scoping

```
def plus(a, b):  
    return a + b  
  
print( type(plus) ) # -> function  
z = plus # assignment (dt: Zuweisung) (no function call!)  
print( type(z) ) # -> function  
z == plus # -> True  
z is plus # -> True  
z(2, 3) # -> 5
```

Functions as Objects → scoping

```
def plus(a, b):  
    return a + b  
  
print( type(plus) ) # -> function  
z = plus # assignment (dt: Zuweisung) (no function call!)  
print( type(z) ) # -> function  
z == plus # -> True  
z is plus # -> True  
z(2, 3) # -> 5  
  
# factory functions ("closures")  
def factory(q):  
    b = q # b is local here  
    def add(a):  
        return b + a # b is "global" here (from outer namespace)  
    return add # an object of type `function` is returned
```


Functions as Objects → scoping

```
def plus(a, b):  
    return a + b
```

```
print( type(plus) ) # -> function  
z = plus # assignment (dt: Zuweisung) (no function call!)  
print( type(z) ) # -> function  
z == plus # -> True  
z is plus # -> True  
z(2, 3) # -> 5
```

```
# factory functions ("closures")
```

```
def factory(q):  
    b = q          # b is local here  
    def add(a):  
        return b + a # b is "global" here (from outer namespace)  
    return add # an object of type `function` is returned
```

```
# add is unknown here
```

```
p1 = factory(7)  
p2 = factory(1.5)  
p1(3) # -> 10  
p2(3) # -> 4.5  
factory("The sun is ")("shining.") # evaluation without assignment
```

Argument- and Type-checking

- very flexible function calls
- frequent source of errors: wrong arguments (values or types)
- type checking often useful (effort-benefit consideration)

```
def some_function(sentence, number1, number2, a_list):  
  
    if not type(sentence) == str: # direct equality checking (not recommended)  
        print("Type error: sentence")  
        exit()  
  
    if not isinstance(number1, int):  
        print("Type error: number1")  
        exit()  
  
    if not isinstance(number2, (int, float)):  
        print("Type error: number2")  
        exit()
```

Argument- and Type-checking

- very flexible function calls
- frequent source of errors: wrong arguments (values or types)
- type checking often useful (effort-benefit consideration)

```
def some_function(sentence, number1, number2, a_list):  
  
    if not type(sentence) == str: # direct equality checking (not recommended)  
        print("Type error: sentence")  
        exit()  
  
    if not isinstance(number1, int):  
        print("Type error: number1")  
        exit()  
  
    if not isinstance(number2, (int, float)):  
        print("Type error: number2")  
        exit()
```

Recommended: compact variant in one line with keyword `assert`:

```
#  
assert isinstance(number1, int) and number1 > 0  
# -> raises an AssertionError if condition is not fulfilled  
...
```

Functions vs. Methods

- Python supports
 - “Procedural Programming” (defining and calling functions)
 - “Functional Programming” (anonymous functions, recursive functions)
 - “Object Oriented Programming” (classes, inheritance, instances)

Functions vs. Methods

- Python supports
 - “Procedural Programming” (defining and calling functions)
 - “Functional Programming” (anonymous functions, recursive functions)
 - “Object Oriented Programming” (classes, inheritance, instances)
- Deep understanding of OOP: difficult
- Many features implemented via OOP → nevertheless easy to use
- Example:

```
city = "Saarbrücken"  
# call some methods of the str object  
s1 = city.upper()    # -> "SAARBRÜCKEN"  
s2 = city.lower()    # -> "saarbrücken"  
b1 = city.startswith("Saar") # -> True
```

- So called *method*: Function, which is attached to an object
- Object is passed as implicit first argument (usually called `self`)

Docstrings

- Strongly recommended:
 - Write documentation and code in the same file and (almost) simultaneously
- Docstrings $\hat{=}$ docs available from the program)
- Allows for automatic creation of nice HTML doc with [Sphinx](#)

```
def calculate(x, mode="normal"):
    """
    Short description: A functions that does something.

    :param x:      main argument (float)
    :param mode:   mode of the algorithm (str)
                   (e.g. "normal" or "reverse")

    :rtype:        result of complicated formula (float)

    After the description of the parameters, typically
    there is more information on the documented function.
    """

    y = x**2*some_other_function(x, mode) + complicated_formula(x)
    return y
```

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

Recommended: using the *context manager* syntax (`with`-statement)

Listing: 01_d_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

Recommended: using the *context manager* syntax (`with`-statement)

Listing: 01_d_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`
- ...

Allgemeine Tipps

- „1st make it run! 2nd make it right! 3rd make it fast“

Allgemeine Tipps

- „1st make it run! 2nd make it right! 3rd make it fast“
- „DRY: Dont repeat yourself“
 - Copy-Paste ist eine typische Fehlerquelle

Allgemeine Tipps

- „1st make it run! 2nd make it right! 3rd make it fast“
- „DRY: Dont repeat yourself“
 - Copy-Paste ist eine typische Fehlerquelle
- Mindestens 50% der Zeit zum Debuggen einplanen

Allgemeine Tipps

- „1st make it run! 2nd make it right! 3rd make it fast“
- „DRY: Dont repeat yourself“
 - Copy-Paste ist eine typische Fehlerquelle
- Mindestens 50% der Zeit zum Debuggen einplanen
- Manchmal kann es sinnvoll sein bewusst Fehler einzufügen:

1

1/0

Allgemeine Tipps

- „1st make it run! 2nd make it right! 3rd make it fast“
- „DRY: Dont repeat yourself“
 - Copy-Paste ist eine typische Fehlerquelle
- Mindestens 50% der Zeit zum Debuggen einplanen
- Manchmal kann es sinnvoll sein bewusst Fehler einzufügen:

1

1/0

- Für (weiter) Fortgeschrittene:
 - Versionsverwaltung mit `git`
 - `unittesting`
 - Style-Guide: PEP8 + automatisches „code-linting“

Numerical Computation with Python (Numpy + Scipy)

- Objective of this part:
 - rough overview of the possibilities
 - no completeness intended
- Structure:
 - Numpy arrays
 - Numpy (basic numerics)
 - Scipy (application oriented numerics)

Numpy arrays (I)

- So far the following container classes (“sequences”) have been presented:

```
list: [1, 2, 3], tuple: (1, 2, 3), str: "1, 2, 3"
```

- Unsuitable for calculations

```
# useful for strings
linie = "-" * 10 # -> "-.-.-.-.-.-.-.-.-.-"

# not useful for numerical calculations
numbers = [3, 4, 5]
res1 = numbers*2 # -> [3, 4, 5, 3, 4, 5]

# not possible at all
res2 = numbers*1.5
res3 = numbers**2
```

Numpy arrays (II)

- For Numpy arrays: Calculations are performed **element-wise**

```
from numpy import array

numbers = [3.0, 4.0, 5.0] # list with float objects

x = array(numbers)
res1 = x*1.5 # -> array([ 4.5,  6. ,  7.5])
res2 = x**2 # -> array([ 9., 16., 25.])
res3 = res1 - res2 # -> array([ -4.5., -10., -18.5])
```

- arrays can have n dimensions

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

print(arr_2d.shape) # -> (2, 3)
```

Numpy arrays (III)

Further possibilities to create, `array` objects:

Listing: 02_a_array_creation.py

```
import numpy as np
x0 = np.arange(10) # like range(...) but with arrays
x1 = np.linspace(-10, 10, 200)
    # 200 values: array([-10., -9.899497, ..., 10])
x2 = np.logspace(1, 100, 500) # 500 values, always same ratio

x3 = np.zeros(10) # see also: np.ones(...)
x4 = np.zeros( (3, 5) ) # Caution: takes only one argument! (= shape)

x5 = np.eye(4) # 4x4 unity matrix
x6 = np.diag( (4, 3.5, 23) ) # 3x3 diagonal matrix with specified diagonal elements

x7 = np.random.rand(5) # array with 5 random numbers (each between 0 and 1)
x8 = np.random.rand(4, 2) # array with 8 random numbers and shape = (4, 2)

from numpy import r_, c_ # "index tricks" for rows and columns
x9 = r_[6, 5, 4.2] # array([ 6.,  5.,  4.2])
x10 = r_[x9, -84, x3[:2]] # array([ 6.,  5.,  4.2, -84,  0.,  1.])
x11 = c_[x9, x6, x5[:1, :]] # stacking in column direction

assert x11.shape == (3, 8)
```

Slicing and Broadcasting

- slicing: address values inside of an array
- analogous to other sequences: `x[start:stop:step]` ;
- first element: `x[0]`
- dimensions are separated by a comma
- negative indices count from the end backwards

Listing: 02_b_slicing.py

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

x1 = a[3] # element with index 3 (-> 6.0)
x2 = a[3:6] # elements 3 to 5 -> array([ 6.,  8., 10.])
x3 = a[-3:] # from 3rd-last element to the end -> array([30., 32., 34.])
# Caution: a, x2 and x3 share the data (they are only "views" to the data)
a[-2:] *= -1 # change the data in a and observe the change in x3:
print(x3) # -> [-30., -32., -34.]

# for 2d arrays: first index -> row; second index column; separator: comma
y1 = A[:, 0] # first column of A (index 0)
y2 = A[1, :3] # first three elements of the second column (index 1)
```

- ∃ “broadcasting”: automatically adapt the shape (e.g. array + scalar)


Broadcasting (Theory)

(optional slide)

- $\hat{=}$ Numpy's handling of arrays with different shapes (for element-wise calculations)
- trivial example: x (2d array) + y (float) $\rightarrow y$ is „blown up“ to match the size of x
- Nontrivial example: 2d array + 1d array = ?
- Rule: The size along the **last axis** of each operand
 - a) must be the same or
 - b) one of those sizes must be one.

- Two examples:

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

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

- Common error message:

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

\rightarrow Recommendation: [read the docs](#) and experiment interactively

- see also: `02_c_broadcasting_example.py`

Broadcasting Example

Listing: 02_c_broadcasting_example.py

```
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, "Caution, the next statements result in an error.")
time.sleep(2)

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

Reminder: $E*b_13$ is **not** matrix vector multiplication (see also slide 53).

Numpy functions

```
import numpy as np
from numpy import sin, pi # Save typing work

t = np.linspace(0, 2*pi, 1000)

x = sin(t) # analog: cos, exp, sqrt, log, log2, ...
xd = np.diff(x) # differentiate numerically
# Caution: xd has one entry less!
X = np.cumsum(x) # integrate numerically (cumulative summation)
```

- No python loops needed → Numpy functions very are fast (like C code)
- Comparison operations:

```
# element-wise:
y1 = np.arange(3) >= 2
    # -> array([False, False, True], dtype=bool)
# for complete array:
y2 = np.all( np.arange(3) >= 0) # -> True
y3 = np.any( np.arange(3) < 0) # -> False
```

Further Numpy Functions

(optional slide)

- `min` , `max` , `argmin` , `argmax` , `sum` (→ scalar values)
- `abs` , `real` , `imag` (→ arrays)
- change shape: `.T` (transpose), `reshape` , `flatten` , `vstack` , `hstack`

linear algebra:

- matrix multiplication:
 - `a@b` (recommended, @-operator was introduced in Python 3.5)
 - `dot(a, b)` (old and safe way)
 - `np.matrix(a)*np.matrix(b)` (not recommended by me)
- Submodule: `numpy.linalg` :
 - `det` , `inv` , `solve` (solve linear equation system), `eig` (eigenvalues and vectors),
 - `pinv` (pseudo inverse), `svd` (singular value decomposition), ...

Scipy

- Own package, based on Numpy
- Offers functionality for
 - Data input & output (e.g. mat format (Matlab))
 - Physical constants
 - More linear algebra
 - Signal processing (Fourier transform, filter, ...)
 - Statistics
 - Optimization
 - Interpolation
 - Numerical Integration (“Simulation”)

scipy.optimize (1)

- Very useful: `fsolve` (docs) and `minimize` (docs)
- `fsolve` : find roots (zeros) of a scalar (nonlinear) function $f : \mathbb{R} \rightarrow \mathbb{R}$ or of a vector function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$
- initial estimation of solution is necessary
- example: approximate solution of the nonlinear equation

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

Listing: 02_f_fsolve_example.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])
# check:
print(sol, sol + 2.3*np.cos(sol)) # -> [-0.72363261] [1.]
```

scipy.optimize (2)

- `minimize` : finds minimum of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ (docs)
- interface to various minimization algorithms.
- allows many optional arguments (e.g. specification of bounds or (in)equation constraints).
- can also solve equations, by minimizing the quadratic **equation error**, see following example (same mathematical problem as before, but different solution approach)

Listing: 02_g_minimize_example.py

```
import numpy as np
from scipy import optimize

def fnc2(x):
    return (x + 2.3*np.cos(x) - 1)**2 # quadratic equation error

res = optimize.minimize(fnc2, 0) # Optimization with initial estimate 0
# check:
print(res.x, res.x + 2.3*np.cos(res.x)) # -> [-0.7236326] [1.00000004]

# now with limits and with changed start estimation -> other solution
res = optimize.minimize(fnc2, 0.5, bounds=[(0, 3)])
# check:
print(res.x, res.x + 2.3*np.cos(res.x)) # -> [2.03999505] [1.00000003]
```

Num. Integration of ODEs (Theory)

- “simulation” = numerical solution of differential equations
- **O**rdinary **D**ifferential **E**quations) in state space representation: $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
 - time derivative $\dot{\mathbf{z}}$ of the state vector \mathbf{z} depends on the state itself (and on t)
- solution of the ODE: time development $\mathbf{z}(t)$ (depends on initial state $\mathbf{z}(0)$)
→ “initial value problem” (IVP), (German: Anfangswertproblem)

Num. Integration of ODEs (Theory)

- “simulation” = numerical solution of differential equations
- **O**rdinary **D**ifferential **E**quations) in state space representation: $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
 - time derivative $\dot{\mathbf{z}}$ of the state vector \mathbf{z} depends on the state itself (and on t)
- solution of the ODE: time development $\mathbf{z}(t)$ (depends on initial state $\mathbf{z}(0)$)
→ “initial value problem” (IVP), (German: Anfangswertproblem)
- example: harmonic oscillator with ODE: $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- preparation: transformation to state space representation
(one ODE of 2nd order → two ODEs of 1st order):
State: $\mathbf{z} = (z_1, z_2)^T$ with $z_1 := y, z_2 := \dot{y} \rightarrow$ two ODEs:

$$\dot{z}_1 = z_2 \quad (\text{“definitional equation”})$$

$$\dot{z}_2 = -2\delta z_2 - \omega^2 z_1 \quad (= \ddot{y})$$

- \exists various integration algorithms (Euler, Runge-Kutta, ...)
- accessible via universal function: `scipy.integrate.solve_ivp` (docs)

Num. Integration of ODEs (Theory)

- “simulation” = numerical solution of differential equations
- **O**rdinary **D**ifferential **E**quations) in state space representation: $\dot{\mathbf{z}} = \mathbf{f}(\mathbf{z}, t)$
 - time derivative $\dot{\mathbf{z}}$ of the state vector \mathbf{z} depends on the state itself (and on t)
- solution of the ODE: time development $\mathbf{z}(t)$ (depends on initial state $\mathbf{z}(0)$)
→ “initial value problem” (IVP), (German: Anfangswertproblem)
- example: harmonic oscillator with ODE: $\ddot{y} + 2\delta\dot{y} + \omega^2 y = 0$
- preparation: transformation to state space representation
(one ODE of 2nd order → two ODEs of 1st order):
State: $\mathbf{z} = (z_1, z_2)^T$ with $z_1 := y, z_2 := \dot{y}$ → two ODEs:

$$\dot{z}_1 = z_2 \quad (\text{“definitional equation”})$$

$$\dot{z}_2 = -2\delta z_2 - \omega^2 z_1 \quad (= \ddot{y})$$

- \exists various integration algorithms (Euler, Runge-Kutta, ...)
- accessible via universal function: `scipy.integrate.solve_ivp` (docs)

detailed explanations in separate notebook: → [simulation_of_dynamical_systems.ipynb](#)

Num. Integration of ODEs (Application)

Listing: 02_d_solve_ivp_example.py

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

delta = .1
omega_2 = 2**2
def rhs(t, z):
    """ rhs means 'right hand side [function]' """
    # argument t must be present in the function head, but it can be ignored in the body
    z1, z2 = z # unpacking the state vector (array) into its two components
    z1_dot = z2
    z2_dot = -(2*delta*z2 + omega_2*z1)
    return [z1_dot, z2_dot]

tt = np.arange(0, 100, .01) # independent variable (time)
z0 = [10, 0] # initial state for z1 and z2 (=y, and y_dot)
res = solve_ivp(rhs, (tt[0], tt[-1]), z0, t_eval=tt) # calling the integration algorithm
zz = res.y # array with the time-development of the state
           # (rows: components; columns: time steps)

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

- function `rhs` is “ordinary” Python object

→ Can be passed as argument to another function (here: `solve_ivp`)

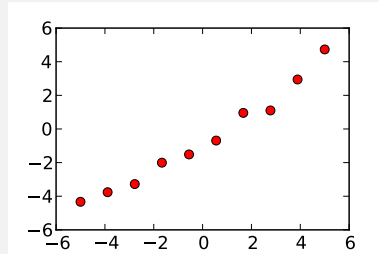
- note: function `odeint` is predecessor of `solve_ivp`.

main difference: argument order and return object

Regression, Interpolation

regression with `numpy.polyfit` ([docs](#)):

(optional slide)

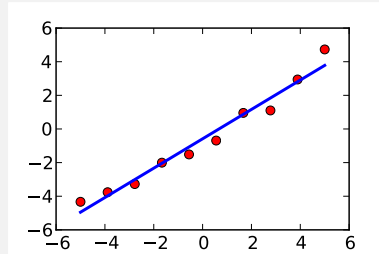


Regression, Interpolation

(optional slide)

regression with `numpy.polyfit` ([docs](#)):

- linear regression (blue)

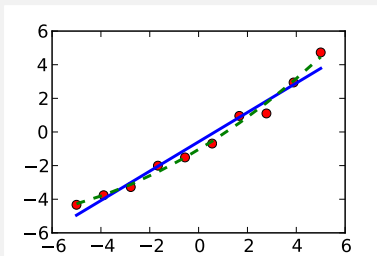


Regression, Interpolation

(optional slide)

regression with `numpy.polyfit` ([docs](#)):

- linear regression (blue)
- or higher order



Regression, Interpolation

(optional slide)

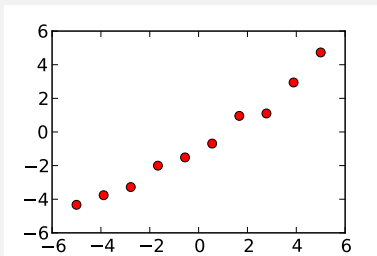
regression with `numpy.polyfit` ([docs](#)):

- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` ([docs](#)):

- piecewise polynomial (\rightarrow „Spline“)
- arbitrary order
(here: orders 0, 1, 2)



Regression, Interpolation

regression with `numpy.polyfit` ([docs](#)):

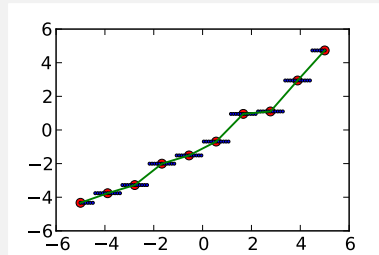
- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` ([docs](#)):

- piecewise polynomial (\rightarrow „Spline“)
- arbitrary order
(here: orders 0, 1, 2)

(optional slide)



Regression, Interpolation

regression with `numpy.polyfit` ([docs](#)):

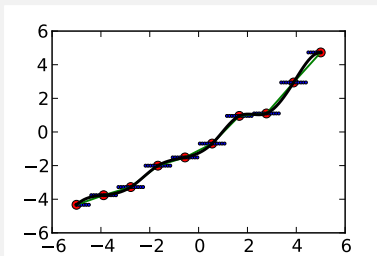
- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` ([docs](#)):

- piecewise polynomial (\rightarrow „Spline“)
- arbitrary order
(here: orders 0, 1, 2)

(optional slide)



Regression, Interpolation

(optional slide)

regression with `numpy.polyfit` ([docs](#)):

- linear regression (blue)
- or higher order

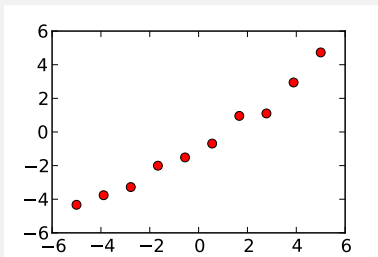
interpolation with

`scipy.interpolation.interp1d` ([docs](#)):

- piecewise polynomial (\rightarrow „Spline“)
- arbitrary order
(here: orders 0, 1, 2)

hybrid form with `scipy.interpolation.splrep` ([docs](#)):

- “smoothed spline” (smoothness adjustable via coefficient)



Regression, Interpolation

(optional slide)

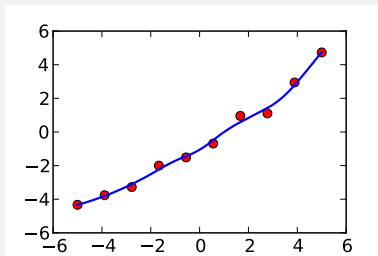
regression with `numpy.polyfit` ([docs](#)):

- linear regression (blue)
- or higher order

interpolation with

`scipy.interpolation.interp1d` ([docs](#)):

- piecewise polynomial (\rightarrow „Spline“)
- arbitrary order
(here: orders 0, 1, 2)



hybrid form with `scipy.interpolation.splrep` ([docs](#)):

- “smoothed spline” (smoothness adjustable via coefficient)

see also: `02_e_interp_example.py`

Regression, Interpolation

(optional slide)

regression with `numpy.polyfit` ([docs](#)):

- linear regression (blue)
- or higher order

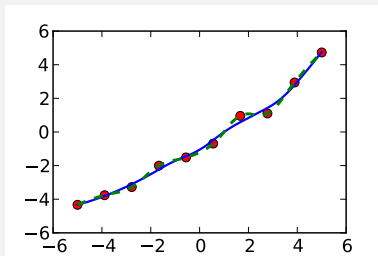
interpolation with

`scipy.interpolation.interp1d` ([docs](#)):

- piecewise polynomial (\rightarrow „Spline“)
- arbitrary order
(here: orders 0, 1, 2)

hybrid form with `scipy.interpolation.splrep` ([docs](#)):

- “smoothed spline” (smoothness adjustable via coefficient)



see also: `02_e_interp_example.py`

Intermediate Summary

- `numpy` arrays
- further `numpy` -functions
- `scipy` functions (numerical integration, optimization, regression, interpolation)

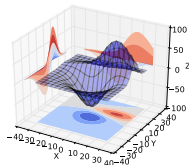
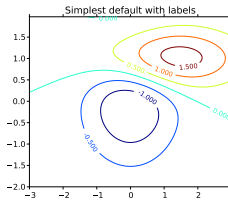
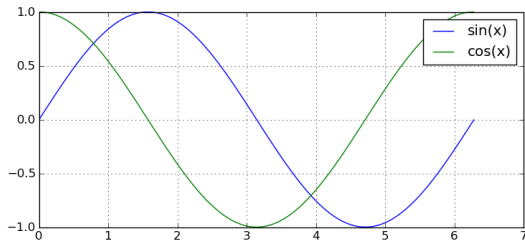
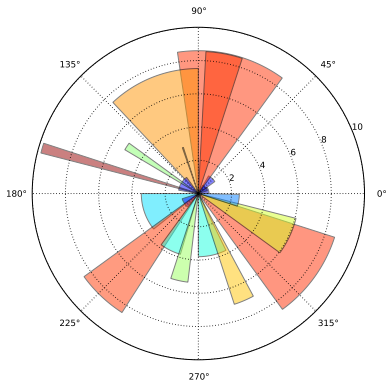
Links

- http://www.scipy.org/Tentative_NumPy_Tutorial
- 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 (extensive)

- <http://docs.scipy.org/doc/scipy/reference/> (tutorial + reference)
- <http://www.scipy.org/Cookbook>

Preview

- Visualization of measurement data, simulation results → better understanding
- Publication-ready graphics for theses, dissertations, scientific articles etc.



Package Overview



- Quasi-standard for 2d plotting with Python
- Also support for some 3d plotting
- Syntax based on Matlab (easy)
- Many plot functions and plot types
- High quality results
- Interactive (zooming, panning)
- Embedding of formulas and symbols
- \LaTeX interface for text and formulas in diagrams
- Saving of plots in as raster graphics (e.g. png) or vector graphics (e.g. pdf)
- Extensive documentation
- Performance:
acceptable, but not ideal for real-time visualization (fast alternative: <https://pyqtgraph.org/>)
- Huge example gallery: <https://matplotlib.org/stable/gallery/>

Simple Example

```
import numpy as np
import matplotlib.pyplot as plt

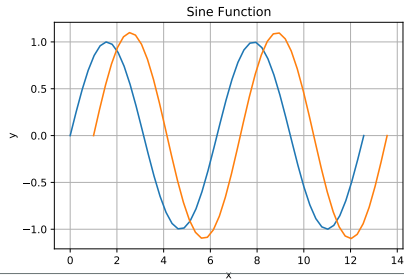
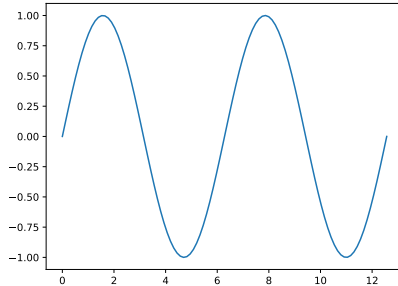
x = np.linspace(0, 4*np.pi, 50)
y = np.sin(x)

plt.plot(x, y)
plt.show()
```

Customize plot:

```
# 1. create the plot:
plt.plot(x, y)
plt.plot(x+1, y*1.1)

# 2. specify properties:
plt.title('Sine Function')
plt.grid(True)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Plot Function

`plot(...)` takes at least one positional argument:

`plot(y)` → plots sequence `y` over its indices

Common pattern: `plot(x, y, 'ro:')` → x-values, y-values, format string
(here: `:` → red, `o` → circle markers, `:` → dotted line),

For more better readability use keyword arguments (kwargs):

```
plt.plot(x, y, color='red', linestyle=':', marker='o', linewidth=2)
```

```
# with short kwargs:
```

```
plt.plot(x, y, c='red', ls=':', marker='o', lw=2)
```

Automatic color cycling and legend

```
plt.plot(x, np.sin(x), label='sin(x)') # light blue
plt.plot(x, np.cos(x), label='cos(x)') # orange
plt.legend() # automatically placed (if not specified by kwarg)
```

See also: docstring of plot function e.g. `plt.plot?` in Jupyter Notebook

Text and Styling

Formulas and symbols via \rightarrow \LaTeX -support

Tip: use raw strings like `r"abc"` to unbind the backslash from special meanings
(like `"\n"` \rightarrow new line character)

```
plt.plot(x[1:], y[1:]/x[1:], label=(r"$\frac{\sin(\phi)}{\phi}$"))  
plt.legend()
```

Fontsize of legend

```
plt.legend(fontsize=20)  # only this time  
# ...  
plt.rcParams['legend.fontsize'] = 18  # from now on
```

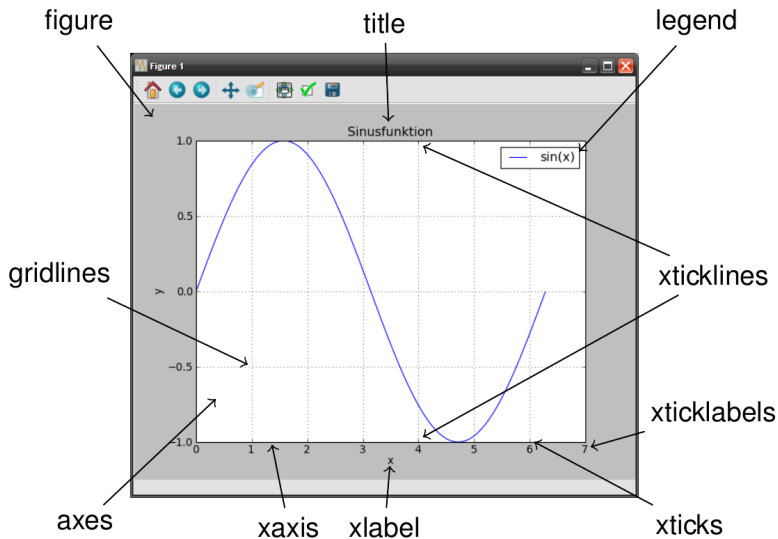
Fontsize of label

```
plt.xlabel('time [s]', fontsize=14)  # only this time  
# ...  
plt.rcParams['axes.labelsize'] = 10  # from now on
```

Change parameter of object after creation

```
line1, = plt.plot(x, y)  
plt.setp(line1, linewidth=3, color="purple")
```

Matplotlib Terminology



Example Script 1

Listing: example-code/03_a_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 # scaling factor millimeter -> inch
8 fig = plt.figure(figsize=(250*mm, 180*mm)) # specify size in mm
9
10 plt.plot(alpha, y, label=r'$\sin(\alpha)$')
11 plt.xlabel(r'$\alpha$ in rad')
12 plt.ylabel('$y$')
13 plt.title('Sine Function')
14 plt.legend() # add legend
15 plt.grid() # toggle grid
16
17 plt.savefig('test.pdf') # file format is inferred from ending (.pdf)
18 plt.show() # display the figure
```

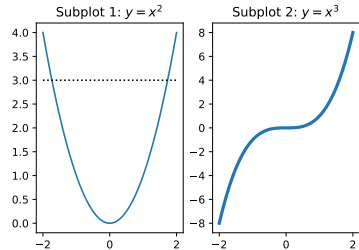
Recommendations:

- Visualization should be implemented separately from slow calculations (e.g. simulation)
- Visualization should be completely automated (no manual interactions)
- Reason: reproducibility; visualization will be created many times before it is ready

Example Script 2

Listing: example-code/03_b_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 # scaling factor millimeter -> inch
7 scale = 0.5 # for convenient proportional scaling
8 fs = np.array([250*mm, 180*mm])*scale # specify scaled size in mm
9
10 # small right margin
11 plt.rcParams['figure.subplot.right'] = .98
12
13 # subplots: 1 row, 2 columns
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") # dotted black horizontal line at bei y=3
23
24 plt.savefig('subplots.pdf') # file format is inferred from ending (.pdf)
25 plt.show() # display the figure
```



Where do I get help?

- Matplotlib is very big and complex → see docs:
<https://matplotlib.org/stable/users/index.html>
- Tip 1: Gallery (<https://matplotlib.org/stable/gallery/>)
Many examples (images and corresponding code)
- Tip 2: Axes class documentation
https://matplotlib.org/stable/api/axes_api.html
- All plot and draw functions are accessible via the `axes` class!
 - `ax.plot()` , `ax.bar()` , `ax.scatter()` , `ax.arrow()` , ...
 - Especially important: keyword arguments
- docstrings of objects and functions, e.g. `plt.plot?` in Jupyter
- cheatsheets in appendix or at <https://matplotlib.org/cheatsheets/>

Frequently Used Properties

```
ax = plt.gca() # get current axes object
```

- `ax.set_aspect('equal')` → aspect ratio 1:1
- `ax.set_xlim(0, 10)` → range of X axis values
- `ax.set_xticklabels(['a', 'b'])` → custom labels
- `ax.legend(loc=1)` → position of legend
- `ax.tick_params(**kwargs)` → optics of axis ticks

Discover configurable properties:

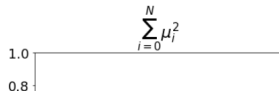
```
import ipyindex
ipyindex.dirsearch("size", plt.rcParams)
# or
ipyindex.dirsearch("tick", plt.rcParams)
```

More on L^AT_EX

Matplotlib comes with its own (reduced) L^AT_EX compiler

See examples above (`example-code/matplotlib2.py`)

or try `plt.title(r"$\sum_{i=0}^N \mu_i^2$")`

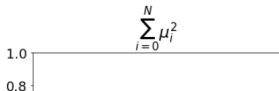


More on \LaTeX

Matplotlib comes with its own (reduced) \LaTeX compiler

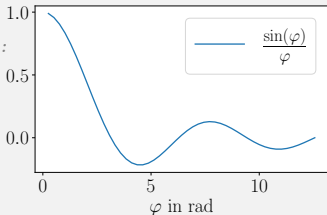
See examples above (`example-code/matplotlib2.py`)

or try `plt.title(r"$\sum_{i=0}^N \mu_i^2$")`



Alternatively, use local \LaTeX -installation (if present)

```
plt.rcParams["text.use_tex"] = True # activate LaTeX rendering
# optional but looks better:
plt.rcParams["font.family"] = "serif"
# the following function call is spread over multiple lines:
plt.plot(
    x[1:], y[1:]/x[1:],
    label=(r"$\frac{\sin(\varphi)}{\varphi}$")
)
plt.xlabel(r"$\varphi$ in rad")
plt.legend(); plt.show()
```



→ all strings will be rendered with \LaTeX

Matplotlib - interactive usage

- assumption until now: usage from a script (.py file)
- `plt.show()` opens a new window (or more)
- program is paused until window is closed
- figure can be manipulated interactively (zoom, pan, ...)

Matplotlib - interactive usage

- assumption until now: usage from a script (.py file)
- `plt.show()` opens a new window (or more)
- program is paused until window is closed
- figure can be manipulated interactively (zoom, pan, ...)
- \exists different behavior: `plt.ion()` # activate interactive mode
- program continues after figure is created
- figure can be manipulated by code after `plt.show()`

Matplotlib - interactive usage

- assumption until now: usage from a script (.py file)
- `plt.show()` opens a new window (or more)
- program is paused until window is closed
- figure can be manipulated interactively (zoom, pan, ...)
- \exists different behavior: `plt.ion()` # activate interactive mode
- program continues after figure is created
- figure can be manipulated by code after `plt.show()`
- Jupyter Notebook: ≥ 2 types of display modes (“backends”)
- activated via “magic commands” (beginning with `%`)

```
%matplotlib inline    # static image; good for exporting  
%matplotlib notebook  # interactive widget inside the notebook
```

Summary

- library for 2d-visualization: `matplotlib`
- `plt.plot` , `plt.show` , `plt.savefig` , `plt.rcParams` , ...
- quite complex (many options)
- best practice:
 - customize examples from <https://matplotlib.org/stable/gallery/>

Cheatsheet: Line Styles

∃ two possibilities: short: as format string: `plot(x, y, '-.')`

long: as keyword argument (with long or short key):

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

short form	long form	description	output
<code>''</code> or <code>''</code>	<code>None</code>	without line	
<code>'-'</code>	<code>'solid'</code>	solid line (default) (<i>Vorgabe</i>)	————
<code>'--'</code>	<code>'dashed'</code>	dashed line	-----
<code>'-.'</code>	<code>'dashdot'</code>	line with dashes and dots	-.-.-.-.
<code>':'</code>	<code>'dotted'</code>	dotted line

Cheatsheet: Marker

∃ two possibilities; short as format string: `plot(x, y, 'o')`

long as keyword argument: `plot(x, y, marker='h')`

Size with `markersize=10` or `ms=10` adjustable

Examples (there are more):

short form		description	output
' '	oder <code>None</code>	without marker (<i>default</i>)	
'.'		dot	●
'o'		circle	○
'D'		diamond	◇
'H'		hexagon	⬡
'+'		plus	+
's'		square	■

Cheatsheet: Colors

In format string, or as keyword argument

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

Gray: numeric value between 0 ($\hat{=}$ black) and 1 ($\hat{=}$ white) as `str` object:

```
plot(x, y, color='0.3')
```

more precise color specification:

- Hexadecimal notation (as in HTML/CSS): `color='#e3e6f7'`
- 3-tuple ($\hat{=}$ Red, Green, Blue): `color=(0.3, 0.8, 0.1)`
- 4-tuple ($\hat{=}$ RGB + Alpha (opacity)): `color=(0.3, 0.8, 0.1, 0.7) # 30% transparency`

Predefined Colors:

short form	long form
'b'	'blue'
'g'	'green'
'r'	'red'
'c'	'cyan'
'm'	'magenta'
'y'	'yellow'
'k'	'black'
'w'	'white'

Extensive cheatsheets:

<https://matplotlib.org/cheatsheets/>

Programming Paradigms (Overview)

- desired: reusability of already implemented functionality
- copy & paste?
 - frequent source of errors (forgetting some necessary changes)
 - later developments → changes in many places required → avoidable effort

→ problem can be addressed with different **paradigms**

Programming Paradigms (Overview)

- desired: reusability of already implemented functionality
- copy & paste?
 - frequent source of errors (forgetting some necessary changes)
 - later developments → changes in many places required → avoidable effort

→ problem can be addressed with different **paradigms**

- What is a programming paradigm?
 - set of rules for formal and structural code design
 - so far in this course: “procedural programming”
 - here: “object-oriented programming”
 - also existing: “functional programming” (Lisp ♥ *recursive* functions calls)
- What is it used for?
 - support for creation of **good** code
 - suggest/prioritize a certain approach
- No dogma!
 - paradigm application depending on concrete problem (and taste)
 - unlike e.g. Java, Python does not enforce a certain paradigm
 - in Python: combinations of paradigms possible

Object-Oriented Programming (OOP)

- description of a complex system as an interaction of **objects**
- objects consist of
 - data ("**attributes**")
 - associated functions ("**methods**")
- objects are **instances** of a **class**
i.e. an object is the concrete variable, the class is the data type

object-orientation is a large field

- enough details for a whole semester
- here: only clarify most important terms and principles

Simple Example

Suppose, you own a personal soccer ball



- The ball has the properties (attributes)
 - radius
 - material
 - color
 - ...
 - (nouns)

- The ball provides certain actions (methods)
 - throw ball (in direction (x, y, z))
 - roll ball (in direction x, y)
 - ...
 - (verbs)

Object Orientation in Python

Your personal soccer ball is a particular sphere

Listing: example-code/05_a_sphere.py

```
class Sphere():
    """A class modeling a spherical objects"""

    def __init__(self, radius, midpoint=(0, 0, 0)):
        """
        Initialization method. Automatically executed when an object is created.
        Corresponds (approximately) to the 'constructor' in other programming
        languages.
        """

        # set attributes:
        self.radius = radius
        self.midpoint = midpoint

    def calc_volume(self):
        r = self.radius
        return (4/3)*np.pi*(r**3)
```

Instantiation

Other balls are spheres, too:

```
# Definition of the class
class Sphere():
    def __init__(self, radius, midpoint=(0,0,0)):
        ...

# instantiation of the class (create variables of that type)
# (arguments are passed to the `__init__` method)
soccer_ball = Sphere(radius=21)
tennis_ball = Sphere(3)
rubber_ball = Sphere(1)

print("radius of soccer ball:", soccer_ball.radius) # access to attribute
V = tennis_ball.calc_volume() # access to method
```

- class `Sphere` only is the “construction plan” or “blueprint”
 - instantiation: creation of **concrete objects** according to the blueprint
 - each object gets its own memory section
- attribute values are independent of each other
(each `sphere` instance has its own radius, center, ...)
- each object has unique memory address (readable with `id(..)`)

Inheritance (in general)

- creation of a new class based on an existing one
- only limited analogy to biological inheritance
- typical case: inheritance from the abstract to the specific
- representation: `base class` \leftarrow `child class` (" \leftarrow " $\hat{=}$ "inherits from")
- example: `animal` \leftarrow `mamal` \leftarrow `dog`
- child class has all attributes/methods of base class
 - value/implementation can be overridden
- additional attributes/methods possible in child class

What is inheritance good for?

- sharing structure and code (attributes and methods)
 - reduces implementation effort
- documentation of similarities between classes

Inheritance (in Python)

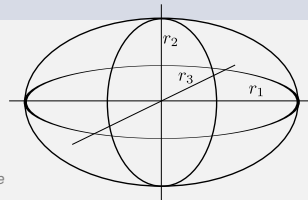
The sphere is a special kind of ellipsoid

```
class Ellipsoid():
    def __init__(self, r1, r2, r3):
        ...

class Sphere(Ellipsoid):
    def __init__(self, radius):

        # In the constructor of child class we call the
        # constructor of the parent class:

        # Sphere is a Ellipsoid where all radii are equal
        Ellipsoid.__init__(self, radius, radius, radius)
```



- class `Sphere` here is derived from class `Ellipsoid`
- attributes and methods are inherited (if not specified explicitly in the child class)
- constructor `__init__` is overridden so that it accepts only one radius

Inheritance Hierarchy

```
class GeometricObject: # (topmost) base class
    ...
class Cuboid(GeometricObject): # level 1 subclass
    ...
class Ellipsoid(GeometricObject): # level 1 subclass
    ...
class Sphere(Ellipsoid): # level 2 subclass
    ...
soccer_ball = Sphere(21) # instance
```

- base class `GeometricObject` implicitly is derived from `object` (builtin type)
- illustration with the help functions `isinstance(...)` und `issubclass(...)`

```
isinstance(soccer_ball, Sphere) # True
isinstance(soccer_ball, GeometricObject) # True
isinstance(soccer_ball, Cuboid) # False
issubclass(Sphere, Ellipsoid) # True
issubclass(Sphere, Cuboid) # False
```

Python-Speciality (1): self

- a **method** is a function belonging to an object
 - when executed, it must be known to which **instance** this method belong
- passing the instance as **implicit** first argument.
- Usually named (convention): `self`
 - in other words:
 - `self` is placeholder for the concrete instance at a time when it does not yet exist

Listing: exampl-code/05_b_self.py

```
# Note: the id(...) function returns a unique identity-number for each object.
# Same number means: same object.

class ClassA():
    def m1(self):
        print(id(self))

    def m2(x): # !! self argument missing
        print(x)

a = ClassA() # create an instance

a.m1() # no explicit argument (but implicitly a is passed)
print(id(a)) # this gives the same number

a.m2() # no error (corresponds to print(id(a)), because x takes the role of self)
a.m2(123) # Error: too many arguments passed (a (implicitly) and 123 (explicitly))
```

Speciality (2): everything is an object

- no “primitive data types” (as in Java or C++)
- everything* is an object (i. e. an instance of a class):
 - numbers (instances of `int`, `float`, `complex`, ...)
 - strings (instances of `str`, `bytes`, ...)
 - functions and classes:

```
class ClassA():  
    pass  
  
def function1():  
    pass  
  
type(ClassA) # -> classobj  
type(function1) # -> function
```

- keywords, operators and other syntax elements are not objects!

```
type(while) # SyntaxError  
type(def) # SyntaxError  
type(class) # SyntaxError  
type(+) # SyntaxError
```

Summary OOP

presented terms

- class
- instance (= object)
- attribute
- method
- constructor
- base class
- inheritance

presented Python constructs

- `class`, `isinstance(...)`, `issubclass(...)`, `self`, `id(...)`, `type(...)`

other OOP related topics:

- multiple inheritance
- static methods
- operator overloading and "magic methods"
- ducktyping
- polymorphism
- encapsulation
- meta class programming
- difference between `__init__` and `__new__`
- class methods
- class variables
- data classes

OOP-related Links

- [official Python doc on classes, instances, `self` etc.](#)
- [demystifying `self`](#)
- [introductory blog post](#)

Data Analysis: Overview

What do you do when you evaluate (measurement) data?

- load data
- select relevant data (or sections)
- determine new variables from existing ones
- increase information density
- visualize results

Data Analysis: Overview

What do you do when you evaluate (measurement) data?

- load data
- select relevant data (or sections)
- determine new variables from existing ones
- increase information density
- visualize results

Learning objectives:

- load / save data
- productive handling of Numpy arrays
- overview, which algorithms are already implemented
- overview on Pandas

Loading / Saving Data

```
import numpy as np

# ... do important calculations then save the result
result_array = np.arange(21).reshape(7, 3)

# same array but with automatic determination of column number
result_array2 = np.arange(21).reshape(7, -1)

np.save("result.npy", result_array) # binary format
np.savetxt("result.dat", result_array) # txt format

# ... in another file ...

array1 = np.load("result.npy") # binary format
array2 = np.loadtxt("result.dat") # txt format
```

∃ other possibilities (e.g. for *.wav files or Matlab format: see [scipy.io.*](#)).

Array Indexing

Basic indexing options of Numpy arrays (see also course03):

- integer numbers (`x[5]`) and “slices” (`x[3:10]`)

Array Indexing

Basic indexing options of Numpy arrays (see also course03):

- integer numbers (`x[5]`) and “slices” (`x[3:10]`)

Advanced indexing option 1: `int`-arrays

- can have arbitrary length; must contain integer values between $-n$ and $+n - 1$
- values are interpreted as indices; values can be repeated and omitted

```
x = np.array([10, 11, 12, 13])  
idcs = np.array([1, 2, 2, 1, 0 -2])  
y = x[idcs] # -> array([11, 12, 12, 11, 10, 12])
```

Array Indexing

Basic indexing options of Numpy arrays (see also course03):

- integer numbers (`x[5]`) and “slices” (`x[3:10]`)

Advanced indexing option 1: `int`-arrays

- can have arbitrary length; must contain integer values between $-n$ and $+n - 1$
- values are interpreted as indices; values can be repeated and omitted

```
x = np.array([10, 11, 12, 13])
ids = np.array([1, 2, 2, 1, 0 -2])
y = x[ids] # -> array([11, 12, 12, 11, 10, 12])
```

Advanced indexing option 2: `boolean` arrays

- length must be the same as indexed array (`x`)
- can only contain values `True` and `False`
- length of result equals number of `True`

```
ids = np.array([True, False, False, True])
x[ids] # -> [10, 13] (only first and last value were selected)

# negate all values that are less than 12:
x[x<12]*=-1 # -> [-10, -11, 12, 13]
```

Numerical Differentiation

- reminder: $\frac{df(x)}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$ ("difference quotient")
- `numpy.diff` for each array element: calc *successor minus predecessor*.
- for approximation of the first derivative of a function: **you** have divide by Δx .
- `numpy.diff(x)` → result is one element shorter than input data `x`

```
import numpy as np
x = np.arange(20) # array([0, 1, 2, 3,...])
xd = np.diff(x) # array([1, 1, 1, ...])
```

- higher derivative orders also possible, see [doc](#)
- opposite operation: `np.cumsum(x)` (cumulative sum $\hat{=}$ discrete integral).

Useful Engineering Tools: Filter and FFT

- filtering (low pass, moving average, ...).

→ package: [scipy.signal](#)

- Representation of the frequency spectrum

→ package: [numpy.fft](#) (Fast Fourier Transform)

“most important algorithm” of the information age

both require quite some background knowledge (therefore not covered here)

Interpolation (1)

- Interpolation (generate intermediate values, change sampling rate, ...)

→ Package: `scipy.interpolate`

Listing: example-code/06_a_interpolation.py

```
import numpy as np
import scipy.interpolate as ip
import matplotlib.pyplot as plt

# original data
x = [1, 2, 3, 4]
y = [2, 0, 1, 3]

plt.plot(x, y, "bx") # blue crosses
plt.savefig("interpolation0.pdf")

# create linear interpolator function
fnc1 = ip.interp1d(x, y)

# achieve higher x-resolution by evaluation of fnc1
xx = np.linspace(1, 4, 20)
plt.plot(xx, fnc1(xx), "r.-") # red solid line and dots
plt.savefig("interpolation1.pdf")
plt.show()
```

Interpolation (1)

- Interpolation (generate intermediate values, change sampling rate, ...)

→ Package: `scipy.interpolate`

Listing: example-code/06_a_interpolation.py

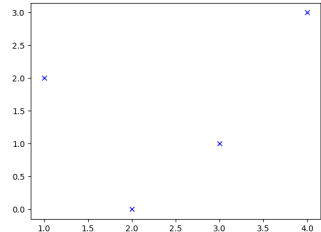
```
import numpy as np
import scipy.interpolate as ip
import matplotlib.pyplot as plt

# original data
x = [1, 2, 3, 4]
y = [2, 0, 1, 3]

plt.plot(x, y, "bx") # blue crosses
plt.savefig("interpolation0.pdf")

# create linear interpolator function
fnc1 = ip.interp1d(x, y)

# achieve higher x-resolution by evaluation of fnc1
xx = np.linspace(1, 4, 20)
plt.plot(xx, fnc1(xx), "r.-") # red solid line and dots
plt.savefig("interpolation1.pdf")
plt.show()
```



Interpolation (1)

- Interpolation (generate intermediate values, change sampling rate, ...)

→ Package: `scipy.interpolate`

Listing: example-code/06_a_interpolation.py

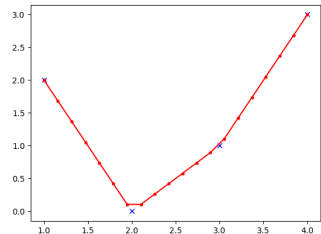
```
import numpy as np
import scipy.interpolate as ip
import matplotlib.pyplot as plt

# original data
x = [1, 2, 3, 4]
y = [2, 0, 1, 3]

plt.plot(x, y, "bx") # blue crosses
plt.savefig("interpolation0.pdf")

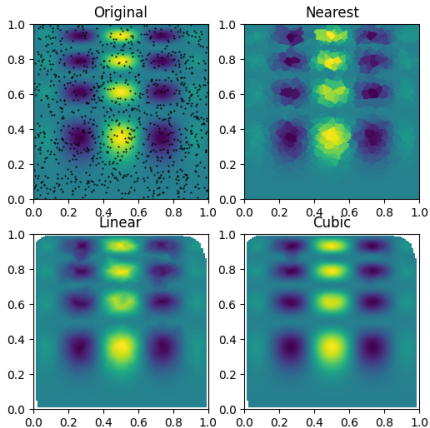
# create linear interpolator function
fnc1 = ip.interp1d(x, y)

# achieve higher x-resolution by evaluation of fnc1
xx = np.linspace(1, 4, 20)
plt.plot(xx, fnc1(xx), "r.-") # red solid line and dots
plt.savefig("interpolation1.pdf")
plt.show()
```



Interpolation (2)

- Works also in higher dimensions with irregularly distributed input data (see `example_code/06_b_griddata.py`, taken from [docs.scipy.org/...](https://docs.scipy.org/))



Regression (or “Fitting”)

→ `numpy.polyfit` and `numpy.polyval`

Listing: example-code/06_c_regression.py

```
import numpy as np
import matplotlib.pyplot as plt

N = 25
xx = np.linspace(0, 5, N)

# cool trick: two assignments in one line
m, n = 2, -1

# evaluate equation of straight line:  $y = mx + n$ 
yy = np.polyval([m, n], xx)
yy_noisy = yy + np.random.randn(N) # add some random noise

# create linear fit (regression 1st order polynomial)
mr, nr = np.polyfit(xx, yy_noisy, 1) # calculate fit
yyr = np.polyval([mr, nr], xx) # evaluate the function

plt.plot(xx, yy, 'go--', label="original")
plt.plot(xx, yy_noisy, 'k.', label="noisy data")
plt.plot(xx, yyr, 'r-', label="regression")
plt.legend()
plt.savefig("regression.png")

plt.show()
```

Regression (or “Fitting”)

→ `numpy.polyfit` and `numpy.polyval`

Listing: example-code/06_c_regression.py

```
import numpy as np
import matplotlib.pyplot as plt

N = 25
xx = np.linspace(0, 5, N)

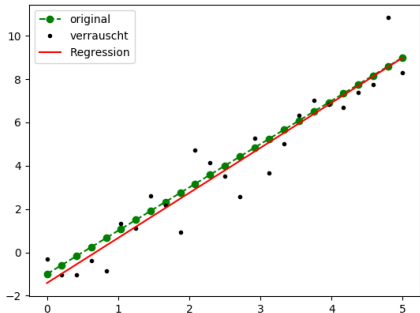
# cool trick: two assignments in one line
m, n = 2, -1

# evaluate equation of straight line:  $y = m \cdot x + n$ 
yy = np.polyval([m, n], xx)
yy_noisy = yy + np.random.randn(N) # add some random noise

# create linear fit (regression 1st order polynomial)
mr, nr = np.polyfit(xx, yy_noisy, 1) # calculate fit
yyr = np.polyval([mr, nr], xx) # evaluate the function

plt.plot(xx, yy, 'go--', label="original")
plt.plot(xx, yy_noisy, 'k.', label="noisy data")
plt.plot(xx, yyr, 'r-', label="regression")
plt.legend()
plt.savefig("regression.png")

plt.show()
```



Regression (or “Fitting”)

→ `numpy.polyfit` and `numpy.polyval`

Listing: example-code/06_c_regression.py

```
import numpy as np
import matplotlib.pyplot as plt

N = 25
xx = np.linspace(0, 5, N)

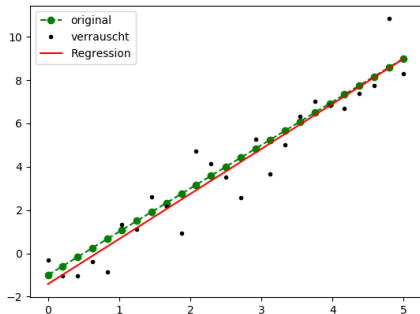
# cool trick: two assignments in one line
m, n = 2, -1

# evaluate equation of straight line:  $y = m \cdot x + n$ 
yy = np.polyval([m, n], xx)
yy_noisy = yy + np.random.randn(N) # add some random noise

# create linear fit (regression 1st order polynomial)
mr, nr = np.polyfit(xx, yy_noisy, 1) # calculate fit
yyr = np.polyval([mr, nr], xx) # evaluate the function

plt.plot(xx, yy, 'go--', label="original")
plt.plot(xx, yy_noisy, 'k.', label="noisy data")
plt.plot(xx, yyr, 'r-', label="regression")
plt.legend()
plt.savefig("regression.png")

plt.show()
```



- higher polynomial orders also possible

“Machine Learning” (ML)

- widely used since 2010's; very successful on some tasks
- can also be understood as **function approximation**
- three important subareas
 - **supervised learning**
 - classification (dog or cat? Mozart or Helene Fischer?)
 - regression (How well will movie X please person Y?)
 - **unsupervised learning** (= automatic cluster detection)
 - **Reinforcing learning** (agent adapts responses to environment to maximize reward)

Python is (arguably) the most important languages in ML:

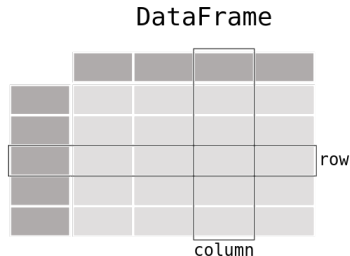
Subjective Link Selection:

- <https://scikit-learn.org> (neural networks, Gaussian processes, and many more).
- <https://pytorch.org> (neural networks)

The Pandas Package

→ **"spreadsheet processing with Python"**

- most important package for "Data Science"
- based on Numpy
- most important data type: `pandas.DataFrame`
 - models a table
 - columns can have names
 - columns can have different data types
- second most important data type: `pandas.Series`
 - models row or column



extensive documentation: <https://pandas.pydata.org/docs/>

Pandas: Create Data Frames

For following code examples see also: `notebooks/pandas_demo.ipynb`

Listing: 06_d_pandas.py

```
1 import pandas as pd
2 import numpy as np
3
4 # create df from a numpy array:
5 arr = np.random.randn(6, 4)
6 df1 = pd.DataFrame(arr, columns=list("ABCD"))
7
8 # create df from a dict
9 shop_articles = {
10     "weight": [10.1, 5.0, 8.3, 7.2],
11     "color": ["red", "green", "blue", "transparent"],
12     "availability": [False, True, True, False],
13     "price": 8.99 # all have the same price
14 }
15 article_numbers = ["A107", "A108", "A109", "A110"]
16 df2 = pd.DataFrame(shop_articles, index=article_numbers)
17
18 # each has its own data type
19 print("column types:\n", df2.dtypes)showspaces
```

Pandas: Save and Load Data

Listing: 06_d_pandas.py

```
25 fname = "things.csv"
26
27 # save the data frame as CSV file (Comma Separated Values )
28 # csv file will also contain header information (column labels)
29 df2.to_csv(fname)
30
31
32
33 # Pandas function to load csv-data into DataFrame
34 # (Detects column names automatically)
35 df2_new = pd.read_csv(fname)
36
37 # display(df2_new) # Jupyter-Notebook-specific
```


Pandas: Read/Write Access to Cells

Listing: 06_d_pandas.py

```
42
43 # access individual values (by verbose index and column):
44 print(df2.loc["A108", "weight"])
45 df2.loc["A108", "weight"] = 3.4 # set new value
46 df2.loc["A108", "weight"] += 2 # increase by two
47
48 # access by numerical indices
49 print(df2.iloc[1, 0]) # row index: 1, column index: 0
50
51 # use slices to change multiple values
52 df2.loc["A108":"A109", "price"] *= 0.30 # 30% discount
53
54 # access multiple columns (-> new df object)
55 print(df2[["price", "weight"]])
56
57 # new column (-> provide a height value for every article)
58 df2["height"] = [10, 20, 30, 40]
59
60 # new row (-> provide a value for every column (weight, color, ...) )
61 df2.loc["X400"] = [15 , "purple" , True , 25.00 , 50]
```

Pandas: Select Data by Boolean Indexing

Listing: 06_d_pandas.py

```
65 # create Series-object with bool-entries
66 idcs = df2["weight"] > 8
67
68 # use this Series-object for indexing
69 print(df2[idcs])
70
71 # similar statement without intermediate variable:
72 print(df2[df2["weight"] < 8])
```

Pandas: Apply Functions

Listing: 06_d_pandas.py

```
75 df2.describe()
76 df2["price"].mean()
77 df2["weight"].median()
78 df2["weight"].max()
79
80 print("shorthand notation (if column label is valid python name)")
81 print(df2.weight == df2["weight"])
82 print(all(df2.weight == df2["weight"]))
83
84 # combine function application with boolean indexing
85 df2[df2.weight>8].weight.mean()
86
87 # apply an arbitrary function (here np.diff) to each (selected) column
88 print(df2[["price", "weight"]].apply(np.diff))
```

Summary of this Part

- Access to data (`np.load` , `pd.read_csv`)
- Selection (integer indexing, boolean indexing of both arrays and data frames)
- Interpolation
- Regression
- Pandas: DataFrames, r/w cells, rows, columns, boolean indexing, functions
- See docs for more details
- See docs for more details (links above)

Advanced Programming Techniques – Outline

- functional programming,
- nested functions
- namespaces
- `import` mechanisms
- exceptions (error handling)
- PEP8, unittests, documentation tools, version management

Functional Programming (and List Comprehensions)

- application of functions to sequences (e.g. lists)
 - keyword `lambda` → anonymous function (“throwaway function”)

```
L = [1, 3, 5, 7, 9]
squares = list(map(lambda z: z**2, L))
big_numbers = list(filter(lambda n: n > 5, L))
# map and filter each return iterators -> convert to list
```

- compact syntax but not so easy to read/understand
- recommended alternative for `map` `filter` `filter`: **list comprehension**:

```
squares = [z**2 for z in L]
big_numbers = [z for z in L if z > 5]
```

- function application for matrices and arrays:
`numpy.ufunc`, `sympy.Matrix.applyfunc`, `pandas.DataFrame.apply`
- typical in functional programming: **recursion** ($\hat{=}$ function that calls itself)

```
def factorial(n): # calculate n!
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Nested Functions (I)

- functions are ordinary objects

→ can be arguments of other function (like `solve_ivp(rhs, ...)` in course03).

```
def func1(x):  
    "a docstring"  
    return x + 1  
  
print(type(func1)) # -> <type 'function'>  
  
def read_the_docs(f):  
    print(f.__doc__)  
  
read_the_docs(func1) # -> "a docstring"
```

Nested Functions (II)

- functions can also be **created** within other functions.
- (important: scopes of variables ("scopes"))

```
def factory(y):  
    def f(x):  
        return x + y  
  
    return f # return the function object  
  
func1 = factory(10)  
func2 = factory(5)  
  
print(func1(2), "; ", func2(2)) # -> 12 ; 7
```


Decorators (I)

“wrapper functions”: functions that are applied to functions and return new functions

Listing: example-code/07_a_decorators.py

```
1 result_cache = dict()
2
3 def cached(func):
4
5     def wrapper(arg):
6         key = (func, arg)
7         if key in result_cache:
8             print("use cached result")
9             return result_cache[key]
10        else:
11            res = func(arg)
12            result_cache[key] = res
13            return res
14        return wrapper
15
16 def func1(x):
17     print("Executing func1 with x =", x)
18     return x**2
19
20 wrapped_func1 = cached(func1)
21
22 # original function gets called
23 print(wrapped_func1(5))
24
25 # value from the cache gets used
26 print(wrapped_func1(5))
```

“wrapper”: (german: Verpackung, Umhüllung)

Example:

- memory for results of calculations
- memory dict object
- key: 2-tuple: (function, argument)
- value: return value of function
- motivation: saving computation time

General:

- decorators are useful for reusable pre- and postprocessing of data

Decorators(II)

- outer (factory) function (i.e. the wrapper) is called “decorator”
- special short syntax to wrap a function: use `@`
- skip the variable assignment : `wrapped_func = decorator(original_func)`
- original function is not available under its own name

Listing: example-code/07_a_decorators.py (28-34)

```
@cached
def func2(x):
    print("Executing func2 with x =", x)
    return 100 + x

print(func2(4))  # original function gets called
print(func2(4))  # value from the cache gets used
```

- decoators can also be applied to methods and classes
- builtin decorators: `staticmethod`, `classmethod`, ...
- more information on decorators:
 - programiz.com/python-programming/decorator
 - thecodeship.com/patterns/guide-to-python-function-decorators/

Namespaces

- namespace: scope of variable names.
- each name is used at most once per namespace → uniqueness
- terms "namespace" and "scope" almost synonymous
- common: nested namespaces
 - in a normal function: 2 levels (global and local namespace)
 - function in function: 3 levels
- Python interpreter searches names from inside to outside
if a name is not found in the local scope the next higher scope is searched
- each module has a global namespace
- keep order and overview with prefixes:

```
import math # from Python's standard lib (no array support)
import numpy # with array support
x = 1.23
a = numpy.arange(4)

print(math.exp(x))
print(numpy.exp(a))
# these funcs have the same name, but need to be separated
# math.exp(a) would cause an error
```

Importing Modules and Packages

- module = Python file
- currently executed file: "main module"
- `import`: import names (+ objects bound to them) into own namespace
- two options:

```
# option 1: import the module
import numpy
x = numpy.arange(5)

# option 2: import single objects into global namespace
from numpy import arange, pi
x = arange(5)*pi
```

- abbreviations (aliases) with keyword `as`:

```
# option 1: import the module
import numpy as np
x = np.arange(5)

# option 2: import single objects into global namespace
from numpy import arange as ar
x = ar(5)
```

Imports (II)

Other abbreviations:

```
# this is possible but considered bad style  
# multiple imports in one line (with or without alias):  
import numpy as np, matplotlib, sympy
```

- wildcard import (*)
- Only recommended for interactive testing
 - (no control which names are imported and overwritten if necessary):

```
from numpy import * # "namespace pollution"
```

Imports (III)

- imported module is executed "normally":
(but only once; `importlib.reload` → force true re-import)

```
# module1.py
def function1(x):
    return x**2 - 3*x + 5

print("abc") # will produce output upon importing this module
Z = 123
```

```
# main.py (located in the same directory as module1.py)
import importlib
import module1 # -> abc
import module1 # no new output
importlib.reload(module1) # -> abc
print(module1.Z + 0.4) # -> 123.4
module1.function1(0) # -> 5
```

- modules are also objects:

```
print(type(module1)) # -> <type 'module'>
```

Packages / Import Paths

- module = Python file
- package = directory containing file `__init__.py` and possibly other modules
- can be nested: `package.subpackage1.subpackage2.module.object`
- file `__init__.py` is allowed to be empty

What is the purpose of modularization?

- reusability, redistribution, exchange
- thematic structuring of code

Packages / Import Paths

- module = Python file
- package = directory containing file `__init__.py` and possibly other modules
- can be nested: `package.subpackage1.subpackage2.module.object`
- file `__init__.py` is allowed to be empty

What is the purpose of modularization?

- reusability, redistribution, exchange
- thematic structuring of code

Import paths:

- Python interpreter searches in certain default directories (incl. current workdir)

```
import sys
print(sys.path)  # -> ['', '/anaconda/lib/python3.8/site-packages', ...]
```

- customization options:
 - `PYTHONPATH` environment variable
 - `my_path.pth` file (in one of the default directories)
 - `sys.path.append(...)`

Exceptions (Error Handling)

- errors are inevitable during programming
- good error handling saves a lot of time and nerves
- Python: "exceptions"
 - are "thrown" at an erroneous position
 - can be "caught" in an error location

```
def F(x, y):  
    return x/y  
def G(z):  
    try:  
        print(F(10, z))  
    except ZeroDivisionError:  
        print("Division by 0 is not allowed!")  
G(5) # -> 2  
G(0) # -> Division by 0 is not allowed!
```

- important types: `Exception` (=base class for all exceptions), `NameError`, `ValueError`, `TypeError`, `AttributeError`, `NotImplementedError`,
- more information: <https://docs.python.org/3/tutorial/errors.html>

Exceptions (II)

Recommendations:

- every source code is based on assumptions made during development
 - occasionally check if assumptions are still true at runtime
- throw exceptions in your own code: `raise`

```
def F(x):  
    if not isinstance(x, (float, int)):  
        msg = f"number expected but got {type(x)}"  
        raise TypeError(msg)  
    return x**2
```

- more convenient (⇒ lower usage hurdle): `assert`

```
def F(x):  
    assert isinstance(x, (float, int))  
    return x**2
```

→ exception type: `AssertionError`

- disadvantage: unspecific error (type information missing) → only for “private use”.

Further Topics

Python Enhancement Proposal #8; short: PEP8, python.org/dev/peps/pep-0008/

- style guide for good code (conventions for naming and formatting)

`unittest` package, <https://docs.python.org/3/library/unittest.html>

- automated testing of your code
- indispensable for larger projects, already pays off for small projects

`sphinx` package, <http://www.sphinx-doc.org/en/stable>

- automatically generate documentation from code (and docstrings).

→ requires good docstrings

version control with [Git](#)

- essential for projects with multiple people but also useful on its own
- recommendation: <https://git-scm.com>, see also [FSFW-git-intro-workshop \(de\)](#)

Further Topics

Python **E**nhancement **P**roposal #8; short: PEP8, python.org/dev/peps/pep-0008/

- style guide for good code (conventions for naming and formatting)

`unittest` package, <https://docs.python.org/3/library/unittest.html>

- automated testing of your code
- indispensable for larger projects, already pays off for small projects

`sphinx` package, <http://www.sphinx-doc.org/en/stable>

- automatically generate documentation from code (and docstrings).

→ requires good docstrings

version control with `Git`

- essential for projects with multiple people but also useful on its own
- recommendation: <https://git-scm.com>, see also [FSFW-git-intro-workshop](#) (de)

still more topics: [typing hints](#), [continuous integration](#), [meta class programming](#), ...

Summary of this Part

- functional programming (`lambda`, `map`, `filter`, list comprehension).
- nested functions (define functions within functions)
- decorators
- namespaces, `import` mechanisms (keeping order)
- exceptions (throwing and catching, `assert`)
- PEP8, unittests, doc tools, version management

Performance Optimization – Outline

- Introduction
- Timing
- General Tips
- Compiled Code

Introduction

What is performance?

- runtime
 - memory requirements (RAM, hard disk)
 - power consumption
- In this lecture: only **execution time** considered
(most important in most cases, easy to measure, correlated with power consumption)

Facts

- Python: slower than compiled languages (interpreters)
 - in many applications: difference not even perceptible (0.1s vs. 0.01s)
 - runtime optimization of code itself often very time consuming
- conflict of goals: execution vs. development time

Introduction

What is performance?

- runtime
 - memory requirements (RAM, hard disk)
 - power consumption
- In this lecture: only **execution time** considered
(most important in most cases, easy to measure, correlated with power consumption)

Facts

- Python: slower than compiled languages (interpreters)
 - in many applications: difference not even perceptible (0.1s vs. 0.01s)
 - runtime optimization of code itself often very time consuming
- conflict of goals: execution vs. development time
- ⇒ general tips to follow from the start
- ⇒ specific optimization of runtime for **critical algorithm parts**

Time Measurement (I)

- module `time`
- `time.time()` returns “epoch-time” (also called “UNIX-timestamp”)
`time` $\hat{=}$ seconds since 01/01/1970 00:00:00.00
- advantage: very simple
- disadvantage: additional (“boilerplate”) code distributed in the program

```
import time

s = 0
start = time.time()

for i in range(100000):
    s += i**(0.5)

print("Duration [s]:", time.time() - start)
```

Time Measurement (II)

- module `timeit`, see [docs](#)
- runtime measurement of a statement (mostly function call)
- good for comparison of code snippets for special problem
- statement must be passed as *string* or *callable*
- advantages: non-invasive, averaging of multiple runs

Time Measurement (II)

- module `timeit`, see [docs](#)
- runtime measurement of a statement (mostly function call)
- good for comparison of code snippets for special problem
- statement must be passed as *string* or *callable*
- advantages: non-invasive, averaging of multiple runs

Listing: example-code/08_a_time-example.py

```
import math; from timeit import timeit

def root1(x=2): return x**0.5

def root2(): return math.sqrt(2)

N = int(1e6)
print("2**0.5:      ", timeit("2**0.5", number=N), "sqrt(2):", timeit("math.sqrt(2)", setup="import math", number=N))

# func calls without argument
print("root1():     ", timeit(root1, number=N), "root2():", timeit(root2, number=N))

# func calls with argument (timeit(root2(x=2), number=N)) would "see" only the return value
# Thus, we need to pass the function call as string. Then the `globals`- keyword argument is also needed.
print("root1(x=2):", timeit("root1(x=2)", number=N, globals=globals()))
```

Time Measurement (II)

- module `timeit`, see [docs](#)
- runtime measurement of a statement (mostly function call)
- good for comparison of code snippets for special problem
- statement must be passed as *string* or *callable*
- advantages: non-invasive, averaging of multiple runs

Listing: example-code/08_a_time-example.py

```
import math; from timeit import timeit

def root1(x=2): return x**0.5

def root2(): return math.sqrt(2)

N = int(1e6)
print("2**0.5:      ", timeit("2**0.5", number=N), "sqrt(2):", timeit("math.sqrt(2)", setup="import math", number=N))

# func calls without argument
print("root1():     ", timeit(root1, number=N), "root2():", timeit(root2, number=N))

# func calls with argument (timeit(root2(x=2), number=N)) would "see" only the return value
# Thus, we need to pass the function call as string. Then the `globals`- keyword argument is also needed.
print("root1(x=2):", timeit("root1(x=2)", number=N, globals=globals()))
```

- See also: “magic macros” for both IPython and Jupyter: `%time` and `%timeit`

Professional Timing: Profiling (I)

- module `cProfile`: detailed runtime analysis of a (possibly very large) program → Find bottleneck.
- profiling creates overhead, i. e. program runs slightly slower than without it
- results as print output or to file for use in analysis tools
- argument is passed as string

Listing: example-code/08_b_profile-example.py

```
import cProfile
import math

def main():
    s = 0
    for i in range(100000):
        s += math.sqrt(i)

cProfile.run("main()")
```

alternative (command line call): `python -m cProfile -s cumtime test.py > test.txt`

- sorted by cumulative time
- `... > test.txt` redirects output to file: `test.txt`
- with option `-o test.prfl` will redirect results in binary format to file `test.prfl` (can then be evaluated with `pstats`, see [docs](#)).

Profiling (II)

Output of the example

100004 function calls in 0.021 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.021	0.021	<string>:1(<module>)
1	0.014	0.014	0.021	0.021	profile-example.py:4(main)
1	0.000	0.000	0.021	0.021	{built-in method builtins.exec}
100000	0.006	0.000	0.006	0.000	{built-in method math.sqrt}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

- shows which function was called how often and how much time it needed
→ find starting points for optimization
- interesting here: only $\approx \frac{1}{3}$ of the runtime for `sqrt` needed
- rest: overhead (function call, loop)

Profiling (II)

Output of the example

100004 function calls in 0.021 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.021	0.021	<string>:1(<module>)
1	0.014	0.014	0.021	0.021	profile-example.py:4(main)
1	0.000	0.000	0.021	0.021	{built-in method builtins.exec}
100000	0.006	0.000	0.006	0.000	{built-in method math.sqrt}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

- shows which function was called how often and how much time it needed
→ find starting points for optimization
- interesting here: only $\approx \frac{1}{3}$ of the runtime for `sqrt` needed
- rest: overhead (function call, loop)
- Further analysis: `pstats`, see [docs](#)

General Tips (I)

- optimize code only when there is an actual need
("Premature optimisation is the root of all evil.")
→ use profiling and identify only the worthwhile jobs.
- optimize only correct code
- use unit tests to ensure correctness of the code during/after rework
 - order: „Make it run. Make it right. Make it fast.“
- use appropriate libraries for respective problem
- e.g. `numpy` for numerics
 - is written in C/Fortran → much faster than pure Python
- Python scripts usually faster than Jupyter Notebooks (rendering overhead)

General Tips (II)

- use appropriate data types: `tuple` or `dict` instead of `list`.
example: “element Lookup”

```
res = 3 in {1: True, 2: True, 3: True} # effort: O(1) (= const)
res = 3 in [1, 2, 3] # effort O(n)
```

- in (nested) loops: move functionality “from inside to outside”.
 - initializations of variables
 - calculations → intermediate results save/cache
 - **execute statements only as often as necessary, but as rarely as possible**
 - “loops in functions” are faster than “functions in loops”
(every function-call costs time)
- create auxiliary **local variables** to avoid “points” (e. g. from object orientation):
 - each point (`obj.attr`) means attributes/member lookup,
 - local caching is worthwhile especially in loops

```
root = math.sqrt
# ...
root(2) # inside a loop avoid name-lookup
```

Outdated Tips (III)

Often recommended but not so effective (anymore) w.r.t speedup

- use iterators (e.g. `range(4)` instead of `[0, 1, 2, 3]`)
 - background: iterators generate function to calculate next element,
 - more memory-efficient than generating whole sequence in advance
- use `list comprehension` instead of `for`-loops

```
r = [ str(k) for k in [1, 2, 3] ]  
# instead of  
r = []  
for k in [1, 2, 3]:  
    r.append(str(k))
```

- vectorize functions for array operations (`numpy.vectorize`), see [docs](#)

```
def f(x):  
    if x > 2: return x*100  
    else:     return x  
  
xx = np.arange(5)  
# f(xx) # -> ValueError  
f_vect = np.vectorize(f)  
f_vect(xx) # -> array([ 0, 1, 2, 300, 400])
```

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.
- high flexibility, but comparatively low execution speed

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.

→ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor

→ high execution speed, low flexibility (e.g. static data types, memory management)

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.

→ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor

→ high execution speed, low flexibility (e.g. static data types, memory management)

Possible combinations (embedding compiled code in Python):

- “Just in Time” compilation of certain code sections (e.g. module `numba`)

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.

→ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor

→ high execution speed, low flexibility (e.g. static data types, memory management)

Possible combinations (embedding compiled code in Python):

- “Just in Time” compilation of certain code sections (e.g. module `numba`)
- compile Python code into `cython`
 - very similar to Python but statically typed and compiled → fast

Compiled Code (Overview)

Python source code:

- Compiled into “bytecode” and executed by the Python *interpreter* at runtime.
→ high flexibility, but comparatively low execution speed

Compiled code (C, C++, Rust, ...):

- Compiled into machine language and processed directly by the processor
→ high execution speed, low flexibility (e.g. static data types, memory management)

Possible combinations (embedding compiled code in Python):

- “Just in Time” compilation of certain code sections (e.g. module [numba](#))
- compile Python code into [cython](#)
 - very similar to Python but statically typed and compiled → fast
- [ctypes](#)
 - Can load external libraries into Python (e.g. *.dll on Windows, *.so on Unix)
→ very powerful and flexible
 - Not considered here; if necessary see [python-c-code-example \(github\)](#)
 - mostly useful if C-library already exists (or is needed anyway, e.g. for target hardware)

Just-in-time-Compilation with `numba`

- Significant acceleration potential for mathematical operations.
- Necessary: `pip install numba`
- Example: "`Mandelbrot set`"
 - (simple math, high numerical effort, visual result).

Listing: example-code/08_c_numba1.py (14-29)

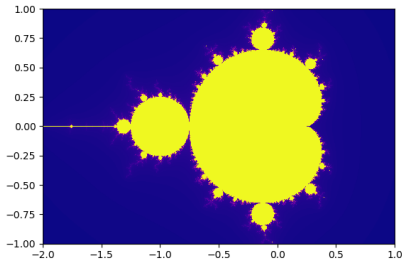
```
# Decorator for just-in-time comp. (-> 30x speedup)
@jit
def mandel(x, y, max_iters):
    """
    Given a complex number x + y*j, determine
    if it is part of the Mandelbrot set given
    a fixed number of iterations.
    """
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 1e3:
            return i
```


Just-in-time-Compilation with `numba`

- Significant acceleration potential for mathematical operations.
- Necessary: `pip install numba`
- Example: "Mandelbrot set"
 - (simple math, high numerical effort, visual result).

Listing: example-code/08_c_numba1.py (14-29)

```
# Decorator for just-in-time comp. (-> 30x speedup)
@jit
def mandel(x, y, max_iters):
    """
    Given a complex number x + y*j, determine
    if it is part of the Mandelbrot set given
    a fixed number of iterations.
    """
    i = 0
    c = complex(x, y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 1e3:
            return i
```



Cython (I)

- Cython is a separate programming language, installation: `pip install cython`
 - Very close to Python but with explicit **static type** information
- can be compiled to C automatically → compilable → faster
- details: see [docs](#)

Cython (I)

- Cython is a separate programming language, installation: `pip install cython`
 - Very close to Python but with explicit **static type** information
- can be compiled to C automatically → compilable → faster
- details: see [docs](#)
 - procedure:
 - Develop algorithm in pure Python (“Make it run” + “Make it right”)
 - Translate Python to Cython manually
 - Translate Cython code to C
 - Compile C code
 - Import / use module created this way “as usual” (→ “Make it fast”)

Cython (I)

- Cython is a separate programming language, installation: `pip install cython`
 - Very close to Python but with explicit **static type** information
- can be compiled to C automatically → compilable → faster
- details: see [docs](#)
 - procedure:
 - Develop algorithm in pure Python (“Make it run” + “Make it right”)
 - Translate Python to Cython manually
 - Translate Cython code to C
 - Compile C code
 - Import / use module created this way “as usual” (→ “Make it fast”)

Typically 3 files, e. g.

- `mandel-cython.pyx` : cython source code
- `mandel-cython-setup.py` : for compiling
- `mandel-cython-main.py` : to import and call compiled code

Listing: example-code/08_d_mandel-cython.pyx

```
# Cython source code
cimport numpy as np # for the special numpy stuff

cdef inline int mandel(double real, double imag, int max_iterations=20):
    """Given a complex number  $x + y*j$ , determine if it is part of the
    Mandelbrot set given a fixed number of iterations. """

    cdef double z_real = 0., z_imag = 0.
    cdef int i

    for i in range(0, max_iterations):
        z_real, z_imag = ( z_real*z_real - z_imag*z_imag + real,
                          2*z_real*z_imag + imag )
        if (z_real*z_real + z_imag*z_imag) >= 1000:
            return i
    # return -1
    return 255

def create_fractal( double min_x, double max_x, double min_y, int nb_iterations,
                   np.ndarray[np.uint8_t, ndim=2, mode="c"] image not None):

    cdef int width, height, x, y, start_y, end_y
    cdef double real, imag, pixel_size

    width = image.shape[0]
    height = image.shape[1]

    pixel_size = (max_x - min_x) / width

    for x in range(width):
        real = min_x + x*pixel_size
        for y in range(height):
            imag = min_y + y*pixel_size
            image[x, y] = mandel(real, imag, nb_iterations)
```

Cython (III)

- script for conversion Cython → C:

Listing: example-code/08_d_mandel-cython-setup.py

```
"script for conversion of cython-code to c-code"

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy # to get includes

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("mandelcy", ["08_d_mandel-cython.pyx"], )],
    include_dirs = [numpy.get_include()],
)
```

- Command: `python 08_d_mandel-cython-setup.py build_ext --inplace`
→ C code is compiled and an importable library is created

Cython (IV)

- Calling the compiled code and visualization of Mandelbrot set:

Listing: example-code/08_d_mandel-cython-main.py

```
import numpy as np
import matplotlib.pyplot as plt
import mandelcy # our Cython module (for the real work)

import time

# define section of the Gaussian number plane
min_x = -1.5
max_x = 0.15
min_y = -1.5
max_y = min_y + max_x - min_x

# to have same section like numba script
# min_x = -2; max_x = 1; min_y = -1.5

nb_iterations = 255

t1 = time.time()
dataarray = np.zeros((500, 500), dtype=np.uint8)
t2 = time.time()
print("Time needed", t2 - t1)

# execution of the compiled code
mandelcy.create_fractal(min_x, max_x, min_y, nb_iterations, dataarray)

dataarray = dataarray.T[::-1, :] # Transpose and reverse order along first axis

plt.imshow(dataarray, extent=(min_x, max_x, min_y, max_x), cmap=plt.cm.plasma)
plt.savefig("mandel-cython.png")
plt.show()
```

Cython (IV)

- Calling the compiled code and visualization of Mandelbrot set:

Listing: example-code/08_d_mandel-cython-main.py

```
import numpy as np
import matplotlib.pyplot as plt
import mandelcy # our Cython module (for the real work)

import time

# define section of the Gaussian number plane
min_x = -1.5
max_x = 0.15
min_y = -1.5
max_y = min_y + max_x - min_x

# to have same section like numba script
# min_x = -2; max_x = 1; min_y = -1.5

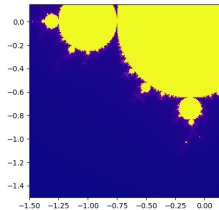
nb_iterations = 255

t1 = time.time()
dataarray = np.zeros((500, 500), dtype=np.uint8)
t2 = time.time()
print("Time needed", t2 - t1)

# execution of the compiled code
mandelcy.create_fractal(min_x, max_x, min_y, nb_iterations, dataarray)

dataarray = dataarray.T[::-1, :] # Transpose and reverse order along first axis

plt.imshow(dataarray, extent=(min_x, max_x, min_y, max_x), cmap=plt.cm.plasma)
plt.savefig("mandel-cython.png")
plt.show()
```



(→ 500x speedup (Py vs Cy))

Summary of this Part

- \exists many ways to tweak Python code to make it faster
- If that is not enough:
 - identify bottle necks using [profiling](#)
- replace critical program parts with compiled code
 - just-in-time compilation [numba](#) (effort: low)
 - manual port to [cython](#) (effort: moderate)
 - custom C code using [ctypes](#) (effort might be considerable)

Summary of this Part

- \exists many ways to tweak Python code to make it faster
- If that is not enough:
 - identify bottle necks using [profiling](#)
- replace critical program parts with compiled code
 - just-in-time compilation [numba](#) (effort: low)
 - manual port to [cython](#) (effort: moderate)
 - custom C code using [ctypes](#) (effort might be considerable)
 - (\exists more possibilities, e.g. [PyPy](#))
- not covered here:
 - [threading/multiprocessing](#), parallelization via [asyncio](#)
 - [pyjion](#) (“drop-in JIT Compiler for Python 3.10”)

The Package Sympy

- Python library for symbolic computations ('calculations with letters')
→ backend of a **computer algebra system** (CAS)
- Possible frontends: own Python script, IPython shell, Jupyter notebook
- Advantage over other products: CAS-functionality inside a real programming language

The Package Sympy

- Python library for symbolic computations ('calculations with letters')
→ backend of a **computer algebra system** (CAS)
- Possible frontends: own Python script, IPython shell, Jupyter notebook
- Advantage over other products: CAS-functionality inside a real programming language
- Goal: overview and introduction
- Structure:
 - Introduction
 - Calculating
 - Substituting
 - Important functions & data types
 - Numerical evaluation of formulas

Sympy Overview

- ways to import the package or objects from it:
 - `import sympy as sp`
 - `from sympy import sin, cos, pi`
 - `from sympy import *` ⚠ caution: “namespace pollution”

Sympy Overview

- ways to import the package or objects from it:
 - `import sympy as sp`
 - `from sympy import sin, cos, pi`
 - `from sympy import *` ⚠ caution: “namespace pollution”
- `sp.symbols`, `sp.Symbol` : create symbols (\neq variables)
- `sp.sin(x)`, `sp.cos(2*pi*t)`, `sp.exp(x)`, ... : mathematical functions
- `sp.Function('f')(x)` : create and evaluate custom functions
- `sp.diff(<expr>, <var>)` or `<expr>.diff(<var>)` : taking derivatives
- `sp.simplify(<expr>)` : simplification
- `<expr>.expand()` : expansion of brackets ($a(x + y) \rightarrow ax + ay$)
- `<expr>.subs(...)` : substitution
- `sp.pprint(<expr>)` : “pretty printing”

Performing Calculations

Listing: example-code/04_a_sympy1.py

```
import sympy as sp
x = sp.Symbol("x")
a, b, c = sp.symbols("a b c") # different ways to create symbols

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 (multiply out)

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

# define custom function
f1 = sp.Function("f") # -> sympy.core.function.f (not evaluated)
g1 = sp.Function("g")(x) # -> g(x) (function evaluated at x)

# taking derivatives
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)

# simplification (example):
y = sp.sin(x)**2+sp.cos(x)**2
print(y == 1) # -> False
print(sp.simplify(y)) # -> 1
print(sp.simplify(y) == 1) # -> True
```

Substitutions: `<expr>.subs(...)`

- comparable to `<str>.replace(old, new)`
- useful for: manual simplifications, (partial) function evaluations, coordinate transformations.

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


Substitutions: `<expr>.subs(...)`

- comparable to `<str>.replace(old, new)`
- useful for: manual simplifications, (partial) function evaluations, coordinate transformations.

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

- call options: 1. two arguments, 2. list with 2-tuples, 3. dict
 1. `<expr>.subs(old, new)`
 2. `<expr>.subs([(old1, new1), (old2, new2), ...])`
 3. `<expr>.subs({old1: new1, old2: new2, ...})`
 - option 2: order of list → substitution order
 - relevant when substituting derivatives (see [example notebook](#))
- important: `.subs(...)` returns new expression (original expr. remains unchanged)

More Important Functions, Methods, Types

- `A = sp.Matrix([[x1, a*x2], [c*x3, sp.sin(x1)]])` : create a matrix
- `A.jacobian([x1, x2, x3])` : Jacobian matrix of a vector (matrix of partial derivatives)
- `sp.solve(x**2 + x - a, x)` : solve (systems of) equations
- `<expr>.atoms()`, `<expr>.atoms(sp.sin)` : “atoms” (of a certain type)
- `<expr>.args` : arguments of the respective class (summands, factors, ...)
- `sp.simplify(...)` : adapt data type (plain Python \rightarrow Sympy)
- `sp.integrate(<expr>, <var>)` : symbolic integration (anti derivative)
- `sp.series(...)` : Taylor series expansion of expression (like $e^x \Big|_{x=0} = \sum_{k=0}^n \frac{1}{k!} x^k$)
- `sp.limit(<var>, <value>)` : limit (like $\lim_{x \rightarrow 0} \frac{\sin(x)}{x} = 1$)
- `<expr>.as_num_denom()` : decomposition into 2-tuple: (*numerator*, *denominator*)
- `sp.poly(x**7+a*x**3+b*x+c, gens=[x], domain="EX")` : Polynomial
- `sp.Piecewise(...)` : piecewise defined function

\rightarrow see docs (and docstrings) for more info (or just try things out)

Numeric Formula Evaluation

- given: expression and values of the individual variables
- wanted: numerical result

Numeric Formula Evaluation

- given: expression and values of the individual variables
- wanted: numerical result
- possible in principle: `expr.subs(num_values).evalf()`

Numeric Formula Evaluation

- given: expression and values of the individual variables
- wanted: numerical result
- possible in principle: `expr.subs(num_values).evalf()`
- better (in terms of speed): `lambdify` (origin of the name: Python's `lambda` functions)
- creates a Python function that you can then call with the arguments

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

# create the function
df_xa_fnc = sp.lambdify((a, b, x), df_xa, modules='numpy')

# call (= evaluate) the function
print( f_xa_fnc(1.2, 0.5, 3.14) )
```

Sympy Support in Jupyter

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

```
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) ##:
```

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

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

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

See Example Notebook

[../notebooks/course09-sympy-demo.html](#)

Sympy Support in Jupyter

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

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

```
%load_ext ipydisplaytools
```

```
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) ##:
```

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

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

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

See Example Notebook

`../notebooks/course09-sympy-demo.html`

for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ output

Sympy Support in Jupyter

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

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

```
%load_ext ipydisplaytools
```

```
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) ##:
```

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

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

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

See Example Notebook

[../notebooks/course09-sympy-demo.html](#)

for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ output

enables special comment:

##:

Sympy Support in Jupyter

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

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

```
%load_ext ipydisplaytools
```

```
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) ##:
```

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

```
---
```

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

```
---
```

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

```
---
```

See Example Notebook

../notebooks/course09-sympy-demo.html

for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ output

enables special comment:

##:

shows the result
of assignments

Sympy Support in Jupyter

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

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

`%load_ext ipydisplaytools`

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) ##:
```

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

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

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

See Example Notebook

`../notebooks/course09-sympy-demo.html`

for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ output

enables special comment:

`##:`

shows the result
of assignments

Sympy Support in Jupyter

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

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

`%load_ext ipydisplaytools`

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) ##:`

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

`---`

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

`---`

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

`---`

See Example Notebook

`../notebooks/course09-sympy-demo.html`

for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ output

enables special comment:

`##:`

shows the result
of assignments

Docs and Links

- docstrings of the respective objects (in Jupyter: e.g. `sp.solve?`)
- <http://docs.sympy.org/latest/tutorial/index.html>
- module reference (e.g.: `solve` -function)
- <http://docs.sympy.org/latest/tutorial/gotchas.html> (pitfalls)

Summary

- symbolic calculations
- substitution
- important functions / data types
- numerical evaluation of functions