```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# """
# Created on Friday  Nov  16 21:31:07 2018
#
# @author: achim
# """
```

Lektion 6

# 1 Wiederholung: Definition von Arrays

```
import numpy as np
```

Definiere Matrix

```
A = np.array([[1, 2], [3, 4]])
A
```

```
array([[1, 2],
       [3, 4]])
```

Definiere Vektor

```
v = np.array([1, 2])
v
```

```
array([1, 2])
```

# 2 Einfache Matrizen / Vektoren

Matrix mit Einsen

```
E = np.ones((3, 2))
E
Ei = np.ones_like(A)
Ei
```

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

Nullmatrix

```
Z = np.zeros((3, 2))
Z
```

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

Einheitsmatrix Identität

```
I = np.eye(3)
I
```

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

## 3 Einfache Operationen für Arrays

transponierte Matrix

```
A
A.T
A.transpose()
```

```
array([[1, 3],
       [2, 4]])
```

Spur

```
A.trace()
```

```
5
```

Eigenschaften

```
A.size
A.shape
A.ndim
```

```
2
```

komplexwertige Matrix

```
C = np.array([[1.+2.j, 2.-1.j], [2.+1.j, 3.-5.j]]);
C
```

```
array([[1.+2.j, 2.-1.j],
       [2.+1.j, 3.-5.j]])
```

transponiert-konjugierte / adjungierte Matrix

```
C.T.conj()
C
C.real
C.imag
```

```
array([[ 2., -1.],
       [ 1., -5.]])
```

### 3.1 Achtung: +, -, * , %, /, ** wirken eintragsweise

Die Standardoperationen wirken eintragsweise auf die Einträge der Arrays.

A * A ist also KEINE Matrix-Matrix-Multiplikation!

eintragsweise +

```
A+A
```

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

eintragsweise *

```
A*A
```

```
array([[ 1,  4],
       [ 9, 16]])
```

eintragsweise /

```
C/A
```

```
array([[1.         +2.         j, 1.         -0.5        j],
       [0.66666667+0.33333333j, 0.75       -1.25       j]])
```

eintragsweise **

```
A**A
```

```
array([[  1,   4],
       [ 27, 256]])
```

### 3.2 Broadcasting / Expansion in der Singleton Dimenson

+, -, * , %, / wirken eintragsweise wobei die fehlende Dimension expandiert wird.

```
v = np.array([-5, 7])
v
```

```
array([-5,  7])
```

Das erste Element von v wird zu jeder zu jedem Element der ersten Spalte addiert

```
A+v
```

```
array([[-4,  9],
       [-2, 11]])
```

Das erste Element von v wird ...

```
A*v

A.astype(float)**v
```

```
array([[1.00000000e+00, 1.28000000e+02],
       [4.11522634e-03, 1.63840000e+04]])
```

zu jedem Eintrag wird 1 addiert

```
A + 1
```

```
array([[2, 3],
       [4, 5]])
```

jeder Eintrag wird durch 3 dividiert

```
A / 3
```

```
array([[0.33333333, 0.66666667],
       [1.        , 1.33333333]])
```

3 wird durch jeden Eintrag dividiert

```
3 / A
```

```
array([[3.  , 1.5 ],
       [1.  , 0.75]])
```

### 3.3 Matrix-Multiplikation

mit np.dot

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

```
array([[ 7, 10],
       [15, 22]])
```

@ Operator verkürzte Notation: @ Statt des "dot"-Befehls kann man auch das @-Zeichen verwenden.

```
A@A
```

```
array([[ 7, 10],
       [15, 22]])
```

Matrix-Vektor Produkt funktioniert genau so

```
v = np.array([1, 2])
v
np.dot(A, v)
```

```
array([ 5, 11])
```

Auch hier kann @ verwendet werden

```
A@v
```

```
array([ 5, 11])
```

Skalarprodukt zweier Vektoren

```
np.dot(v, v)
```

```
5
```

Beachte: gleiche Resultate

```
np.dot(v, v.T)
```

```
5
```

v@v

```
np.dot(v.T, v)
```

```
5
```

Wieso ist das so? 'v' ist KEIN Zeilen/Spaltenvektor (hat nur eine Dimension), also bewirkt das Transponieren nichts.

Achtung

```
xxl = np.array([i**2 for i in range(18952)])
xxl@xxl
```

```
-9130673722756036796
```

Kreuzprodukt / Vektorprodukt mit np.cross

```
x = np.array([1, 0, 0])
y = np.array([0, 1, 0])
np.cross(x, y)
```

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

Dyadisches Produkt / tensorielles Produkt (Multiplikation eines Spaltenvektor mit einem Zeilenvektor liefert Matrix)

```
np.outer(v, v)
```

```
array([[1, 2],
       [2, 4]])
```

Mehr Informationen zu Operationen auf Arrays finden Sie unter

https://www.python-kurs.eu/numpy_numerische_operationen_auf_arrays.php

### 3.4 Übersichtlicher: Definiere Vektoren als Zeilen- bzw. Spaltenvektor

@ funktionert dann immer so wie man es aus der linearen Algebra erwartet, d.h. Zeilenvektor @ Spaltenvektor liefert skalare Größe, Spaltenvektor @ Zeilenvektor liefert Matrix

```
v = np.array([1, 0]).reshape(2, 1)  # Spaltenvektor
u = np.array([0, 1]).reshape(1, 2)  # Zeilenvektor
A = np.array([[1, 2], [3, 4]])
```

Zahl

```
u@v
```

```
array([[0]])
```

Matrix

```
v@u
```

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

Matrix-Vektor-Produkt

```
A@v
```

```
array([[1],
       [3]])
```

Das klappt auch von links

```
u@A
```

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

Achtung, hier stimmen die Dimensionen nicht mehr!

```
A@u
```

```
---------------------------------------------------------------------------ValueError
call last)<ipython-input-1-8cd09a6d7873> in
<module>()
----> 1A@u
ValueError: shapes (2,2) and (1,2) not aligned:
2 (dim 1) != 1 (dim 0)
```

Weitere Informationen zu reshape: https://www.python-kurs.eu/numpy_dimensionen_anpassen.php

# 4 Weitere Arrayoperationen

np.amax(A) oder A.max() liefert das maximale Matrixelement:

```
np.max(A)
A.max()
```

```
4
```

Maximum entlang der ersten Achse

```
A.max(1)
```

```
array([2, 4])
```

Maximum der 1. Zeile

```
A
np.max(A[0, :])
```

```
2
```

"flattens" A in ein 1-dim. Array

```
A.flatten()
```

```
array([1, 2, 3, 4])
```

np.argmin(B) / np.argmax(B) liefert Index bezüglich "flattened array" des kleinsten / größten Matrixelements Falls das größte / kleinste Element mehrfach vorkommt, dann erhält man denersten Index

```
B = np.array([[2, 1], [1, 2]])
B
B.max()
B.argmax()
```

```
0
```

liefert das (erste) maximale Element

```
B.flatten()[np.argmax(B)]
```

```
2
```

Unterschied zu np.maximum

```
display(A,B,np.maximum(A,B))
```

```
array([[1, 2],
       [3, 4]])
```

```
array([[2, 1],
       [1, 2]])
```

```
array([[2, 2],
       [3, 4]])
```

broadcasting wird angewendet

```
v = np.array([5,1])
v
np.maximum(A,v)
```

```
array([[5, 2],
       [5, 4]])
```

## 4.1 Weitere Möglichkeiten zur Definition von Arrays

```
A = np.arange(12)
```

A ist ein 1D array

```
A.shape
```

```
(12,)
```

A.reshape(i,j) transformiert A in ein array mit i Zeilen und j Spalten B hat zwei Dimensionen

```
B = A.reshape(3, 4)
B.shape
B
```

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

Die Anzahl der Elemente ändert sich natürlich nicht

```
A.size-B.size
```

```
0
```

Wichtig: A und B zeigen noch immer auf die selben Einträge im Speicher!

```
A[3] = -42
B[0,0] = -567

C = A.reshape(3, 4).copy()
C[0] = 123
display(A, B, C)
```

```
array([-567,    1,    2,  -42,    4,    5,    6,    7,    8,    9,
10,
         11])
```

```
array([[-567,    1,    2,  -42],
       [   4,    5,    6,    7],
       [   8,    9,   10,   11]])
```

```
array([[123, 123, 123, 123],
       [  4,   5,   6,   7],
       [  8,   9,  10,  11]])
```

## 4.2 np.diag, np.triu, np.tril

```
A = np.arange(16).reshape(4, 4)
A
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Diagonale von A

```
np.diag(A)
```

```
array([ 0,  5, 10, 15])
```

Untere Dreiecksmatrix

```
np.tril(A)
```

```
array([[ 0,  0,  0,  0],
       [ 4,  5,  0,  0],
       [ 8,  9, 10,  0],
       [12, 13, 14, 15]])
```

Obere Dreiecksmatrix

```
np.triu(A)
```

```
array([[ 0,  1,  2,  3],
       [ 0,  5,  6,  7],
       [ 0,  0, 10, 11],
       [ 0,  0,  0, 15]])
```

np.diag kann auch Diagonalmatrizen erzeugen

```
B = np.diag(v.flatten())
B
```

```
array([[5, 0],
       [0, 1]])
```

np.diag für Nebendiagonalen

```
A
NU = np.diag(A, k=1)  # obere Nebendiagonale
NU
NL = np.diag(A, k=-1) # untere Nebendiagonale
NL
```

```
array([ 4,  9, 14])
```

Benutze np.diag zur Definition von Matrizen

```
x = [1, 2, 3]
A = np.diag(x, k=-1) + np.diag(x, k=1)
A
```

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

### 4.3 Index Arrays

Wir können Integer-Arrays verwenden um andere Arrays zu definieren

1. Beispiel zur Verwendung von Index-Arrays

```
# Array mit den ersten 8 Zweierpotenzen
a = 2**np.arange(8);
a
# Array mit geraden Zahlen von 0 bis 7
i = 2*np.arange(4); # Index-Array
i
# Jedes zweite Element von a in b speichern
b = a[i]
b
```

```
array([ 1,  4, 16, 64])
```

Beachte: Diesen Vektor können wir auch folgendermaßen erzeuge:

```
a[::2]
```

```
array([ 1,  4, 16, 64])
```

Die Konstruktion unter Verwendung von Index-Arrays ist aber im Allgemeinen flexibler

2. Beispiel zur Verwendung von Index-Arrays

```
# Erzeuge einen Vektor, der die Gegendiagonale einer Matrix enthält
A = np.arange(16).reshape(4, 4)
A
i = np.arange(4)
j = i[::-1]
j
i
b = A[i, j]
b
```

```
array([ 3,  6,  9, 12])
```

# 5 Verwende boolsche Ausdrücke bei der Definition von Matrizen

Bsp: Setze alle negativen Matrixeinträge auf Null

```
A
B = A - A.T
B
B<0
B[B < 0] = 0
B
```

```
array([[0, 0, 0, 0],
       [3, 0, 0, 0],
       [6, 3, 0, 0],
       [9, 6, 3, 0]])
```

elementweises logisches “oder” und “und”

```
B = A - A.T
B
```

```
array([[ 0, -3, -6, -9],
       [ 3,  0, -3, -6],
       [ 6,  3,  0, -3],
       [ 9,  6,  3,  0]])
```

Welche Elemente von B sind kleiner als -5 ODER größer als 6?

```
IA = np.logical_or(B < -5, B > 6)
IA
```

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

Welche Elemente von B sind kleiner als 2 UND größer als 0?

```
np.logical_and(B < 7, B > 0)

B[IA] = 100
B
```

```
array([[  0,  -3, 100, 100],
       [  3,   0,  -3, 100],
       [  6,   3,   0,  -3],
       [100,   6,   3,   0]])
```

# 6 Kronecker Produkt

Erzeugt ein zusammengesetztes Array aus Blöcken des 2. Arrays unter Verwendung der durch das 1. Array beschriebenen Skalierung

einmal so rum

```
np.kron([1, 10, 100], [5, 6, 7])
```

```
array([  5,   6,   7,  50,  60,  70, 500, 600, 700])
```

anders rum kommt was anderes raus

```
np.kron([5, 6, 7], [1, 10, 100])
```

```
array([  5,  50, 500,   6,  60, 600,   7,  70, 700])
```

np.kron(np.eye(2), np.ones((2, 2)))

```
np.kron(np.ones((2, 2)),np.eye(2))
```

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

## 6.1 vstack und hstack

Füge eine Zeile oder Spalte b zu einer 3x3 Matrix A hinzu

Zeile anhängen mit np.vstack:

```
A = np.ones((3, 3))
b = np.array((2, 2, 2))
np.vstack([A, b])
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [2., 2., 2.]])
```

Spalte anhängen mit np.hstack

```
A = np.ones((3, 3))
b = np.array((2, 2, 2)).reshape(3, 1)  # erzeugt Spaltenvektor
np.hstack([A, b])
```

```
array([[1., 1., 1., 2.],
       [1., 1., 1., 2.],
       [1., 1., 1., 2.]])
```

Alternativ kann man auch die (mächtigere) Funktion “np.concatenate” verwenden (siehe Hilfe)