w-python-numpy-grundlagen

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

2025-08-26

Table of contents

Preamble

Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Maik Poetzsch, und Sebastian Seipel. 2024. „Bausteine Computergestützter Datenanalyse. Werkzeugbaustein NumPy“. https://github.com/bausteine-der-datenanalyse/w-python-numpy-grundlagen.

BibTeX-Vorlage

```
@misc{BCD-Styleguide-2024,
 title={Bausteine Computergestützter Datenanalyse. Werkzeugbaustein NumPy},
 author={Arnold, Lukas and Arnold, Simone and Baitsch, Matthias and Fehr, Marc and Poetzsch,
 year={2024},
 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 Introduction to NumPy

NumPy is a powerful library for Python that is used for numerical computing and data analysis. The name "NumPy" is an acronym for "Numerical Python".
NumPy itself is mainly written in the C programming language, which is why NumPy is generally very fast.

NumPy allows efficient work with small and large vectors and matrices that would otherwise be cumbersome to implement in native Python. It also provides the ability to perform calculations with vectors and matrices easily—even for very large datasets.

This introduction will help you understand and use the basics of NumPy.

1.1 Advantages & Disadvantages

In most cases, operations with NumPy data structures are faster. However, unlike native Python lists, NumPy arrays can only hold one data type per list.

Why is NumPy often faster?

NumPy implements a more efficient memory storage for lists.
Native Python stores list contents in scattered memory locations, wherever space is available.

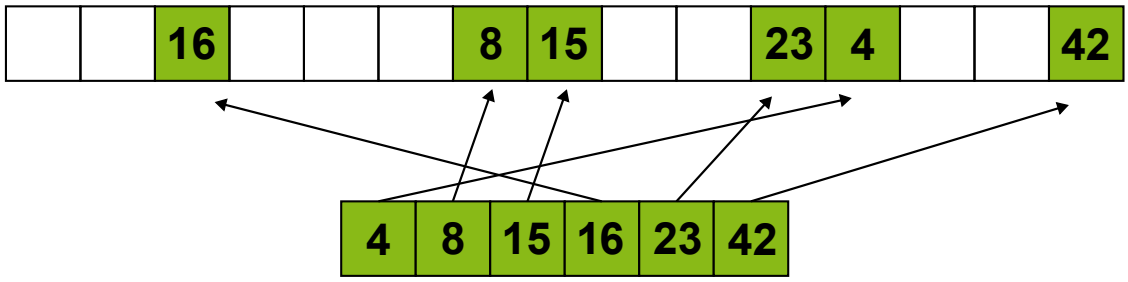


Figure 1.1: Storage of data in native Python

In contrast, NumPy arrays and matrices are stored in contiguous blocks of memory, allowing for more efficient data access.

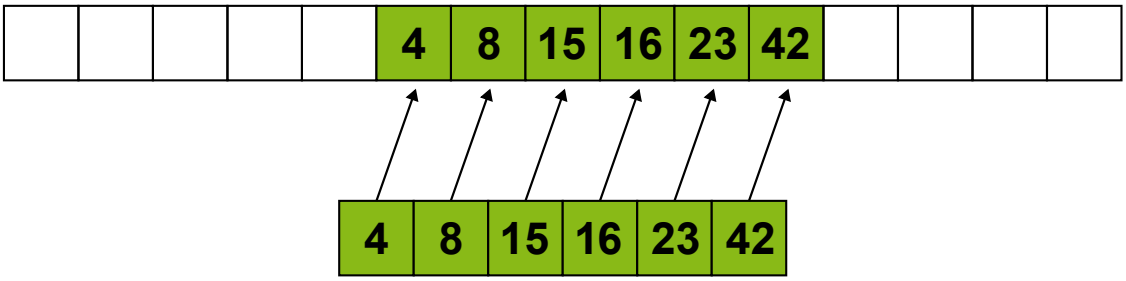


Figure 1.2: Storage of data in NumPy

However, this also means that expanding a list is much faster than expanding arrays or matrices. Lists can use any free space, while arrays and matrices have to be copied to a new location in memory.

1.2 Importing the Package

NumPy is imported with the following line. It is a global convention to use the alias `np`.

```
import numpy as np
```

1.3 References

All functions introduced here can be found in the (English) NumPy documentation: Documentation

2 Creating NumPy Arrays

In Python, vectors are typically represented by lists and matrices by nested lists. For example, the vector

$$(1, 2, 3, 4, 5, 6)$$ and the matrix $$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

can be natively created in Python like this:

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

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

print(lst)
print(matrix)
```

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

If you want to use NumPy arrays, you can use the `np.array()` command:

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

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

print(lst)
print(matrix)
```

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

If you look at the output from the `print()` commands, two things stand out. First, the commas are gone, and second, the matrix is printed in a clean, readable format.

It is also possible to create higher-dimensional arrays. This requires another level of nesting. In the following example, a three-dimensional matrix is created:

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

It is considered good practice to always initialize arrays. NumPy offers three functions to create pre-initialized arrays. Alternatively, arrays can be initialized with fixed values. You can use `np.zeros()` to set all values to 0 or `np.ones()` to initialize all values with 1. These functions take the shape in the form `[rows, columns]`. If you want to initialize all elements with a specific value, use the `np.full()` function.

```
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]])
```

> How could you create arrays filled with a specific value x?
>
> The trick is to initialize an array with `np.ones()` and then multiply the array by the number `x`. In the following example, `x = 5`:
>
> ```
> np.ones([2,3]) * 5
> ```
>
> ```
> array([[5., 5., 5.],
>        [5., 5., 5.]])
> ```

If you want to create a vector with evenly spaced values, e.g., for an axis in a plot, NumPy offers two options. Use `np.linspace(start, stop, #values)` or `np.arange(start, stop, step)` to generate such arrays.

```
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])
```

Practice Exercise: Array Creation

Create one NumPy array for each of the following tasks:

1. with the values 1, 7, 42, 99
2. ten times the number 5
3. with numbers from 35 **up to and including** 50
4. with all even numbers from 20 **up to and including** 40
5. a matrix with 5 columns and 4 rows filled with the value 4
6. with 10 values evenly spaced from 22 **up to and including** 40

Solution

```
# 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 Size, Structure, and Type

If you're unsure about the structure or shape of an array, or if you want to use this information for loops, NumPy offers the following functions to retrieve it:

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

`np.shape()` returns the length of each dimension in the form of a tuple.

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

```
(2, 3)
```

The native Python function `len()` only returns the length of the first dimension, i.e., the number of elements in the outer brackets. In the example above, `len()` sees the two lists `[1, 2, 3]` and `[4, 5, 6]`.

```
len(matrix)
```

```
2
```

The function `np.ndim()` returns the number of dimensions, unlike `np.shape()`.

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

```
2
```

> The output of `np.ndim()` can also be derived using `np.shape()` and a native Python function. How?
>
> `np.ndim()` simply returns the length of the tuple from `np.shape()`:

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

```
2
```

If you want to know the total number of elements in an array, you can use the function `np.size()`.

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

```
6
```

NumPy arrays can contain various data types. Below we have three different arrays with different data types:

```
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(["Monday", "Tuesday", "Wednesday"])
```

With the method `np.dtype`, we can retrieve the data type of arrays. Usually, this returns the type along with a number that represents the number of bytes needed to store the values. The array *typ_a* has the data type `int64`, meaning whole numbers.

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

```
int64
```

The array *typ_b* has the data type `float64`, where float represents floating-point numbers.

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

```
float64
```

The array *typ_c* has the data type `U8`, where `U` stands for Unicode. The text is stored in Unicode format.

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

```
<U9
```

Below is a table of typical data types you'll commonly encounter in NumPy:

Table 3.1: Typical Data Types in NumPy

Data Type	NumPy Name	Examples
Boolean	`bool`	[True, False, True]
Integer	`int`	[-2, 5, -6, 7, 3]
Unsigned Integer	`uint`	[1, 2, 3, 4, 5]
Floating Point	`float`	[1.3, 7.4, 3.5, 5.5]
Complex Numbers	`complex`	[-1 + 9j, 2 - 77j, 72 + 11j]
Text (Unicode)	`U`	["monday", "tuesday"]

Intermediate Exercise: Reading Array Information

Given the following 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]]])
```

Visually determine the number of dimensions and the length of each dimension. What is the data type of the elements in this matrix?
Then verify your results by applying the appropriate NumPy functions.

Solution

```
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]]])

num_dimensions = np.ndim(matrix)
print("Number of dimensions: ", num_dimensions)

dimension_lengths = np.shape(matrix)
print("Lengths of each dimension: ", dimension_lengths)

print(matrix.dtype)
```

```
Number of dimensions:  3
Lengths of each dimension:  (3, 3, 4)
int64
```

4 Working with Arrays

4.1 Arithmetic Functions

One major advantage of NumPy is working with arrays. Without NumPy, you would either have to use a `loop` or a `list comprehension` to perform operations on all values in a list. NumPy eliminates this inconvenience.

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

b = np.array([9, 8, 7, 6, 5])
```

Basic mathematical operations like addition can be expressed in two ways: either using the `np.add()` function or simply with the `+` operator.

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

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

```
a + b
```

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

Without NumPy, the operation would look like this:

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

print(result)
```

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

For other types of arithmetic, there are functions like: `np.subtract()`, `np.multiply()`, and `np.divide()`.

Higher-level mathematical operations also have functions:

- `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)`

> **Working with Trigonometric Functions**
>
> Just like with a calculator, a common error when using trigonometric functions (sin, cos, ...) is to input degrees instead of radians. However, the trigonometric functions in NumPy expect values in radians.
> To easily convert between degrees and radians, NumPy provides the functions `np.deg2rad()` and `np.rad2deg()`.

4.2 Comparisons

NumPy arrays can also be compared with one another. Let's look at the following two arrays:

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

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

To check whether these arrays are identical, we can use the `==` comparator. This compares the arrays element-wise.

```
a == b
```

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

You can also compare arrays using the `>` and `<` operators:

```
a < b
```

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

When comparing arrays with floating point numbers, it is often necessary to allow for some tolerance due to small rounding errors in computations.

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

```
np.False_
```

For this case, NumPy offers a comparison function `np.isclose(a,b,atol)`, where `atol` stands for absolute tolerance. In the following example, an absolute tolerance of 0.001 is used.

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

```
True
```

> **Why is 0.1 + 0.2 not equal to 0.3?**
>
> Numbers are internally represented in binary. Just like 1/3 cannot be represented precisely with a finite number of decimal digits, some numbers must be rounded in binary representation.
>
> ```
> a = 0.1
> b = 0.2
> print(a + b)
> ```
>
> ```
> 0.30000000000000004
> ```

4.3 Aggregation Functions

For many types of analysis, we need functions such as sum or mean. Let's start with an example array `a`:

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

The sum is calculated using the `np.sum()` function.

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

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

Of course, you can also determine the minimum and maximum of an array. The functions are `np.min()` and `np.max()`.

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

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

If you want the position of the maximum value instead of the value itself, use `np.argmax` instead of `np.max`.

For statistical analysis, common functions are `np.mean()` for the mean, `np.median()` for the median, and `np.std()` for the standard deviation.

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

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

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

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

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

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

Exercise: Working with Arrays

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

1. Create a new array that contains the sine values of the added arrays `a` and `b`.
2. Calculate the sum, mean, and standard deviation of the elements in `a`.
3. Find the largest and smallest values in both `a` and `b`.

Solution

```
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 Basic Slicing with Numeric Indices

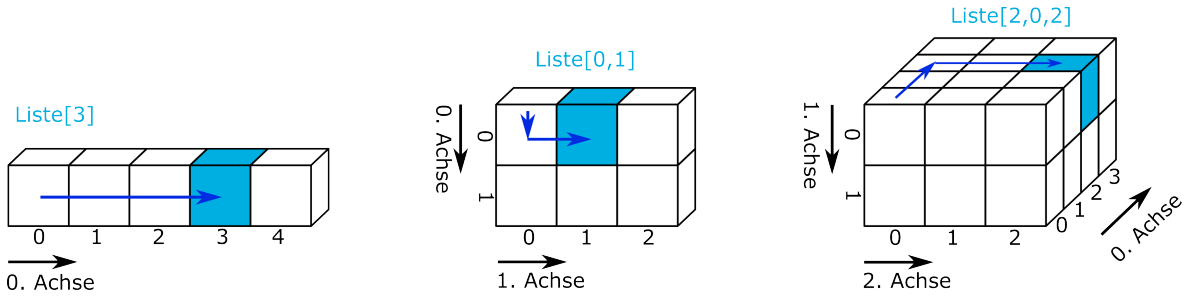


Figure 5.1: Accessing individual axes for one-, two-, and three-dimensional arrays including examples

To select data within an array, use the following formats:

1. `[a]` returns the single value at position `a`
2. `[a:b]` returns all values from position `a` up to (but not including) `b`
3. `[a:b:c]` returns values from position `a` to `b-1` in steps of `c`

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

```
# Selecting the first element
list[0]
```

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

```
# Selecting the last element
list[-1]
```

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

```
# Selecting a range of elements
list[1:4]
```

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

For two-dimensional arrays, you use a comma to separate the selection along the first and second dimensions.

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

```
# Selecting a single element
matrix[1,1]
```

```
np.int64(5)
```

For three-dimensional arrays, an additional index is used, again separated by commas. The order remains: first, second, then third dimension.

```
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]]]
```

```
# Selecting a specific element
matrix_3d[1,0,2]
```

```
np.int64(9)
```

5.2 Slicing with Boolean Values (Boolean Masking)

Boolean slicing uses a boolean mask to select specific elements from an array. The mask is an array of the same length as the original, consisting of `True` or `False` values.

```
# Create a sample array
a = np.array([1, 2, 3, 4, 5, 6])

# Create the mask
maske = a > 3

print(maske)
```

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

This results in an array of boolean values. Applying this mask to the original array returns all elements where the mask is `True`.

```
# Applying the mask
print(a[maske])
```

```
[4 5 6]
```

Warning

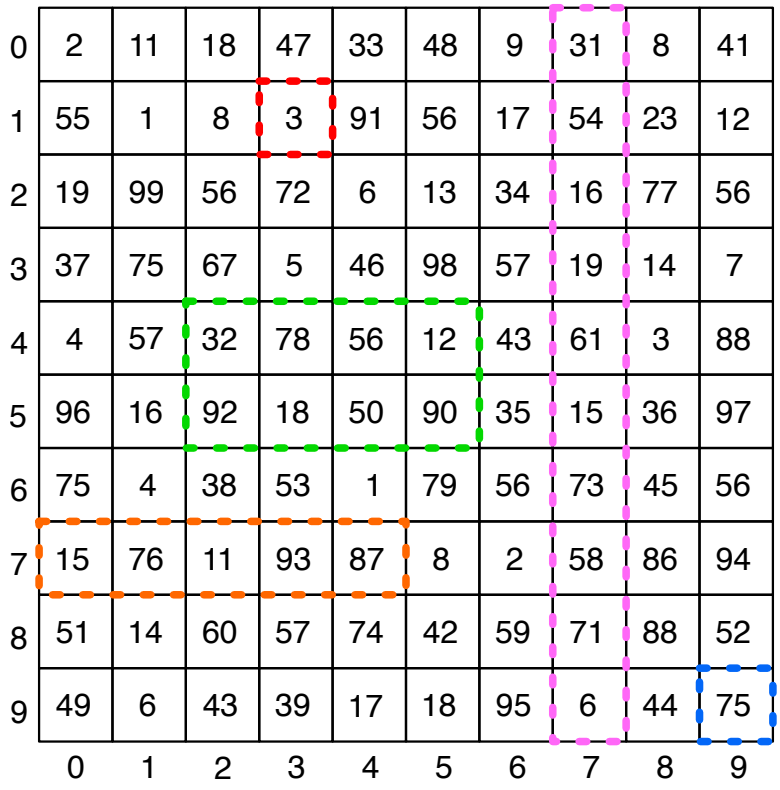
Using boolean arrays is only possible with the NumPy module. This approach cannot be applied to native Python lists. In such cases, you must iterate over the list manually.

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

```
[4, 5, 6]
```

Mini Exercise: Array Slicing

Select the colored sections of the array `matrix` using the slicing techniques you just learned.




```
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]
])
```

Solution

- Red: `matrix[1,3]`
- Green: `matrix[4:6,2:6]`
- Pink: `matrix[:,7]`
- Orange: `matrix[7,:5]`
- Blue: `matrix[-1,-1]`

6 Array Manipulation

6.1 Changing the Shape

Various functions allow us to change the shape and contents of arrays.

One of the most important array operations is transposing. This operation switches rows with columns and vice versa.

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

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

If we now transpose this array, we get:

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

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

If we have this matrix and want to turn it into a vector, we can use the `np.flatten()` function:

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

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

To return to a two-dimensional data structure, we use the function `np.reshape(target, shape)`:

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

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

If we want to expand, shrink, or modify the content of an existing array, NumPy also provides suitable functions.

If we have an empty array or want to add elements to an existing array, we use the `np.append()` function. This function appends a value to the existing array.

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

new_array = np.append(array, 7)
print(new_array)
```

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

Sometimes we need to insert a value not at the end but at a specific position in the array. The appropriate tool here is the function `np.insert(array, position, insertion)`. In the following example, the number 7 is inserted at the third position.

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

new_array = np.insert(array, 3, 7)
print(new_array)
```

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

Just like we can insert new elements, we can also delete elements. For this, we use the function `np.delete(array, position)`, which takes the array and the position of the element to delete.

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

new_array = np.delete(array, 3)
print(new_array)
```

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

Lastly, let's look at joining two arrays. In the following example, array `b` is appended to array `a` using the function `np.concatenate((array a, array b))`.

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

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

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

6.2 Sorting Arrays

NumPy also provides the ability to sort arrays. In the following example, we start with an unsorted array. Using the `np.sort()` function, we obtain a sorted array.

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

sorted_array = np.sort(unsorted)

print(sorted_array)
```

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

6.3 Sublists with Unique Values

When working with data where, for example, projects are assigned employee IDs, there may be a finite number of employee IDs that appear multiple times if an employee works on several projects.

If we want a list where each number appears only once, we can use the `np.unique` function.

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

unique_values = np.unique(list_with_duplicates)

print(unique_values)
```

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

If we also set the option `return_counts=True`, a second variable will store how often each value occurs.

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

unique_values, counts = np.unique(list_with_duplicates, return_counts=True)

print(counts)
```

```
[2 3 2 1 1]
```

Mini Exercise: Array Manipulation

Given the following two-dimensional array named `matrix`:

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

1. Change the shape of the array `matrix` into a one-dimensional array.
2. Sort the one-dimensional array in ascending order.
3. Change the shape of the sorted array into a two-dimensional array with 2 rows and 6 columns.
4. Determine the unique elements in the original array `matrix` and print them.

Solution

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

# 1. Change to a one-dimensional array
flat_array = matrix.flatten()

# 2. Sort the one-dimensional array in ascending order
sorted_array = np.sort(flat_array)

# 3. Reshape the sorted array into a 2x6 array
reshaped_array = sorted_array.reshape(2, 6)

# 4. Find the unique elements in the original array
unique_elements_original = np.unique(matrix)
```

7 Reading and Writing Files

The `numpy` module provides functions for reading and writing structured text files.

7.1 Reading Files

To read structured text files, such as CSV (comma-separated values) files, the `np.loadtxt()` function can be used. It takes the filename to be read as its main argument, along with other options to define the data structure. The return value is a (multi-dimensional) array.

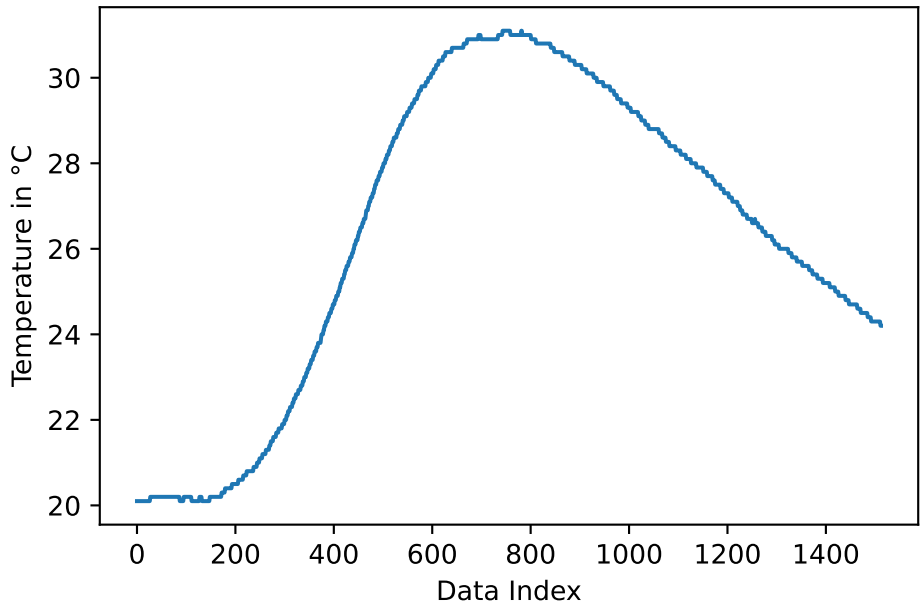
In the following example, the file TC01.csv is read and its contents are plotted. The first line of the file is ignored, as it is interpreted as a comment (starting with the `#` character).

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

```
print("Data:", data)
print("Shape:", data.shape)
```

```
Data: [20.1 20.1 20.1 ... 24.3 24.2 24.2]
Shape: (1513,)
```

```
plt.plot(data)
plt.xlabel('Data Index')
plt.ylabel('Temperature in °C');
```



By default, the `np.loadtxt()` function expects comma-separated values. If the data is separated by a different character, the `delimiter = ""` option can be used to specify a different delimiter. For example, if the separator is a semicolon, the function call would look like: `np.loadtxt(data.txt, delimiter = ";")`

If the file starts with lines containing additional information such as units or experiment details, these can be skipped using the `skiprows= #rows` option.

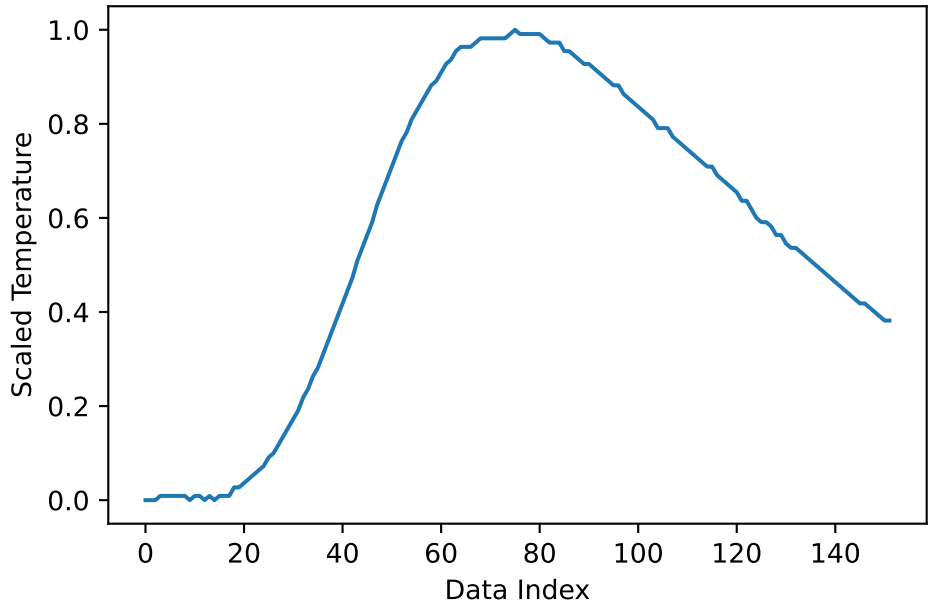
7.2 Writing Files

To write arrays to files, you can use the `np.savetxt()` function available in `numpy`. At a minimum, it requires the array to be written and a filename. Numerous formatting and structuring options are also available.

The following example scales the data read earlier and writes every tenth value to a new file. A comment (`header` argument) is also added at the beginning of the file. The output format of the numbers can be specified using the `fmt` argument. The format is similar to the formatting style used in formatted strings.

```
value_range = np.max(data) - np.min(data)
data_scaled = (data - np.min(data)) / value_range
data_scaled = data_scaled[::10]
```

```
plt.plot(data_scaled)
plt.xlabel('Data Index')
plt.ylabel('Scaled Temperature');
```



When writing the file, a multi-line comment is defined using the newline character `\n`. The floating-point numbers are formatted with `%5.2f`, which means 5 total characters with 2 digits after the decimal point.

```
# Assignment is split into multiple lines due to the narrow display in the script
comment = f'Data from {filename} scaled to the range ' + \
          '0 to 1 \noriginal min / max: ' + \
          f'{np.min(data)}/{np.max(data)}'
new_filename = '01-daten/TC01_scaled.csv'

np.savetxt(new_filename, data_scaled,
           header=comment, fmt='%5.2f')
```

To illustrate, the first lines of the newly created file are printed.

```
# Read the first lines of the newly created file
file = open(new_filename, 'r')
for i in range(10):
    print(file.readline(), end='')
file.close()
```

```
# Data from 01-daten/TC01.csv scaled to the range 0 to 1
# original min / max: 20.1/31.1
 0.00
 0.00
 0.00
 0.01
 0.01
 0.01
 0.01
 0.01
```

8 Working with Images

Images are digitally stored as matrices. Each pixel contains three color values: red, green, and blue. These three values (ranging from 0–255) are combined to produce all desired colors.

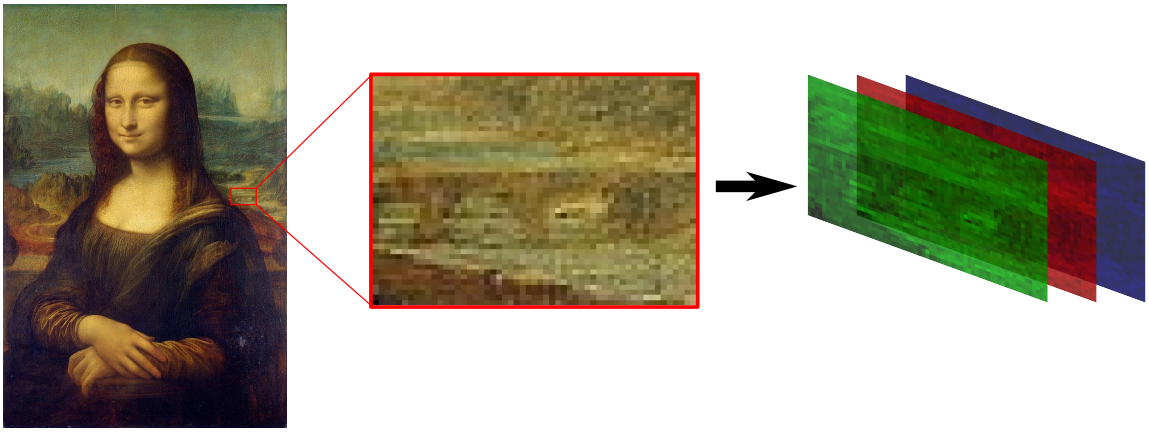


Figure 8.1: A high-resolution image consists of many pixels. Each pixel contains 3 color values: one each for green, blue, and red.

Thanks to the digital nature of images, they can be easily processed using NumPy tools. In the following example, we use the Mona Lisa as our image. It can be found at this link.

Let's import this image using the `imread()` function from the matplotlib package. You'll see that it is a three-dimensional NumPy array.

```
import matplotlib.pyplot as plt

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

```
Shape: (1024, 677, 3)
```

Let's take a look at a snippet of the data using the `print()` function.

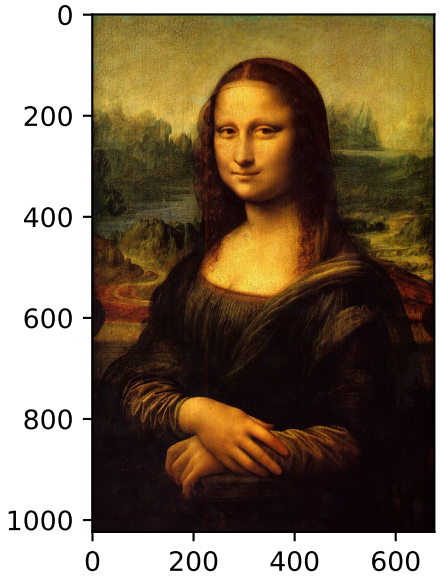
```
print(data)
```

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

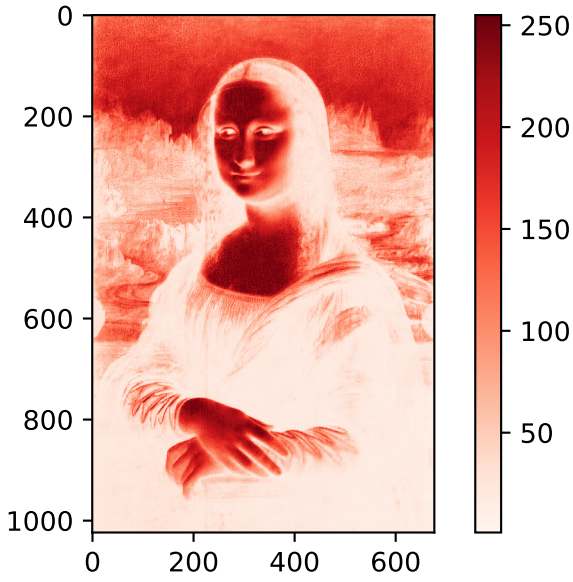
Using the `plt.imshow` function, the image can be displayed in true color. This works because the function interprets the last index of the dataset as color information (red, green, blue). If there were a fourth layer, it would be interpreted as an individual transparency value.

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



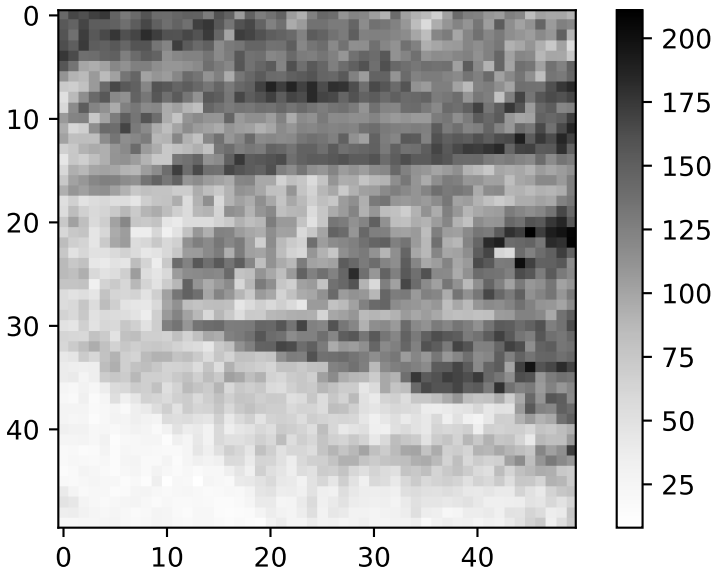
Of course, the individual color channels can also be viewed separately. We do this by fixing the last index. Here we look only at the red component of the image. When displaying a simple array, the output is in grayscale. With the option `cmap='Reds'`, we can adjust the color scale.

```
# Use the red color scale 'Reds'
plt.imshow(data[:,:,0], cmap='Reds')
plt.colorbar()
plt.show()
```



Since the image data is stored as arrays, many operations such as slicing or other computations are possible. The example below shows a cropped area of the red channel.

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



Now let's consider a more complex operation on image data, the Laplace operator. It can be used to detect the edges of objects. For each pixel $B_{i,j}$ – except those at the border – the following value $\phi_{i,j}$ is computed:

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

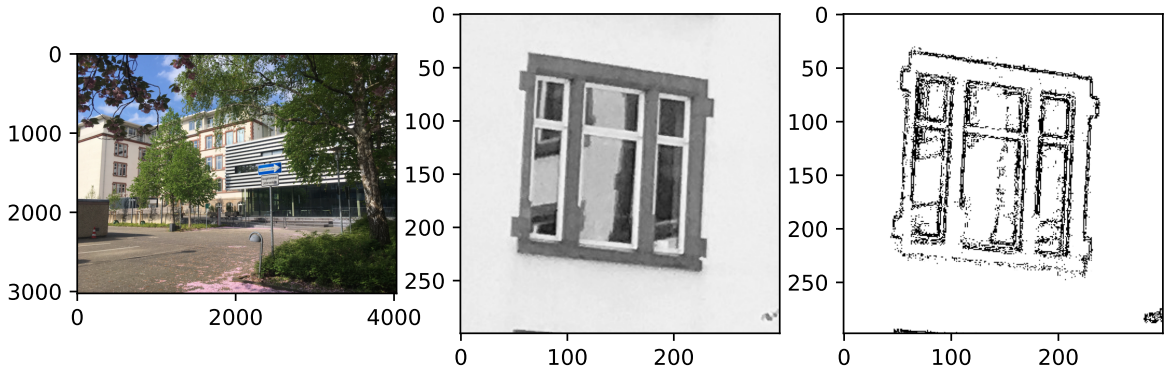
The following function implements this operation. In addition, all values of ϕ below a threshold are set to zero, and those above are set to 255.

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

    # Create a copy of the data as a float array
    region = np.array(data, dtype=float)

    # Split the equation into two parts
    lapx = region[2:, :] - 2 * region[1:-1, :] + region[:-2, :]
    lapy = region[:, 2:] - 2 * region[:, 1:-1] + region[:, :-2]

    # Combine the parts and take the absolute value
    lap = np.abs(lapx[:,1:-1] + lapy[1:-1, :])

    # Thresholding
    lap[lap > threshold] = 255
    lap[lap < threshold] = 0

    return lap
```

Now let's examine a photo of the Haspel Campus in Wuppertal: Image. Applying the Laplace operator to a cropped section gives the following result:

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

lap = img_lap(region)

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(region, cmap="Greys_r");
```

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



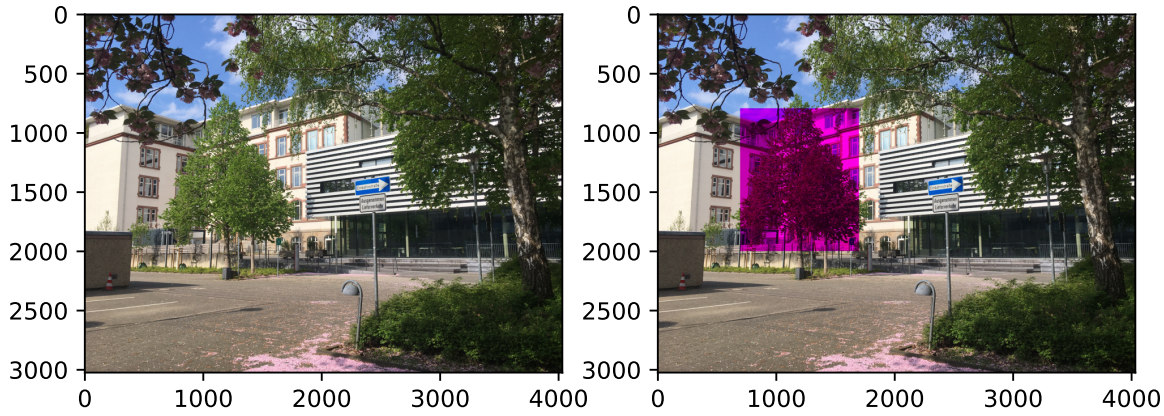
We can now clearly see the outlines of the window.

If we want to edit a color component and then reassemble the image, we use the function `np.dstack((red, green, blue)).astype('uint8')`, where `red`, `green`, and `blue` are the individual 2D arrays. Let's try removing the green color from the tree on the left.

It's important that the data is in `uint8` format after recombining, which is why we use `.astype('uint8')`.

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

# Store individual color channels in arrays
red = np.array(data[:, :, 0], dtype=float)
green = np.array(data[:, :, 1], dtype=float)
blue = np.array(data[:, :, 2], dtype=float)

# Set the green area of the tree on the left to 0
green_new = green.copy()
green_new[800:2000, 700:1700] = 0

composed = np.dstack((red, green_new, blue)).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(composed)
```



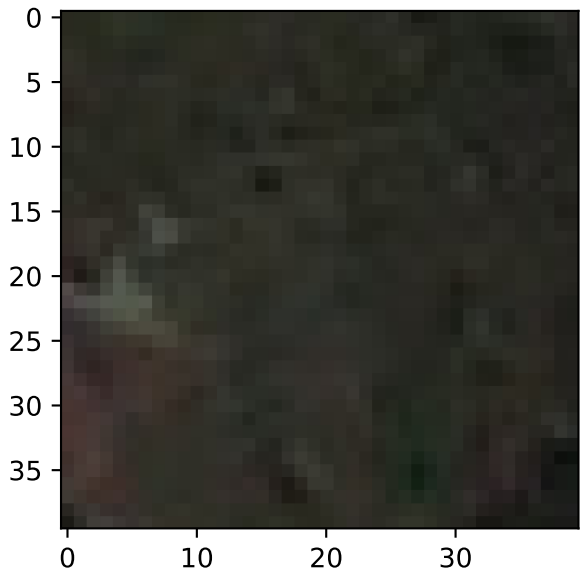
Mini Exercise: Image Manipulation

Load the following image of the Haspel Campus in Wuppertal:
Image
Extract the blue component and display the middle row of the image, as well as any selected image section.

Solution

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

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

shape = data.shape
print("Shape:", shape)

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

row = data[int(shape[0]/2), :, 2]
print(row)

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

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



9 Learning Objective Check

Welcome to the learning objective check!

This self-assessment is designed to help you review your understanding of the topics covered so far and to allow you to independently evaluate your learning progress. It is structured so that you can identify your strengths and weaknesses and work specifically on areas that still need improvement.

You have two options to test your knowledge. You can use the quiz, which will guide you automatically through the different topics. Alternatively, you will find standard questions below, similar to those used in the script so far.

Please take enough time to work through the questions calmly. Be honest with yourself and try to solve the tasks without any aids in order to get a realistic picture of your current level of knowledge. If you encounter difficulties with a question, it is an indication that you should practice more in that area.

Good luck with the exercises and your continued learning!

Task 1

How is the NumPy package typically imported?

Task 2

Use NumPy to create the following arrays:

1. Convert the list [1, 2, 3] into a NumPy array
2. A one-dimensional array containing the numbers from 0 to 9
3. A two-dimensional array of shape 3×33×3 filled with ones
4. A one-dimensional array containing numbers from 10 to 50 (inclusive) in steps of 5

Task 3

What is the difference between the functions `np.ndim`, `np.shape`, and `np.size`?

Task 4

What data type does the following array have? Which function can be used to read the data type of an array?

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

Task 5

Perform the following mathematical operations on these two arrays:

a = [5, 1, 3, 6, 4] and b = [6, 5, 2, 6, 9]

1. Add both arrays
2. Calculate the element-wise product of a and b
3. Add 3 to each entry in array a

Task 6

a = [9, 2, 3, 1, 3]

1. Calculate the mean and standard deviation of array a
2. Determine the minimum and maximum of the array

Task 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. Extract the first row
2. Extract the last column
3. Extract the submatrix consisting of rows 2 to 4 and columns 1 to 3

Task 8

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

1. Reshape the array into a two-dimensional matrix with shape 4×5
2. Reshape the array into a two-dimensional matrix with shape 5×4
3. Reshape the array into a three-dimensional matrix with shape 2×2×5
4. Flatten the three-dimensional array from Task 3 back into a one-dimensional array
5. Transpose the 4×5 matrix from Task 1

Task 9

Which two functions can be used to read data from a file and save data to a file?

Task 10

You want to isolate the pixel data of one color channel from an image. What do you need to do?

Solutions

Task 1

```
import numpy as np
```

Task 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]
```

Task 3

- `np.ndim`: Returns the number of dimensions
- `np.shape`: Returns the lengths of each dimension
- `np.size`: Returns the total number of elements

Task 4

Since these are floating-point numbers, the data type is `float64`.

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

```
float64
```

Task 5

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

# 1.
result = a + b
print("The sum of both vectors is:", result)

# 2.
result = a * b
print("The product of both vectors is:", result)

# 3.
result = a + 3
print("a plus 3 is:", result)
```

```
The sum of both vectors is: [11  6  5 12 13]
The product of both vectors is: [30  5  6 36 36]
a plus 3 is: [8 4 6 9 7]
```

Task 6

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

# 1.
mean = np.mean(a)
print("Mean:", mean)

std_dev = np.std(a)
print("Standard deviation:", std_dev)

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

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

```
Mean: 3.6
```

```
Standard deviation: 2.8000000000000003
Minimum: 1
Maximum: 9
```

Task 7

```python
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. First row
print(matrix[0, :])

# 2. Last column
print(matrix[:, -1])

# 3. Submatrix from rows 2 to 4 and columns 1 to 3
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]]
```

Task 8

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

# 1. Reshape to 4x5
matrix_4x5 = array.reshape(4, 5)

# 2. Reshape to 5x4
matrix_5x4 = array.reshape(5, 4)

# 3. Reshape to 2x2x5
matrix_2x2x5 = array.reshape(2, 2, 5)

# 4. Flatten
flattened_array = matrix_2x2x5.flatten()

# 5. Transpose the 4x5 matrix
transposed_matrix = matrix_4x5.T

# Optional outputs
print("Original array:", array)
print("4x5 matrix:\n", matrix_4x5)
print("5x4 matrix:\n", matrix_5x4)
print("2x2x5 matrix:\n", matrix_2x2x5)
print("Flattened array:", flattened_array)
print("Transposed 4x5 matrix:\n", transposed_matrix)
```

```
Original 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]]]
Flattened array: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
Transposed 4x5 matrix:
 [[ 1  6 11 16]
 [ 2  7 12 17]
 [ 3  8 13 18]
 [ 4  9 14 19]
 [ 5 10 15 20]]
```

Task 9

The appropriate functions are `np.loadtxt()` and `np.savetxt()`.

Task 10

Image data is typically stored as a 3D matrix, where each color component (usually red, green, and blue) is stored in a separate channel. If the image is stored in a matrix called `data`, you can isolate a color channel using slicing: `data[:, :, 0]`, where `0`, `1`, or `2` selects the red, green, or blue channel respectively.

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 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]]
```

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.

Table 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 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']
```

11 Exam Questions

Exercise 1

A rectangular concrete beam is subjected to a uniformly distributed load along its length. The stress distribution along the length of the beam shall be analyzed. The beam has a length of 10 meters, a width of 0.3 meters, and a height of 0.5 meters. The uniformly distributed load is 5000 N/m.

1. Create a NumPy array `x` with 100 evenly spaced points along the length of the beam from 0 to 10 meters.
2. Calculate the bending moments $M(x)$ along the length of the beam using the formula:

   $$M(x) = \frac{w \cdot x \cdot (L - x)}{2}$$

   where w is the distributed load (in N/m), x is the position along the beam (in m), and L is the length of the beam (in m).
3. Compute the maximum bending stress $\sigma_{\max}(x)$ at each point along the beam using the formula:

   $$\sigma_{\max}(x) = \frac{M(x) \cdot c}{I}$$

   where c is the distance from the neutral axis to the outermost fiber of the beam (in m), and I is the moment of inertia.
   The moment of inertia of a rectangular cross-section is given by:

   $$I = \frac{b \cdot h^3}{12}$$

   where b is the width (in m) and h is the height of the beam (in m).
4. Determine the maximum bending stress.
5. Plot the stress distribution $\sigma_{\max}(x)$ along the length of the beam.