03h_NumPy_Advanced

0.1 Fortgeschrittenes Thema: Weitere Funktionalitäten von NumPy

In diesem "Anhang" stellen wir im Schnelldurchlauf noch "einige" weitere Möglichkeiten von NumPy vor, die immer wieder hilfreich sein können. Außerdem diskutieren wir Fragen der Immutabilität von ndarrays, und fortgeschrittene Möglichkeiten zur Indizierung von (Teil-) Matrizen.

0.1.1 Zusammensetzen von Matrizen

Wir beginnen mit den Methoden hstack() und vstack(), mit denen Matrizen aus Teilmatrizen zusammengesetzt werden können:

[ ]:
```
import numpy as np

# Wir legen zunächst drei Matrizen bzw. Vektoren an:
A = np.array([[1, 2, 3], [3, 2, 1]])
B = np.array([[4, 5, 6]])
C = np.array([[6, 5]]).T
# B ist ein Zeilenvektor, C ist ein Spaltenvektor!
print(A, B, C, sep="\n---\n")
print()

# Matrizen vertikal aneinander hängen:
# np.vstack( ) bekommt ein Tupel übergeben!
print(np.vstack((A,B)))
print()

# Matrizen horizontal aneinander hängen:
# np.hstack( ) bekommt ein Tupel übergeben!
print(np.hstack((A, C)))
```

In manchen Anwendungsfällen setzen sich bestimmte Matrizen $A \in \mathbb{R}^{n \times n}$ aus kleineren Matrizen zusammen, man spricht von **Blockmatrizen**:

$$A = \left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right).$$

Wir betrachten

$$A = \begin{pmatrix} 0_{n \times n} & I_{n \times n} \\ B & 0_{n \times n} \end{pmatrix},$$

wobei $0_{n\times n}$ die n-dimensionale Matrix, $I_{n\times n}$ die n-dimensionale Einheitsmatrix und $B \in \mathbb{R}^{n\times n}$ eine spezielle Systemmatrix ist.

[1]:
```
import numpy as np

n = 3
# Anlegen einer n-dimensionalen Matrix (in Anwendungen hat
# diese Matrix natürlich eine andere Struktur und ist nicht
# zufällig!)
B = np.random.random((n,n))
# Abspeichern der Null- und Einheitsmatrix
Z = np.zeros((n,n))
I = np.eye(n)

# Zusammenbau der Matrix:
A = np.vstack((np.hstack((Z,I)), np.hstack((B,Z))))

print(A)
```

```
[[0.         0.         0.         1.         0.         0.        ]
 [0.         0.         0.         0.         1.         0.        ]
 [0.         0.         0.         0.         0.         1.        ]
 [0.96743086 0.83556261 0.92969968 0.         0.         0.        ]
 [0.4894793  0.6176453  0.36771992 0.         0.         0.        ]
 [0.81024688 0.16851292 0.00818795 0.         0.         0.        ]]
```

Um Matrizen gleicher Form anzulegen, können wir natürlich zuerst die Dimensionen auslesen. Die folgenden Befehle zeigen, dass dies auch abgekürzt werden kann:

[2]:
```
import numpy as np

A = np.random.random((3,6))
print(A)
print()

# Erstellen einer "ähnlichen" Matrix, d.h. einer Matrix
# mit gleicher Form (shape ist gleich).
print(np.zeros_like(A))
print()
print(np.ones_like(A))
print()
print(np.full_like(A, 2))
```

```
[[0.56589671 0.19747265 0.53113003 0.23109075 0.76859567 0.2701753 ]
 [0.83954862 0.64065998 0.65650808 0.92595212 0.85992112 0.67231083]
 [0.33002765 0.48300671 0.97294634 0.72909353 0.74501355 0.34027234]]

[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
```

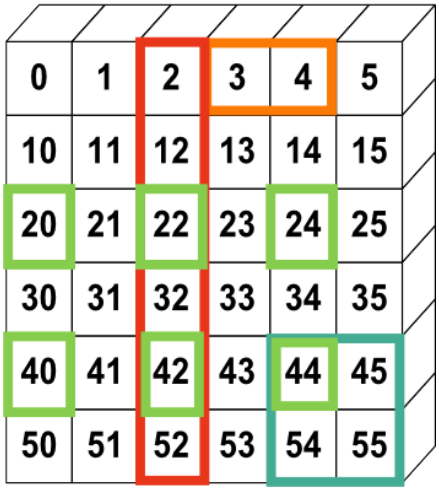
```
 [0. 0. 0. 0. 0. 0.]]

[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]

[[2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2.]]
```

0.1.2 Quiz

Wie können die folgenden farbig markierten Blöcke in numpy indiziert werden?



Quelle: http://www.scipy-lectures.org und https://github.com/scipy-lectures/scipy-lecture-notes, Gaël Varoquaux et al., CC-BY

[3]:
```
import numpy as np

# Wir sehen in Kürze warum dieser Befehl funktioniert:
A = np.array([[0,1,2,3,4,5]]) + np.array([[0,10,20,30,40,50]]).T
print(A)

# Orange:
print(A[0,3:5])

# Rot:
print(A[:, 2])

# Grün
print(A[2::2, 0::2])
```

```
# Türkis:
print(A[-2:,-2:])
```

```
[[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]
[3 4]
[ 2 12 22 32 42 52]
[[20 22 24]
 [40 42 44]]
[[44 45]
 [54 55]]
```

0.1.3 Matrix-Ausschnitte

Wir betrachten einige Beispiele zur Modifikation eines ndarray:

[4]:
```
import numpy as np

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

# Überschreiben einer einzelnen Zahl
# in der Matrix:
M[1,2] = 2
print(M)

# Austauschen einer Zeile:
M[2,:] = np.array([1, 2, 3])
#print(M)
M[2,:] = [1, 2, 3]
print(M)

# Matrix in Matrix kopieren:
M[1:3,:] = np.array([[1.,2.,3.],[-1.,-2.,-3.]])
print(M)

# Zeilenvektor in Matrix kopieren:
# Hierbei ist es wichtig ein zweidimensionales
# Objekt zu erzeugen!
#M[1:4,2:3] = np.array([1.,0.,-1.0]).T #schlägt fehl!
M[1:4,2:3] = np.array([[1.],[0.],[-1.0]])
print(M)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]]
[[ 1  2  3]
 [ 4  5  2]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]]
[[ 1  2  3]
 [ 4  5  2]
 [ 1  2  3]
 [10 11 12]
 [13 14 15]]
[[ 1  2  3]
 [ 1  2  3]
 [-1 -2 -3]
 [10 11 12]
 [13 14 15]]
[[ 1  2  3]
 [ 1  2  1]
 [-1 -2  0]
 [10 11 -1]
 [13 14 15]]
```

Wir betrachten erneut das Beispiel

$$A = \begin{pmatrix} 0_{n\times n} & I_{n\times n} \\ B & 0_{n\times n} \end{pmatrix}.$$

Diesmal schreiben wir allerdings die Blöcke direkt in die Matrix $A \in \mathbb{R}^{2n\times 2n}$:

[5]:
```
import numpy as np

n = 3
A = np.zeros((2*n,2*n))
B = np.random.random((n,n))

# Einheitsmatrix oben rechts:
A[0:n,n:2*n] = np.eye(n)
# Blockmatrix unten links:
A[n:2*n,0:n] = B

print(A)
```

```
[[0.         0.         0.         1.         0.         0.        ]
 [0.         0.         0.         0.         1.         0.        ]
 [0.         0.         0.         0.         0.         1.        ]
```

```
 [0.53439192 0.57737822 0.14339324 0.         0.         0.        ]
 [0.34641329 0.39757884 0.66338936 0.         0.         0.        ]
 [0.92406293 0.74042529 0.21097672 0.         0.         0.        ]]
```

0.1.4 Inplace Operationen, Copies, Referenzen und Views

Zur Motivation dieses Abschnitts betrachten wir den folgenden Codeschnipsel:

[6]:
```
import numpy as np

ones = np.ones((2,2))

A = ones
A += 5
A = A + ones

# Welches Ergebnis erwarten wir hier?
print(A)
print(ones)
```

```
[[12. 12.]
 [12. 12.]]
[[6. 6.]
 [6. 6.]]
```

Das Verhalten ist ähnlich wie bei Listen und anderen Datenstrukturen: Mit der Zeile A = ones zeigt die Matrix A auf das gleiche Objekt im Speicher, also auf die Matrix ones.

Durch die Zeile A += 5 wird damit nicht nur A, sondern auch ones geändert zur Matrix [[6,6],[6,6]], denn beides sind ja nur Verweise auf dieselbe Stelle im Speicher.

In der nachfolgenden Zeile wird eine **neue** Matrix im Speicher abgelegt und auf A + ones gesetzt. Damit hat A nichts mehr mit ones zu tun.

Operationen wie A += 5 sind sogenannte **inplace-Operationen**. Sie verändern die Matrix A direkt, ohne ein neues Objekt im Speicher anzulegen. Wir kennen dieses Prinzip bereits von Listen.

Das folgende Beispiel zeigt, dass ein Ausschnitt einer Matrix in numpy eine echte "Referenz" (View) und keine Kopie ist:

[7]:
```
import numpy as np

A = np.ones((5,6))

B = A[2:4,2:5]
B *= 2

print(A)
```

```
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
```

```
 [1. 1. 2. 2. 2. 1.]
 [1. 1. 2. 2. 2. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

Der in B gespeicherte Ausschnitt zeigt auf die gleichen Daten in der Matrix A. Dies funktioniert allerdings nur bei "zusammenhängenden" Ausschnitten.

Um den Ausschnitt als unabhängige Kopie zu erhalten, kann die Methode .copy() verwendet werden:

[8]:
```
import numpy as np

A = np.ones((5,6))
B = A[2:4,2:5].copy()
B *= 2
print(A)
print(B)
```

```
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
[[2. 2. 2.]
 [2. 2. 2.]]
```

0.1.5 Indizierung mittels Integer-Matrizen:

Wir können aus einer Matrix durch Angabe der entsprechenden Zeilen und Spalten gezielt Bereiche auswählen. Hierfür definieren wir eine ganzzahlige (d.h. aus int bestehende) Matrix beziehungsweise einen Vektor:

[10]:
```
import numpy as np

# Anlegen einer Testmatrix:
A = np.arange(50).reshape(5,10)
print(A)

# Auswahl der zweiten und vierten Zeile:
rows = [1, 3]
print(A[rows, :])

# Alternativ kann die Auswahlmatrix auch als
# numpy Array angelegt werden. Hierbei muss
# allerdings der Datentyp auf integer geändert
# werden:
columns = np.array([2, 5, 6], dtype=int)
# Auswahl der 3., 6. und 7. Spalte:
B = A[:, columns]
```

```
print(B)
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]]
[[10 11 12 13 14 15 16 17 18 19]
 [30 31 32 33 34 35 36 37 38 39]]
[[ 2  5  6]
 [12 15 16]
 [22 25 26]
 [32 35 36]
 [42 45 46]]
```

[11]:
```
import numpy as np

x = np.array([[0,  1,  2], [3,  4,  5], [6,  7,  8], [9, 10, 11]])
print(x)

# auch die gleichzeitige Indizierung von Zeilen und Spalten ist möglich:
rows = np.array([[0,0],[3,3]])
cols = np.array([[0,2],[0,2]])
y = x[rows,cols]

print("Die Eckelemente der Matrix x sind:\n", y)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
Die Eckelemente der Matrix x sind:
 [[ 0  2]
 [ 9 11]]
```

0.1.6 Beispiel: Berechnung von Vektornormen und Ermittlung der größten Elemente

Gegeben seien Vektoren $x^1, x^2, \ldots, x^m \in \mathbb{R}^n$. Wir suchen die k größten Vektoren, gemessen nach deren euklidischer Norm $\|x\| := \sqrt{\sum_{l=1}^{n} x_l^2}$.

[12]:
```
import numpy as np

# Wir speichern alle Vektoren in einem numpy Array:
# X = [x1 x2 ... xm]
n = 4
m = 20
k = 2
X = np.random.random((n,m))
```

```
print(X)

# Die euklidische Norm kann durch komponentenweises
# Quadrieren, aufsummieren entlang der Dimension 0
# (also spaltenweise) und anschließende Wurzelbildung
# berechnet werden. In numpy ein Einzeiler:
norms = np.sqrt(np.sum(X**2, axis=0))
print(norms)

# I speichert nun ein Array mit einer Indizierung
# so dass norms[I] sortiert ist:
I = np.argsort(norms)
print(norms[I])
print(I)

# Die größten Normen erhalten wir dann durch
# Auswahl der k letzten Einträge:
print(norms[I[-k:]])
print("Die {:d} größten Vektoren sind:".format(k))
print(X[:,I[-k:]])
```

```
[[0.63143622 0.6381661  0.43584958 0.15954705 0.04430214 0.82552866
  0.96567682 0.27802834 0.82175764 0.37676258 0.77177992 0.39818726
  0.69472445 0.0719457  0.31605854 0.23672704 0.42076358 0.25992616
  0.16863394 0.30466913]
 [0.40325407 0.79579666 0.01646407 0.56879695 0.33901188 0.60974108
  0.95033066 0.25995518 0.48749365 0.71014017 0.11487734 0.48355207
  0.10856232 0.68070299 0.96275273 0.27048761 0.35162081 0.90217788
  0.76353246 0.10802836]
 [0.68696204 0.01754047 0.78623564 0.926821   0.41906936 0.52164908
  0.5553215  0.27265753 0.95242547 0.3960686  0.54475561 0.58432347
  0.54318867 0.20649933 0.07954561 0.08802753 0.19865378 0.21076479
  0.69493207 0.61243448]
 [0.4978437  0.5796371  0.29059565 0.54614303 0.59377744 0.27979596
  0.75095522 0.87803205 0.18709495 0.77995839 0.70072632 0.05180879
  0.71991907 0.52060845 0.29360597 0.64637036 0.21834034 0.06998022
  0.19532121 0.64763213]]
[1.13185279 1.17338619 0.94490647 1.22729565 0.80317028 1.18477226
 1.64559284 0.99506726 1.36200385 1.18804651 1.18178556 0.85819215
 1.14357563 0.88442517 1.05797816 0.74481322 0.62274795 0.96478265
 1.06418992 0.9481544 ]
[0.62274795 0.74481322 0.80317028 0.85819215 0.88442517 0.94490647
 0.9481544  0.96478265 0.99506726 1.05797816 1.06418992 1.13185279
 1.14357563 1.17338619 1.18178556 1.18477226 1.18804651 1.22729565
 1.36200385 1.64559284]
[16 15  4 11 13  2 19 17  7 14 18  0 12  1 10  5  9  3  8  6]
[1.36200385 1.64559284]
Die 2 größten Vektoren sind:
```

```
[[0.82175764 0.96567682]
 [0.48749365 0.95033066]
 [0.95242547 0.5553215 ]
 [0.18709495 0.75095522]]
```

0.1.7 Logische Indizierung

Ein weitverbreitetes und nützliches Konstrukt ist die sogenannte **logische Indizierung**. Hierbei werden Teile eines Arrays addressiert, die über eine bool Matrix definiert werden:

[13]:
```
import numpy as np

A = np.array([1, 2, 3, 4, 5, 6])
# Anlegen einer Auswahlmatrix:
I = np.array([True, True, True, False, False, True], dtype=bool)
print("A[I] =", A[I])

# Die Indexmatrix I kann auch dazu verwendet werden,
# bestimmte Teile einer Matrix zu überschreiben:
A[I] = 0
print(A)
```

```
A[I] = [1 2 3 6]
[0 0 0 4 5 0]
```

[14]:
```
import numpy as np

# Einige weitere Einsatzmöglichkeiten:

A = np.arange(20).reshape((4,5))
# Alle Zahlen kleiner als 4
print(A)
print(A<4)

# Matrixausschnitte in einer Zeile:
print(A[A<4])

# Alle Zahlen die |x-10| < 5 erfüllen:
print(A[np.fabs(A-10) < 5])
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
[[ True  True  True  True False]
 [False False False False False]
 [False False False False False]
 [False False False False False]]
[0 1 2 3]
```

```
[ 6  7  8  9 10 11 12 13 14]
```

0.1.8 Broadcasting

Bei arithmetischen Matrizenoperationen sind Matrizen manchmal strukturiert (identische Zeilen/Spalten), und es ist ggf. einige Arbeit, diese Matrizen anzulegen.

Broadcasting ist eine Möglichkeit in NumPy Matrizen und Vektoren unterschiedlicher Dimension zu kombinieren.

Es wird hierbei durch Angabe einer Zeile (Spalte) einer Matrix diese implizit auf die richtige Breite erweitert (daher der Begriff "Broadcasting"), so dass die arithmetische Operation Sinn ergibt. Man spart sich also etwas Tipp-Arbeit.

Als erstes betrachten wir die komponentenweise Multiplikation einer Matrix mit einem Zeilenvektor, der implizit durch Ergänzen von Zeilenkopien auf die richtige Dimension gebracht wird:

$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix} * \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

[15]:
```
import numpy as np

A = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
B = np.array([[1, 2, 3]])
print(A * B)
```

```
[[11 24 39]
 [21 44 69]
 [31 64 99]]
```

Völlig analog funktioniert dies mit einem Spaltenvektor:

$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

[17]:
```
A = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
B = np.array([[1, 2, 3]]).T
print(A*B)
```

```
[[11 12 13]
 [42 44 46]
 [93 96 99]]
```

Und für richtig faule Menschen:

$$\begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix} * \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

[18]:
```
A = np.array([[10, 20, 30]])
B = np.array([[1, 2, 3]]).T
print(A*B)
```

```
[[10 20 30]
 [20 40 60]
 [30 60 90]]
```

0.1.9 Beispiel: Punkte mit geringstem Abstand

Gegeben seien Vektoren $x_1, \ldots, x_m \in \mathbb{R}^n$ und ein Vektor x^*. Wir suchen die k Punkte mit geringstem Abstand zu x^*:

[19]:
```
import numpy as np

# Anlegen m zufälliger Vektoren im R^n
# und speichern der Vektoren in einer Matrix
# X=[x1 x2 ... xm]
n = 2
m = 100
X = np.random.random((n,m))
x_star = np.array([[0.5, 0.5]]).T
k = 3

# Berechnen aller Abstände zum Vektor x_star:
distances = np.sqrt( np.sum((X - x_star)**2, axis=0) )

# Kleinste Abstände finden:
I = np.argsort(distances)
result = X[:,I[0:k]]
print(result)
```

```
[[0.48267513 0.54321634 0.57446515]
 [0.4567739  0.56343912 0.46837208]]
```

0.1.10 Beispiel: Auswertung einer 2D Funktion

Wir betrachten die Funktion $f(x,y) = xy$. Wir wollen die Funktion auf einem zweidimensionalen Gitter in der Box $[0,1] \times [0,1]$ auswerten. Dafür bieten sich die Broadcasting-Funktion von NumPy an:

[20]:
```
import numpy as np

# Anlegen des Gitters:
x = np.linspace(0,1,num=5).reshape((1,5))
y = np.linspace(0,1,num=5).reshape((5,1))
print(x)
print(y)
f = lambda x,y: x*y
```

```
points = f(x,y)
print(points)
```

```
[[0.   0.25 0.5  0.75 1.  ]]
[[0.  ]
 [0.25]
 [0.5 ]
 [0.75]
 [1.  ]]
[[0.     0.     0.     0.     0.    ]
 [0.     0.0625 0.125  0.1875 0.25  ]
 [0.     0.125  0.25   0.375  0.5   ]
 [0.     0.1875 0.375  0.5625 0.75  ]
 [0.     0.25   0.5    0.75   1.    ]]
```

0.2 Impressum

0.2.1 Programmierkurs Python, Dominik Göddeke https://www.ians.uni-stuttgart.de, Universität Stuttgart

Version vom April 2023

Gefördert durch die Stiftung Innovation in der Hochschullehre. (alle Rechte am Logo vorbehalten)

Gefördert mit Mitteln der Deutschen Forschungsgemeinschaft (EXC 2075 - 390740016) im Rahmen der Exzellenzstrategie.

[ ]: