

w-python-numpy-grundlagen

Lukas Arnold

Matthias Baitsch

Simone Arnold

Marc Fehr

Sebastian Seipel

Florian Bagemihl

Maik Poetzsch

2025-11-14

Inhaltsverzeichnis

Preamble	3
Intro	4
1 Einführung NumPy	6
2 Erstellen von NumPy arrays	8
3 Größe, Struktur und Typ	12
4 Rechnen mit Arrays	16
5 Slicing	21
6 Array Manipulation	26
7 Lesen und Schreiben von Dateien	31
8 Arbeiten mit Bildern	35
9 Lernzielkontrolle	43
10 Übung	51
11 Klausurfragen	61

Preamble



Bausteine Computergestützter Datenanalyse von Lukas Arnold, Simone Arnold, Florian Bagemihl, Matthias Baitsch, Marc Fehr, Franca Hollmann, Maik Poetzsch und Sebastian Seipel. „Werkzeugbaustein NumPy“ von Marc Fehr und Maik Poetzsch ist lizenziert unter [CC BY 4.0](#). Das Werk ist abrufbar unter <https://github.com/bausteine-der-datenanalyse/w-python-numpy-grundlagen>. Ausgenommen von der Lizenz sind alle Logos und anders gekennzeichneten Inhalte. 2025

Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Franca Hollmann, Maik Poetzsch, und Sebastian Seipel. 2025. „Bausteine Computergestützter Datenanalyse. Werkzeugbaustein NumPy“. <https://github.com/bausteine-der-datenanalyse/w-python-numpy-grundlagen>.

BibTeX-Vorlage

```
@misc{BCD-numpy-2025,  
  title={Bausteine Computergestützter Datenanalyse. Werkzeugbaustein NumPy},  
  author={Arnold, Lukas and Arnold, Simone and Baitsch, Matthias and Fehr, Marc and Hollmann, Maik and Poetzsch, Sebastian},  
  year={2025},  
  url={https://github.com/bausteine-der-datenanalyse/w-python-numpy-grundlagen}}
```

Intro

Voraussetzungen

- Grundlagen Python
- Einbinden von zusätzlichen Paketen
- Plotten mit Matplotlib

Verwendete Pakete und Datensätze

Pakete

- NumPy
- Matplotlib

Datensätze

- TC01.csv
- Bild: Mona Lisa
- Bild: Campus

Bearbeitungszeit

Geschätzte Bearbeitungszeit: 2h

Lernziele

- Einleitung: was ist NumPy, Vor- und Nachteile
- Nutzen des NumPy-Moduls
- Erstellen von NumPy-Arrays
- Slicing

- Lesen und schreiben von Dateien
- Arbeiten mit Bildern

1 Einführung NumPy

NumPy ist eine leistungsstarke Bibliothek für Python, die für numerisches Rechnen und Datenanalyse verwendet wird. Daher auch der Name NumPy, ein Akronym für “Numerisches Python” (englisch: “Numeric Python” oder “Numerical Python”). NumPy selbst ist hauptsächlich in der Programmiersprache C geschrieben, weshalb NumPy generell sehr schnell ist.

NumPy bietet ein effizientes Arbeiten mit kleinen und großen Vektoren und Matrizen, die so ansonsten nur umständlich in nativem Python implementiert werden würden. Dabei bietet NumPy auch die Möglichkeit, einfach mit Vektoren und Matrizen zu rechnen und das auch für sehr große Datenmengen.

Diese Einführung wird Ihnen dabei helfen, die Grundlagen von NumPy zu verstehen und zu nutzen.

1.1 Vorteile & Nachteile

Fast immer sind Operationen mit Numpy Datenstrukturen schneller. Im Gegensatz zu nativen Python Listen kann man dort aber nur einen Datentyp pro Liste speichern.

i Warum ist numpy oftmals schneller?

NumPy implementiert eine effizientere Speicherung von Listen im Speicher. Nativ speichert Python Listeninhalte aufgeteilt, wo gerade Platz ist.

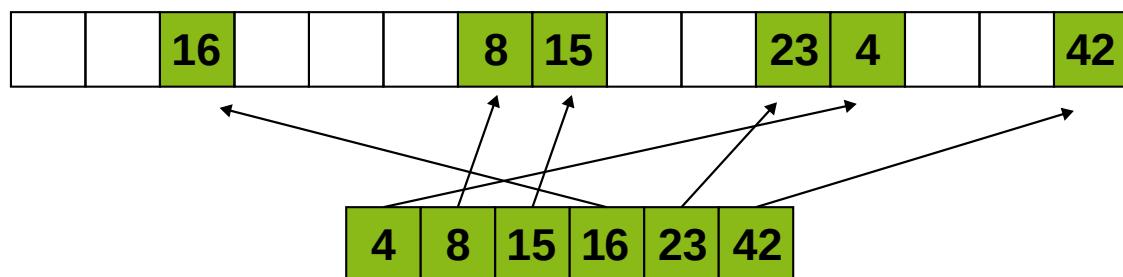


Abbildung 1.1: Speicherung von Daten in nativem Python

Dagegen werden NumPy-Arrays und Matrizen zusammenhängend gespeichert, was einen effizienteren Datenaufruf ermöglicht.

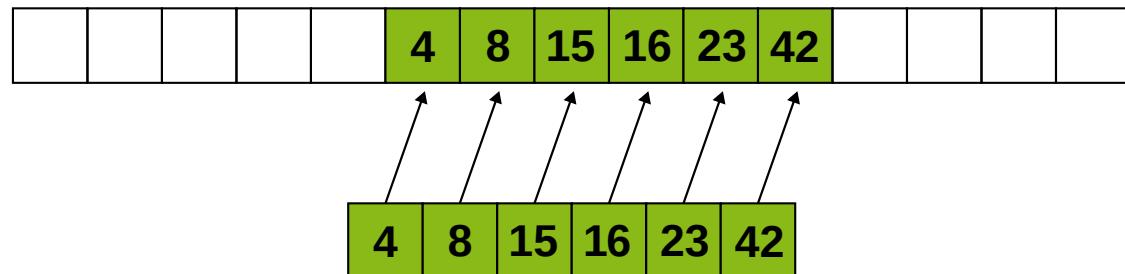


Abbildung 1.2: Speicherung von Daten bei Numpy

Dies bedeutet aber auch, dass eine Erweiterung einer Liste deutlich schneller ist als eine Erweiterung von Arrays oder Matrizen. Bei Listen kann jeder freie Platz genutzt werden, während Arrays und Matrizen an einen neuen Ort im Speicher kopiert werden müssen.

1.2 Einbinden des Pakets

NumPy wird über folgende Zeile eingebunden. Dabei hat sich global der Standard entwickelt, als Alias `np` zu verwenden.

```
import numpy as np
```

1.3 Referenzen

Sämtliche hier vorgestellten Funktionen lassen sich in der (englischen) NumPy-Dokumentation nachschlagen: [Dokumentation](#)

2 Erstellen von NumPy arrays

Typischerweise werden in Python Vektoren durch Listen und Matrizen durch geschachtelte Listen ausgedrückt. Beispielsweise würde man den Vektor

$$(1, 2, 3, 4, 5, 6) \quad \text{und die Matrix} \quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

nativ in Python so erstellen:

```
liste = [1, 2, 3, 4, 5, 6]

matrix = [[1, 2, 3], [4, 5, 6]]

print(liste)
print(matrix)
```

```
[1, 2, 3, 4, 5, 6]
[[1, 2, 3], [4, 5, 6]]
```

Möchte man jetzt NumPy-Arrays verwenden, benutzt man den Befehl `np.array()`.

```
liste = np.array([1, 2, 3, 4, 5, 6])

matrix = np.array([[1, 2, 3], [4, 5, 6]])

print(liste)
print(matrix)
```

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

Betrachtet man die Ausgaben der `print()` Befehle, fallen zwei Sachen auf: Zum einen fallen die Kommata weg und zum anderen wird die Matrix passend ausgegeben.

Es gibt auch die Möglichkeit, höherdimensionale Arrays zu erstellen. Dabei wird eine neue Ebene der Verschachtelung benutzt. Im folgenden Beispiel wird eine dreidimensionale Matrix erstellt.

```
matrix_3d = np.array([[ [1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

Es gilt als “good practice”, Arrays immer zu initialisieren. Dafür bietet NumPy drei Funktionen, um vorinitialisierte Arrays zu erzeugen. Alternativ können Arrays auch mit festgesetzten Werten initialisiert werden. Dafür kann entweder die Funktion `np.zeros()` verwendet werden, die alle Werte auf 0 setzt, oder aber `np.ones()`, welche alle Werte mit 1 initialisiert. Der Funktion wird die Form im Format [Reihen, Spalten] übergeben. Möchte man alle Einträge auf einen spezifischen Wert setzen, kann man den Befehl `np.full()` benutzen.

```
np.zeros([2,3])
```

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

```
np.ones([2,3])
```

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

```
np.full([2,3],7)
```

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

💡 Wie könnte man Arrays, die mit einer Zahl x gefüllt sind, auch erstellen?

Der Trick besteht hierbei darin, ein Array mit `np.ones()` zu initialisieren und dieses Array dann mit der Zahl x zu multiplizieren. Im folgenden Beispiel ist `x = 5`.

```
np.ones([2,3]) * 5
```

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

Möchte man zum Beispiel für eine Achse in einem Plot einen Vektor mit gleichmäßig verteilten Werten erstellen, bieten sich in NumPy zwei Möglichkeiten. Mit den Befehlen `np.linspace(Start,Stopp,#Anzahl Werte)` und `np.arange(Start,Stopp,Abstand zwischen Werten)` können solche Arrays erstellt werden.

```
np.linspace(0,1,11)
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

```
np.arange(0,10,2)
```

```
array([0, 2, 4, 6, 8])
```

💡 Zwischenübung: Array Erstellung

Erstellen Sie jeweils ein NumPy-Array, mit dem folgenden Inhalt:

1. mit den Werten 1, 7, 42, 99
2. zehn mal die Zahl 5
3. mit den Zahlen von 35 **bis einschließlich** 50
4. mit allen geraden Zahlen von 20 **bis einschließlich** 40
5. eine Matrix mit 5 Spalten und 4 Reihen mit dem Wert 4 an jeder Stelle
6. mit 10 Werten die gleichmäßig zwischen 22 und einschließlich 40 verteilt sind

Lösung

```
# 1.  
print(np.array([1, 7, 42, 99]))
```

```
[ 1  7 42 99]
```

```
# 2.  
print(np.full(10,5))
```

```
[5 5 5 5 5 5 5 5 5 5]
```

```
# 3.  
print(np.arange(35, 51))
```

```
[35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50]
```

```
# 4.  
print(np.arange(20, 41, 2))
```

```
[20 22 24 26 28 30 32 34 36 38 40]
```

```
# 5.  
print(np.full([4,5],4))
```

```
[[4 4 4 4]  
 [4 4 4 4]  
 [4 4 4 4]  
 [4 4 4 4]]
```

```
# 6.  
print(np.linspace(22, 40, 10))
```

```
[22. 24. 26. 28. 30. 32. 34. 36. 38. 40.]
```

3 Größe, Struktur und Typ

Wenn man sich nicht mehr sicher ist, welche Struktur oder Form ein Array hat oder diese Größen zum Beispiel für Schleifen nutzen möchte, bietet NumPy folgende Funktionen für das Auslesen dieser Größen an.

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
```

`np.shape()` gibt die Längen der Dimensionen in Form eines Tupels zurück.

```
np.shape(matrix)
```

```
(2, 3)
```

Die native Python Funktion `len()` gibt dagegen nur die Länge der ersten Dimension, also die Anzahl der Elemente in den äußeren Klammern wieder. Im obigen Beispiel würde `len()` also die beiden Listen `[1, 2, 3]` und `[4, 5, 6]` sehen.

```
len(matrix)
```

```
2
```

Die Funktion `np.ndim()` gibt im Gegensatz zu `np.shape()` nur die Anzahl der Dimensionen zurück.

```
np.ndim(matrix)
```

```
2
```

💡 Die Ausgabe von `np.ndim()` kann mit `np.shape()` und einer nativen Python-Funktion erreicht werden. Wie?

`np.ndim()` gibt die Länge der Liste von `np.shape()` aus.

```
len(np.shape(matrix))
```

2

Möchte man die Anzahl aller Elemente in einem Array ausgeben, kann man die Funktion `np.size()` benutzen.

```
np.size(matrix)
```

6

NumPy-Arrays können verschiedene Datentypen haben. Im Folgenden haben wir drei Arrays mit jeweils unterschiedlichem Datentyp.

```
typ_a = np.array([1, 2, 3, 4, 5])
typ_b = np.array([0.1, 0.2, 0.3, 0.4, 0.5])
typ_c = np.array(["Montag", "Dienstag", "Mittwoch"])
```

Mit der Methode `np.dtype` können wir den Datentyp von Arrays ausgeben lassen. Meist wird dabei der Typ plus eine Zahl ausgegeben, welche die zum Speichern benötigte Bytezahl angibt. Das Array `typ_a` beinhaltet den Datentyp `int64`, also ganze Zahlen.

```
print(typ_a.dtype)
```

`int64`

Das Array `typ_b` beinhaltet den Datentyp `float64`, wobei `float` für Gleitkommazahlen steht.

```
print(typ_b.dtype)
```

`float64`

Das Array `typ_c` beinhaltet den Datentyp `U8`, wobei das U für Unicode steht. Hier wird als Unicodetext gespeichert.

```
print(typ_c.dtype)
```

<U8

Im folgenden finden Sie eine Tabelle mit den typischen Datentypen, die sie häufig antreffen.

Tabelle 3.1: Typische Datentypen in NumPy

Datentyp	Numpy Name	Beispiele
Wahrheitswert	bool	[True, False, True]
Ganze Zahl	int	[-2, 5, -6, 7, 3]
Positive ganze Zahl	uint	[1, 2, 3, 4, 5]
Kommazahl	float	[1.3, 7.4, 3.5, 5.5]
komplexe Zahl	complex	[-1 + 9j, 2-77j, 72 + 11j]
Textzeichen	U	["montag", "dienstag"]

💡 Zwischenübung: Array-Informationen auslesen

Gegeben sei folgende Matrix:

```
matrix = np.array([[ [ 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]]])
```

Bestimmen Sie durch Anschauen die Anzahl an Dimensionen und die Länge jeder Dimension. Von welchem Datentyp ist der Inhalt dieser Matrix?

Überprüfen Sie daraufhin Ihre Ergebnisse, indem Sie die passenden NumPy-Funktionen anwenden.

Lösung

```
matrix = np.array([[[ 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]]])

anzahl_dimensionen = np.ndim(matrix)

print("Anzahl unterschiedlicher Dimensionen: ", anzahl_dimensionen)

laenge_dimensionen = np.shape(matrix)

print("Länge der einzelnen Dimensionen: ", laenge_dimensionen)

print(matrix.dtype)

Anzahl unterschiedlicher Dimensionen: 3
Länge der einzelnen Dimensionen: (3, 3, 4)
int64
```

4 Rechnen mit Arrays

4.1 Arithmetische Funktionen

Ein großer Vorteil von NumPy ist das Rechnen mit Arrays. Ohne NumPy müsste man entweder eine Schleife oder aber `list comprehension` benutzen, um mit sämtlichen Werten in der Liste zu rechnen. In NumPy fällt diese Unannehmlichkeit weg.

```
a = np.array([1, 2, 3, 4, 5])  
b = np.array([9, 8, 7, 6, 5])
```

Normale mathematische Operationen wie die Addition lassen sich auf zwei Arten ausdrücken: Entweder über die `np.add()` Funktion oder aber simpel über den Operator `+`.

```
np.add(a,b)  
  
array([10, 10, 10, 10, 10])  
  
a + b  
  
array([10, 10, 10, 10, 10])
```

Ohne NumPy würde die Operation folgendermaßen aussehen:

```
ergebnis = np.ones(5)  
for i in range(len(a)):  
    ergebnis[i] = a[i] + b[i]  
  
print(ergebnis)
```

```
[10. 10. 10. 10. 10.]
```

Für die anderen Rechenarten existieren auch Funktionen: `np.subtract()`, `np.multiply()` und `np.divide()`.

Auch für die höheren Rechenoperationen gibt es Funktionen:

- `np.exp(a)`
- `np.sqrt(a)`
- `np.power(a, 3)`
- `np.sin(a)`
- `np.cos(a)`
- `np.tan(a)`
- `np.log(a)`
- `a.dot(b)`

⚠️ Arbeiten mit Winkelfunktionen

Wie auch am Taschenrechner birgt das Arbeiten mit den Winkelfunktionen (`sin`, `cos`, ...) die Fehlerquelle, dass man nicht mit Radian-Werten, sondern mit Grad-Werten arbeitet. Die Winkelfunktionen in NumPy erwarten jedoch Radian-Werte.
Für eine einfache Umrechnung bietet NumPy die Funktionen `np.deg2rad()` und `np.rad2deg()`.

4.2 Vergleiche

NumPy-Arrays lassen sich auch miteinander vergleichen. Betrachten wir die folgenden zwei Arrays:

```
a = np.array([1, 2, 3, 4, 5])  
  
b = np.array([9, 2, 7, 4, 5])
```

Möchten wir feststellen, ob diese zwei Arrays identisch sind, können wir den `==`-Komparator benutzen. Dieser vergleicht die Arrays elementweise.

```
a == b  
  
array([False, True, False, True, True])
```

Es ist außerdem möglich Arrays mit den `>`- und `<`-Operatoren zu vergleichen:

```
a < b
```

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

Möchte man Arrays mit Gleitkommazahlen vergleichen, ist es oftmals nötig, eine gewisse Toleranz zu benutzen, da bei Rechenoperationen minimale Rundungsfehler entstehen können.

```
a = np.array(0.1 + 0.2)
b = np.array(0.3)
a == b
```

```
np.False_
```

Für diesen Fall gibt es eine Vergleichsfunktion `np.isclose(a,b,atol)`, wobei `atol` für die absolute Toleranz steht. Im folgenden Beispiel wird eine absolute Toleranz von 0,001 verwendet.

```
a = np.array(0.1 + 0.2)
b = np.array(0.3)
print(np.isclose(a, b, atol=0.001))
```

```
True
```

i Warum ist $0.1 + 0.2$ nicht gleich 0.3 ?

Zahlen werden intern als Binärzahlen dargestellt. So wie $1/3$ nicht mit einer endlichen Anzahl an Ziffern korrekt dargestellt werden kann, müssen Zahlen ggf. gerundet werden, um im Binärsystem dargestellt zu werden.

```
a = 0.1
b = 0.2
print(a + b)
```

```
0.30000000000000004
```

4.3 Aggregatfunktionen

Für verschiedene Auswertungen benötigen wir Funktionen, wie etwa die Summen- oder die Mittelwert-Funktion. Starten wir mit einem Beispiel-Array a:

```
a = np.array([1, 2, 3, 4, 8])
```

Die Summe wird über die Funktion `np.sum()` berechnet.

```
np.sum(a)
```

```
np.int64(18)
```

Natürlich lassen sich auch der Minimalwert und der Maximalwert eines Arrays ermitteln. Die beiden Funktionen lauten `np.min()` und `np.max()`.

```
np.min(a)
```

```
np.int64(1)
```

Möchte man nicht das Maximum selbst, sondern die Position des Maximums bestimmen, wird statt `np.max()` die Funktion `np.argmax()` verwendet.

Für statistische Auswertungen werden häufig die Funktion für den Mittelwert `np.mean()`, die Funktion für den Median `np.median()` und die Funktion für die Standardabweichung `np.std()` verwendet.

```
np.mean(a)
```

```
np.float64(3.6)
```

```
np.median(a)
```

```
np.float64(3.0)
```

```
np.std(a)
```

```
np.float64(2.4166091947189146)
```

Zwischenübung: Rechnen mit Arrays

Gegeben sind zwei eindimensionale Arrays a und b:

a = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100]) und b = np.array([5, 15, 25, 35, 45, 55, 65, 75, 85, 95])

1. Erstellen Sie ein neues Array, das die Sinuswerte der addierten Arrays a und b enthält.
2. Berechnen Sie die Summe, den Mittelwert und die Standardabweichung der Elemente in a.
3. Finden Sie den jeweils größten und den kleinsten Wert in a und b.

Lösung

```
a = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
b = np.array([5, 15, 25, 35, 45, 55, 65, 75, 85, 95])

# 1.
sin_ab = np.sin(a + b)

# 2.
sum_a = np.sum(a)
mean_a = np.mean(a)
std_a = np.std(a)

# 3.
max_a = np.max(a)
min_a = np.min(a)
max_b = np.max(b)
min_b = np.min(b)
```

5 Slicing

5.1 Normales Slicing mit Zahlenwerten

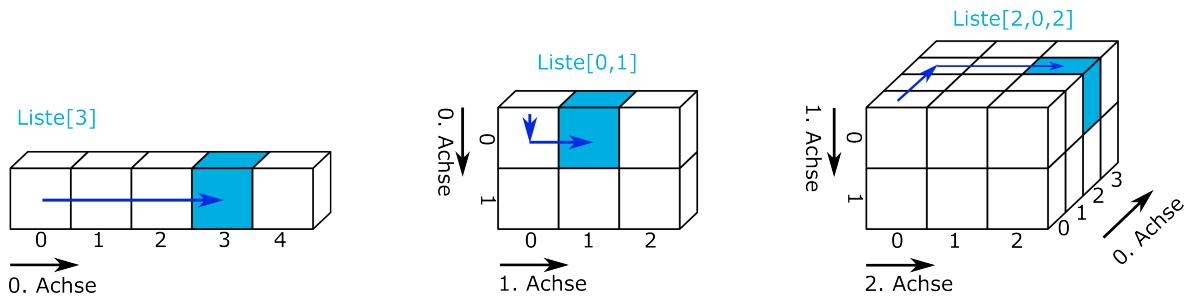


Abbildung 5.1: Ansprechen der einzelnen Achsen für den ein-, zwei- und dreidimensionalen Fall inkl. jeweiligem Beispiel

Möchte man Daten innerhalb eines Arrays auswählen, so geschieht das in der Form:

1. `array[a]` wobei ein einzelner Wert an Position a ausgegeben wird.
2. `array[a:b]` wobei alle Werte von Position a bis Position b-1 ausgegeben werden.
3. `array[a:b:c]` wobei die Werte von Position a bis Position b-1 mit einer Schrittweite von c ausgegeben werden.

```
liste = np.array([1, 2, 3, 4, 5, 6])
```

```
# Auswählen des ersten Elements  
liste[0]
```

```
np.int64(1)
```

```
# Auswählen des letzten Elements  
liste[-1]
```

```
np.int64(6)
```

```
# Auswählen einer Reihe von Elementen
liste[1:4]

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

Für zweidimensionale Arrays wählt man mit einer zweiten, durch ein Komma von der ersten getrennten Zahl die zweite Dimension aus.

```
matrix = np.array([[1, 2, 3], [4, 5, 6])

# Auswählen eines Elements
matrix[1,1]

np.int64(5)
```

Für dreidimensionale Arrays wählt man mit einer weiteren, ebenfalls durch ein Komma getrennten Zahl die dritte Dimension aus. Dabei wird diese jedoch an die erste Stelle gesetzt.

```
matrix_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(matrix_3d)

[[[ 1  2  3]
 [ 4  5  6]]

 [[ 7  8  9]
 [10 11 12]]]

# Auswählen eines Elements
matrix_3d[1,0,2]

np.int64(9)
```

5.2 Slicing mit logischen Werten (boolesche Masken)

Beim logischen Slicing wird eine boolesche Maske verwendet, um bestimmte Elemente eines Arrays auszuwählen. Die Maske ist ein Array gleicher Länge wie das Original, das aus den Werten `True` oder `False` besteht.

```
# Erstellen wir ein Beispiel Array
a = np.array([1, 2, 3, 4, 5, 6])

# Erstellen der Maske
maske = a > 3

print(maske)
```

```
[False False False  True  True  True]
```

Wir erhalten also ein Array mit boolschen Werten. Verwenden wir diese Maske nun zum slicen, erhalten wir alle Werte an den Stellen, an denen die Maske den Wert **True** besitzt.

```
# Anwenden der Maske
print(a[maske])
```

```
[4 5 6]
```

Mehrere Masken können mit dem Operator ***** zu einem logischen UND verknüpft werden.

```
# 2 Masken
gerade_zahlen = a % 2 == 0
kleiner_3 = a < 3
print(gerade_zahlen, kleiner_3, sep = '\n')

# logisches UND beider Masken
print(a[kleiner_3 * gerade_zahlen])
```

```
[False  True False  True False  True]
[ True  True False False False]
[2]
```

⚠ Warnung

Das Verwenden von booleschen Arrays ist nicht für native Python-Listen möglich. Hier muss durch die Liste iteriert werden.

```

a = [1, 2, 3, 4, 5, 6]
ergebnis = [x for x in a if x > 3]
print(ergebnis)

```

[4, 5, 6]

💡 Zwischenübung: Array-Slicing

Wählen Sie die farblich markierten Bereiche aus dem Array "matrix" mit den eben gelernten Möglichkeiten des Array-Slicing aus.

0	2	11	18	47	33	48	9	31	8	41
1	55	1	8	3	91	56	17	54	23	12
2	19	99	56	72	6	13	34	16	77	56
3	37	75	67	5	46	98	57	19	14	7
4	4	57	32	78	56	12	43	61	3	88
5	96	16	92	18	50	90	35	15	36	97
6	75	4	38	53	1	79	56	73	45	56
7	15	76	11	93	87	8	2	58	86	94
8	51	14	60	57	74	42	59	71	88	52
9	49	6	43	39	17	18	95	6	44	75

Darstellung der in der folgenden Code-Zelle erstellten Matrix."}

{fig-alt:“Grafische

```
matrix = np.array([
    [2, 11, 18, 47, 33, 48, 9, 31, 8, 41],
    [55, 1, 8, 3, 91, 56, 17, 54, 23, 12],
    [19, 99, 56, 72, 6, 13, 34, 16, 77, 56],
    [37, 75, 67, 5, 46, 98, 57, 19, 14, 7],
    [4, 57, 32, 78, 56, 12, 43, 61, 3, 88],
    [96, 16, 92, 18, 50, 90, 35, 15, 36, 97],
    [75, 4, 38, 53, 1, 79, 56, 73, 45, 56],
    [15, 76, 11, 93, 87, 8, 2, 58, 86, 94],
    [51, 14, 60, 57, 74, 42, 59, 71, 88, 52],
    [49, 6, 43, 39, 17, 18, 95, 6, 44, 75]
])
```

Lösung

- Rot: matrix[1,3]
- Grün: matrix[4:6,2:6]
- Pink: matrix[:,7]
- Orange: matrix[7,:5]
- Blau: matrix[-1,-1]

6 Array Manipulation

6.1 Ändern der Form

Durch verschiedene Funktionen lassen sich die Form und die Einträge der Arrays verändern.

Eine der wichtigsten Array-Operationen ist das Transponieren. Dabei werden Reihen in Spalten und Spalten in Reihen umgewandelt.

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(matrix)
```

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

Transponieren wir dieses Array nun, erhalten wir:

```
print(np.transpose(matrix))
```

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

Wollen wir nun aus dieser Matrix einen Vektor erstellen, so können wir die Methode `np.flatten()` benutzen:

```
vector = matrix.flatten()
print(vector)
```

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

Um wieder eine zweidimensionale Datenstruktur zu erhalten, benutzen wir die Funktion `np.reshape(Ziel, Form)`

```
print(np.reshape(matrix, [3, 2]))
```

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

Möchten wir den Inhalt eines bereits bestehenden Arrays erweitern, verkleinern oder ändern, bietet NumPy ebenfalls die passenden Funktionen.

Haben wir ein leeres Array oder wollen wir ein schon volles Array erweitern, benutzen wir die Funktion `np.append()`. Dabei hängen wir einen Wert an das bereits bestehende Array an.

```
liste = np.array([1, 2, 3, 4, 5, 6])

neue_liste = np.append(liste, 7)
print(neue_liste)
```

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

Gegebenenfalls ist es nötig, einen Wert nicht am Ende, sondern an einer beliebigen Position im Array einzufügen. Das passende Werkzeug ist hierfür die Funktion `np.insert(Array, Position, Einschub)`. Im folgenden Beispiel wird an der Indexposition 3 die Zahl 7 eingesetzt.

```
liste = np.array([1, 2, 3, 4, 5, 6])

neue_liste = np.insert(liste, 3, 7)
print(neue_liste)
```

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

Wenn sich neue Elemente einfügen lassen, können natürlich auch Elemente gelöscht werden. Hierfür wird die Funktion `np.delete(Array, Position)` benutzt, die ein Array und die Position der zu löschenen Funktion übergeben bekommt.

```
liste = np.array([1, 2, 3, 4, 5, 6])

neue_liste = np.delete(liste, 3)
print(neue_liste)
```

```
[1 2 3 5 6]
```

Zuletzt wollen wir uns noch die Verbindung zweier Arrays anschauen. Im folgenden Beispiel wird dabei das Array `b` an das Array `a` mithilfe der Funktion `np.concatenate((Array a, Array b))` angehängt.

```
a = np.array([1, 2, 3, 4, 5, 6])
b = np.array([7, 8, 9, 10])

neue_liste = np.concatenate((a, b))
print(neue_liste)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

6.2 Sortieren von Arrays

NumPy bietet auch die Möglichkeit, Arrays zu sortieren. Im folgenden Beispiel starten wir mit einem unsortierten Array. Mit der Funktion `np.sort()` erhalten wir ein sortiertes Array.

```
import numpy as np
unsortiert = np.array([4, 2, 1, 6, 3, 5])

sortiert = np.sort(unsortiert)

print(sortiert)
```

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

6.3 Unterlisten mit einzigartigen Werten

Arbeitet man mit Daten, bei denen zum Beispiel Projekten Personalnummern zugeordnet werden, hat man Daten mit einer endlichen Anzahl an Personalnummern, die jedoch mehrfach vorkommen können, wenn diese an mehr als einem Projekt gleichzeitig arbeiten.

Möchte man nun eine Liste erstellen, die jede Nummer nur einmal enthält, kann die Funktion `np.unique()` verwendet werden.

```

import numpy as np
liste_mit_dopplungen = np.array([4, 1, 1, 6, 3, 4, 7, 3, 3])

einzigartige_werte = np.unique(liste_mit_dopplungen)

print(einzigartige_werte)

```

[1 3 4 6 7]

Setzt man dann noch die Option `return_counts=True` kann in einer zweiten Variable gespeichert werden, wie oft jeder Wert vorkommt.

```

import numpy as np
liste_mit_dopplungen = np.array([4, 1, 1, 6, 3, 4, 7, 3, 3])

einzigartige_werte, anzahl = np.unique(liste_mit_dopplungen, return_counts=True)

print(anzahl)

```

[2 3 2 1 1]

Zwischenübung: Array-Manipulation

Gegeben ist das folgende zweidimensionale Array `matrix`:

```

matrix = np.array([
    [4, 7, 2, 8],
    [1, 5, 3, 6],
    [9, 2, 4, 7]
])

```

1. Ändern Sie die Form des Arrays `matrix` in ein eindimensionales Array.
2. Sortieren Sie das eindimensionale Array in aufsteigender Reihenfolge.
3. Ändern Sie die Form des sortierten Arrays in ein zweidimensionales Array mit 2 Zeilen und 6 Spalten.
4. Bestimmen Sie die eindeutigen Elemente im ursprünglichen Array `matrix` und geben Sie diese aus.

Lösung

```
matrix = np.array([
    [4, 7, 2, 8],
    [1, 5, 3, 6],
    [9, 2, 4, 7]
])

# 1. Ändern der Form in ein eindimensionales Array
flat_array = matrix.flatten()

# 2. Sortieren des eindimensionalen Arrays in aufsteigender Reihenfolge
sorted_array = np.sort(flat_array)

# 3. Ändern der Form des sortierten Arrays in ein 2x6-Array
reshaped_array = sorted_array.reshape(2, 6)

# 4. Bestimmen der eindeutigen Elemente im ursprünglichen Array
unique_elements_original = np.unique(matrix)
```

7 Lesen und Schreiben von Dateien

Das Modul NumPy stellt Funktionen zum Lesen und Schreiben von strukturierten Textdateien bereit.

7.1 Lesen von Dateien

Zum Lesen von strukturierten Textdateien, z. B. im CSV-Format (comma separated values), kann die Funktion `np.loadtxt()` verwendet werden. Diese bekommt als Argumente den einzulesenden Dateinamen und weitere Optionen zur Definition der Struktur der Daten. Der Rückgabewert ist ein (mehrdimensionales) Array.

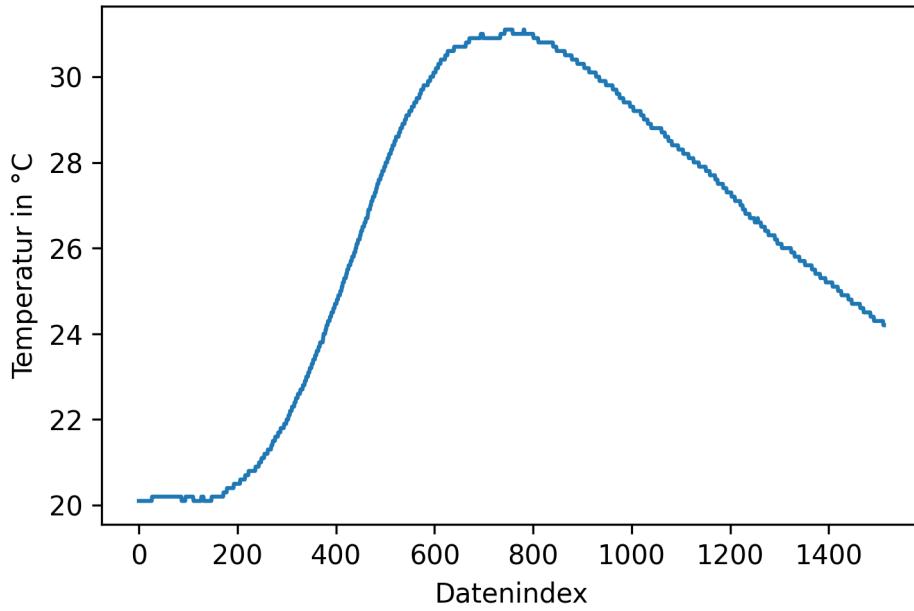
Im folgenden Beispiel wird die Datei `TC01.csv` eingelesen und deren Inhalt graphisch dargestellt. Die erste Zeile der Datei wird dabei ignoriert, da sie als Kommentar – eingeleitet durch das #-Zeichen – interpretiert wird.

```
dateiname = '01-daten/TC01.csv'  
daten = np.loadtxt(dateiname)
```

```
print("Daten:", daten)  
print("Form:", daten.shape)
```

```
Daten: [20.1 20.1 20.1 ... 24.3 24.2 24.2]  
Form: (1513,)
```

```
plt.plot(daten)  
plt.xlabel('Datenindex')  
plt.ylabel('Temperatur in °C');
```



Standardmäßig erwartet die `np.loadtxt()`-Funktion Komma separierte Werte. Werden die Daten durch ein anderes Trennzeichen getrennt, kann mit der Option `delimiter = ""` ein anderes Trennzeichen ausgewählt werden. Beispielsweise würde der Funktionsaufruf bei einem Semikolon folgendermaßen aussehen: `np.loadtxt(data.txt, delimiter = ";")`.

Enthalten die ersten Zeilen einer Datei zusätzliche Informationen wie die verwendeten Einheiten oder eine Beschreibung des Experiments, können diese Zeilen mit der Option `skiprows = #Reihen` übersprungen werden.

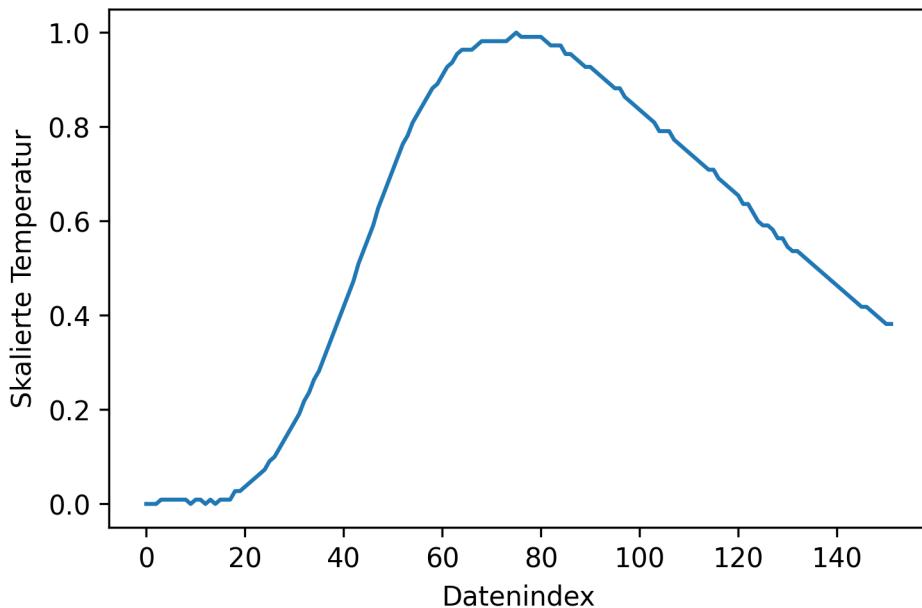
7.2 Schreiben von Dateien

Zum Schreiben von Arrays in Dateien, kann die in NumPy verfügbare Funktion `np.savetxt()` verwendet werden. Dieser müssen mindestens die zu schreibenden Arrays als auch ein Dateiname übergeben werden. Darüber hinaus sind zahlreiche Formatierungs- bzw. Strukturierungsoptionen möglich.

Folgendes Beispiel skaliert die oben eingelesenen Daten und schreibt jeden zehnten Wert in eine Datei. Dabei wird auch ein Kommentar (`header`-Argument) am Anfang der Datei erzeugt. Das Ausgabeformat der Zahlen kann mit dem `fmt`-Argument angegeben werden. Das Format ähnelt der Darstellungsweise, welche bei den formatierten Zeichenketten vorgestellt wurde.

```
wertebereich = np.max(daten) - np.min(daten)
daten_skaliert = ( daten - np.min(daten) ) / wertebereich
daten_skaliert = daten_skaliert[::10]
```

```
plt.plot(daten_skaliert)
plt.xlabel('Datenindex')
plt.ylabel('Skalierte Temperatur');
```



Beim Schreiben der Datei wird ein mehrzeiliger Kommentar mithilfe des Zeilenumbruchzeichens \n definiert. Die Ausgabe der Gleitkommazahlen wird mit %5.2f formatiert, was 5 Stellen insgesamt und zwei Nachkommastellen entspricht.

```
# Zuweisung ist auf mehrere Zeilen aufgeteilt, aufgrund der
# schmalen Darstellung im Skript
kommentar = f'Daten aus {dateiname} skaliert auf den Bereich ' + \
            '0 bis 1 \noriginales Min / Max:' + \
            f'{np.min(daten)}/{np.max(daten)}'
neu_dateiname = '01-daten/TC01_skaliert.csv'

np.savetxt(neu_dateiname, daten_skaliert,
           header=kommentar, fmt='%5.2f')
```

Zum Veranschaulichen werden die ersten Zeilen der neuen Datei ausgegeben.

```
# Einlesen der ersten Zeilen der neu erstellten Datei
datei = open(neu_dateiname, 'r')
for i in range(10):
    print( datei.readline() , end=' ')
datei.close()

# Daten aus 01-daten/TC01.csv skaliert auf den Bereich 0 bis 1
# originales Min / Max:20.1/31.1
0.00
0.00
0.00
0.01
0.01
0.01
0.01
0.01
0.01
```

8 Arbeiten mit Bildern

Bilder werden digital als Matrizen gespeichert. Dabei werden pro Pixel drei Farbwerte (rot, grün, blau) gespeichert. Aus diesen drei Farbwerten (Wertebereich 0-255) werden dann alle gewünschten Farben zusammengestellt.

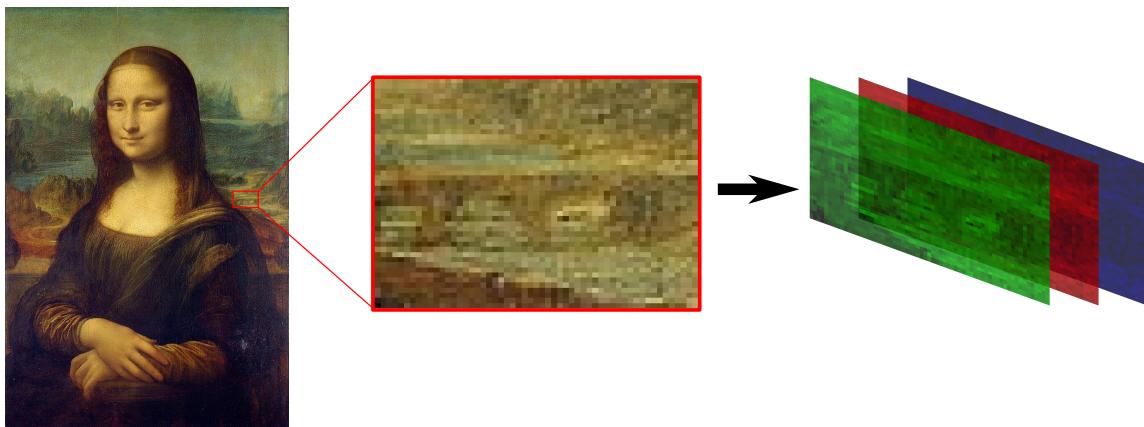


Abbildung 8.1: Ein hochauflößtes Bild besteht aus sehr vielen Pixeln. Jedes Pixel enthält 3 Farbwerte, einen für die Farbe Rot, einen für Grün und einen für Blau.

Aufgrund der digitalen Darstellung von Bildern lassen sich diese mit den Werkzeugen von NumPy leicht bearbeiten. Wir verwenden für folgendes Beispiel als Bild die Mona Lisa. Das Bild ist unter folgendem [Link](#) zu finden.

Importieren wir dieses Bild nun mit der Funktion `imread()` aus dem matplotlib-package, sehen wir, dass es sich um ein dreidimensionales NumPy-Array handelt.

```
import matplotlib.pyplot as plt

data = plt.imread("00-bilder/mona_lisa.jpg")
print("Form:", data.shape)
```

Form: (1024, 677, 3)

Schauen wir uns einmal mit der `print()`-Funktion diese Daten an.

```
print(data)
```

```
[[[ 68  62  38]
 [ 88  82  56]
 [ 92  87  55]
 ...
 [ 54  97  44]
 [ 68 110  60]
 [ 69 111  63]]
```

```
[[ 65  59  33]
 [ 68  63  34]
 [ 83  78  46]
 ...
 [ 66 103  51]
 [ 66 103  52]
 [ 66 102  56]]
```

```
[[ 97  90  62]
 [ 87  80  51]
 [ 78  72  38]
 ...
 [ 79 106  53]
 [ 62  89  38]
 [ 62  88  41]]
```

```
...
```

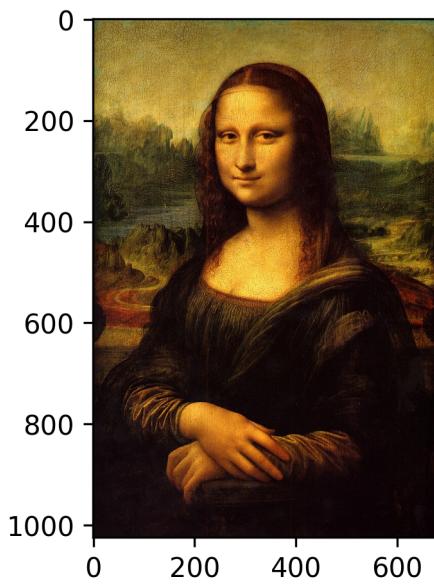
```
[[ 25  14  18]
 [ 21  10  14]
 [ 20   9  13]
 ...
 [ 11   5   9]
 [ 11   5   9]
 [ 10   4   8]]
```

```
[[ 23  12  16]
 [ 23  12  16]
 [ 21  10  14]
 ...
 [ 11   5   9]
 [ 11   5   9]]
```

```
[ 10   4   8]]  
  
[[ 22  11  15]  
 [ 26  15  19]  
 [ 24  13  17]  
 ...  
 [ 11   5   9]  
 [ 10   4   8]  
 [  9   3   7]]]
```

Mit der Funktion `plt.imshow()` kann das Bild in Echtfarben dargestellt werden. Dies funktioniert, da die Funktion die einzelnen Ebenen, hier der letzte Index des Datensatzes, als Farbinformationen (Rot, Grün, Blau) interpretiert. Wäre noch eine vierte Ebene dabei, würde sie als Transparenzwert verwendet worden.

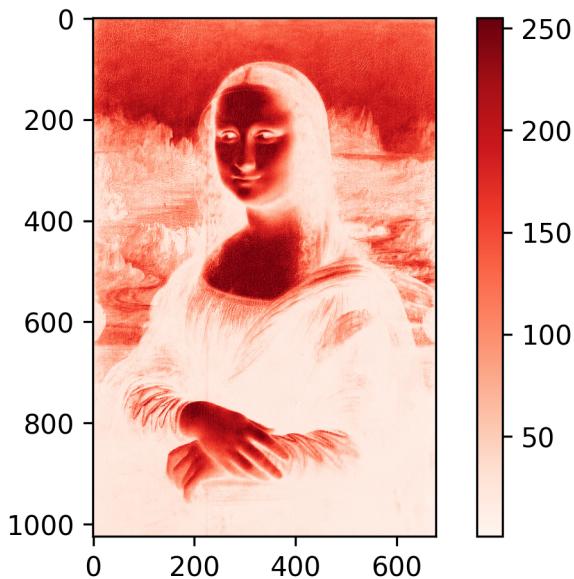
```
plt.imshow(data)
```



Natürlich können auch die einzelnen Farbebenen individuell betrachtet werden. Dazu wird der letzte Index festgehalten. Hier betrachten wir nur den roten Anteil des Bildes. Stellen wir ein einfaches Array dar, werden die Daten mit der Standardfarbpalette ‘viridis’ ausgegeben. Mit Hilfe der Option `cmap='Reds'` können wir die Farbskala anpassen.

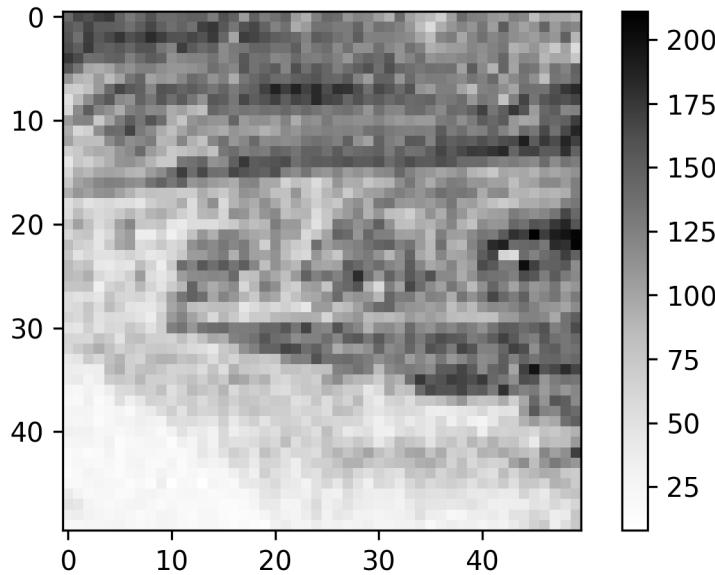
```
# Als Farbskala wird die Rotskala  
# verwendet 'Reds'
```

```
plt.imshow( data[:, :, 0], cmap='Reds' )
plt.colorbar()
plt.show()
```



Da die Bilddaten als Arrays gespeichert sind, sind viele der möglichen Optionen, z. B. zur Teilauswahl oder Operationen, verfügbar. Das untere Beispiel zeigt einen Ausschnitt im Rotkanal des Bildes.

```
bereich = np.array(data[450:500, 550:600, 0], dtype=float)
plt.imshow( bereich, cmap="Greys" )
plt.colorbar()
```



Betrachten wir nun eine komplexere Operation an Bilddaten, den [Laplace-Operator](#). Er kann genutzt werden, um Ränder von Objekten zu identifizieren. Dazu wird für jeden Bildpunkt $B_{i,j}$ – außer an den Rändern – folgender Wert $\phi_{i,j}$ berechnet:

$$\phi_{i,j} = |B_{i-1,j} + B_{i,j-1} - 4 \cdot B_{i,j} + B_{i+1,j} + B_{i,j+1}|$$

Folgende Funktion implementiert diese Operation. Darüber hinaus werden alle Werte von ϕ unterhalb eines Schwellwerts auf Null und oberhalb auf 255 gesetzt.

```
def img_lap(data, schwellwert=25):

    # Erstellung einer Kopie der Daten, nun jedoch als
    # Array mit Gleitkommazahlen
    bereich = np.array(data, dtype=float)

    # Aufteilung der obigen Gleichung in zwei Teile
    lapx = bereich[2:, :] - 2*bereich[1:-1, :] + bereich[:-2, :]
    lapy = bereich[:, 2:] - 2*bereich[:, 1:-1] + bereich[:, :-2]

    # Zusammenführung der Teile und Bildung des Betrags
    lap = np.abs(lapx[:, 1:-1] + lapy[1:-1, :])

    # Schwellwertanalyse
    lap[lap > schwellwert] = 255
```

```

lap[lap < schwellwert] = 0

return lap

```

Betrachten wir ein Bild vom Haspel Campus in Wuppertal. Die Anwendung des Laplace-Operators auf den oberen Bildausschnitt ergibt folgende Ausgabe:

```

data = plt.imread('01-daten/campus_haspel.jpeg')
bereich = np.array(data[1320:1620, 400:700, 1], dtype=float)

lap = img_lap(bereich)

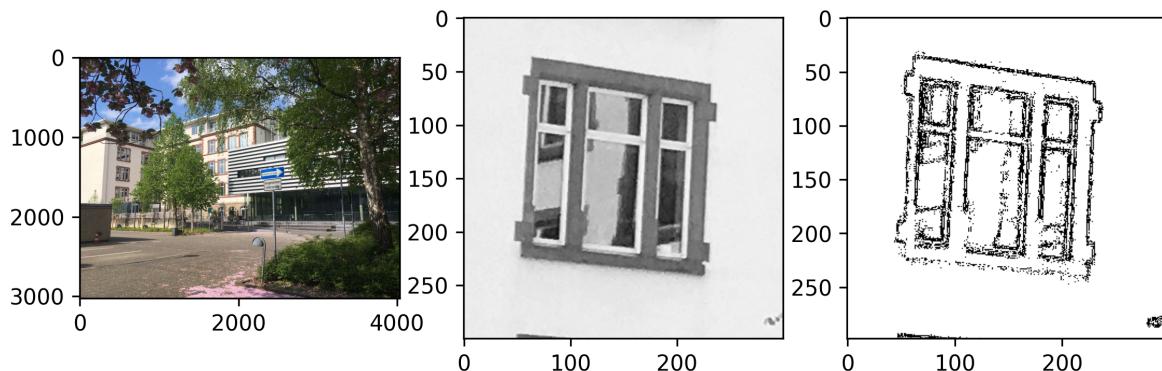
plt.figure(figsize=(9, 3))

ax = plt.subplot(1, 3, 1)
ax.imshow(data, cmap="Greys_r")

ax = plt.subplot(1, 3, 2)
ax.imshow(bereich, cmap="Greys_r");

ax = plt.subplot(1, 3, 3)
ax.imshow(lap, cmap="Greys");

```



Wir können damit ganz klar die Formen des Fensters erkennen.

Wollen wir zum Beispiel eine Farbkomponente bearbeiten und dann das Bild wieder zusammensetzen, benötigen wir die Funktion `np.dstack((rot, grün, blau)).astype('uint8')`, wobei `rot`, `grün` und `blau` die jeweiligen 2D-Arrays sind. Versuchen wir nun, die grüne Farbe aus dem Baum links zu entfernen.

Wichtig ist, dass die Daten nach dem Zusammensetzen im Format `uint8` vorliegen, deswegen die Methode `.astype('uint8')`.

```
data = plt.imread('01-daten/campus_haspel.jpeg')

# Speichern der einzelnen Farben in Arrays
rot = np.array(data[:, :, 0], dtype=float)
gruen = np.array(data[:, :, 1], dtype=float)
blau = np.array(data[:, :, 2], dtype=float)

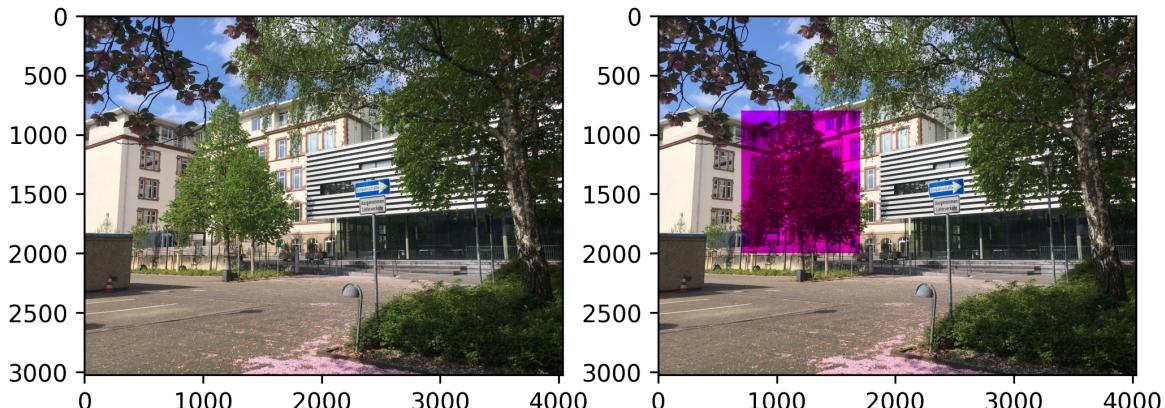
# Setzen wir den Bereich des linken Baumes im Array auf 0
gruen_neu = gruen.copy()
gruen_neu[800:2000, 700:1700] = 0

zusammengesetzt = np.dstack((rot, gruen_neu, blau)).astype('uint8')

plt.figure(figsize=(8, 5))

ax = plt.subplot(1, 2, 1)
ax.imshow(data, cmap="Greys_r")

ax = plt.subplot(1, 2, 2)
ax.imshow(zusammengesetzt)
```



💡 Zwischenübung: Bilder bearbeiten

Lesen Sie folgendes Bild vom Haspel Campus in Wuppertal ein: [Bild](#).
Extrahieren Sie den blauen Anteil und lassen Sie sich die Zeile in der Mitte des Bildes sowie einen beliebigen Bildausschnitt ausgeben.

Lösung

```
import numpy as np
import matplotlib.pyplot as plt

data = plt.imread('01-daten/campus_haspel.jpeg')

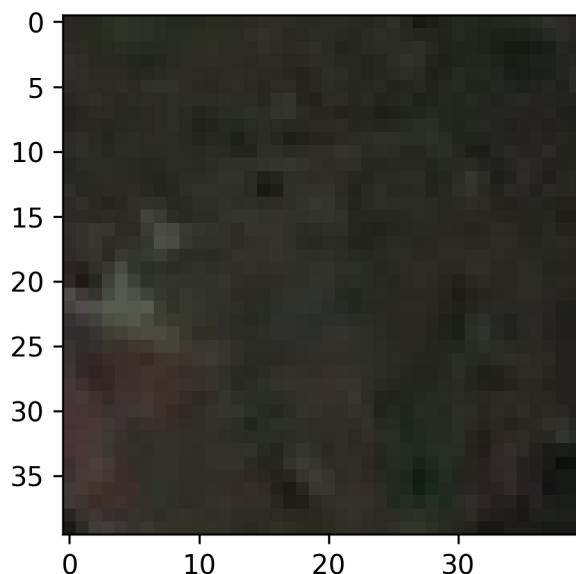
form = data.shape
print("Form:", data.shape)

blau = data[:, :, 2]
plt.imshow(blau, cmap='Blues')

zeile = data[int(form[0]/2), :, 2]
print(zeile)

ausschnitt = data[10:50, 10:50, :]
plt.imshow(ausschnitt)
```

Form: (3024, 4032, 3)
[221 220 220 ... 28 28 28]



9 Lernzielkontrolle

Herzlich willkommen zur Lernzielkontrolle!

Diese Selbstlernkontrolle dient dazu, Ihr Verständnis der bisher behandelten Themen zu überprüfen und Ihnen die Möglichkeit zu geben, Ihren Lernfortschritt eigenständig zu bewerten. Sie ist so konzipiert, dass Sie Ihre Stärken und Schwächen erkennen und gezielt an den Bereichen arbeiten können, die noch verbessert werden müssen.

Es stehen hier zwei Möglichkeiten zur Verfügung, Ihr Wissen zu prüfen. Sie können das Quiz benutzen, welches Sie automatisch durch die verschiedenen Themen führt. Alternativ finden Sie darunter normale Fragen, wie Sie bisher im Skript verwendet wurden.

Bitte nehmen Sie sich ausreichend Zeit für die Bearbeitung der Fragen und gehen Sie diese in Ruhe durch. Seien Sie ehrlich zu sich selbst und versuchen Sie, die Aufgaben ohne Hilfsmittel zu lösen, um ein realistisches Bild Ihres aktuellen Wissensstands zu erhalten. Sollten Sie bei einer Frage Schwierigkeiten haben, ist dies ein Hinweis darauf, dass Sie in diesem Bereich noch weiter üben sollten.

Viel Erfolg bei der Bearbeitung und beim weiteren Lernen!

Aufgabe 1

Wie wird das NumPy-Paket typischerweise eingebunden?

Aufgabe 2

Erstellen Sie mit Hilfe von NumPy die folgenden Arrays:

1. Erstellen sie aus der Liste [1, 2, 3] ein NumPy-Array.
2. Ein eindimensionales Array, das die Zahlen von 0 bis 9 enthält.
3. Ein zweidimensionales Array der Form 3×3 , das nur aus Einsen besteht.
4. Ein eindimensionales Array, das die Zahlen von 10 bis 50 (einschließlich) in Schritten von 5 enthält.

Aufgabe 3

Was ist der Unterschied zwischenden den Funktionen `np.ndim()`, `np.shape()` und `np.size()`?

Aufgabe 4

Welchen Datentyp hat folgendes Array? Mit welcher Funktion kann man den Datentyp eines Arrays auslesen?

```
vector = np.array([ 4.8,  8.2, 15.6, 16.6, 23.2, 42.8 ])
```

Aufgabe 5

Führen Sie mit den folgenden zwei Arrays diese mathematischen Operationen durch:

`a = np.array([5, 1, 3, 6, 4])` und `b = np.array([6, 5, 2, 6, 9])`

1. Addieren Sie beide Arrays.
2. Berechnen Sie das elementweise Produkt von a und b.
3. Addieren Sie zu jedem Eintrag von a 3 dazu.

Aufgabe 6

`a = np.array([9, 2, 3, 1, 3])`

1. Bestimmen Sie Mittelwert und Standardabweichung für das Array a.
2. Bestimmen Sie Minimum und Maximum des Arrays.

Aufgabe 7

```
matrix = np.array([
    [ 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]
```

```
])
```

1. Extrahieren Sie die erste Zeile.
2. Extrahieren Sie die letzte Spalte.
3. Extrahieren Sie die Untermatrix, die aus den Zeilen 2 bis 4 und den Spalten 1 bis 3 besteht.

Aufgabe 8

```
array = np.arange(1, 21)
```

1. Ändern Sie die Form des Arrays in eine zweidimensionale Matrix der Form 4×5 .
2. Ändern Sie die Form des Arrays in eine zweidimensionale Matrix der Form 5×4 .
3. Ändern Sie die Form des Arrays in eine dreidimensionale Matrix der Form $2 \times 2 \times 5$.
4. Flachen Sie das dreidimensionale Array aus Aufgabe 3 wieder zu einem eindimensionalen Array ab.
5. Transponieren Sie die 4×5 -Matrix aus Aufgabe 1.

Aufgabe 9

Mit welchen zwei Funktionen können Daten aus einer Datei gelesen und in einer Datei gespeichert werden?

Aufgabe 10

Sie möchten aus einem Bild die Bilddaten einer Farbkomponente isolieren. Was müssen Sie dafür tun?

Lösung

Aufgabe 1

```
import numpy as np
```

Aufgabe 2

```
# 1.  
np.array([1, 2, 3])  
  
# 2.  
print(np.arange(10))  
  
# 3.  
print(np.ones((3, 3)))  
  
# 4.  
print(np.arange(10, 51, 5))  
  
[0 1 2 3 4 5 6 7 8 9]  
[[1. 1. 1.]  
 [1. 1. 1.]  
 [1. 1. 1.]]  
[10 15 20 25 30 35 40 45 50]
```

Aufgabe 3

`np.ndim()`: Gibt die Anzahl der Dimensionen zurück. `np.shape()`: Gibt die Längen der einzelnen Dimensionen wieder. `np.size()`: Gibt die Anzahl aller Elemente aus.

Aufgabe 4

Da es sich hier um Gleitkommazahlen handelt, ist der Datentyp `float64`.

```
vector = np.array([ 4.8,  8.2, 15.6, 16.6, 23.2, 42.8 ])  
print(vector.dtype)  
  
float64
```

Aufgabe 5

```

a = np.array([5, 1, 3, 6, 4])
b = np.array([6, 5, 2, 6, 9])

# 1.
ergebnis = a + b
print("Die Summe beider Vektoren ergibt: ", ergebnis)

# 2.
ergebnis = a * b
print("Das Produkt beider Vektoren ergibt: ", ergebnis)

# 3.
ergebnis = a + 3
print("Die Summe von a und 3 ergibt: ", ergebnis)

```

Die Summe beider Vektoren ergibt: [11 6 5 12 13]
 Das Produkt beider Vektoren ergibt: [30 5 6 36 36]
 Die Summe von a und 3 ergibt: [8 4 6 9 7]

Aufgabe 6

```

a = np.array([9, 2, 3, 1, 3])

# 1.
mittelwert = np.mean(a)
print("Der Mittelwert ist: ", mittelwert)

standardabweichung = np.std(a)
print("Die Standardabweichung von a beträgt: ", standardabweichung)

# 2.
minimum = np.min(a)
print("Das Minimum beträgt: ", minimum)

maximum = np.max(a)
print("Das Maximum beträgt: ", maximum)

```

Der Mittelwert ist: 3.6
 Die Standardabweichung von a beträgt: 2.8000000000000003
 Das Minimum beträgt: 1
 Das Maximum beträgt: 9

Aufgabe 7

```
matrix = np.array([
    [ 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]
])

# 1. Erste Zeile
print(matrix[0,:])

# 2. Letzte Spalte
print(matrix[:, -1])

# 3. Ausschnitt
print(matrix[1:4, 0:3])
```

```
[1 2 3 4 5]
[ 5 10 15 20 25]
[[ 6  7  8]
 [11 12 13]
 [16 17 18]]
```

Aufgabe 8

```

array = np.arange(1, 21)

# 1. Ändern der Form in eine 4x5-Matrix
matrix_4x5 = array.reshape(4, 5)

# 2. Ändern der Form in eine 5x4-Matrix
matrix_5x4 = array.reshape(5, 4)

# 3. Ändern der Form in eine 2x2x5-Matrix
matrix_2x2x5 = array.reshape(2, 2, 5)

# 4. Abflachen der 2x2x5-Matrix zu einem eindimensionalen Array
flattened_array = matrix_2x2x5.flatten()

# 5. Transponieren der 4x5-Matrix
transposed_matrix = matrix_4x5.T

# Ausgabe der Ergebnisse (optional)
print("Originales Array:", array)
print("4x5-Matrix:\n", matrix_4x5)
print("5x4-Matrix:\n", matrix_5x4)
print("2x2x5-Matrix:\n", matrix_2x2x5)
print("Abgeflachtes Array:", flattened_array)
print("Transponierte 4x5-Matrix:\n", transposed_matrix)

```

```

Originales Array: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
4x5-Matrix:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
5x4-Matrix:
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]]
2x2x5-Matrix:
[[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
```

```
[[11 12 13 14 15]
 [16 17 18 19 20]]
```

Abgeflachtes Array: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]

Transponierte 4x5-Matrix:

```
[[ 1 6 11 16]
 [ 2 7 12 17]
 [ 3 8 13 18]
 [ 4 9 14 19]
 [ 5 10 15 20]]
```

Aufgabe 9

Die passenden Funktionen sind `np.loadtxt()` und `np.savetxt()`.

Aufgabe 10

Typischerweise sind Bilddaten große Matrizen, wobei die Farben in drei unterschiedlichen Matrizen gespeichert werden. Dabei ist die Farbreihenfolge oft “Rot”, “Grün” und “Blau”. Dementsprechen müssen wir, wenn die Daten in der Matrix `data` gespeichert sind, mit Slicing eine Dimension auswählen: `data[:, :, 0]`, wobei die Zahl 0-2 für die jeweilige Farbe steht.

10 Übung

10.1 Aufgabe 1 Filmdatenbank

In der ersten Aufgabe wollen wir fiktive Daten für Filmbewertungen untersuchen. Das Datenset ist dabei vereinfacht und beinhaltet folgende Spalten:

1. Film ID
2. Benutzer ID
3. Bewertung

Hier ist das Datenset:

```
import numpy as np

bewertungen = np.array([
    [1, 101, 4.5],
    [1, 102, 3.0],
    [2, 101, 2.5],
    [2, 103, 4.0],
    [3, 101, 5.0],
    [3, 104, 3.5],
    [3, 105, 4.0]
])
```

💡 a) Bestimmen Sie die niedrigste und höchste Bewertung, die jemals gegeben wurde

Lösung

```
niedrigste_bewertung = np.min(bewertungen[:,2])  
  
print("Die niedrigste jemals gegebene Bewertung ist:", niedrigste_bewertung)  
  
hoechste_bewertung = np.max(bewertungen[:,2])  
  
print("Die höchste jemals gegebene Bewertung ist:", hoechste_bewertung)  
  
Die niedrigste jemals gegebene Bewertung ist: 2.5  
Die höchste jemals gegebene Bewertung ist: 5.0
```

💡 b) Nennen Sie alle Bewertungen für Film 1

Lösung

```
bewertungen_film_1 = bewertungen[np.where(bewertungen[:,0]==1)]  
  
print("Bewertungen für Film 1:\n", bewertungen_film_1)  
  
Bewertungen für Film 1:  
[[ 1. 101. 4.5]  
 [ 1. 102. 3. ]]
```

💡 c) Nennen Sie alle Bewertungen von Person 101

Lösung

```
bewertungen_101 = bewertungen[np.where(bewertungen[:,1]==101)]  
  
print("Bewertungen von Person 101:\n", bewertungen_101)
```

Bewertungen von Person 101:

```
[[ 1. 101. 4.5]  
 [ 2. 101. 2.5]  
 [ 3. 101. 5. ]]
```

💡 d) Berechnen Sie die mittlere Bewertung für jeden Film und geben Sie diese nacheinander aus

Lösung

```
for ID in [1, 2, 3]:  
  
    mittelwert = np.mean(bewertungen[np.where(bewertungen[:,0]==ID),2])  
  
    print("Die mittlere Bewertung für Film", ID, "beträgt:", mittelwert)
```

Die mittlere Bewertung für Film 1 beträgt: 3.75

Die mittlere Bewertung für Film 2 beträgt: 3.25

Die mittlere Bewertung für Film 3 beträgt: 4.1666666666666667

💡 e) Finden Sie den Film mit der höchsten Bewertung

Lösung

```
index_hoechste_bewertung = np.argmax(bewertungen[:,2])  
  
print(bewertungen[index_hoechste_bewertung,:])  
  
[ 3. 101. 5.]
```

💡 f) Finden Sie die Person mit den meisten Bewertungen

Lösung

```
einzigartige_person, anzahl = np.unique(bewertungen[:, 1], return_counts=True)  
meist_aktive_person = einzigartige_person[np.argmax(anzahl)]  
print("Personen mit den meisten Bewertungen:", meist_aktive_person)
```

Personen mit den meisten Bewertungen: 101.0

💡 g) Nennen Sie alle Filme mit einer Wertung von 4 oder besser.

Lösung

```
index_bewertung_besser_vier = bewertungen[:, 2] >= 4  
print("Filme mit einer Wertung von 4 oder besser:")  
print(bewertungen[index_bewertung_besser_vier, :])
```

Filme mit einer Wertung von 4 oder besser:

```
[[ 1.  101.    4.5]  
 [ 2.  103.    4. ]  
 [ 3.  101.    5. ]  
 [ 3.  105.    4. ]]
```

💡 h) Film Nr. 4 ist erschienen. Der Film wurde von Person 102 mit einer Note von 3.5 bewertet. Fügen Sie diesen zur Datenbank hinzu.

Lösung

```
neue_bewertung = np.array([4, 102, 3.5])

bewertungen = np.append(bewertungen, [neue_bewertung], axis=0)

print(bewertungen)

[[ 1. 101. 4.5]
 [ 1. 102. 3. ]
 [ 2. 101. 2.5]
 [ 2. 103. 4. ]
 [ 3. 101. 5. ]
 [ 3. 104. 3.5]
 [ 3. 105. 4. ]
 [ 4. 102. 3.5]]
```

💡 i) Person 102 hat sich Film Nr. 1 nochmal angesehen und hat das Ende jetzt doch verstanden. Dementsprechend soll die Bewertung jetzt auf 5.0 geändert werden.

Lösung

```
bewertungen[(bewertungen[:, 0] == 1) &
              (bewertungen[:, 1] == 102), 2] = 5.0

print("Aktualisieren der Bewertung:\n", bewertungen)
```

Aktualisieren der Bewertung:

```
[[ 1. 101. 4.5]
 [ 1. 102. 5. ]
 [ 2. 101. 2.5]
 [ 2. 103. 4. ]
 [ 3. 101. 5. ]
 [ 3. 104. 3.5]
 [ 3. 105. 4. ]
 [ 4. 102. 3.5]]
```

10.2 Aufgabe 2 - Kryptographie - Caesar-Chiffre

In dieser Aufgabe wollen wir Text sowohl ver- als auch entschlüsseln.

Jedes Zeichen hat über die sogenannte ASCII-Tabelle einen Zahlenwert zugeordnet.

Tabelle 10.1: Ascii-Tabelle

Buchstabe	ASCII Code	Buchstabe	ASCII Code
a	97	n	110
b	98	o	111
c	99	p	112
d	100	q	113
e	101	r	114
f	102	s	115
g	103	t	116
h	104	u	117
i	105	v	118
j	106	w	119
k	107	x	120
l	108	y	121
m	109	z	122

Der Einfachheit halber ist im Folgenden schon der Code zur Umwandlung von Buchstaben in Zahlenwerte und wieder zurück aufgeführt. Außerdem beschränken wir uns auf Texte mit kleinen Buchstaben.

Ihre Aufgabe ist nun, die Zahlenwerte zu verändern.

Zunächste wollen wir eine einfache Caesar-Chiffre anwenden. Dabei werden alle Buchstaben um eine gewisse Anzahl verschoben. Ist Beispielsweise der der Verschlüsselungswert “1” wird aus einem A ein B, einem M, ein N. Ist der Wert “4” wird aus einem A ein E und aus einem M ein Q. Die Verschiebung findet zyklisch statt, das heißt, bei einer Verschiebung von 1 wird aus einem Z ein A.

```
import numpy as np

# Funktion, die einen Buchstaben in ihren ASCII-Wert umwandelt
def buchstabe_zu_ascii(c):
    return np.array([ord(c)])

# Funktion, die einen ASCII-Wert in den passenden Buchstaben umwandelt
```

```
def ascii_zu_buchstabe(a):
    return chr(a)
```

- 💡 1. Überlegen Sie sich zunächst, wie man diese zyklische Verschiebung mathematisch ausdrücken könnte (Hinweis: Modulo Rechnung)

Lösung

$$\text{ASCII}_{\text{verschoben}} = (\text{ASCII} - 97 + \text{Versatz}) \bmod 26 + 97$$

- 💡 2. Schreiben Sie Code, der mit einer Schleife alle Zeichen umwandelt.

Zunächst sollen alle Zeichen in ASCII-Code umgewandelt werden. Dann wird die Formel auf die Zahlenwerte angewendet und schlussendlich in einer dritten Schleife wieder alle Werte in Buchstaben übersetzt.

Lösung

```
import numpy as np

# Funktion, die einen Buchstaben in ihren ASCII-Wert umwandelt
def buchstabe_zu_ascii(c):
    return ord(c)

# Funktion, die einen ASCII-Wert in den passenden Buchstaben umwandelt
def ascii_zu_buchstabe(a):
    return chr(a)

klartext = "abradabra"
versatz = 3

umgewandelter_text = []
verschluesselte_zahl = []
verschluesselter_text= []

for buchstabe in klartext:
    umgewandelter_text.append(buchstabe_zu_ascii(buchstabe))
print(umgewandelter_text)

for zahl in umgewandelter_text:
    verschluesselt = (zahl - 97 + versatz) % 26 + 97
    verschluesselte_zahl.append(verschluesselt)
print(verschluesselte_zahl)

for zahl in verschluesselte_zahl:
    verschluesselter_text.append(ascii_zu_buchstabe(zahl))
print(verschluesselter_text)

[97, 98, 114, 97, 107, 97, 100, 97, 98, 114, 97]
[100, 101, 117, 100, 110, 100, 103, 100, 101, 117, 100]
['d', 'e', 'u', 'd', 'n', 'd', 'g', 'd', 'e', 'u', 'd']
```

- 💡 3. Ersetzen Sie die Schleife, indem Sie die Rechenoperation mit einem NumPy-Array durchführen

Lösung

```
import numpy as np

# Funktion, die einen Buchstaben in ihren ASCII-Wert umwandelt
def buchstabe_zu_ascii(c):
    return ord(c)

# Funktion, die einen ASCII-Wert in den passenden Buchstaben umwandelt
def ascii_zu_buchstabe(a):
    return chr(a)

klartext = "abradabra"
versatz = 3

umgewandelter_text = []
verschluesselte_zahl = []
verschluesselter_text= []

for buchstabe in klartext:
    umgewandelter_text.append(buchstabe_zu_ascii(buchstabe))
print(umgewandelter_text)

umgewandelter_text = np.array(umgewandelter_text)
verschluesselte_zahl = (umgewandelter_text - 97 + versatz) % 26 + 97
print(verschluesselte_zahl)

for zahl in verschluesselte_zahl:
    verschluesselter_text.append(ascii_zu_buchstabe(zahl))
print(verschluesselter_text)

[97, 98, 114, 97, 107, 97, 100, 97, 98, 114, 97]
[100 101 117 100 110 100 103 100 101 117 100]
['d', 'e', 'u', 'd', 'n', 'd', 'g', 'd', 'e', 'u', 'd']
```

💡 4. Schreiben sie den Code so um, dass der verschlüsselte Text entschlüsselt wird.

Lösung

```
import numpy as np

# Funktion, die einen Buchstaben in ihren ASCII-Wert umwandelt
def buchstabe_zu_ascii(c):
    return ord(c)

# Funktion, die einen ASCII-Wert in den passenden Buchstaben umwandelt
def ascii_zu_buchstabe(a):
    return chr(a)

versatz = 3

umgewandelter_text = []
verschluesselte_zahl = []
entschluesselter_text= []

for buchstabe in verschluesselter_text:
    umgewandelter_text.append(buchstabe_zu_ascii(buchstabe))
print(umgewandelter_text)

umgewandelter_text = np.array(umgewandelter_text)
verschluesselte_zahl = (umgewandelter_text - 97 - versatz) % 26 + 97
print(verschluesselte_zahl)

for zahl in verschluesselte_zahl:
    entschluesselter_text.append(ascii_zu_buchstabe(zahl))
print(entschluesselter_text)

[100, 101, 117, 100, 110, 100, 103, 100, 101, 117, 100]
[ 97  98 114  97 107  97 100  97  98 114  97]
['a', 'b', 'r', 'a', 'k', 'a', 'd', 'a', 'b', 'r', 'a']
```

11 Klausurfragen

Aufgabe 1

Ein rechteckiger Träger aus Beton wird entlang seiner Länge mit einer gleichmäßig verteilten Last belastet. Die Spannungsverteilung entlang der Länge des Trägers soll analysiert werden. Der Träger hat eine Länge von 10 Metern und eine Breite von 0,3 Metern. Die Höhe des Trägers beträgt 0,5 Meter. Die gleichmäßig verteilte Last beträgt 5000 N/m.

1. Erstellen Sie ein NumPy-Array x mit 100 gleichmäßig verteilten Punkten entlang der Länge des Trägers von 0 bis 10 Metern.
2. Berechnen Sie die Biegemomente $M(x)$ entlang der Länge des Trägers unter Verwendung der Formel:

$$\left[M(x) = \frac{w \cdot x \cdot (L - x)}{2} \right]$$

wobei w die verteilte Last (in N/m), x die Position entlang des Trägers (in m) und L die Länge des Trägers (in m) ist.

3. Berechnen Sie die maximale Biegespannung σ_{max} an jedem Punkt entlang des Trägers unter Verwendung der Formel:

$$\left[\sigma_{max}(x) = \frac{M(x) \cdot c}{I} \right]$$

wobei c der Abstand von der neutralen Faser zur äußersten Faser des Trägers (in m) und I das Flächenträgheitsmoment ist. Das Flächenträgheitsmoment eines rechteckigen Querschnitts ist:

$$\left[I = \frac{b \cdot h^3}{12} \right]$$

wobei b die Breite (in m) und h die Höhe des Trägers (in m) ist.

4. Bestimmen Sie die maximale Biegespannung
5. Plotten Sie die Spannungsverteilung $\sigma_{max}(x)$ entlang der Länge des Trägers.