w-python-numpy-grundlagen

Lukas Arnold Simone Arnold Florian Bagemihl
Matthias Baitsch Marc Fehr Maik Poetzsch
Sebastian Seipel

2024-06-18

Inhaltsverzeichnis

Preamble

Intro

Voraussetzungen

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

Verwendete Pakete und Datensätze

- NumPy
- Matplotlib

Bearbeitungszeit

Geschätzte Bearbeitungszeit: 42 Stunden

Lernziele

- Einleitung: was ist NumPy, Vor- und Nachteile
- Nutzen des NumPy-Moduls
- Erstellen von NumPy-Arrays
- Slicing
- Referenzen
- Lesen und schreiben von Dateien
- Bilder

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.

Warum ist numpy oftmals schneller?

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

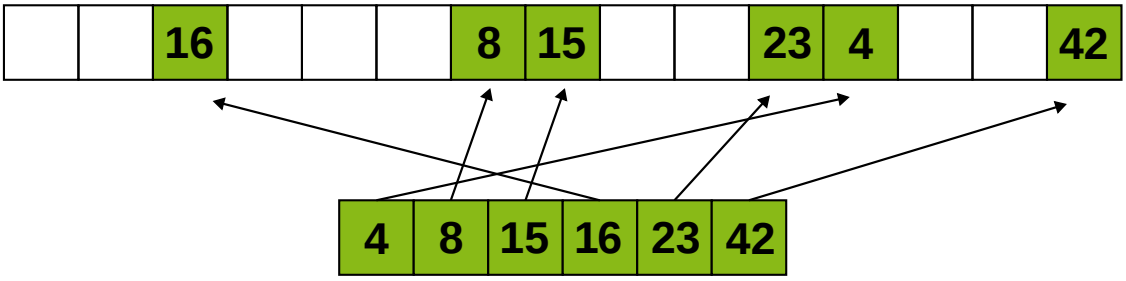


Abbildung 1.1: Speicherung von Daten in nativem Python

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

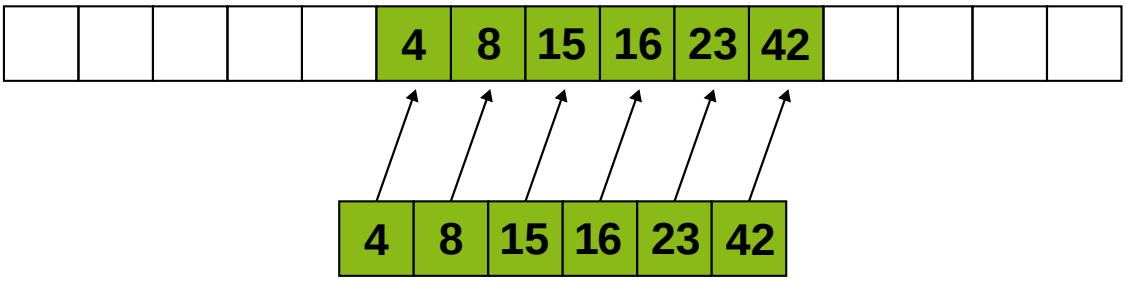


Abbildung 1.2: Speicherung von Daten bei Numpy

Dies bedeutet aber auch, dass es eine Erweiterung der 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

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

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 drei-dimensionale 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 auch Arrays die mit einer Zahl x gefühlt sind erstellen?

Der Trick beseht hierbei ein Array mit `np.ones()` zu initialisiere 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,Stop,#Anzahl Werte)` und `np.arrange(Start,Stop,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 einschlieslich 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 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 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 einzelnen Dimension in Form einer Liste 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.

```
len(matrix)
```

```
2
```

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

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

```
2
```

> Die Ausgabe von `np.ndim()` kann mit `np.size()`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 beinhalten. Im folgenden haben wir drei verschiedene Arrays mit einem jeweils anderen Datentypen.

```
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 Datentypen 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 Datentypen int64, also ganze Zahlen.

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

```
int64
```

Das Array *typ_b* beinhaltet den Datentypen float64, wobei float für Gleitkommazahlen steht.

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

```
float64
```

Das Array *typ_c* beinhaltet den Datentypen 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 Zahlen	`uint`	[1, 2, 3, 4, 5]
Kommazahlen	`float`	[1.3, 7.4, 3.5, .5.5]
komplexe zahlen	`complex`	[-1 + 9j, 2-77j, 72 + 11j]
Textzeichen	`U`	[“montag”, “dienstag”]

4 Rechnen mit Arrays

4.1 Arithmetische Funktionen

Ein großer Vorteil an 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 das `+` Zeichen.

```
np.add(a,b)
```

```
array([10, 10, 10, 10, 10])
```

```
a + b
```

```
array([10, 10, 10, 10, 10])
```

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

Auch für die anderen höheren Rechenoperationen gibt es ebenfalls 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.grad2rad()`und `np.rad2grad()`.

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
```

```
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 Summer wird über die Funktion `np.sum()` berechnet.

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

```
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)
```

```
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)
```

```
3.6
```

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

```
3.0
```

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

```
2.4166091947189146
```

5 Slicing

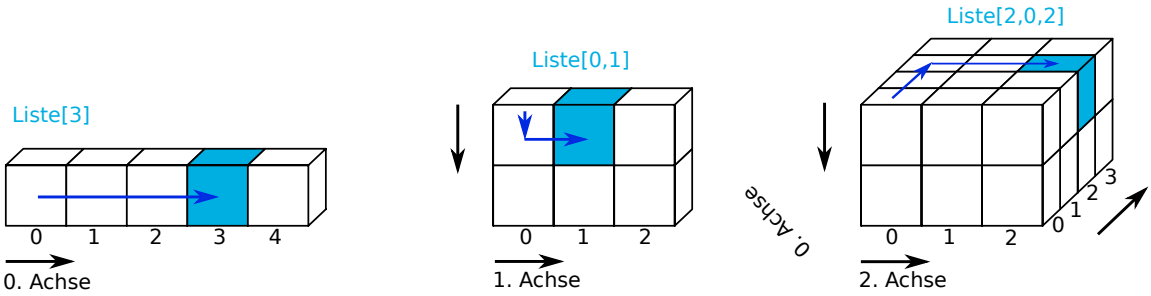


Abbildung 5.1: Ansprechen der einzelnen Achsen für den ein-, zwei- und dreidimensionallen Fall inkl. jeweiligem Beispiel

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

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

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

```
1
```

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

```
6
```

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

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

Für zwei-dimensionale Arrays wählt man getrennt durch ein Komma mit einer zweiten Zahl die zweite Dimension aus.

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

```
# Auswählen einer Elements
matrix[1,1]
```

```
5
```

Für drei-dimensionale Arrays wählt man getrennt durch ein Komma mit einer weiteren Zahl die dritte Dimension aus. Dabei wird dieses jedoch an die erste Stelle gesetzt.

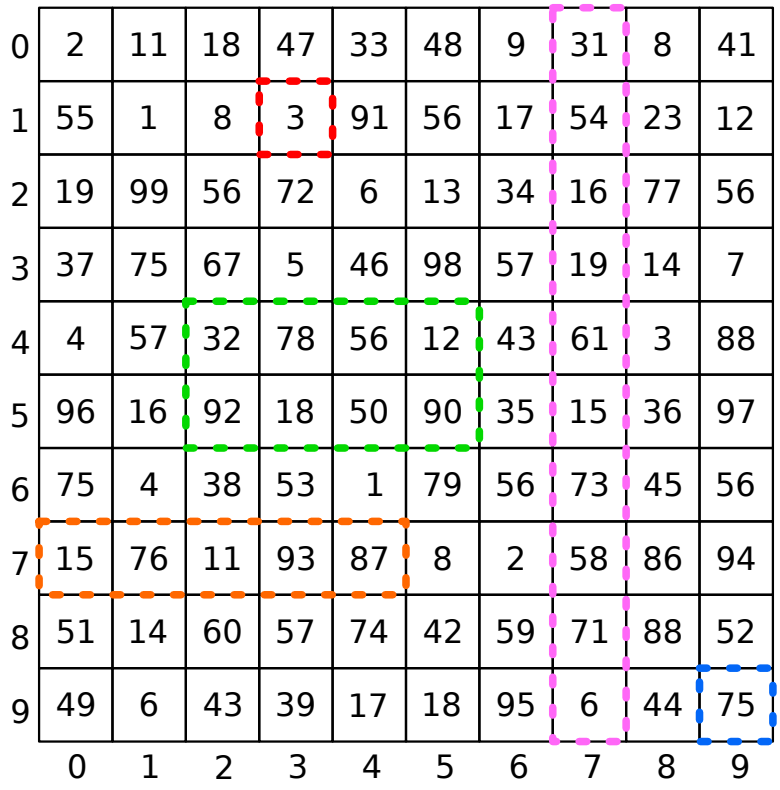
```
matrix_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(matrix_3d)
```

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

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

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

```
9
```

> Zwischenübung: Array-Slicing
>
> Wählen Sie die farblich markierten Bereiche aus dem Array “matrix” mit den eben gelernten Möglichkeiten des Array-Slicing aus.



Lösung

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

Haben wir ein nun diese Matrix und wollen daraus einen Vektor erstellen so können wir die Funktion `np.ravel()` benutzen:

```
vector = np.ravel(matrix)
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 hier die Funktion `np.insert(Array, Position, Einschub)`. Im folgenden Beispiel wird an der dritten Stelle 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 löschenden 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 Projekte 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 die jede Nummer nur einmal enthält, kann die Funtkion `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]
```

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 `np.loadtxt()`-Funktion verwendet werden. Diese bekommt als Argumente den einzulesenden Dateinamen und weitere Optionen zur Definition der Struktur der Daten. Der Rückgabewert ist ein (mehrdimensionales) Array.

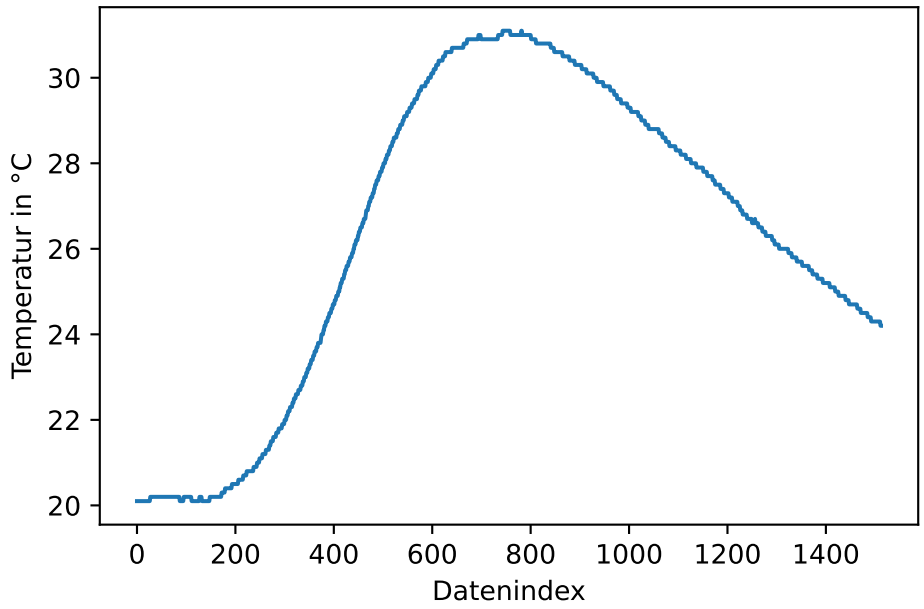
Im folgenden Beispiel wird die Datei TC01.csv eingelesen und deren Inhalt graphisch dargestellt. Die erste Zeile der Datei wird dabei ignoriert, da sie als Kommentar – eingeleitet durch das #-Zeichen – interpretiert wird.

```
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');
```



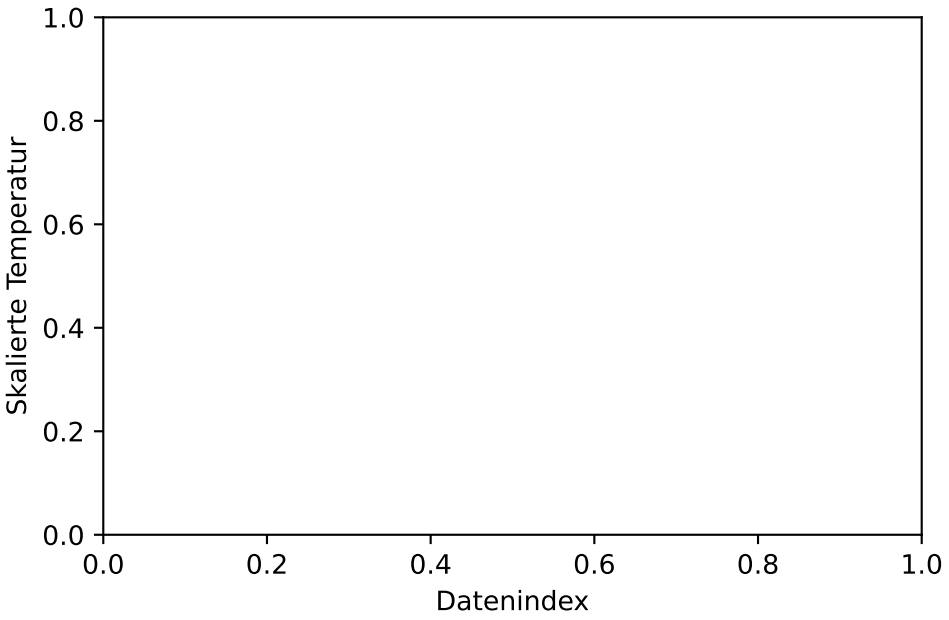
7.2 Schreiben von Dateien

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 schreib 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.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 Beriech' + \
             '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 Beriech0 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
```

8 Arbeiten mit Bildern

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

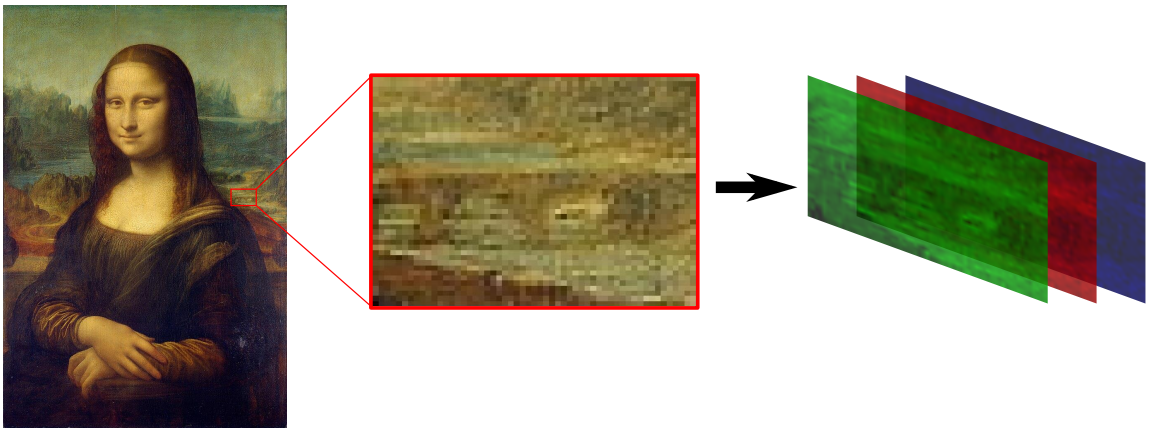


Abbildung 8.1: Ein hochaufgelöstes Bild besteht aus sehr vielen Pixeln. Jedes Pixel enthät 3 Farbwerte, einen für die Fabre Grün, einen für Blau und einen für Rot.

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

Importieren wir dieses Bild nun mit der Funktion `imread()`aus dem matplotlib-package, sehen wir das es 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 einen Ausschnitt dieser Daten an.

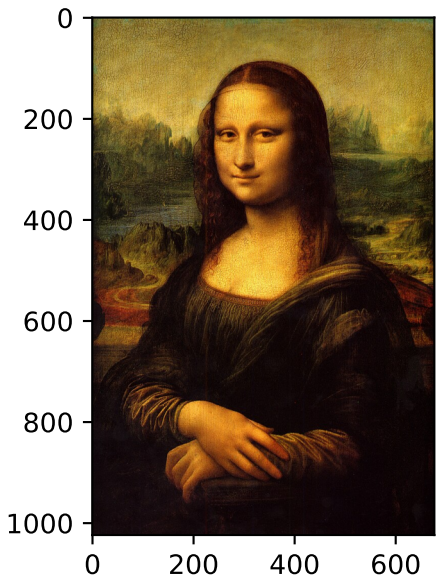
```
print(data)
```

```
[[[ 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]]]
```

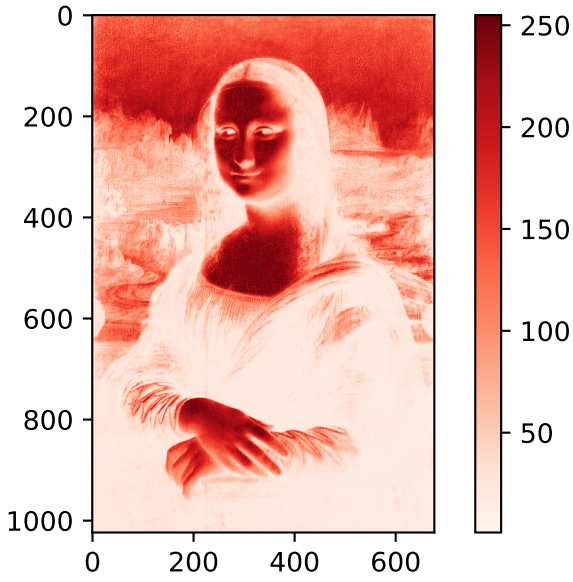
Mit der Funktion `plt.imshow` kann das Bild in Echtfarben dargestellt werden. Dies funktioniert, da die Funktion die einzelnen Ebenen, hier der letzte Index, des Datensatzes als Farbinformationen (rot, grün, blau) interpretiert. Wäre noch eine vierte Ebene dabei, würde sie als individueller Transparenzwert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

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



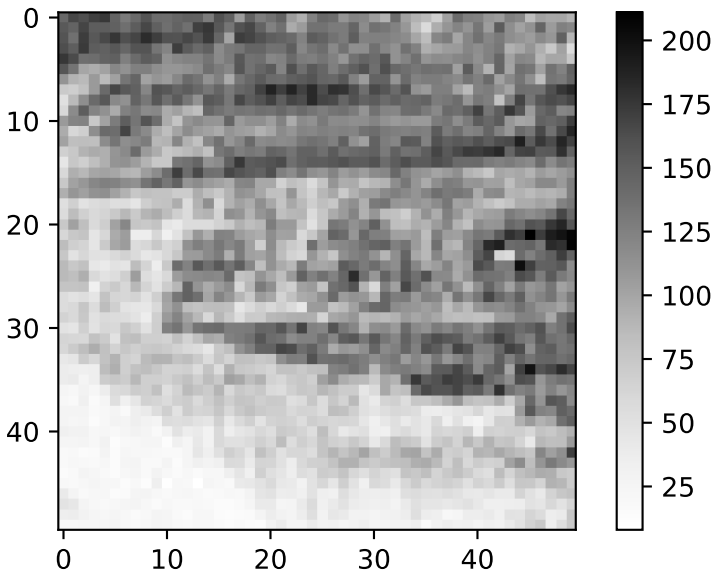
Natürlich können auch die einzelnen Farbebenen individuell betrachtet werden. Dazu wird der letzte Index festgehalten.

```
# Als Farbskale 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()
```



9 Übung

9.1 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 jemals niedrigste und höchste Bewertung, die je gegeben wurde

Lösung

```
niedrigste_bewertung = np.min(bewertungen[:,2])

print("Die niedrigste jemals gegebene Bertung ist:", niedrigste_bewertung)

hoechste_bewertung = np.max(bewertungen[:,2])

print("Die hoechste jemals gegebene Bertung ist:", hoechste_bewertung)
```

```
Die niedrigste jemals gegebene Bertung ist: 2.5
Die hoechste jemals gegebene Bertung 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: 4.166666666666667
```

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_aktiver_person = einzigartige_person[np.argmax(anzahl)]

print("Personen mit den meisten Bewertungen:", meist_aktiver_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 Berwertung 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]]
```

9.2 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 9.1: 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 Zahlenwerten 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 = "abrakadabra"
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 = "abrakadabra"
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']
```