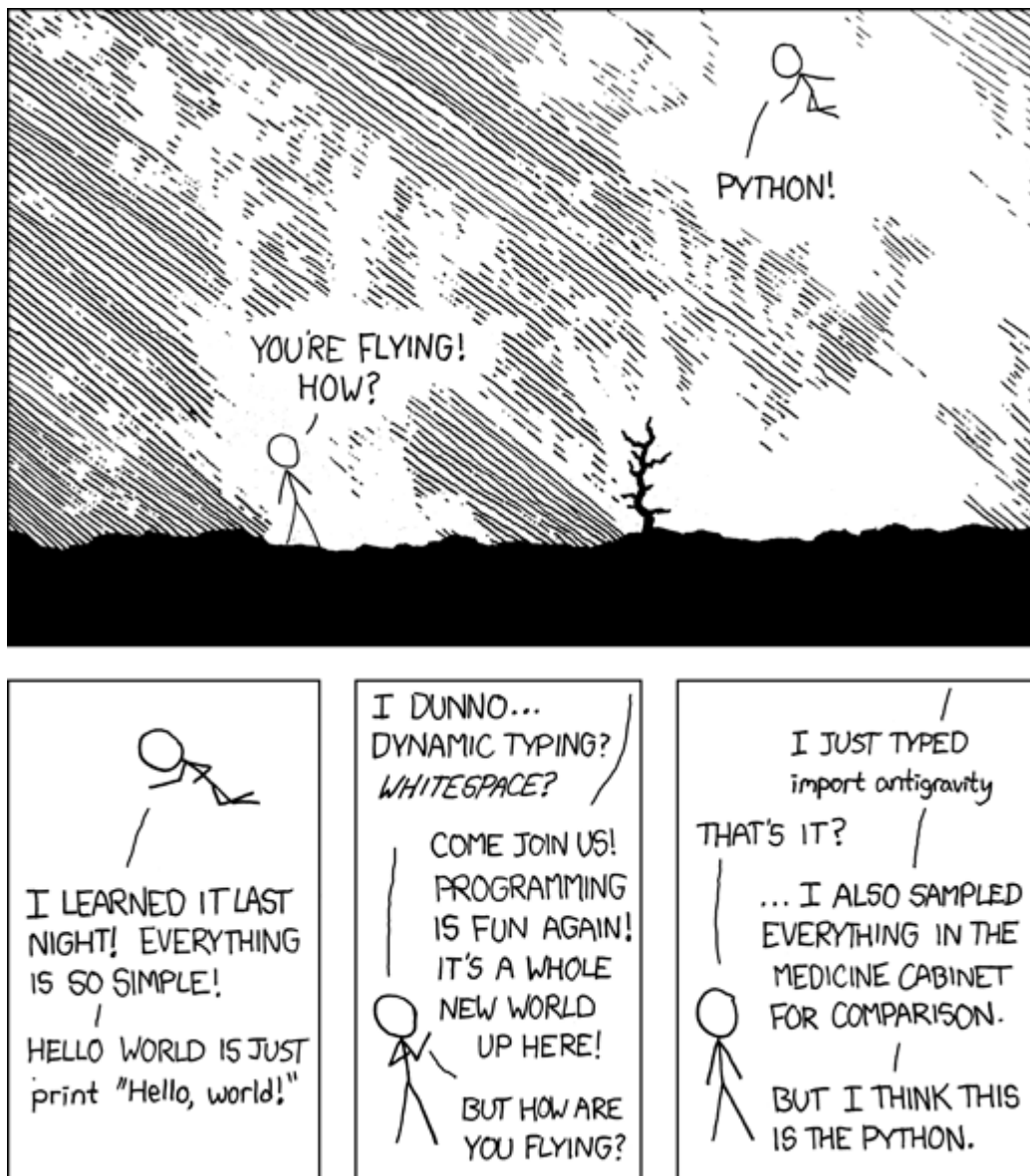


# Introduction to Python

---



## Introduction to Python

[General FAQ](#)

[Installation](#)

[Windows](#)

[Linux](#)

[MAC](#)

[Most important shortcuts in PyCharm](#)

[What is Python? Why Python?](#)

[Python 2.7 and Python 3.x](#)

[pip3 \(pip installs package\)](#)

[Python Console and Jupyter Notebook](#)

[Python Syntax Crashkurs](#)

[Hello, world!](#)

[Comments](#)

- Variable types
- Operators
- Loops and Conditions
- Basic Slicing and Indexing
- Methods and functions
- Exercise: Prime numbers
- Classes
- import package
- Lokale Imports
- Matplotlib
  - Exercise: Plot Collector
  - Exercise: Performance Optimization
- NumPy (Numeric Python)
  - Warum ist NumPy so schnell?
  - Fazit
  - n-dimensionales Indexing / Slicing mit numpy
  - Operationen in Numpy

## Image Processing

- Exposition for prettier slides
  - Step by step
- Discrete 2d-Convolution Filter
- Clipping
- Ich möchte noch mehr machen, weiß aber nicht was!
- References

# General FAQ

---

## Installation

### Windows

[Download Python3](#) When installing check "Add Path Variable". [Download PyCharm \(Community Edition\)](#)

### Linux

```
# install PyCharm IDE via snap package manager
sudo snap install pycharm-community --classic

# python3 is shipped with the most linux distributions so
# you DON'T NECESSARILY NEED TO INSTALL python. You can use your system's interpreter.
# If this should not be the case:
sudo apt-get install python3.7-dev
# IMPORTANT NOTE: When using your system's interpreter NEVER update your pip.
# This might cause many problems with your OS!
```

## MAC

[Download Python3](#) or get it with *brew*. [Download PyCharm \(Community Edition\)](#).

## Most important shortcuts in PyCharm

Shortcut	Effect
<code>Ctrl + P</code>	Show possible arguments
<code>Ctrl + Space</code>	Autocomplete
<code>Ctrl + Alt + L</code>	Apply code conventions
<code>Ctrl + Shift + F10</code>	Run current opened script
<code>Shift + F10</code>	Run script
<code>Ctrl + B</code>	Go to declaration/implementation
<code>Shift + F6</code>	Refactoring
<code>Ctrl + D</code>	Insert copy of the line to the next line
<code>Ctrl + Shift + Up/Down</code>	Move line upwards/downwards

## What is Python? Why Python?

- Interpreted High-Level Language
  - Multiple Programming paradigms supported (Procedural, Object-oriented, Functional, Imperative)
  - Garbage collected
  - Interpreter has to be installed on target system (No shipped runtime as in Java; No binaries as in C++)
- 'Pythonic' design philosophy
  - [PEP 20 - The Zen of Python](#)
  - [PEP 8 - Style Guide for Python Code](#)
- Cross-platform
  - Linux, Unix, Windows, Android, iOS
  - ARM architecture (e.g. [Raspberry Pi](#), [NVIDIA Jetson](#), [MicroPython on pyboard](#))
- Use case:
  - De facto standard for Machine Learning, Neural Nets (e.g. [Tensorflow](#), [PyTorch](#), [Caffe](#))
  - Data Mining (Alternative to R)
  - Scientific Computing (Alternative to Matlab by importing `numpy` and `matplotlib`; see [NumPy for Matlab users](#))
  - Web (e.g. Django)
  - Frontend (e.g. Qt, Tkinter)

## Python 2.7 and Python 3.x

Python 2.7 wird immernoch oft verwendet, da viele ältere Projekte noch darauf basieren, aber im Januar 2020 wird der Support eingestellt. **Wir verwenden hier deshalb ausschließlich Python 3!** Python 3 ist **nicht vollständig abwärtskompatibel**. In vielen Foreneinträgen sind Beispiele und Lösungen zu finden, die unter Python 3 nicht funktionieren - Finger weg davon! Es sollte stets auf die Version geachtet werden, die gerade behandelt wird. Die auffälligsten Unterschiede im Code, die direkt ins Auge springen sind folgende:

```
# Python 2.7
print "'print' is a keyword without brackets and integers need a cast " + str(42)

# Python 3.x
print("'print' is a function that", "can concatenate multiple strings", "comma seperated",
      "and numbers like", 42, "are okay too.")

# Python 2.7 only: It is possible to import some newer features from Python 3 with this
statement
from __future__ import *
```

## pip3 (pip installs package)

pip3 ist ein Paketverwaltungsprogramm, mit dem schnell und bequem Python-Bibliotheken per Konsolenbefehl installiert werden können.

```
# install numpy packages for python3 via pip
# (for windows use only 'pip')
pip3 install numpy

# if a requirement file is provided by a repository it can be used to install the required
packages automatically
pip3 install -r requirements.txt
```

## Python Console and Jupyter Notebook

Python kann (ähnlich wie Matlab) aus der Konsole heraus aufgerufen werden. Dies eignet sich zwar gut zum Testen und Zeigen von Beispielen, aber nicht zur Softwareentwicklung, weshalb diese Methode hier nicht vertieft werden soll.

```
>>> import numpy as np
>>> example = np.array([3,4,5])
>>> print("My example array looks like", example)
My example array looks like [3 4 5]
```

Jupyter Notebook verfolgt dagegen den Ansatz, kleinere Codeblöcke im Browser darzustellen und in eine Art Dokumentation einzubinden. Dies ist gut für Forschung und Lehre, aber ebenfalls nicht zur Entwicklung umfangreicherer Projekte geeignet.

## Python Syntax Crashkurs

---

### Hello, world!

```
print("Hello, world!")
```

By the way: Python is case sensitive!

### Comments

```
"""I'm a Docstring
Docstrings support multiple lines and are invoked with three double quotes.
"""

'''Multiline comments
Invoked with three single quotes.
'''

# simple comment with a hash
```

### Variable types

```
'''Datatypes must not be specified explicitly - in python they are dynamic.
This means that a variable can change it's datatype if necessary.
'''

## ATOMIC DATATYPES
# initialize x as integer
x = 5
# overwrite x with a float
x = 1.2
# overwrite x with a string
x = "Hello, world!"
# bool
x = True
x = False

## COLLECTIONS
# lists are sorted sets of values
my_list = [3.14, 2.71, 'foo', 42, [4, 2], 9] # list can contain floats, strings and
integers and another list
my_list.append(0.29) # add a new element to the list
```

```

my_list += ['bar'] # short hand for appending to lists is the '+' operator
print(my_list[0]) # prints the first item of the list: '3.14'

# tuples are like lists, but they are immutable
my_tuple = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

# dictionaries are unsorted sets. They link a key to a value.
my_dict = {'pi': 3.14,
           'e': 2.71,
           'dictionary': 'woerterbuch'}
print(my_dict['dictionary']) # prints 'woerterbuch'
my_dict['sqrt_2'] = 1.41 # add a new entry to the dictionary
print(my_dict['pi']) # get a dictionary's entry by it's key

```

## Operators

```

print(2+3)      # 5          addition
print(3*4)      # 12         multiplication
print(2**10)    # 1024       power
print(7/3)      # 2.3333333333 division
print(7//3)     # 2          integer division
print(7%3)      # 1          modulo division

i += 1 # increment (i++ is not possible in python)

```

## Loops and Conditions

```

# instead of {curved brackets} python uses indentation (with 4 spaces)
if x > 5:
    print("Hello, world!")
    print("Everything indented will be in the condition")

print("This is not indented anymore so it does not care about the if condition.")

if some_kind_of_boolean_flag:
    print("I'm in the if-condition!")
elif y == 9:
    print("I'm in the else-if-condition!")
else:
    print("I'm in the else-condition!")

# todo: xor???
while x <= 42 or x == 4711:
    print(x)
    x += 1

```

```

# For loops in python need an iterable to loop over like a list or array.
my_list = [2, 4, 6, 8]
for it in my_list:
    print(it)

# The 'range' and function can provide such an iterable.
# 'range(10)' gives 0..9
for it in range(10):
    print(it)

# The 'range' function provides further optional parameters to declare start and stepwidth.
# To see, what arguments a function can take, use the 'Ctrl + P' shortcut of PyCharm!
for it in range(2, 2048, 8):
    print(it)

# The 'enumerate' function provides the elements of a list or array and counts them
for index, element in enumerate(my_list):
    print(index, element)

```

## Basic Slicing and Indexing

```

# create a python list
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# access single values by index
print(lst[5])    # 5
print(lst[-1])   # 9 --> negative numbers count backwards from the end

# you can access a subset by slicing in the form of [start:end]
print(lst[2:6])  # [2, 3, 4, 5]          --> all values from index 2 to 6 (exclusive)
print(lst[5:])   # [5, 6, 7, 8, 9]      --> no end given means take all to the end
print(lst[:3])   # [0, 1, 2, 3, 4, 5, 6] --> all up to the last three values

```

## Methods and functions

```

# PEP 8: Function names should be lowercase, with words separated by underscores
# as necessary to improve readability.
def my_function(a, b):
    sigma = a + b
    return sigma

# call function with arguments
c = my_function(2, 3)

# A function can have positional arguments and optional arguments with default values

```

```
def analyze_array(array, start=0, end=None):
    sigma = sum(array[start:end])
    mean = sigma / len(array[start:end])

    # A function can have multiple return values
    return sigma, mean

my_list = [2, 4, 6, 8]
# multiple return values are written comma separated
sum_all, mean_all = analyze_array(my_list) # call function without optional args
sum_some, mean_some = analyze_array(my_list, end=-1) # call function with optional args

print(sum_all, mean_all)
print(sum_some, mean_some)
```

## Exercise: Prime numbers

Schreiben Sie ein skript, das...

- ...alle Primzahlen von 2 bis 1000 in einer Liste speichert.
- ...die 12. Primzahl ausgibt.

## Classes

```
# PEP 8: Class names should normally use the CapWords convention.
class MyClass:
    def __init__(self, a, b):
        """Constructor"""
        self.a = a
        self.b = b

    def multiply(self):
        return self.a * self.b

    def pythagoras(self):
        return math.sqrt(self.a ** 2 - self.b ** 2)

    def set_values(self, a, b):
        self.a = a
        self.b = b

# get new instance
my_instance = MyClass(2, 5)

# call its functions
print(my_instance.pythagoras())
my_instance.set_values(7, 13)
```



# import package

Eine der größten Stärken von Python liegt in den Bibliotheken, die für die meisten Problemstellungen bereits vorgefertigte Funktionen bieten. Wie bereits erwähnt, kann mit Pip3 ein benötigtes Paket schnell installiert, und mit der import Funktion sofort verwendet werden:

```
# install needed packages in your operating system terminal
# (for windows use only 'pip')
pip3 install numpy
```

```
# in your python file import the package with the namespace of your choice
import time

import tqdm
import numpy as np

# and you can use them
my_numpy_array = np.array([[1, 2, 3],
                           [4, 5, 6]])

my_numpy_array_reshaped = np.reshape(my_numpy_array, [np.size(my_numpy_array)])

for element in tqdm.tqdm(my_numpy_array_reshaped, desc="Waiting"):
    time.sleep(element)
```

Die wichtigsten Standardbibliotheken	
os	Operating System. Dateien öffnen/speichern, Dateipfade, etc.
sys	Zugriff auf den verwendeten Python Interpreter
math	Mathematische Operationen und Konstanten
time	Timer, Wartezeiten, etc.
timeit	Performancetesting von kurzen Codeabschnitten
argparse	Parsen von Argumenten beim Kommandozeilenaufruf

Die wichtigsten third party packages	
numpy	Performante Berechnungen und bequeme Handhabung von Vektoren, Matrizen, mehrdimensionalen Matrizen und generellen numerischen Operationen
pandas	Performer Zugriff auf komplexe Datenstrukturen. Beispielsweise import von *.csv Dateien.
matplotlib	Grafische Darstellung von Daten und mathematischer Funktionen
scipy	Lineare Algebra, numerische Integration, Interpolation, FFT, Signalverarbeitung, Bildverarbeitung

## Lokale Imports

Um Codeteile zu kapseln und wiederverwendbar zu gestalten, kann jedes Python Skript als Modul importiert werden.

Angenommen die beiden Dateien `main.py` und `utils.py` befinden sich im gleichen Ordner.

```
# this is the python utility file 'utils.py'
# defined functions and globals can be used by other files when imported
GLOBAL_STRING_EXAMPLE = "dubidubida"

def fancy_function():
    print("fancy")

def knowing_function():
    print(42)

if __name__ == '__main__':
    # this will only be executed if this file is executed (not imported)
    fancy_function()
    knowing_function()
```

```
# this is the python main file 'main.py'
import utils # import local python file

print(utils.GLOBAL_STRING_EXAMPLE) # use imported file as module
utils.fancy_function()
```

## Matplotlib

Das Standardtool zum plotten von mathematischen Funktionen oder Daten ist Matplotlib.

```
import matplotlib.pyplot as plt

# add a scatter plot
dataset_0 = [1, 10, 12, 0.5, 4, 3, 2, -3]
plt.scatter(list(range(len(dataset_0))), # values for x axis
            dataset_0) # values for y axis

# add a function plot
dataset_1 = []
for it in range(8):
    dataset_1 += [it ** 2]

plt.plot(dataset_1)

# show plot window
plt.show()
```

## Exercise: Plot Collector

```
# in terminal (for Windows use 'pip' instead of 'pip3')
pip3 install matplotlib
```

```
# in python script
import matplotlib.pyplot as plt
```

1. Erstellen Sie eine Liste mit Werten (Datensatz)
2. Plotten Sie diesen mit matplotlib
3. Erstellen Sie eine Klasse, die...
  1. ...mehrere Datensätze enthalten kann.
  2. ...eine Möglichkeit bietet, weitere Datensätze hinzuzufügen
  3. ...eine Möglichkeit bietet, alle gesammelten Datensätze in einem Plot darzustellen.
  4. ...eine Möglichkeit bietet, die Anzahl der aktuell hinzugefügten Datensätze auszugeben

## Exercise: Performance Optimization

Sum of multiples of two numbers. If we list all the natural numbers below 10 that are multiples of 3 or 5 we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of a or b below n. When you found your first solution, think again how exactly the described sequence of numbers works. Try to find a pattern and how you could design an algorithm to speed up the calculation time.

```
# the timeit module can be used to compare the performance of small code snippets
```

```
import timeit

def solution_simple(a, b, n):
    """This is my simple and naive approach to the problem"""
    pass # <code goes here>

if __name__ == '__main__':
    a = 3
    b = 5
    n = 10
    repetitions = 50

    # stop time
    time_solution_simple = timeit.timeit(lambda: solution_simple(a, b, n),
                                         number=repetitions) / repetitions

    print("solution_simple", time_solution_simple)
```

For different solutions to the problem see the file

`./Exercise_PerformanceOptimization/performance_optimization.py`.

## NumPy (Numeric Python)

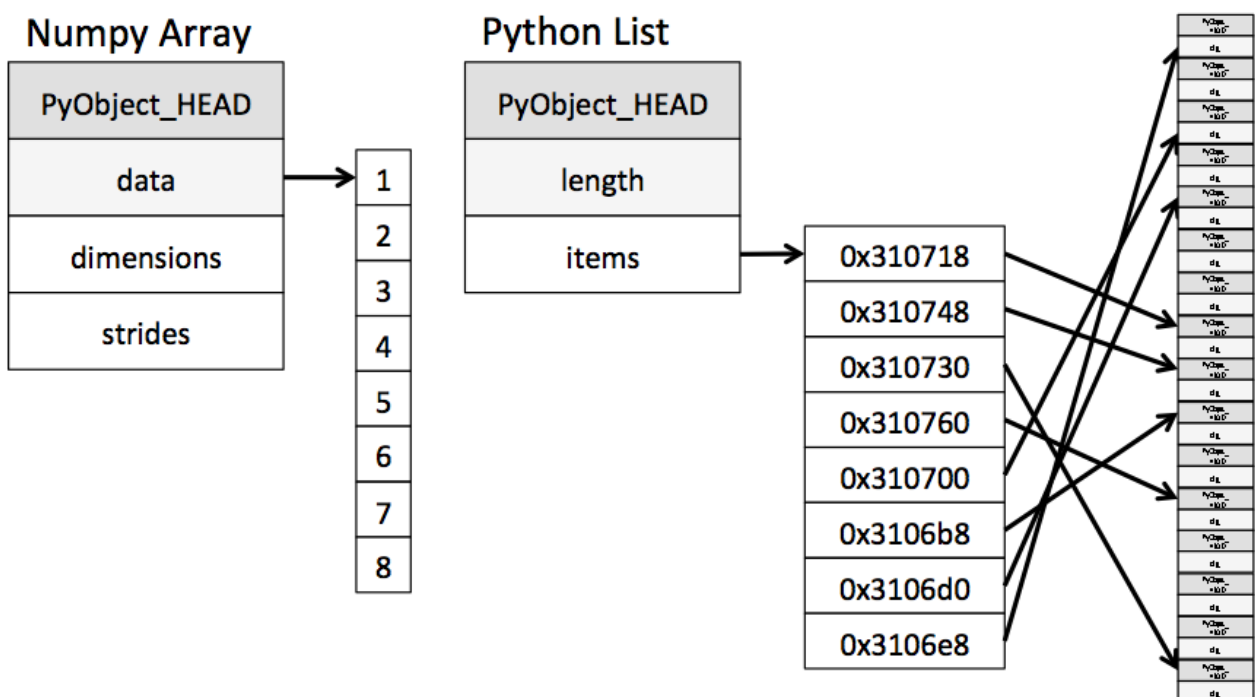
### Warum ist NumPy so schnell?

NumPy ist ein Python Paket für numerische Operationen und das Kernelement für Scientific Computing in Python. Der Standarddatentyp hier ist das n-dimensionale `ndarray`. NumPy bietet darüber hinaus diverse Funktionen zur Verarbeitung von Matrizen. Diese sind sehr viel schneller als die Python-internen Standardoperationen und Datentypen. Um einmal ein paar Gründe dafür zu nennen:

1. **Precompiled C** Die meisten Funktionen in NumPy sind in C geschrieben und sind vorcompiliert. Python dagegen wird interpretiert. D.h. der Standardcompiler in Python (CPython) nimmt nur geringfügige Prozessoptimierungen vor, um zur Laufzeit schnell arbeiten zu können. Mit dem C-Compiler dagegen genießt man diesen Vorteil und redundante Operationen werden "wegoptimiert".
2. **Statically typed** Wenn der Python Interpreter auf eine Python-Variable zugreift, weiß er zunächst lediglich, dass es sich um ein Objekt handelt - nicht aber, was für eine Art von Objekt oder wie groß es ist. Das heißt der Interpreter muss erst im Object-Header der Variable nachschauen, dass es sich beispielsweise um einen Integer handelt. Dies kostet Zeit. In NumPy dagegen haben alle Elemente eines Arrays ab der Deklaration den gleichen Datentyp und die Größe des Arrays ist festgelegt. (Sobald das Array verändert werden soll, wird ein neues angelegt und das alte gelöscht.) Somit wird die Abfrage des Typs übersprungen.
3. **Vectorization instead of loops** Grundsätzlich gilt es, Schleifen stets zu vermeiden, wenn man eine hohe Performanz erreichen möchte, weil dabei jedes Element einzeln betrachtet und verarbeitet werden muss. Wenn man aber dieselbe Operation auf mehrere Elemente anwenden will, kann man dagegen sogenannte *Vektorisierung* nutzen. Dabei parallelisiert man die Verarbeitung, indem man beispielsweise SIMD (Single Instruction Multiple Data) Funktionen der CPU nutzt. Dies ist bei der Anwendung von

Operationen auf Matrizen der Fall, weshalb das NumPy-Paket auch vektorisiert geschrieben ist. Deshalb kann es große Arrays so schnell verarbeiten.

4. **Locality of reference** Listen in Python sind Arrays von Pointern. Deshalb sind die Daten einer Python List nicht zwangsläufig hintereinander im Speicher abgelegt. NumPy, sowie C-/Fortran-Arrays dagegen allokalieren einen Speicherbereich. Dies ist dadurch überhaupt möglich, dass sie einen statischen Datentyp und eine feste Länge haben. Somit ist der erforderliche Speicherbereich für ein Array in NumPy bekannt und die Daten können zusammenhängend im Speicher abgelegt werden. Durch diese [Räumliche Lokalität](#) kann auf die Daten schneller zugegriffen werden, als wenn sie unsortiert verteilt wären. Dieser Effekt lässt sich beobachten, wenn man in NumPy eine 2D-Matrix anlegt und die gleiche Operation einmal auf dessen Zeilen und einmal auf dessen Spalten anwendet. In C werden Arrays spaltenweise im Speicher angelegt und in Fortran zeilenweise. NumPy unterstützt beide Möglichkeiten. Mit dem Skript `./Exercise_PerformanceOptimization/numpy_performance.py` kann dieser Performanceunterschied beobachtet werden.



## Fazit

Python-interne Variablen und Operationen sind nicht direkt langsam. Sie sind lediglich die falsche Wahl für die Anwendung numerischer Operationen auf große Datenmengen.

## n-dimensionales Indexing / Slicing mit numpy

```
# numpy can handle multidimensional arrays
import numpy as np

# syntax for a 2d array is like an array of arrays
arr_2d = np.array([[0, 1, 2],
                   [3, 4, 5]])
```

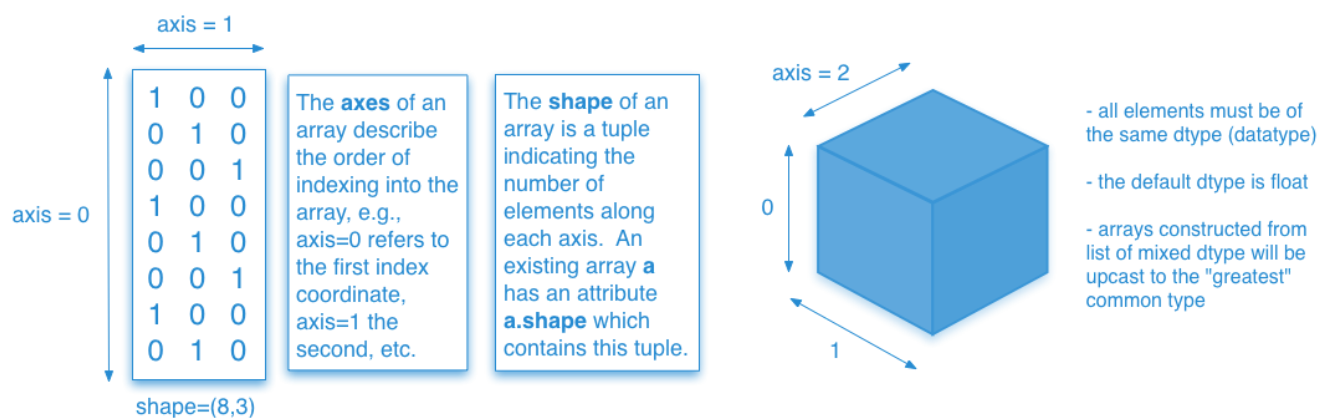
```

print(arr_2d[1, 2])    # 5
print(arr_2d[:, 1])    # [1, 4] --> ':' means all values of that axis

# same for 3d array (more than three are possible too)
arr_3d = np.zeros([2, 3, 4]) # create 2x3x4 array filled with zeroes
arr_3d[1, 1, 1] = 1
print(arr_3d)          # [[0. 0. 0. 0.]
                        #  [0. 0. 0. 0.]
                        #  [0. 0. 0. 0.]
                        #  [0. 0. 0. 0.]
                        #  [0. 1. 0. 0.]
                        #  [0. 0. 0. 0.]]

```

## Anatomy of an array



## Operationen in Numpy

```

# Numpy arrays aren't lists! Lists are Stack-like collections.
my_list = [0, 1, 2]
print(my_list * 2) # prints [0, 1, 2, 0, 1, 2] --> instead of multiplication the list is
                  # concatenated with itself

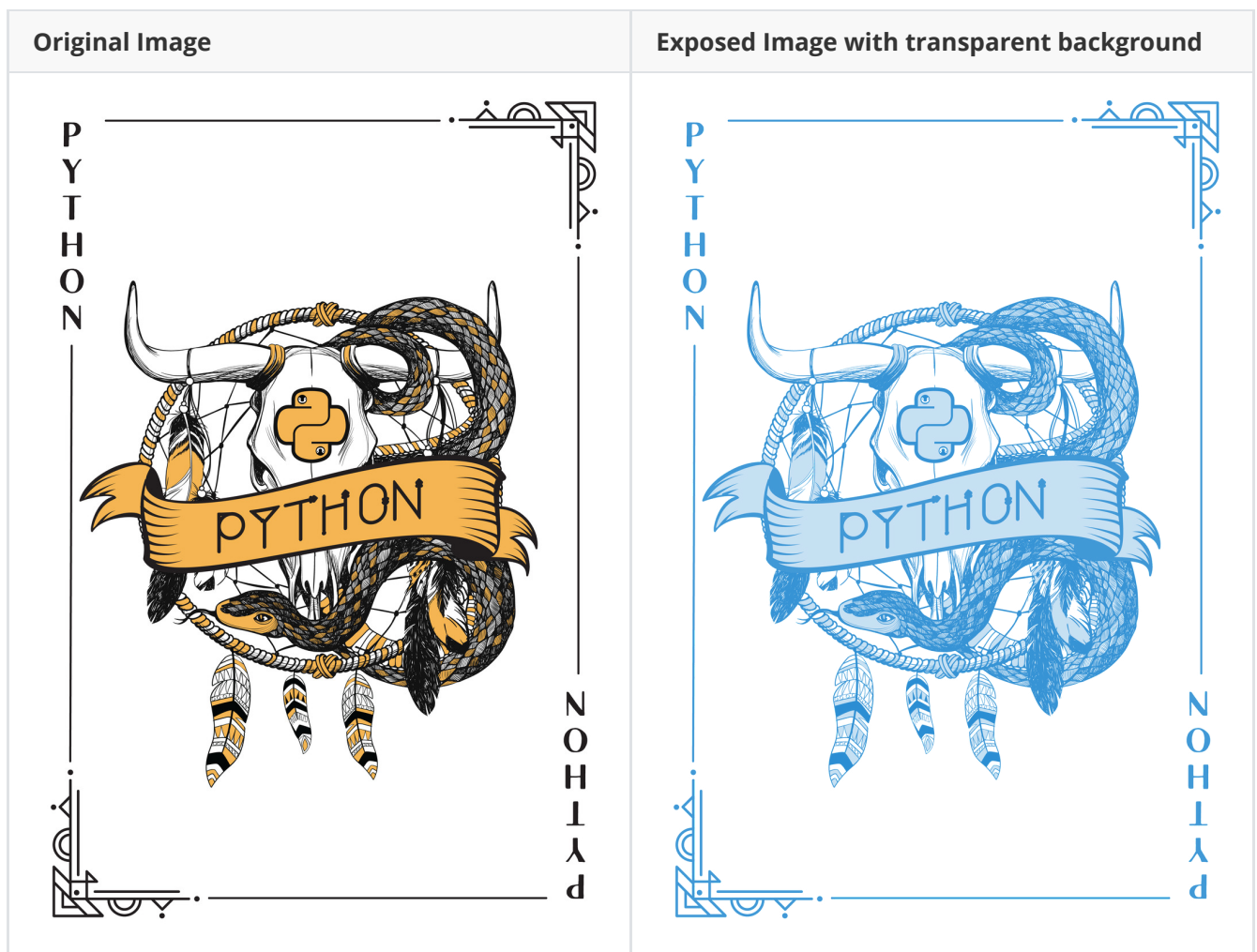
# Numpy arraydot_products are made for numeric operations:
my_narray = np.array([0, 1, 2])
print(my_narray * 2) # prints [0, 2, 4] --> Multiplication

# Numpy gets fast by avoiding for loops. Instead it's own functions should be preferred.
# For example:
shape = np.shape(arr_2d) # Gives the dimensions as tuple: (2, 3)
size = np.size(arr_2d)   # Gives the number of elements: 6
dotproduct = np.dot(arr_2d, [1, 2, 3]) # Dot product of two arrays: array([8, 26])

```

## Image Processing

## Exposition for prettier slides

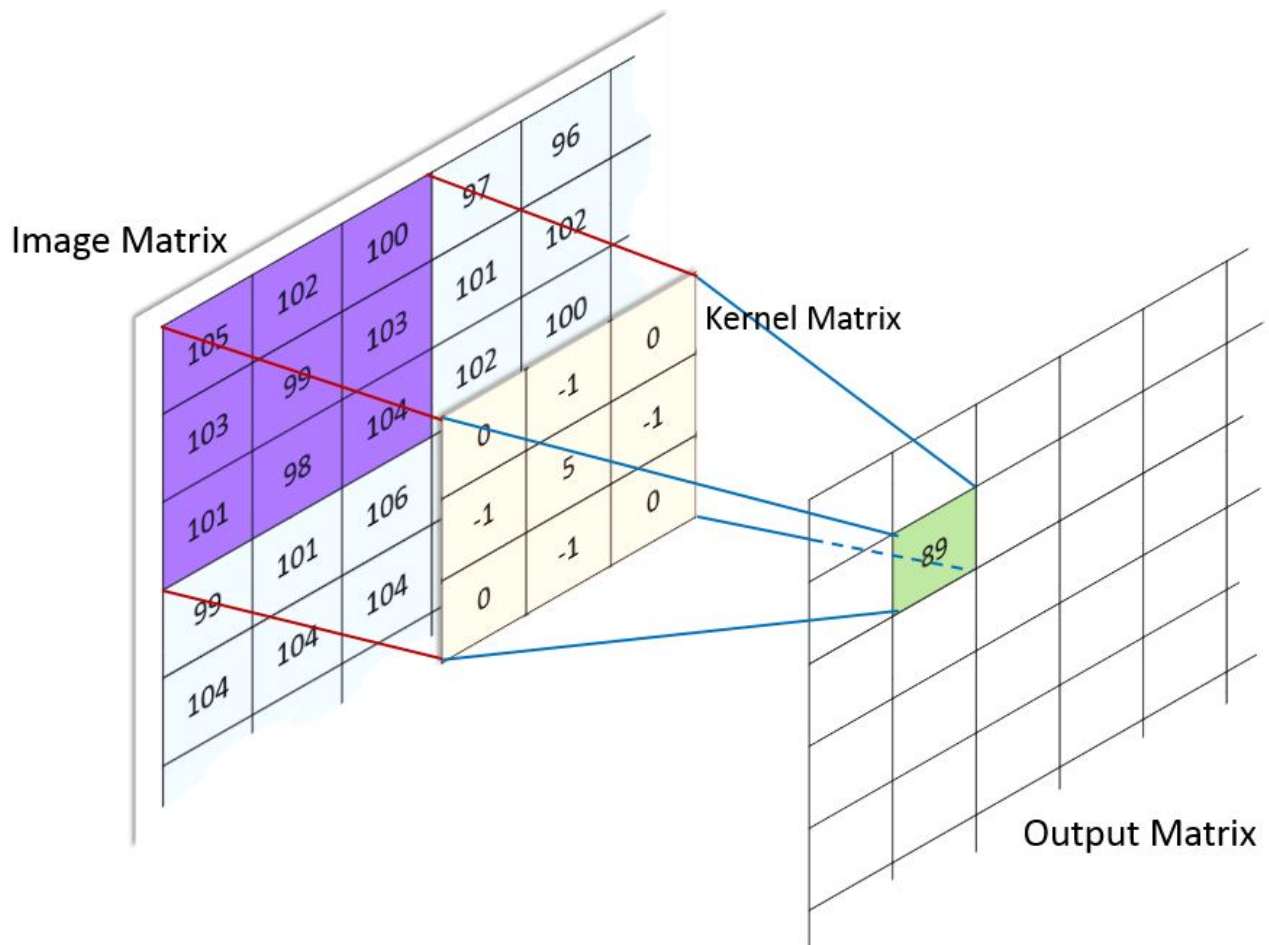


### Step by step

- Load image `io.imread('./filename.jpg')`
- Convert to grayscale `np.....`
- Generate empty 4-channel array, same size as input image  
`np.empty([height, width, 4])`
- Set RGB to chosen color
- Inverted alpha to grayscale of input image
- Save output image

## Discrete 2d-Convolution Filter

Eine diskrete Faltung (en.: convolution) sieht in der Praxis so aus, dass ein Ausschnitt aus der Image Matrix mit einer Kernel Matrix multipliziert und dann aufsummiert wird. Das Ergebnis wird an die entsprechende Stelle in der Output Matrix geschrieben. Dieser Prozess wird für jeden Pixel ausgeführt.



```
# for image processing we can use these libraries
import numpy as np # numeric python
from skimage import io # load and save image files
from scipy import ndimage # n-dimensional image processing

# different kernels will have a different effect on the image
'''BLUR
For a simple blur like the so called 'Box Blur' we calculate the arithmetic mean.
We weight every pixel the same and normalize so that the sum of the kernel is 1.
Think about, why the sum of the kernel must be 1.
'''
kernel_box_blur = np.array([[1, 1, 1],
                             [1, 1, 1],
                             [1, 1, 1]])/9

'''EDGE DETECTION
For an edge detection the sum of the kernel must be zero.
This way the value in the output matrix will be zero, which means black,
if there is no difference between the centerpoint and its surrounding
pixels.
'''
kernel_edge_detection = np.array([[-1, -1, -1],
```



```
[-1, 8, -1],  
[-1, -1, -1]])
```

```
'''SHARPEN
```



```
To sharpen an image we use the same principle as for the edge detection.  
The only difference is that we have a sum of 1 caused by a higher weight  
on the centerpoint.
```

```
'''
```

```
kernel_sharpen = np.array([[0, -1, 0],  
                           [-1, 5, -1],  
                           [0, -1, 0]])
```

```
# CONVOLUTION
```

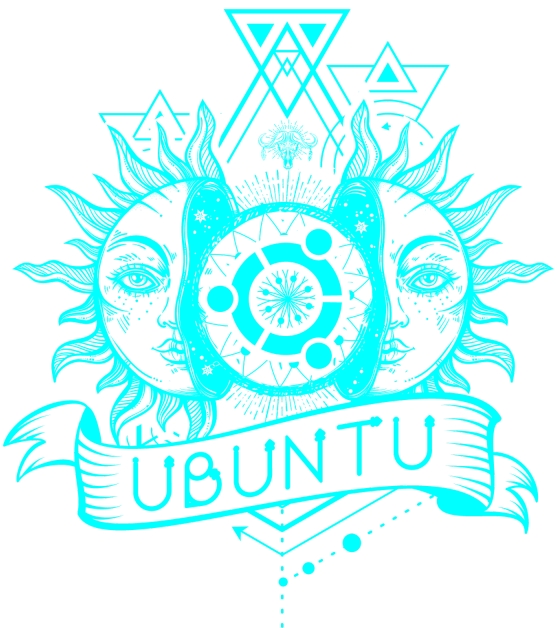
```
img_convolved = ndimage.convolve(img, kernel, mode='reflect')
```

Original Image	Blurred and edge detection
 The original Ubuntu logo is a high-contrast black and white image. It features a central circular emblem with a sunburst pattern, flanked by two stylized faces with large eyes. Above the faces are several geometric triangles, and below them is a banner with the word 'UBUNTU' in a bold, sans-serif font. The entire logo is set against a solid black background.	 The result of applying a blur and edge detection filter to the original logo. The image is rendered in a light blue/gray color on a white background. The edges of the logo's components are clearly defined, while the interior areas are blurred, creating a soft, ethereal effect.

Visit [the regarding Wikipedia article](#) for more examples.

## Clipping

Bonus: If background isn't pure black or white, clip and normalize values.

Without clipping	With clipping
	

## Ich möchte noch mehr machen, weiß aber nicht was!

- Weitere Zahlenprobleme finden sich unter [projecteuler.net](http://projecteuler.net)
- Image Processing
  - Weiterer Filter zur Kantenerkennung: Laplace of Gaussian Hier kann mit der Kernelgröße experimentiert werden. Mit `ndimage.generic_filter` aus dem `scipy` Paket können auch andere Berechnungen als Multiplikation und Summe mit einem Kernel implementiert werden, um andere Effekte zu erzielen.
  - Mit dem `argparse` Modul kann ein Commandline Interface erstellt werden. Damit können Ein- und Ausgabepfade sowie Flags (z.B. für hellen/dunklen Hintergrund) gesetzt werden. Befehle für die Pfadangaben finden sich in `os.path`.

## References

- <http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/python/anatomyarray.png>
- <https://imgs.xkcd.com/comics/python.png>
- <https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/>
- <https://linux.pictures/content/1-projects/39-python-card-jpg/python-card.jpg>

- <https://i.stack.imgur.com/vxEa3.jpg>