lektion5a

November 6, 2024

1 Einführung in NumPy

NumPy ist ein Paket das mathematische Funktionen, einen Zufallszahlengenerator, grundlegende Verfahren der linearen Algebra und vieles mehr zur Verfügung stellt.

[1]:
```
import numpy as np
```

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

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

[3]:
```
np.pi
```

[3]:
```
3.141592653589793
```

[ ]:
```
np.sin(np.pi)
```

Wir werden uns anschauen, wie man unter Nutzung von NumPy Vektoren und Matrizen definiert und einfache Operationen durchführt.

1.1 Vektoren und Matrizen in NumPy erzeugen (Teil 1)

[7]:
```
u = np.array([1, 2, 3])      # 1D Array (Vektor)
```

[8]:
```
v = np.array([4, 5])
```

[9]:
```
A = np.array([[3, 2, 1], [3, 5, 4], [8, 6, 7]])  # 2D Array (Matrix)
```

[10]:
```
type(u), type(v), type(A)
```

[10]:
```
(numpy.ndarray, numpy.ndarray, numpy.ndarray)
```

[11]:
```
u, v, A
```

[11]:
```
(array([1, 2, 3]),
 array([4, 5]),
 array([[3, 2, 1],
        [3, 5, 4],
```

```
       [8, 6, 7]]))
```

[12]:
```
w = np.array(5)    # eine skalare Größe (OD array)
```

[13]:
```
w
```

[13]:
```
array(5)
```

[14]:
```
3*u  # jedes Element wird mit 3 multipliziert
```

[14]:
```
array([3, 6, 9])
```

[15]:
```
e = np.array([1, 1, 1])
e
```

[15]:
```
array([1, 1, 1])
```

[16]:
```
u - 3*e
```

[16]:
```
array([-2, -1,  0])
```

1.2 Attribute eines NumPy Arrays

ndim Anzahl der Dimensionen eines NumPy arrays anzeigen:

[17]:
```
w.ndim
```

[17]:
```
0
```

[18]:
```
A.ndim
```

[18]:
```
2
```

Es gibt **ndim** auch als Methode (Funktion):

[20]:
```
np.ndim(u)
```

[20]:
```
1
```

[21]:
```
np.ndim(A)
```

[21]:
```
2
```

Die Elemente des **shape** Tupel geben die Größe der entsprechenden Array Dimension an.

Ebenso wie ndim gibt es auch shape sowohl als Attribut als auch als Methode.

[22]:
```
A.shape
```

[22]: (3, 3)

[23]:
```
u.shape
```

[23]: (3,)

[24]:
```
w.shape
```

[24]: ()

[25]:
```
np.shape(A)
```

[25]: (3, 3)

[26]:
```
np.shape(u)
```

[26]: (3,)

size gibt die Anzahl der Elemente des Arrays an:

[27]:
```
A.size
```

[27]: 9

[28]:
```
np.size(A)
```

[28]: 9

[29]:
```
u.size
```

[29]: 3

[30]:
```
np.size(u)
```

[30]: 3

1.3 vstack und hstack

Füge eine Zeile oder Spalte zur 3×3 Matrix A hinzu

[31]:
```
# array vertikal zusammenkleben mit np.vstack
np.vstack((A, e))
```

[31]:
```
array([[3, 2, 1],
       [3, 5, 4],
       [8, 6, 7],
       [1, 1, 1]])
```

[32]:
```
np.hstack((A, e)) # array horizontal zusammenkleben mit np.vstack
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[32], line 1
----> 1 np.hstack((A, e)) # array horizontal zusammenkleben mit np.vstack

File /local/home/schaedle/miniconda3/envs/compla24/lib/python3.12/site-packages
 ↪numpy/_core/shape_base.py:364, in hstack(tup, dtype, casting)
    362     return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)
    363 else:
--> 364     return _nx.concatenate(arrs, 1, dtype=dtype, casting=casting)

ValueError: all the input arrays must have same number of dimensions, but the␣
 ↪array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)
```

[33]:
```
e.reshape(3, 1)
```

[33]:
```
array([[1],
       [1],
       [1]])
```

[40]:
```
np.hstack((A, e.reshape(3, 1))) # array horizontal zusammenkleben mit np.vstack
```

[40]:
```
array([[3, 2, 1, 1],
       [3, 5, 4, 1],
       [8, 6, 7, 1]])
```

[41]:
```
np.hstack((A, A))
```

[41]:
```
array([[3, 2, 1, 3, 2, 1],
       [3, 5, 4, 3, 5, 4],
       [8, 6, 7, 8, 6, 7]])
```

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

1.4 Zugriff auf Einträge und ändern von Einträgen

[42]:
```
C = np.vstack((A, 2*A))
C
```

[42]:
```
array([[ 3,  2,  1],
       [ 3,  5,  4],
       [ 8,  6,  7],
       [ 6,  4,  2],
       [ 6, 10,  8],
```

```
       [16, 12, 14]])
```

[43]:
```
C[0, 0] #Element links oben in C
```

[43]:
```
np.int64(3)
```

[44]:
```
C[-1, -2]
```

[44]:
```
np.int64(12)
```

[45]:
```
C[-1, -1] = 101 # Element rechts unten
C
```

[45]:
```
array([[  3,   2,   1],
       [  3,   5,   4],
       [  8,   6,   7],
       [  6,   4,   2],
       [  6,  10,   8],
       [ 16,  12, 101]])
```

[46]:
```
C[2, :]
```

[46]:
```
array([8, 6, 7])
```

[48]:
```
C[2, :] = -1
C
```

[48]:
```
array([[  3,   2,   1],
       [  3,   5,   4],
       [ -1,  -1,  -1],
       [  6,   4,   2],
       [  6,  10,   8],
       [ 16,  12, 101]])
```

[50]:
```
C[1::2, [1]]
```

[50]:
```
array([[ 5],
       [ 4],
       [12]])
```

[51]:
```
C[1::2, 1]
```

[51]:
```
array([ 5,  4, 12])
```

[52]:
```
C[2:4, 1:3]
```

[52]:
```
array([[-1, -1],
       [ 4,  2]])
```

1.5 Mathematische Operationen für NumPy Arrays

1.5.1 Addition, Subtraktion, Multiplikation, Divions, modulo, potenzieren

*, / , % und ** wirken eintragsweise, wie + und -

[53]:
```
u
```

[53]:
```
array([1, 2, 3])
```

[57]:
```
a = np.array(range(4,7))
a
```

[57]:
```
array([4, 5, 6])
```

[58]:
```
u*a
```

[58]:
```
array([ 4, 10, 18])
```

[59]:
```
u**u
```

[59]:
```
array([ 1,  4, 27])
```

[60]:
```
A*A
```

[60]:
```
array([[ 9,  4,  1],
       [ 9, 25, 16],
       [64, 36, 49]])
```

[64]:
```
A
```

[64]:
```
array([[3, 2, 1],
       [3, 5, 4],
       [8, 6, 7]])
```

[61]:
```
A/A
```

[61]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

[62]:
```
A.T  # transponierte Matrix
```

[62]:
```
array([[3, 3, 8],
       [2, 5, 6],
```

```
       [1, 4, 7]])
```

[63]:
```
A.T  # transponierte Matrix
```

[63]:
```
array([[3, 3, 8],
       [2, 5, 6],
       [1, 4, 7]])
```

1.5.2 Der @ Operator

@ ist als ___**matmul**___ implementiert und definiert auf np.arrays eine allgmeine Matrixmultplikation.

Skalarprodukt von Vektoren

[65]:
```
u@u
```

[65]:
```
np.int64(14)
```

[66]:
```
np.dot(u, u)
```

[66]:
```
np.int64(14)
```

[67]:
```
A@u
```

[67]:
```
array([10, 25, 41])
```

[68]:
```
u@A
```

[68]:
```
array([33, 30, 30])
```

[69]:
```
A@A
```

[69]:
```
array([[23, 22, 18],
       [56, 55, 51],
       [98, 88, 81]])
```

[70]:
```
np.dot(A, A)
```

[70]:
```
array([[23, 22, 18],
       [56, 55, 51],
       [98, 88, 81]])
```

1.5.3 reshape

Mit **.reshape** oder **np.newaxis** kann man aus einem *flachen* Vektor eine Zeilen- oder Spaltenvektor machen

[71]:
```
u_spalte = u.reshape(-1, 1) # Spaltenvektor
u_spalte
```

[71]:
```
array([[1],
       [2],
       [3]])
```

[72]:
```
u_spalte_ = u[:, np.newaxis]
u_spalte_
```

[72]:
```
array([[1],
       [2],
       [3]])
```

[73]:
```
u[0] = 100
u_spalte, u_spalte_  # u_spalte und u_s_ sind nur Ansichten von u und zeigen␣
 ↪auf dieselben Einträge im Speicher
```

[73]:
```
(array([[100],
        [  2],
        [  3]]),
 array([[100],
        [  2],
        [  3]]))
```

[74]:
```
a = a.reshape(1, -1)
a, a.ndim, a.shape
```

[74]:
```
(array([[4, 5, 6]]), 2, (1, 3))
```

[75]:
```
u_zeile = u[np.newaxis, :]
u_zeile
```

[75]:
```
array([[100,   2,   3]])
```

[76]:
```
u_spalte@u_zeile, u_zeile@u_spalte  # Skalarprodukt und äußeres Produkt
```

[76]:
```
(array([[10000,   200,   300],
        [  200,     4,     6],
        [  300,     6,     9]]),
 array([[10013]]))
```

[77]:
```
u_spalte@A
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[77], line 1
```

```
----> 1 u_spalte@A

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, wit
  ↪gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 1)
```

[78]:
```
u_zeile@A
```

[78]:
```
array([[330, 228, 129]])
```

1.5.4 Weitere Arrayoperationen

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

[79]:
```
np.max(A), A.max()
```

[79]:
```
(np.int64(8), np.int64(8))
```

[80]:
```
A.max(1) # Man kann auch das Maximum entlang der ersten Achse
```

[80]:
```
array([3, 5, 8])
```

[81]:
```
A.max(0) # Man kann auch das Maximum entlang der nullten Achse
```

[81]:
```
array([8, 6, 7])
```

[82]:
```
A
```

[82]:
```
array([[3, 2, 1],
       [3, 5, 4],
       [8, 6, 7]])
```

[83]:
```
np.max(A[0, :])  # oder das Maximum der 1. Zeile
```

[83]:
```
np.int64(3)
```

"flatten" A in ein 1-dim. Array umwandeln

[84]:
```
A.flatten()
```

[84]:
```
array([3, 2, 1, 3, 5, 4, 8, 6, 7])
```

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

[86]:
```
A.argmax(), A.flatten()[np.argmax(A)]
```

[86]:
```
(np.int64(6), np.int64(8))
```

[87]:
```
B = np.array([3*i-4 for i in range(9)]).reshape(3, 3)
B
```

[87]:
```
array([[-4, -1,  2],
       [ 5,  8, 11],
       [14, 17, 20]])
```

[88]:
```
np.maximum(A, B) # elementweises maximum (analog np.minimum)
```

[88]:
```
array([[ 3,  2,  2],
       [ 5,  8, 11],
       [14, 17, 20]])
```

1.5.5 Kronecker Produkt

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

[89]:
```
np.kron([1, 10, 100], [5, 6, 7])
```

[89]:
```
array([  5,   6,   7,  50,  60,  70, 500, 600, 700])
```

[91]:
```
I2 = np.array([[1, 0], [0, 2]])
E2 = np.array([[1, 1], [2, 2]])
np.kron(I2, E2), np.kron(E2, I2)
```

[91]:
```
(array([[1, 1, 0, 0],
        [2, 2, 0, 0],
        [0, 0, 2, 2],
        [0, 0, 4, 4]]),
 array([[1, 0, 1, 0],
        [0, 2, 0, 2],
        [2, 0, 2, 0],
        [0, 4, 0, 4]]))
```

1.6 Definition spezieller Matrizen

[92]:
```
E = np.ones((3, 3))   # Matrix aus Einsen
Z = np.zeros((3, 3))  # Nullmatrix
I = np.eye(3)   # Identität
A = np.arange(16)
```

[93]:
```
E, Z, I, A
```

[93]:
```
(array([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]),
```

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

[94]:
```
np.eye(4, 6, k=1)  # erste obere Nebendiagonale
```

[94]:
```
array([[0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 1., 0.]])
```

[95]:
```
np.eye(4, 6, k=-1)  # erste untere Nebendiagonale
```

[95]:
```
array([[0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.]])
```

[96]:
```
np.eye(4, 6)   # Default Wert ist k=0, Hauptdiagonale
```

[96]:
```
array([[1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0.]])
```

[97]:
```
A = A.reshape(4, 4)
A
```

[97]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[98]:
```
np.diag(A)
```

[98]:
```
array([ 0,  5, 10, 15])
```

[99]:
```
# Untere Dreiecksmatrix (lower triangular)
np.tril(A)
```

[99]:
```
array([[ 0,  0,  0,  0],
       [ 4,  5,  0,  0],
       [ 8,  9, 10,  0],
```

```
       [12, 13, 14, 15]])
```

[100]:
```
# Obere Dreiecksmatrix (upper triangular)
np.triu(A)
```

[100]:
```
array([[ 0,  1,  2,  3],
       [ 0,  5,  6,  7],
       [ 0,  0, 10, 11],
       [ 0,  0,  0, 15]])
```

[101]:
```
# np.diag kann auch Diagonalmatrizen erzeugen
B = np.diag(v.flatten())
B
```

[101]:
```
array([[4, 0],
       [0, 5]])
```

[102]:
```
# np.diag für Nebendiagonalen
NU = np.diag(A, k=1)  # obere Nebendiagonale
NU
```

[102]:
```
array([ 1,  6, 11])
```

[103]:
```
NL = np.diag(A, k=-1) # untere Nebendiagonale
NL
```

[103]:
```
array([ 4,  9, 14])
```

[104]:
```
# Benutze np.diag zur Definition von Matrizen
x = [1, 2, 3]
A = np.diag(x, k=-1) + np.diag(x, k=1)
A
```

[104]:
```
array([[0, 1, 0, 0],
       [1, 0, 2, 0],
       [0, 2, 0, 3],
       [0, 0, 3, 0]])
```

[ ]: