Numpy_CrashKurs

August 3, 2021

1 Numpy

1.1 Crash-Kurs

Wir wollen die wichtigsten Möglichkeiten, die Numpy bietet, kurz darstellen. Numpy ist enorm mächtig, daher können wir unmöglich alle Funktionalitäten ansprechen. Wir konzentrieren uns vielmehr auf die Eigenschaften, die wohl am häufigsten verwendet werden.

Kurz zusammengefasst die wichtigsten Vorteile von Numpy-Arrays, wenn man diese mit Python-Listen vergleicht:

- Weniger Speicherbedarf
- Wesentlich schneller
- Mathematische Operationen auf Arrays / Matirzen möglich (z.B. Multiplikation eines Skalars mit einer Matrix)
- Sehr viele eingebaute mathematische Funktionen (Numpy.linalg.*)

1.1.1 Erstellen eines Numpy-Arrays und Eigenschaften ermitteln

Aber der Reihe nach: Erstellen wir ein Numpy-Array und sehen wir uns an, wie wir dieses verwenden können. Alles beginnt mit dem Import von *Numpy*. Als Alias sollte immer *np* verwendet werden. Ein Numpy-Array (weiter nur als "Array" bezeichnet) wird mit der Funktion *array* erstellt, der man eine Liste übergibt. Wir erstellen ein 1-dimensionales Array:

[1]:
```
import numpy as np

a = np.array([4,6,1,23,42,9])
print(a)
```

```
[ 4  6  1 23 42  9]
```

Mit Hilfe von *ndim* kann die Dimension eines Arrays ermittelt werden, mit *shape* die Anzahl der Zeilen und Spalten:

[2]:
```
print(a.ndim)
print(a.shape)
```

```
1
(6,)
```

Erstellen wir ein 2-dimensionales Array, also eine Matrix, mit 3 Zeilen und 3 Spalten (quadratische Matrix):

[3]:
```
m = np.array([[1,2,3], [4,5,6], [7,8,9]])
print(m)
print(m.ndim)
print(m.shape)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
2
(3, 3)
```

Wir können auch den Datentyp der Elemente im Array ermitteln. Im Gegensatz zu Python-Listen sind bei Numpy-Arrays übrigens nur ein Datentyp für alle Elemente erlaubt!

[4]:
```
print(m.dtype)
```

```
int32
```

Es wurde als Datentyp also standardmäßig 4-Byte-Integer verwendet. Wenn man sich sicher ist, dass die Zahlen in einem Array nicht zu groß werden, kann man den Datentyp beim Erstellen des Arrays auch vorab definieren und somit weiter Speicherplatz sparen:

[5]:
```
m = np.array([[1,2,3], [4,5,6], [7,8,9]], dtype="int8")
print(m.dtype)

m_float = np.array([[1,2,3], [4,5,6], [7,8,9]], dtype="float")
print(m_float.dtype)

print(m_float)
```

```
int8
float64
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

Mit *itemsize* kann man den Speicherbedarf in Byte für ein Element ermitteln, mit *nbytes* den Speicherbedarf für das gesamte Array.

[6]:
```
print(m.itemsize)
print(m_float.itemsize)

print(m.nbytes)
print(m_float.nbytes)

# nbytes ist analog  zu
print(m_float.size * m_float.itemsize)
```

```
1
8
9
```

```
72
72
```

Wollen wir ein Array erstellen mit Werten \[von .. bis), so verwenden wir *arange*. Als 3. Argument kann man eine Schrittweite angeben.

[7]:
```
a = np.arange(10)
print(a)

# Mit Schrittweite
a = np.arange(10, 20, 3)
print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
[10 13 16 19]
```

Insbesondere für das Erstellen von Diagramm benötigt man häufig Werte in bestimmten Grenzen, die mit Zwischenwerten mit gleichem Abstand aufgefüllt werden. Dies bewerkstelligt man mit *linspance*:

[8]:
```
x = np.linspace(10, 20, 15)
print(x)
```

```
[10.         10.71428571 11.42857143 12.14285714 12.85714286 13.57142857
 14.28571429 15.         15.71428571 16.42857143 17.14285714 17.85714286
 18.57142857 19.28571429 20.        ]
```

1.1.2 Zugriff auf Elemente eines Arrays

Der Zugriff entspricht weitgehend dem auf Python-Listen. Auch der *Slicing-Operator :* kann verwendet werden. Wie für Programmierer gewohnt beginnt der Index bei 0 zu zählen. Wir erstellen ein 1-Dimensionales Array und greifen auf verschiedene Elemente zu.

[9]:
```
a = np.array([4,6,1,23,42,9])

# Erstes Element ausgeben
print(a[0])

# Letztes Element ausgeben
print(a[5])
```

```
4
9
```

Mit dem negativen Indizes wird auf die letzten Elemente zugegriffen. So können wir das letzte Element eines Arrays auch ausgeben, wenn wir die Anzahl der Elemente nicht kennen.

[10]:
```
# Letztes Element ausgeben
print(a[-1])

# Vorletztes Element ausgeben
```

```
print(a[-2])
```

```
9
42
```

Mit dem Slicing-Operator kann man einen Bereich definieren.

[11]:
```
# Gib die ersten 3 Zahlen aus (Index 0 bis 2)
print(a[:3])

# Gib die letzten 3 Zahlen aus
print(a[-3:])

# Gib 2. bis 5. Element aus (Index 1 bis 4)
print(a[1:5])
```

```
[4 6 1]
[23 42  9]
[ 6  1 23 42]
```

Analoges gilt für 2- oder auch mehrdimensionale Arrays. Wie in der Mathematik üblich entspricht der erste Index der Zeile, der 2. Indexwert der Spalte einer Matrix.

[12]:
```
m = np.array([[1,2,3], [4,5,6], [7,8,9]])
print(m)

print("1. Zeile:")
print(m[0,:])

print("1. Spalte:")
print(m[:,0])

print("Element in der 2. Zeile und 3. Spalte")
print(m[1,2])
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
1. Zeile:
[1 2 3]
1. Spalte:
[1 4 7]
Element in der 2. Zeile und 3. Spalte
6
```

Mit einem weiteren Wert kann auch die Schrittweite definiert werden, also wenn wir aus einem Array zum Beispiel nur auf jedes 3. Element zugreifen wollen:

[13]:
```
v = np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
print(v[2 : -2 : 3])
```

```
[ 3  6  9 12]
```

1.1.3 Ändern eines Arrays

Numpy-Arrays sind "mutable". Man kann also ein Element ändern:

[14]:
```
m = np.array([[1,2,3], [4,5,6], [7,8,9]])
print(m)

print("Überschreibe das Element in der 2. Zeile und 2. Spalte (5) mit 100:")

m[1,1] = 100
print(m)

print("Ersetze die 1. Zeile durch die Werte 1000, 2000 und 3000")
x = np.array([1000,2000,3000])
m[0,:] = x
print(m)


print("Ersetze 2. Spalte durch die Werte -1, -2, -3")
m[:,1] = [-1, -2, -3]
print(m)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Überschreibe das Element in der 2. Zeile und 2. Spalte (5) mit 100:
[[  1   2   3]
 [  4 100   6]
 [  7   8   9]]
Ersetze die 1. Zeile durch die Werte 1000, 2000 und 3000
[[1000 2000 3000]
 [   4  100    6]
 [   7    8    9]]
Ersetze 2. Spalte durch die Werte -1, -2, -3
[[1000   -1 3000]
 [   4   -2    6]
 [   7   -3    9]]
```

1.1.4 Kopieren eines Arrays

Achtung beim Kopieren! Weist man ein Array einfach einer anderen Variablen zu, so wird nur die Referenz kopiert, nicht aber das Array! Beide Variablen referenzieren also dasselbe Array:

[15]:
```
a = np.array([4,6,1,23,42,9])
b = a

a[0] = 1000
```

```
print(b)
```

```
[1000    6    1   23   42    9]
```

Um ein Array zu kopieren verwendet man stattdessen die Funktion *copy*:

[16]:
```
a = np.array([4,6,1,23,42,9])
b = a.copy()

a[0] = 1000
print(b)
```

```
[ 4  6  1 23 42  9]
```

1.1.5 Erstellen von Arrays / Matrizen mit vorab definierten Werten

Möchte man Matrizen erstellen, die definierte Werte enthält, so helfen folgende Funktionen:

[17]:
```
print("Erstelle eine Matrix mit 0-en")
m1 = np.zeros([3,3])
print(m1)

print("Erstelle eine Matrix mit 1-en")
m2 = np.ones([3,3])
print(m2)

print("Erstelle eine Matrix mit einer definierten Zahl")
m3 = np.full([3,3], 42)
print(m3)

print("Erstelle eine quadr. Matrix mit bestimmten Elementen in der␣
 ↪Haupt-Diagonalen")
m4 = np.diag([11,12,13])
print(m4)

print("Erstelle eine Einheitsmatrix E")
m5 = np.identity(3)
print(m5)

print("Erstelle eine Matrix in der Größe einer anderen Matrix und fülle diese␣
 ↪Matrix mit einem bestimmten Wert")
m6 = np.full_like(m5, -1)
print(m6)
```

```
Erstelle eine Matrix mit 0-en
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
Erstelle eine Matrix mit 1-en
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
Erstelle eine Matrix mit einer definierten Zahl
[[42 42 42]
 [42 42 42]
 [42 42 42]]
Erstelle eine quadr. Matrix mit bestimmten Elementen in der Haupt-Diagonalen
[[11  0  0]
 [ 0 12  0]
 [ 0  0 13]]
Erstelle eine Einheitsmatrix E
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Erstelle eine Matrix in der Größe einer anderen Matrix und fülle diese Matrix
mit einem bestimmten Wert
[[-1. -1. -1.]
 [-1. -1. -1.]
 [-1. -1. -1.]]
```

Es ist auch sehr einfach möglich, eine Matrix mit Zufallszahlen zu erstellen. Im Beispiel erstellen wir eine 3x3-Matrix mit ganzzahligen Zufallszahlen zwischen -5 und 5.

[18]:
```
zuf_int = np.random.randint(-5,5, (3,3))
print(zuf_int)
```

```
[[ 4  0 -5]
 [ 2 -3 -5]
 [ 1 -5  4]]
```

Mit *uniform* können auch pseudozufällige gleichverteilte Zufallszahlen vom Typ float erstellt werden. Man übergibt den Min- und Maxwert sowie die Anzahl Zeilen und Spalten als Tupel.

[19]:
```
zuf_gleichverteilt = np.random.uniform(-5., 5., (3,3))
print(zuf_gleichverteilt)
```

```
[[ 0.55838715  0.19610932 -3.01283559]
 [-4.96171547 -4.78197947  0.45455165]
 [ 0.68906862 -1.86651288 -3.12886648]]
```

Analog können auch normalverteilte Werte erstellt werden. Dazu übergibt man den Erwartungswert μ und die Standardabweichung σ.

[20]:
```
zuf_normalverteilt = np.random.uniform(0, 1, (5,5))
print(zuf_normalverteilt)
```

```
[[0.49278288 0.52780241 0.62782097 0.77250136 0.77211805]
 [0.49379849 0.94629722 0.47589337 0.19637635 0.0599124 ]
 [0.3526466  0.75403966 0.46411332 0.11238336 0.65776305]
```

```
 [0.94145096 0.93900038 0.82300052 0.31818087 0.41915935]
 [0.45581135 0.46379684 0.34211512 0.84100075 0.04334392]]
```

1.1.6 Rechnen mit Matrizen: Lineare Algebra

Numpy Arrays entfalten ihre Mächtigkeit erst so richtig, wenn man mathematische Operationen auf ihnen ausführt. Die Funktionen der Linearen Algebra, zum Beispiel um eine Inverse oder eine Determinante einer Matrix zu berechnen, finden wir im Paket *np.linalg*). Einfache Operationen auf Array, also zum Beispiel das Multiplizieren oder Addieren mit einem Skalr, können direkt auf ein Array ausgeführt werden.

[21]:
```
v = np.array([-8, 2, 4])
print(v)
print(3 * v)
print(v + 5)
print(v / 2)
```

```
[-8  2  4]
[-24   6  12]
[-3  7  9]
[-4.  1.  2.]
```

Arrays können auch sehr einfach addiert oder multipliziert werden (solange die Anforderungen für diese Rechenoperationen erfüllt sind!).

[22]:
```
v1 = np.array([-3, 2, 4])
v2 = np.array([1, 0, -2])

print(v1 + v2)
print(v1 * v2)

m1 = np.array([[-3,-2,6], [4,-1,3], [-2,0,1]])
m2 = np.array([[0,1,2], [1,2,-3], [1,2,-5]])
print(m1)

print("Matrix m1 multipliziert mit Vektor v1")
print(m1 * v1)

print("Matrix m1 multipliziert mit Matrix m2")
print(m2)
print(m1 * m2)
```

```
[-2  2  2]
[-3  0 -8]
[[-3 -2  6]
 [ 4 -1  3]
 [-2  0  1]]
Matrix m1 multipliziert mit Vektor v1
[[  9  -4  24]
 [-12  -2  12]
```

```
 [  6   0   4]]
Matrix m1 multipliziert mit Matrix m2
[[ 0  1  2]
 [ 1  2 -3]
 [ 1  2 -5]]
[[ 0 -2 12]
 [ 4 -2 -9]
 [-2  0 -5]]
```

Im obigen Beispiel wurden die Matrizen m1 und m2 multipliziert. Hier wird allerdings jedes Element mit jedem analogen Element der anderen Matrix multipliziert. Möchte man Matrizen im Sinne der Linearen Algebra multiplizieren, so verwendt man *matmul* oder kürzer den @-Operator.

[23]:
```
print(np.matmul(m1,m2))
print(m1 @ m2)
print(m1 @ v1)
```

```
[[  4   5 -30]
 [  2   8  -4]
 [  1   0  -9]]
[[  4   5 -30]
 [  2   8  -4]
 [  1   0  -9]]
[29 -2 10]
```

Die Determinante berechnet man mit *det*.

[24]:
```
print(np.linalg.det(m1))
print(np.linalg.det(m2))
```

```
11.000000000000002
2.0
```

Berechnen der Inverse einer Matrix mit *inv*

[25]:
```
m1_inv = np.linalg.inv(m1)
print(m1_inv)
print(np.round(m1_inv @ m1))
```

```
[[-0.09090909  0.18181818  0.        ]
 [-0.90909091  0.81818182  3.        ]
 [-0.18181818  0.36363636  1.        ]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Eigenwert und Eigenvektor einer Matrix liefert die Funktion *eig*, die ein Tupel zurückgibt. Das erste Element des Tupels entspricht den Eigenwerten, das zweite Element die dazugehörigen Eigenvektoren spaltenweise in einer Matrix.

Es muss gelten:

$A \cdot v = \lambda \cdot v$ wobei v ein Eigenvektor und λ der zugehörige Eigenwert ist.

[26]:
```
eigw, eigv = np.linalg.eig(m2)
print(eigw)
print(eigv)

print("Beispiel: Matrix * Eigenvektor = Eigenwert * Eigenvektor")
print(m2 @ eigv[:,0])

# Liefert dasselbe Ergebnis wie
print(eigw[0] * eigv[:,0])
```

```
[-4.46685999 -0.25937481  1.72623479]
[[ 0.4473362  -0.93676805 -0.71153494]
 [-0.43236839  0.34612912 -0.6375588 ]
 [-0.78290989 -0.05157754 -0.29535878]]
Beispiel: Matrix * Eigenvektor = Eigenwert * Eigenvektor
[-1.99818817  1.93132908  3.49714886]
[-1.99818817  1.93132908  3.49714886]
```

Um eine Matrix zu transponieren gibt es zwei Möglichkeiten: Funktion *transpose* oder mit Hilfe von $m.T$:

[27]:
```
m = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(m)
print(np.transpose(m))
print(m.T)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 4 7]
 [2 5 8]
 [3 6 9]]
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

1.1.7 Rechnen mit Matrizen: Statistik

Numpy liefert auch jede Menge statistischer Funktionen. Hier eine Auswahl.

[28]:
```
a = np.array([3,5,9,11,17,4,2,5,78])

print("Kleinster und größter Wert")
print(np.min(a), np.max(a))

print("Mittelwert")
print(np.mean(a))
```

```
print("Median")
print(np.median(a))

print("Varianz")
print(np.var(a))

print("Standardabweichung")
print(np.std(a))
```

```
Kleinster und größter Wert
2 78
Mittelwert
14.88888888888889
Median
5.0
Varianz
517.6543209876544
Standardabweichung
22.75201795418715
```

Verwendet man 2- oder mehrdimensionale Arrays, so können Statistiken zeilen- oder spaltenweise definiert werden. Dazu verwendet man das Attribut *axis*. Die Verwendung von *axis* ist etwas gewöhnungsbedürftig und nicht gerade intuitiv!

Man stelle sich ein Koordinatensystem vor, im 2-dimensionalen also eine X- und eine Y-Achse. Die *axis* werden einfach durchnummeriert: X entspricht dem Wert 0, Y dem Wert 1. Im 3-dimensionalen könnte man also entsprechend *axis* den Wert 2 zuweisen. Dieses Koordinatensystem "legt" man auf ein Array. Wenn wir also zum Beispiel die Standardabweichung über *axis=0* berechnen wollen, so gehen wir zeilenweise vor und berechnen die Summe der Werte in den Spalten(!). Das ist zugegebenermaßen etwas verwirrend ;-)

Beispiel: Wir definieren eine 2x2-Matrix und berechnen statistische Werte einmal bezüglich der Zeilen (axis=0), einmal bezüglich der Spalten (axis=1):

[29]:
```
m = np.array([[1,2,3],[4,5,42],[7,8,9]])
print(m)

print("Zeile mit den größtem Wert in jeder Spalte")
print(np.max(m, axis=0))

print("Spalte mit dem größtem Wert in jeder Zeile")
print(np.max(m, axis=1))

print("Standardabweichungen der über die  Zeilen")
print(np.std(m, axis=0))

print("Standardabweichungen der über die  Spalten")
print(np.std(m, axis=1))
```

```
print("axis=0: Wir aggregieren über die Zeilen und berechnen die Summen der␣
 ↪Werte (Spaltensummen)")
print(np.sum(m, axis=0))

print("axis=1: Wir aggregieren über die Spalten und berechnen die Summen der␣
 ↪Werte (Zeilensummen)")
print(np.sum(m, axis=1))
```

```
[[ 1  2  3]
 [ 4  5 42]
 [ 7  8  9]]
Zeile mit den größtem Wert in jeder Spalte
[ 7  8 42]
Spalte mit dem größtem Wert in jeder Zeile
[ 3 42  9]
Standardabweichungen der über die  Zeilen
[ 2.44948974  2.44948974 17.1464282 ]
Standardabweichungen der über die  Spalten
[ 0.81649658 17.68238295  0.81649658]
axis=0: Wir aggregieren über die Zeilen und berechnen die Summen der Werte
(Spaltensummen)
[12 15 54]
axis=1: Wir aggregieren über die Spalten und berechnen die Summen der Werte
(Zeilensummen)
[ 6 51 24]
```

1.1.8 Ändern von Arrays

Numpy Arrays können nicht nur bezüglich ihrer Inhalte, sondern auch bezüglich ihrer Struktur geändert werden. Allerdings wird dabei immer ein neues Array erstellt, was natürlich Rechenzeit und zusätzlichen Speicherbedarf erfordert.

Beginnen wir mit der Methode *reshape*, die die Zeilen/Spalten neu anordnet. Häufig verwendet man diese Methode bereits beim Erstellen eines Arrays, da man sich die umständliche Tipperei der eckigen Klammern spart.

[30]:
```
m = np.array([1,2,3,4,5,6,7,8,9]).reshape(3,3)
print(m)

m = np.arange(0,15).reshape(3,5)
print(m)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

Mit *vstack* können Arrays vertikal zu einem neuen Array verbunden werden. Analoges erledigt *hstack* für horizontales Verbinden.

[31]:
```
v1 = np.array([1,2,3,4])
v2 = np.array([5,6,7,8])

print(np.vstack([v1, v2]))

print(np.hstack([v1,v2]))
```

```
[[1 2 3 4]
 [5 6 7 8]]
[1 2 3 4 5 6 7 8]
```

Mit *concatenate* können ebenso Arrays verbunden werden.

[32]:
```
m1 = np.array([[1,2],[3,4]])
m2 = np.array([[11,12],[13,14]])
print(m1)
print(m2)

# Zeilenweise anfügen
print(np.concatenate((m1,m2), axis=0))

# Spaltenweise anfügen
print(np.concatenate((m1,m2), axis=1))
```

```
[[1 2]
 [3 4]]
[[11 12]
 [13 14]]
[[ 1  2]
 [ 3  4]
 [11 12]
 [13 14]]
[[ 1  2 11 12]
 [ 3  4 13 14]]
```

Mit *delete* kann man element, spalten- oder zeilenweise löschen. Löscht man ein Element aus einer 3x3-Matrix, so kann diese natürlich nicht mehr als 3x3-Matrix dargestellt werden, da es nach dem Löschen nur noch 8 Elemente gibt. Daher wird die Matrix durch das Löschen "flachgeklopft".

[33]:
```
m = np.arange(9).reshape(3,3)
print(m)

print("Löschen des 1. Elements")
print(np.delete(m, 1))

print("Man beachte: m gibt es immer noch! Es wurde ein neues Array erstellt!")
```

```
print(m)

print("Lösche die Elemente 1,3,5,7 und 8 und erstelle aus den restlichen Daten␣
 ↪eine 2x2-Matrix")
print(np.delete(m,[1,3,5,7,8]).reshape(2,2))

print("Lösche die zweite Zeile (index=1)")
print(np.delete(m,1, axis=0))

print("Lösche die zweite Spalte")
print(np.delete(m,1, axis=1))
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Löschen des 1. Elements
[0 2 3 4 5 6 7 8]
Man beachte: m gibt es immer noch! Es wurde ein neues Array erstellt!
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Lösche die Elemente 1,3,5,7 und 8 und erstelle aus den restlichen Daten eine
2x2-Matrix
[[0 2]
 [4 6]]
Lösche die zweite Zeile (index=1)
[[0 1 2]
 [6 7 8]]
Lösche die zweite Spalte
[[0 2]
 [3 5]
 [6 8]]
```