

# **Vorlesung Ingenieurinformatik**

Lukas Arnold	Simone Arnold	Kristian Börger
Karen De Lannoye	Tristan Hehnen	Keyvan Najarianeraghi
Sven Orzel	My Linh Würzburger	Florian Bagemihl
Matthias Baitsch	Marc Fehr	Maik Poetzsch
	Sebastian Seipel	

2025-07-24

# Table of contents

<b>Übersicht</b>	<b>3</b>
Allgemeine Infos . . . . .	3
Kontakt . . . . .	3
Abschlussarbeiten . . . . .	3
English Version . . . . .	3
<b>I Grundlagen Python</b>	<b>4</b>
<b>Werkzeugbaustein Python</b>	<b>5</b>
Voraussetzungen . . . . .	5
Lernziele . . . . .	5
<b>1 Einführung</b>	<b>7</b>
<b>2 Willkommen bei Python!</b>	<b>8</b>
2.1 Lernziele dieses Kapitels . . . . .	8
2.2 Ihr erstes Programm . . . . .	8
2.3 Variablen – Namen für Werte . . . . .	9
<b>3 Datentypen verstehen</b>	<b>11</b>
3.1 Lernziele dieses Kapitels . . . . .	11
3.2 Einleitung . . . . .	11
3.3 Die wichtigsten Datentypen . . . . .	11
3.4 Unterschiede zwischen <code>int</code> und <code>float</code> . . . . .	12
3.5 Was sind Booleans ( <code>bool</code> )? . . . . .	12
3.6 Rechnen mit Zahlen . . . . .	13
3.7 Arbeiten mit Text . . . . .	14
3.8 Umwandlung von Datentypen (Typecasting) . . . . .	14
<b>4 Entscheidungen und Wiederholungen</b>	<b>16</b>
4.1 Lernziele dieses Kapitels . . . . .	16
4.2 Bedingungen mit <code>if, elif, else</code> . . . . .	16
4.3 Vergleichsoperatoren . . . . .	17
4.4 Wiederholungen mit <code>while</code> . . . . .	17
4.5 Schleifen mit <code>for</code> und <code>range()</code> . . . . .	18

4.6 Was macht <code>range()</code> genau? . . . . .	18
4.6.1 Varianten: . . . . .	18
<b>5 Mehrere Werte speichern</b>	<b>21</b>
5.1 Was ist eine Liste? . . . . .	21
5.2 Teile aus Listen ausschneiden – Slicing . . . . .	21
5.2.1 Syntax: <code>liste[start:stop]</code> . . . . .	22
5.3 Über Listen iterieren . . . . .	22
5.4 Erweiterung: Bedingte Ausgaben . . . . .	23
5.5 Durchschnitt berechnen . . . . .	23
5.6 Listen erweitern: <code>.append()</code> . . . . .	23
5.7 Verschachtelte Schleifen . . . . .	24
5.8 Listen sortieren . . . . .	24
<b>6 Wiederverwendbarer Code mit Funktionen</b>	<b>26</b>
6.1 Lernziele dieses Kapitels . . . . .	26
6.2 Eine Funktion definieren . . . . .	26
6.3 Parameter übergeben . . . . .	27
6.4 Rückgabewerte mit <code>return</code> . . . . .	27
6.5 Beispiel: Umrechnungen . . . . .	28
6.5.1 Euro zu US-Dollar . . . . .	28
6.6 Parameter mit Standardwerten . . . . .	29
<b>7 Arbeiten mit Dateien</b>	<b>30</b>
7.1 Lernziele dieses Kapitels . . . . .	30
7.2 Eine Datei einlesen . . . . .	30
7.3 Zeilenweise lesen . . . . .	31
7.4 Alle Zeilen auf einmal lesen mit <code>readlines()</code> . . . . .	31
7.5 In eine Datei schreiben . . . . .	32
7.6 Zeilenweise schreiben mit Schleife . . . . .	32
7.7 Dateien manuell schließen mit <code>close()</code> . . . . .	33
<b>8 Lernzielkontrolle</b>	<b>35</b>
8.1 Aufgabe 1: Datentypen und Typecasting . . . . .	35
8.2 Aufgabe 2: Altersprüfung mit Bedingung . . . . .	35
8.3 Aufgabe 3: Zahlen filtern und mitteln . . . . .	35
8.4 Aufgabe 4: Wochentage & Slicing . . . . .	36
8.5 Aufgabe 5: Funktion mit Parameter & Standardwert . . . . .	36
8.6 Aufgabe 6: Hobbys in Datei schreiben . . . . .	36
8.7 Aufgabe 7: Datei lesen & <code>.split()</code> verwenden . . . . .	36
8.8 Aufgabe 8: Namen sortieren und speichern . . . . .	37
8.9 Aufgabe 9: Zahlen durch 3 ausgeben . . . . .	37
8.10 Aufgabe 10: Verschachtelte Schleifen . . . . .	37

<b>II Matplotlib</b>	<b>38</b>
<b>III Intro</b>	<b>40</b>
Voraussetzungen . . . . .	41
Verwendete Pakete und Datensätze . . . . .	41
Bearbeitungszeit . . . . .	41
Lernziele . . . . .	41
<b>9 Einführung</b>	<b>42</b>
9.1 Warum Matplotlib? . . . . .	42
9.2 Alternativen zu Matplotlib . . . . .	42
9.3 Erstes Beispiel: Einfache Linie plotten . . . . .	42
9.4 Nächste Schritte . . . . .	43
<b>10 Diagrammtypen in Matplotlib</b>	<b>44</b>
10.1 1. Liniendiagramme ( <code>plt.plot()</code> ) . . . . .	44
10.2 2. Streudiagramme ( <code>plt.scatter()</code> ) . . . . .	45
10.3 3. Balkendiagramme ( <code>plt.bar()</code> ) . . . . .	46
10.4 4. Histogramme ( <code>plt.hist()</code> ) . . . . .	47
10.5 5. Boxplots ( <code>plt.boxplot()</code> ) . . . . .	48
10.6 6. Heatmaps ( <code>plt.imshow()</code> ) . . . . .	49
10.7 Fazit . . . . .	50
<b>11 Anpassung und Gestaltung von Plots in Matplotlib</b>	<b>51</b>
11.1 1. Achsentitel und Diagrammtitel . . . . .	51
11.2 2. Anpassung der Achsen . . . . .	52
11.3 3. Farben und Linienstile . . . . .	53
11.3.1 Wichtige Farben (Standardfarben in Matplotlib) . . . . .	53
11.3.2 Wichtige Linienstile . . . . .	53
11.4 4. Mehrere Plots mit Subplots . . . . .	54
11.5 5. Speichern von Plots . . . . .	56
11.6 Fazit . . . . .	56
<b>12 Erweiterte Techniken in Matplotlib</b>	<b>57</b>
12.1 1. Logarithmische Skalen . . . . .	57
12.2 2. Twin-Achsen für verschiedene Skalierungen . . . . .	58
12.3 3. Annotationen in Diagrammen . . . . .	59
12.4 Fazit . . . . .	60
<b>13 Best Practices in Matplotlib: Fehler und Verbesserungen</b>	<b>61</b>
13.1 1. Fehlende Beschriftungen . . . . .	61
13.1.1 Schlechtes Beispiel . . . . .	61
13.1.2 Besseres Beispiel . . . . .	62

13.2	2. Ungünstige Farbwahl . . . . .	63
13.2.1	Schlechtes Beispiel . . . . .	63
13.2.2	Besseres Beispiel . . . . .	64
13.3	3. Keine sinnvolle Achsenkalierung . . . . .	65
13.3.1	Schlechtes Beispiel . . . . .	65
13.3.2	Besseres Beispiel . . . . .	66
13.4	4. Überladung durch zu viele Linien . . . . .	67
13.4.1	Schlechtes Beispiel . . . . .	67
13.4.2	Besseres Beispiel . . . . .	68
13.5	Fazit . . . . .	69
<b>IV</b>	<b>Numpy</b>	<b>70</b>
<b>V</b>	<b>Intro</b>	<b>72</b>
	Voraussetzungen . . . . .	73
	Verwendete Pakete und Datensätze . . . . .	73
	Pakete . . . . .	73
	Datensätze . . . . .	73
	Bearbeitungszeit . . . . .	73
	Lernziele . . . . .	73
<b>14</b>	<b>Einführung NumPy</b>	<b>74</b>
14.1	Vorteile & Nachteile . . . . .	74
14.2	Einbinden des Pakets . . . . .	75
14.3	Referenzen . . . . .	75
<b>15</b>	<b>Erstellen von NumPy arrays</b>	<b>76</b>
<b>16</b>	<b>Größe, Struktur und Typ</b>	<b>80</b>
<b>17</b>	<b>Rechnen mit Arrays</b>	<b>84</b>
17.1	Arithmetische Funktionen . . . . .	84
17.2	Vergleiche . . . . .	85
17.3	Aggregatfunktionen . . . . .	87
<b>18</b>	<b>Slicing</b>	<b>89</b>
18.1	Normales Slicing mit Zahlenwerten . . . . .	89
18.2	Slicing mit logischen Werten (Boolesche Masken) . . . . .	90
<b>19</b>	<b>Array Manipulation</b>	<b>94</b>
19.1	Ändern der Form . . . . .	94
19.2	Sortieren von Arrays . . . . .	96

19.3 Unterlisten mit einzigartigen Werten . . . . .	96
<b>20 Lesen und Schreiben von Dateien</b>	<b>99</b>
20.1 Lesen von Dateien . . . . .	99
20.2 Schreiben von Dateien . . . . .	100
<b>21 Arbeiten mit Bildern</b>	<b>103</b>
<b>22 Lernzielkontrolle</b>	<b>111</b>
Aufgabe 1 . . . . .	111
Aufgabe 2 . . . . .	111
Aufgabe 3 . . . . .	112
Aufgabe 4 . . . . .	112
Aufgabe 5 . . . . .	112
Aufgabe 6 . . . . .	112
Aufgabe 7 . . . . .	112
Aufgabe 8 . . . . .	113
Aufgabe 9 . . . . .	113
Aufgabe 10 . . . . .	113
Aufgabe 1 . . . . .	113
Aufgabe 2 . . . . .	113
Aufgabe 3 . . . . .	114
Aufgabe 4 . . . . .	114
Aufgabe 5 . . . . .	114
Aufgabe 6 . . . . .	115
Aufgabe 7 . . . . .	116
Aufgabe 8 . . . . .	116
Aufgabe 9 . . . . .	118
Aufgabe 10 . . . . .	118
<b>23 Übung</b>	<b>119</b>
23.1 Aufgabe 1 Filmdatenbank . . . . .	119
23.2 Aufgabe 2 - Kryptographie - Caesar-Chiffre . . . . .	123
<b>VI Datenanalyse und Modellierung</b>	<b>128</b>
Einleitung . . . . .	129
Lernziele dieses Kapitels . . . . .	129
CSV-Dateien: Ein typisches Format für Messdaten . . . . .	129
Fehlende Werte erkennen und bereinigen . . . . .	130
Statistische Kennzahlen berechnen . . . . .	130
Daten visualisieren . . . . .	130

<b>24 Interpolation – Lücken schließen</b>	<b>132</b>
24.1 Übersicht . . . . .	133
24.2 Polynome . . . . .	133
24.3 Interpolation . . . . .	137
<b>25 Fitting</b>	<b>143</b>
<b>26 Splines</b>	<b>146</b>
26.1 Definition . . . . .	146
26.2 Kubische Splines . . . . .	146
26.3 Anwendung . . . . .	147
<b>27 Trendglättung – Rauschen reduzieren</b>	<b>150</b>
<b>28 Anwendungsbeispiele</b>	<b>151</b>
28.1 Anwendung: Ballonfahrt-Daten analysieren . . . . .	151
28.2 Anwendung: Balkenverformung im Bauingenieurwesen . . . . .	152
28.3 Zusammenfassung . . . . .	153
<b>VII Algorithmen</b>	<b>155</b>
Einführung . . . . .	156
Definition . . . . .	156
Beispiele . . . . .	156
<b>29 Von der Idee zum Code</b>	<b>158</b>
29.1 Kapitelübersicht . . . . .	164
<b>30 Sortieralgorithmen</b>	<b>165</b>
30.1 Selectionsort . . . . .	165
30.2 Bubblesort . . . . .	168
<b>31 Eigenschaften</b>	<b>172</b>
31.1 Terminiertheit . . . . .	172
31.2 Determiniertheit . . . . .	172
31.3 Effizienz . . . . .	173
31.4 Komplexität . . . . .	173
31.4.1 Komplexität Selectionsort . . . . .	174
31.4.2 Komplexität Bubblesort . . . . .	176
<b>32 Numerische Algorithmen</b>	<b>179</b>
32.1 Newton-Raphson-Verfahren . . . . .	179
32.1.1 Anwendungen . . . . .	179
32.1.2 Grundidee . . . . .	179

32.1.3 Beispiel 1 . . . . .	180
32.2 Schritt 0 . . . . .	182
32.3 Schritt 1 . . . . .	183
32.4 Schritt 2 . . . . .	184
32.5 Schritt 3 . . . . .	185
32.6 Schritt 4 . . . . .	186
32.6.1 Beispiel 2 . . . . .	186
32.7 Schritt 0 . . . . .	189
32.8 Schritt 1 . . . . .	190
32.9 Schritt 2 . . . . .	191
32.10 Schritt 3 . . . . .	192
32.11 Schritt 4 . . . . .	193
32.12 Euler-Verfahren . . . . .	193
32.12.1 Beispiel 1 . . . . .	194
<b>33 Exkurs: Interne Darstellung von Zahlen und Zeichen</b>	<b>195</b>
33.1 Analoge und digitale Signale . . . . .	195
33.1.1 Analoge Signale . . . . .	195
33.1.2 Digitale Signale . . . . .	195
33.1.3 Vergleich: Analoge vs. digitale Signale . . . . .	196
33.1.4 Umwandlung analoger zu digitaler Signale . . . . .	196
33.2 Digitale Zahlendarstellung . . . . .	196
33.2.1 Dualsystem . . . . .	197
33.2.2 Hexadezimalsystem . . . . .	197
33.3 Binäre Maßeinheiten . . . . .	198
33.4 Geschwindigkeit der Datenübertragung . . . . .	198
33.5 Darstellung ganzer Zahlen . . . . .	198
33.6 Umrechnung zwischen Dezimal- und Binärzahlen . . . . .	200
33.6.1 Von Binär nach Dezimal . . . . .	200
33.6.2 Von Dezimal nach Binär . . . . .	201
33.7 Darstellung reeller Zahlen . . . . .	201
33.8 Zeichendarstellung . . . . .	202
<b>34 Algorithmische Umrechnung von Zahlen: Dezimal Binär</b>	<b>204</b>
34.1 1. Umrechnung: Dezimal → Binär . . . . .	204
34.1.1 Grundidee . . . . .	204
34.1.2 Flussdiagramm . . . . .	205
34.1.3 Pseudocode . . . . .	206
34.1.4 Python-Implementierung . . . . .	206
34.2 2. Umrechnung: Binär → Dezimal . . . . .	206
34.2.1 Grundidee . . . . .	206
34.2.2 Flussdiagramm . . . . .	207
34.2.3 Pseudocode . . . . .	208

34.2.4 Python-Implementierung . . . . .	208
34.3 3. Mathematische Komplexität . . . . .	208
<b>VIII Numerik</b>	<b>209</b>
<b>35 Integration</b>	<b>211</b>
35.0.1 Ober- und Untersumme . . . . .	211
35.1 Definition . . . . .	212
35.2 Beispiel . . . . .	212
35.3 Interpolation . . . . .	216
35.3.1 Trapezregel . . . . .	217
35.3.2 Simpsonregel . . . . .	220
35.4 Monte-Carlo . . . . .	223
<b>36 Differentiation</b>	<b>227</b>
36.1 Taylor-Entwicklung . . . . .	227
36.2 Differenzenformeln . . . . .	229
36.2.1 Erste Ableitung erster Ordnung . . . . .	229
36.2.2 Erste Ableitung zweiter Ordnung . . . . .	231
36.3 Zweite Ableitung zweiter Ordnung . . . . .	233
36.4 Fehlerbetrachtung . . . . .	234
36.4.1 Approximationsfehler . . . . .	234
36.4.2 Rundungsfehler . . . . .	236

# Übersicht

## Allgemeine Infos

Die Vorlesung *Ingenieurinformatik* an der Bergischen Universität Wuppertal wurde vom im Jahr 2019 gebildeten Lehrstuhl [Computational Civil Engineering \(CCE\)](#) übernommen. Der CCE-Lehrstuhl beschäftigt sich hauptsächlich mit der Erforschung und Entwicklung neuer computergestützter Modelle. Im Zentrum der Anwendung steht die numerische Simulation der Brand- und Rauchausbreitung in Gebäuden.

Da sich das Skript in der Entwicklung befinden freuen wir uns über konstruktive Anregungen und Ihr Feedback. So können Sie Ihre Nachfolger unterstützen.

**Alle organisatorischen Informationen zum Ablauf finden Sie auf der [CCE-Webseite zur Ingenieurinformatik](#).**

## Kontakt

So erreichen Sie uns: \* Als Teilnehmer der Vorlesung: am besten über den zugehörigen [Moodle-Kurs](#) an der Bergischen Universität Wuppertal \* Externe Interessenten benutzten am besten unsere Emailliste \* Kontaktmöglichkeiten zu einzelnen Personen finden Sie auf der [Mitarbeiterwebseite](#)

## Abschlussarbeiten

Wir bieten Abschlussarbeiten (BA, MA, PhD) zu vielen verschiedenen Themen an \* eine Themenübersicht und bereits betreuter Arbeiten finden Sie auf der [Webseite der Abschlussarbeiten](#) \* Bei der Themenfindung kann auch die [Übersicht unserer Publikationen](#) helfen \* Bei Interesse kontaktieren Sie bitte Lukas Arnold

## English Version

Is an english version planned? Not yet, please contact Lukas Arnold.

# **Part I**

# **Grundlagen Python**

# Werkzeugbaustein Python



Bausteine Computergestützter Datenanalyse von Lukas Arnold, Simone Arnold, Florian Bagemihl, Matthias Baitsch, Marc Fehr, Maik Poetzsch und Sebastian Seipel. Werkzeugbaustein Python von Maik Poetzsch ist lizenziert unter [CC BY 4.0](#). Das Werk ist abrufbar auf [GitHub](#). Ausgenommen von der Lizenz sind alle Logos Dritter und anders gekennzeichneten Inhalte. 2025

Zitievorschlag

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

BibTeX-Vorlage

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

## Voraussetzungen

Keine Voraussetzungen

## Lernziele

In diesem Baustein werden die Grundzüge der Programmierung mit Python vermittelt. In diesem Baustein lernen Sie ...

- Grundlagen des Programmierens
- Ausgaben in Python, Grundlegende Datentypen, Flusskontrolle

- die Dokumentation zu lesen und zu verwenden
- Module und Pakete laden

# 1 Einführung



Figure 1.1: Logo der Programmiersprache Python

Python Logo von Python Software Foundation steht unter der [GPLv3](#). Die Wort-Bild-Marke ist markenrechtlich geschützt: <https://www.python.org/psf/trademarks/>. Das Werk ist abrufbar auf [wikimedia](#). 2008

# 2 Willkommen bei Python!

Python ist eine moderne Programmiersprache, die sich besonders gut für Einsteigerinnen und Einsteiger eignet. Sie ist leicht verständlich und wird in vielen Bereichen eingesetzt – von der Datenanalyse bis hin zur Webentwicklung.

In diesem Kurs lernen Sie Python Schritt für Schritt anhand praktischer Beispiele.

## 2.1 Lernziele dieses Kapitels

Am Ende dieses Kapitels können Sie:

- einfache Python-Programme schreiben,
- Text auf dem Bildschirm ausgeben,
- erste Variablen definieren und verwenden.

## 2.2 Ihr erstes Programm

Die ersten Schritte in einer neuen Programmiersprache sind immer die gleichen. Wir lassen uns die Worte ‘Hello World’ ausgeben. Dazu nutzen wir den print-Befehl `print()`:

```
print("Hallo Welt!")
```

Hallo Welt!

**Was passiert hier?** - `print()` ist eine sogenannte **Funktion**, die etwas auf dem Bildschirm ausgibt. - Der Text "Hello World!" wird angezeigt. - Texte (auch „Strings“ genannt) stehen immer in Anführungszeichen.

## 2.3 Variablen – Namen für Werte

Variablen sind wie beschriftete Schubladen: Sie speichern Informationen unter einem Namen.

```
name = "Frau Müller"  
alter = 32
```

Sie können diese Variablen verwenden, um dynamische Ausgaben zu erzeugen:

```
print(name + " ist " + str(alter) + " Jahre alt.")
```

Frau Müller ist 32 Jahre alt.

Zu beachten ist hier, dass sie versuchen sowohl eine Zahl, als auch Text auszugeben. Daher müssen wir mit der Funktion ‘str()’ die Zahl in Text umwandeln.

### Aufgabe: Begrüßung mit Alter

Schreiben Sie ein Programm, das Sie mit Ihrem Namen begrüßt:

Hello Frau Müller!

Tipp: In Python können Sie Texte mit + zusammenfügen. Denken Sie daran, dass Strings in Anführungszeichen stehen müssen.

### Lösung

```
mein_name = "Ihr Name hier"  
print("Hallo " + mein_name + "!")
```

Hello Ihr Name hier!

Erweitern Sie Ihr Programm so, dass es eine Begrüßung inklusive Alter ausgibt:

Hello Frau Müller!  
Sie sind 32 Jahre alt.

Tipp: Verwenden Sie print() mehrmals oder fügen Sie Texte zusammen.

### Lösung

```
name = "Frau Müller"  
alter = 32  
  
print("Hallo " + name + "!")  
print("Sie sind " + str(alter) + " Jahre alt.")
```

Hallo Frau Müller!  
Sie sind 32 Jahre alt.

# 3 Datentypen verstehen

## 3.1 Lernziele dieses Kapitels

Am Ende dieses Kapitels können Sie:

- die wichtigsten Datentypen unterscheiden,
- mit Zahlen und Texten rechnen bzw. arbeiten,
- einfache Berechnungen und Ausgaben erstellen.

## 3.2 Einleitung

In Python gibt es verschiedene **Datentypen**. Diese beschreiben, **welche Art von Daten** Sie in Variablen speichern. Das ist wichtig, weil viele Operationen – wie zum Beispiel + – je nach Datentyp etwas anderes bedeuten:

- + bei Zahlen bedeutet **Addition**,
- + bei Text bedeutet **Zusammenfügen** (Konkatenation).

Bevor wir also mit komplexeren Programmen arbeiten, sollten wir verstehen, welche Datentypen es gibt und wie man mit ihnen umgeht.

## 3.3 Die wichtigsten Datentypen

Hier sind die grundlegenden Datentypen in Python:

Typ	Beispiel	Bedeutung
int	10	Ganze Zahl
float	3.14	Kommazahl
str	"Hallo"	Text (String)
bool	True, False	Wahrheitswert (Ja/Nein)

Sie können den Typ einer Variable mit der Funktion `type()` herausfinden:

```
wert = 42
print(type(wert)) # Ausgabe: <class 'int'>
```

```
<class 'int'>
```

### 3.4 Unterschiede zwischen int und float

In Python unterscheidet man zwischen **ganzen Zahlen** (`int`) und **Kommazahlen** (`float`):

- `int` steht für „integer“ – also ganze Zahlen wie 1, 0, -10
- `float` steht für „floating point number“ – also Zahlen mit Dezimalstellen wie 3.14, 0.5, -2.0

```
a = 10      # int
b = 2.5     # float

print("a:", a, "| Typ:", type(a))
print("b:", b, "| Typ:", type(b))
```

```
a: 10 | Typ: <class 'int'>
b: 2.5 | Typ: <class 'float'>
```

#### Important

Die Unterscheidung ist wichtig: Manche Rechenoperationen verhalten sich je nach Datentyp leicht unterschiedlich.

### 3.5 Was sind Booleans (`bool`)?

Ein **Boolean** ist ein Wahrheitswert: Er kann nur zwei Zustände annehmen:

- `True` (wahr)
- `False` (falsch)

Solche Werte begegnen uns zum Beispiel bei Fragen wie:

- Ist die Temperatur über 30 °C?
- Hat die Datei einen bestimmten Namen?
- Ist die Liste leer?

```

ist_sonnig = True
hat_regenschirm = False

print("Sonnig:", ist_sonnig)
print("Regenschirm dabei?", hat_regenschirm)
print("Typ von 'ist_sonnig':", type(ist_sonnig))

```

```

Sonnig: True
Regenschirm dabei? False
Typ von 'ist_sonnig': <class 'bool'>

```

Booleans werden besonders in **Bedingungen** und **Vergleichen** verwendet, was Sie in Kapitel 4 genauer kennenlernen.

## 3.6 Rechnen mit Zahlen

Python kann wie ein Taschenrechner verwendet werden:

Operator	Beschreibung
+, -	Addition / Subtraktion
*, /	Multiplikation / Division
//, %	Ganzzahlige Division / Rest
**	Potenzieren

```

a = 10
b = 3

print("Addition:", a + b)
print("Subtraktion:", a - b)
print("Multiplikation:", a * b)
print("Potenzieren", a**b)
print("Division:", a / b)
print("Ganzzahlige Division:", a // b)
print("Division mit Rest:", a % b)

```

```

Addition: 13
Subtraktion: 7
Multiplikation: 30

```

```
Potenzieren 1000
Division: 3.333333333333335
Ganzzahlige Division: 3
Division mit Rest: 1
```

### i Note

```
// bedeutet: Ganzzahldivision, das Ergebnis wird abgerundet. Alternativ gibt es auch %.
Hier wird eine Ganzzahldivision durchgeführt und der Rest ausgegeben.
```

## 3.7 Arbeiten mit Text

Texte (Strings) können miteinander kombiniert werden:

```
vorname = "Anna"
nachname = "Beispiel"
print("Willkommen, " + vorname + " " + nachname + "!")
```

```
Willkommen, Anna Beispiel!
```

Wenn Sie Text und Zahlen kombinieren wollen, müssen Sie die Zahl in einen String umwandeln:

```
punkte = 95
print("Sie haben " + str(punkte) + " Punkte erreicht.")
```

```
Sie haben 95 Punkte erreicht.
```

## 3.8 Umwandlung von Datentypen (Typecasting)

Manchmal müssen Sie einen Wert von einem Datentyp in einen anderen umwandeln – z. B. eine Zahl in einen Text (String), damit sie ausgegeben werden kann.

Das nennt man **Typecasting**. Hier sind die wichtigsten Funktionen dafür:

Funktion	Beschreibung	Beispiel
str()	Zahl → Text	str(42) → "42"

Funktion	Beschreibung	Beispiel
<code>int()</code>	Text/Zahl → ganze Zahl	<code>int("10") → 10</code>
<code>float()</code>	Text/Zahl → Kommazahl	<code>float("3.14") → 3.14</code>

```
# Beispiel: Zahl als Text anzeigen
punkte = 100
print("Sie haben " + str(punkte) + " Punkte.")

# Beispiel: String in Zahl umwandeln und berechnen
eingabe = "3.5"
wert = float(eingabe) * 2
print("Doppelt so viel:", wert)
```

Sie haben 100 Punkte.  
Doppelt so viel: 7.0

Achten Sie beim Umwandeln darauf, dass der Inhalt auch wirklich passt – `int("abc")` führt zu einem Fehler.

### 💡 Aufgabe: Alter in Tagen

Berechnen Sie, wie alt eine Person in Tagen ist.

```
alter_jahre = 32
tage = alter_jahre * 365
print("Sie sind ungefähr " + str(tage) + " Tage alt.")
```

Sie sind ungefähr 11680 Tage alt.

Tipp: Denken Sie an die Umwandlung in einen String, wenn Sie die Zahl ausgeben möchten.

### Lösung

```
alter = 32
tage = alter * 365
print("Sie sind ungefähr " + str(tage) + " Tage alt.")
```

Sie sind ungefähr 11680 Tage alt.

# 4 Entscheidungen und Wiederholungen

Programme müssen oft Entscheidungen treffen – zum Beispiel abhängig von einer Benutzereingabe oder einem bestimmten Wert. Ebenso müssen bestimmte Aktionen mehrfach durchgeführt werden.

Dafür gibt es zwei zentrale Elemente in Python:

- Kontrollstrukturen: `if`, `elif`, `else`
- Schleifen: `while` und `for`

## 4.1 Lernziele dieses Kapitels

Am Ende dieses Kapitels können Sie:

- Bedingungen formulieren und mit `if`, `elif`, `else` nutzen,
- Vergleichsoperatoren verwenden (`==`, `<`, `!=`, ...),
- Wiederholungen mit `while` und `for` umsetzen.

## 4.2 Bedingungen mit `if`, `elif`, `else`

```
alter = 17

if alter >= 18:
    print("Sie sind volljährig.")
else:
    print("Sie sind minderjährig.")
```

Sie sind minderjährig.

Mehrere Fälle unterscheiden:

```

note = 2.3

if note <= 1.5:
    print("Sehr gut")
elif note <= 2.5:
    print("Gut")
elif note <= 3.5:
    print("Befriedigend")
else:
    print("Ausreichend oder schlechter")

```

Gut

### 4.3 Vergleichsoperatoren

Ausdruck	Bedeutung
a == b	gleich
a != b	ungleich
a < b	kleiner als
a > b	größer als
a <= b	kleiner oder gleich
a >= b	größer oder gleich

### 4.4 Wiederholungen mit while

```

zähler = 0

while zähler < 5:
    print("Zähler ist:", zähler)
    zähler += 1

```

Zähler ist: 0  
Zähler ist: 1  
Zähler ist: 2  
Zähler ist: 3  
Zähler ist: 4

### Important

Achten Sie auf eine Abbruchbedingung – sonst läuft die Schleife endlos!

## 4.5 Schleifen mit `for` und `range()`

Wenn Sie eine Schleife **genau eine bestimmte Anzahl von Malen** durchlaufen möchten, nutzen Sie `for` mit `range()`:

```
for i in range(5):
    print("Durchlauf:", i)
```

Durchlauf: 0  
Durchlauf: 1  
Durchlauf: 2  
Durchlauf: 3  
Durchlauf: 4

Start- und Endwert festlegen:

```
for i in range(1, 6):
    print(i)
```

1  
2  
3  
4  
5

## 4.6 Was macht `range()` genau?

Die Funktion `range()` erzeugt eine Abfolge von Zahlen, über die Sie mit einer `for`-Schleife iterieren können.

### 4.6.1 Varianten:

```
range(5)
```

ergibt: 0, 1, 2, 3, 4 (startet bei 0, endet **vor** 5)

```
range(2, 6)
```

ergibt: 2, 3, 4, 5 (startet bei 2, endet **vor** 6)

```
range(1, 10, 2)
```

ergibt: 1, 3, 5, 7, 9 (Schrittweite = 2)

`range()` erzeugt keine echte Liste, sondern ein sogenanntes „range-Objekt“, das wie eine Liste verwendet werden kann.

 Aufgabe: Zähle von 1 bis 10

Nutzen Sie eine `for`-Schleife, um die Zahlen von 1 bis 10 auszugeben.

**Lösung**

```
for i in range(1, 11):
    print(i)
```

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

 Aufgabe: Gerade Zahlen ausgeben

Geben Sie alle geraden Zahlen von 0 bis 20 aus. Tipp: Eine Zahl ist gerade, wenn `zahl % 2 == 0`.

**Lösung**

```
for zahl in range(0, 21):
    if zahl % 2 == 0:
        print(zahl)
```

0  
2  
4  
6  
8  
10  
12  
14  
16  
18  
20

# 5 Mehrere Werte speichern

Bisher haben Sie einzelne Werte in Variablen gespeichert. Doch was, wenn Sie eine ganze Reihe von Zahlen, Namen oder Werten auf einmal speichern möchten?

Dafür gibt es in Python **Listen**. In diesem Kapitel lernen Sie außerdem, wie man mit **for**-Schleifen über Listen iteriert.

## 5.1 Was ist eine Liste?

Eine Liste ist eine geordnete Sammlung von Werten eines Datentyps.

```
namen = ["Ali", "Bente", "Carlos"]
noten = [1.7, 2.3, 1.3, 2.0]
```

Auf Elemente greifen Sie mit eckigen Klammern zu:

```
print(namen[0]) # erstes Element
print(noten[-1]) # letztes Element
```

```
Ali
2.0
```

## 5.2 Teile aus Listen ausschneiden – Slicing

Mit dem sogenannten **Slicing** können Sie gezielt Ausschnitte aus einer Liste entnehmen. Dabei geben Sie an, **wo der Ausschnitt beginnt und wo er endet** (der Endwert wird **nicht** mehr mitgenommen):

```
zahlen = [10, 20, 30, 40, 50, 60]
print(zahlen[1:4]) # Ausgabe: [20, 30, 40]
```

```
[20, 30, 40]
```

### 5.2.1 Syntax: liste[start:stop]

- **start**: Index, bei dem das Slicing beginnt (inklusive)
- **stop**: Index, an dem es endet (exklusiv)
- Der Startwert kann auch weggelassen werden: [:3] → erstes bis drittes Element
- Ebenso der Endwert: [3:] → ab dem vierten Element bis zum Ende
- Ganze Kopie: [:]

```
print(zahlen[:3])    # [10, 20, 30]
print(zahlen[3:])    # [40, 50, 60]
print(zahlen[:])     # vollständige Kopie
```

```
[10, 20, 30]
[40, 50, 60]
[10, 20, 30, 40, 50, 60]
```

#### Note

Sie können auch mit negativen Indizes arbeiten (-1 ist das letzte Element):

```
print(zahlen[-3:])  # [40, 50, 60]
[40, 50, 60]
```

## 5.3 Über Listen iterieren

Mit einer **for**-Schleife können Sie über jedes Element in einer Liste iterieren:

```
namen = ["Ali", "Bente", "Carlos"]

for name in namen:
    print("Hallo", name + "!")
```

```
Hallo Ali!
Hallo Bente!
Hallo Carlos!
```

## 5.4 Erweiterung: Bedingte Ausgaben

Sie können in der Schleife mit `if` filtern:

```
temperaturen = [14.2, 17.5, 19.0, 21.3, 18.4]

for t in temperaturen:
    if t > 18:
        print(t, "ist ein warmer Tag")
```

```
19.0 ist ein warmer Tag
21.3 ist ein warmer Tag
18.4 ist ein warmer Tag
```

## 5.5 Durchschnitt berechnen

Python stellt nützliche Funktionen bereit, z.B. `sum()` und `len()`:

```
noten = [1.7, 2.3, 1.3, 2.0]

durchschnitt = sum(noten) / len(noten)
print("Durchschnittsnote:", round(durchschnitt, 2))
```

```
Durchschnittsnote: 1.82
```

## 5.6 Listen erweitern: `.append()`

Manchmal kennen Sie die Listenelemente nicht vorher – dann können Sie neue Werte **nachträglich hinzufügen**:

```
namen = []

namen.append("Ali")
namen.append("Bente")

print(namen)
```

```
['Ali', 'Bente']
```

### Note

Die Methode `.append()` hängt einen neuen Wert an das Ende der Liste.

## 5.7 Verschachtelte Schleifen

Wenn Sie mit **mehrdimensionalen Daten** arbeiten – z. B. eine Tabelle mit mehreren Zeilen – können Sie Schleifen **ineinander verschachteln**:

```
wochentage = ["Mo", "Di", "Mi"]
stunden = [1, 2, 3]

for tag in wochentage:
    for stunde in stunden:
        print(f"{tag}, Stunde {stunde}")
```

```
Mo, Stunde 1
Mo, Stunde 2
Mo, Stunde 3
Di, Stunde 1
Di, Stunde 2
Di, Stunde 3
Mi, Stunde 1
Mi, Stunde 2
Mi, Stunde 3
```

Das ergibt:

```
Mo, Stunde 1
Mo, Stunde 2
Mo, Stunde 3
Di, Stunde 1
...

```

## 5.8 Listen sortieren

Mit `sorted()` können Sie Listen **alphabetisch oder numerisch sortieren**:

```
namen = ["Zoe", "Anna", "Ben"]
sortiert = sorted(namen)

print(sortiert)
```

```
['Anna', 'Ben', 'Zoe']
```

### Important

Die Original-Liste bleibt **unverändert**.

Wenn Sie die Liste direkt verändern möchten, geht das mit:

```
namen.sort()
```

# 6 Wiederverwendbarer Code mit Funktionen

Stellen Sie sich vor, Sie müssen eine bestimmte Berechnung mehrfach im Programm durchführen. Anstatt den Code jedes Mal neu zu schreiben, können Sie ihn in einer **Funktion** bündeln.

Funktionen sind ein zentrales Werkzeug, um Code:

- übersichtlich,
- wiederverwendbar und
- testbar zu machen.

## 6.1 Lernziele dieses Kapitels

Am Ende dieses Kapitels können Sie:

- eigene Funktionen mit `def` erstellen,
- Parameter übergeben und Rückgabewerte nutzen,
- Funktionen sinnvoll in Programmen einsetzen.

## 6.2 Eine Funktion definieren

Eine Funktion besteht aus folgenden Teilen:

1. **Definition mit `def`**
2. **Funktionsname**
3. **Parameter in Klammern (optional)**
4. **Einrückung für den Funktionskörper**
5. (optional) **`return`-Anweisung**

Beispiel:

```
def hallo(name="Gast"):  
    begruessung = "Hallo " + name + "!"  
    return begruessung
```

Fangen wir mit dem ersten Stichwort an. Funktionen werden mit `def` definiert und können beliebig oft aufgerufen werden:

```
def begruessung():
    print("Hallo und willkommen!")
```

Sie wird erst ausgeführt, wenn Sie sie aufrufen:

```
begruessung()
```

```
Haloo und willkommen!
```

## 6.3 Parameter übergeben

Funktionen können Eingabewerte (Parameter) erhalten:

```
def begruessung(name):
    print("Hallo", name + "!")

begruessung("Alex")
```

```
Haloo Alex!
```

## 6.4 Rückgabewerte mit `return`

Eine Funktion kann auch einen Wert **zurückgeben**:

```
def quadrat(zahl):
    return zahl * zahl

ergebnis = quadrat(5)
print(ergebnis)
```

## 6.5 Beispiel: Umrechnungen

### 6.5.1 Euro zu US-Dollar

```
def euro_zu_usd(betrag_euro):
    wechselkurs = 1.09
    return betrag_euro * wechselkurs

print("20 € sind", euro_zu_usd(20), "US-Dollar.")
```

20 € sind 21.8 US-Dollar.

#### 💡 Aufgabe: Begrüßung mit Name

Erstellen Sie eine Funktion `begruesse(name)`, die den Namen in einem Begrüßungstext verwendet:

```
Hallo Fatima, schön dich zu sehen!
```

#### Lösung

```
def begruesse(name):
    print("Hallo", name + ", schön dich zu sehen!")

begruesse("Fatima")
```

```
Hallo Fatima, schön dich zu sehen!
```

#### 💡 Aufgabe: Temperaturumrechnung

Schreiben Sie eine Funktion, die Celsius in Fahrenheit umrechnet:  
Formel: [  $F = C \times 1.8 + 32$  ]

#### Lösung

```
def celsius_zu_fahrenheit(c):
    return c * 1.8 + 32

print(celsius_zu_fahrenheit(20))
```

## 6.6 Parameter mit Standardwerten

Sie können Parametern **Standardwerte** zuweisen. So kann die Funktion auch ohne Angabe eines Werts aufgerufen werden:

```
def begruessung(name="Gast"):
    print("Hallo", name + "!")

begruessung()          # Hallo Gast!
begruessung("Maria")   # Hallo Maria!
```

Hallo Gast!  
Hallo Maria!

### print() vs. return

Diese beiden Begriffe werden oft verwechselt:

Ausdruck	Bedeutung
print()	zeigt einen Text auf dem Bildschirm
return	gibt einen Wert an den Aufrufer zurück

Beispiel:

```
def verdoppeln(x):
    return x * 2

# Ausgabe sichtbar machen
print(verdoppeln(5))  # Ausgabe: 10
```

10

# 7 Arbeiten mit Dateien

Programme arbeiten oft nicht nur mit Benutzereingaben, sondern auch mit **Textdateien** – zum Beispiel um Daten zu speichern oder zu laden.

Python bietet einfache Funktionen, um:

- Dateien **zu öffnen**,
- ihren **Inhalt zu lesen** oder **hineinzuschreiben**,
- und die Datei **wieder zu schließen**.

## 7.1 Lernziele dieses Kapitels

Am Ende dieses Kapitels können Sie:

- Dateien mit `open()` öffnen,
- Inhalte aus Textdateien einlesen,
- Texte in Dateien schreiben,
- mit `with`-Blöcken sicher und einfach arbeiten.

## 7.2 Eine Datei einlesen

```
# Beispiel: Datei lesen
with open("01-daten/beispiel.txt", "r") as datei:
    inhalt = datei.read()
    print(inhalt)
```

Dies ist ein Test.

- "r" steht für `read` (lesen).
- `with` sorgt dafür, dass die Datei nach dem Lesen automatisch geschlossen wird.
- `read()` liest den **gesamten Inhalt** der Datei als String.

## 7.3 Zeilenweise lesen

```
with open("01-daten/beispiel.txt", "r") as datei:  
    for zeile in datei:  
        print("Zeile:", zeile.strip())
```

Zeile: Dies ist ein Test.

### Note

.strip() entfernt Leerzeichen und Zeilenumbrüche am Anfang und Ende.

### Aufgabe: Datei lesen

Angenommen, es gibt eine Datei `gruesse.txt` mit folgendem Inhalt:

```
Hallo Anna  
Guten Morgen Ben  
Willkommen Carla
```

Schreiben Sie ein Programm, das jede Zeile einzeln einliest und mit `print(...)` wiedergibt.

### Lösung

```
with open("01-daten/gruesse.txt", "r") as f:  
    for zeile in f:  
        print(zeile.strip())
```

```
Hallo Anna  
Guten Morgen Ben  
Willkommen Carla
```

## 7.4 Alle Zeilen auf einmal lesen mit `readlines()`

Statt über eine Datei zu iterieren, können Sie alle Zeilen auf einmal als Liste einlesen:

```
with open("01-daten/beispiel.txt", "r") as f:  
    zeilen = f.readlines()  
    print(zeilen)
```

```
['Dies ist ein Test.]
```

### ! Important

Oftmals bestehen die eingelesenen Zeilen aus Werten, die durch ein spezifisches Trennzeichen z.B. „;“. Um diese Zeilen dann in die einzelnen Werte zu trennen, benutzen wir die `.split()`-Funktion.

```
zeile = "319,12,14,190,342"  
print(zeile.split(","))
```

```
['319', '12', '14', '190', '342']
```

Jede Zeile endet mit `\n`, deshalb kann eine Nachbearbeitung mit `.strip()` sinnvoll sein:

```
for zeile in zeilen:  
    print(zeile.strip())
```

```
Dies ist ein Test.
```

## 7.5 In eine Datei schreiben

```
with open("ausgabe.txt", "w") as datei:  
    datei.write("Das ist eine neue Zeile.\n")  
    datei.write("Und noch eine.")
```

- `"w"` steht für `write` (schreiben).
- Achtung: Eine vorhandene Datei wird überschrieben!

## 7.6 Zeilenweise schreiben mit Schleife

```
daten = ["Apfel", "Banane", "Kirsche"]

with open("obst.txt", "w") as f:
    for eintrag in daten:
        f.write(eintrag + "\n")
```

### ! Important

Jede Zeile endet mit \n für einen Zeilenumbruch.

### 💡 Aufgabe: Liste in Datei schreiben

Gegeben ist eine Liste von Städten:

```
staedte = ["Berlin", "Hamburg", "München"]
```

- Schreiben Sie ein Programm, das jede Stadt in eine neue Zeile einer Datei staedte.txt schreibt.

### Lösung

```
staedte = ["Berlin", "Hamburg", "München"]

with open("staedte.txt", "w") as f:
    for stadt in staedte:
        f.write(stadt + "\n")
```

## 7.7 Dateien manuell schließen mit close()

Wenn Sie **keinen with-Block** verwenden, müssen Sie die Datei selbst schließen – sonst bleibt sie geöffnet:

```
datei = open("01-daten/beispiel.txt", "w")
datei.write("Dies ist ein Test.")
datei.close()
```

! Important

`close()` ist wichtig, damit Änderungen gespeichert werden und die Datei nicht gesperrt bleibt.

**Empfehlung:** Nutzen Sie immer `with open(...)`, da Python die Datei dann automatisch schließt – auch bei Fehlern.

# 8 Lernzielkontrolle

## 8.1 Aufgabe 1: Datentypen und Typecasting

Gegeben sind folgende Werte:

- `wert1 = "42"`
- `wert2 = 3.5`
- `wert3 = True`

Gib mit `type()` den Datentyp jedes Werts aus und wandle jeden in einen anderen sinnvollen Typ um. Gib die Ergebnisse aus.

---

## 8.2 Aufgabe 2: Altersprüfung mit Bedingung

Gegeben sei `alter = 16`.

Gib abhängig vom Alter aus:

- „Volljährig“, wenn das Alter 18 ist
  - „Minderjährig“, wenn das Alter < 18 ist
- 

## 8.3 Aufgabe 3: Zahlen filtern und mitteln

Gegeben sei die Liste:

```
zahlen = [5, 8, 13, 20, 33, 40]
```

- Gib alle Zahlen > 10 aus.
  - Berechne und gib den Durchschnitt dieser Zahlen aus.
-

## 8.4 Aufgabe 4: Wochentage & Slicing

Gegeben sei:

```
tage = ["Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"]
```

- Gib alle Wochentage aus, die mit „S“ beginnen.
  - Gib nur die Arbeitstage (Mo–Fr) mit Slicing aus.
- 

## 8.5 Aufgabe 5: Funktion mit Parameter & Standardwert

Erstelle eine Funktion `begruesse(name, sprache="de")`, die je nach Sprache Folgendes ausgibt:

- Deutsch: „Hallo !“
  - Englisch: „Hello !“
- 

## 8.6 Aufgabe 6: Hobbys in Datei schreiben

Gegeben sei eine Liste `hobbies = ["Lesen", "Kochen", "Sport"]`.

Schreibe jeden Eintrag in eine neue Zeile der Datei `hobbies.txt`.

---

## 8.7 Aufgabe 7: Datei lesen & `.split()` verwenden

Angenommen, eine Datei enthält die Zeile:

Ali,Bente,Carlos,Dana

- Zerlege die Zeichenkette mit `.split(",")`.
  - Gib jeden Namen einzeln aus.
-

## 8.8 Aufgabe 8: Namen sortieren und speichern

Gegeben sei:

```
namen = ["Zoe", "Anna", "Lukas", "Ben"]
```

- Sortiere die Liste alphabetisch.
  - Speichere die sortierte Liste in eine Datei `sortiert.txt`, ein Name pro Zeile.
- 

## 8.9 Aufgabe 9: Zahlen durch 3 ausgeben

Erstelle mit einer Schleife eine Liste aller Zahlen zwischen 1 und 20, die durch 3 teilbar sind, und gib sie aus.

---

## 8.10 Aufgabe 10: Verschachtelte Schleifen

Gegeben seien:

```
personen = ["Ali", "Bente"]
hobbys = ["Lesen", "Sport"]
```

Erstelle eine Ausgabe wie:

```
Ali hat das Hobby: Lesen
Ali hat das Hobby: Sport
Bente hat das Hobby: Lesen
Bente hat das Hobby: Sport
```

## **Part II**

# **Matplotlib**

Bausteine Computergestützter Datenanalyse. „Werkzeugbaustein Plotting in Python“ von Lukas Arnold, Simone Arnold, Florian Bagemihl, Matthias Baitsch, Marc Fehr, Maik Poetzsch und Sebastian Seipel ist lizenziert unter [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/). Das Werk ist abrufbar unter <https://github.com/bausteine-der-datenanalyse/w-python-plotting>. Ausgenommen von der Lizenz sind alle Logos und anders gekennzeichneten Inhalte. 2024



## Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Maik Poetzsch, und Sebastian Seipel. 2024. „Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Plotting in Python“. <https://github.com/bausteine-der-datenanalyse/w-python-plotting>.

## BibTeX-Vorlage

```
@misc{BCD-Styleguide-2024,  
    title={Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Plotting in Python},  
    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-plotting}}
```

## **Part III**

### **Intro**

## **Voraussetzungen**

- Grundlagen Python
- Einbinden von zusätzlichen Paketen

## **Verwendete Pakete und Datensätze**

- matplotlib

## **Bearbeitungszeit**

Geschätzte Bearbeitungszeit: 1h

## **Lernziele**

- Einleitung: wie visualisiere ich Daten in Python
- Anpassen von Plots
- Do's & Dont's für wissenschaftliche Plots

# 9 Einführung

Matplotlib ist eine der bekanntesten Bibliotheken zur Datenvisualisierung in Python. Sie ermöglicht das Erstellen statischer, animierter und interaktiver Diagramme mit hoher Flexibilität.

## 9.1 Warum Matplotlib?

- **Breite Unterstützung:** Funktioniert mit NumPy, Pandas und SciPy.
- **Hohe Anpassbarkeit:** Vollständige Kontrolle über Diagramme.
- **Integration in Jupyter Notebooks:** Ideal für interaktive Datenanalyse.
- **Kompatibilität:** Unterstützt verschiedene Ausgabeformate (PNG, SVG, PDF etc.).

## 9.2 Alternativen zu Matplotlib

Während Matplotlib leistungsstark ist, gibt es Alternativen, die für bestimmte Zwecke besser geeignet sein können:  
- **Seaborn:** Basiert auf Matplotlib, erleichtert statistische Visualisierung.  
- **Plotly:** Erzeugt interaktive Plots, gut für Dashboards.  
- **Bokeh:** Ideal für Web-Anwendungen mit interaktiven Visualisierungen.

## 9.3 Erstes Beispiel: Einfache Linie plotten

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

# Beispiel-Daten
t = np.linspace(0, 10, 100)
y = np.sin(t)

# Erstellen des Plots
plt.plot(t, y, label='sin(t)')
plt.xlabel('Zeit (s)')
```

```
plt.ylabel('Amplitude')
plt.title('Einfaches Linien-Diagramm')
plt.legend()
plt.show()
```

Dieses einfache Beispiel zeigt, wie man mit Matplotlib eine **Sinuskurve** visualisieren kann.

## 9.4 Nächste Schritte

Im nächsten Kapitel werden wir uns mit den verschiedenen Diagrammtypen beschäftigen, die Matplotlib bietet.

# 10 Diagrammtypen in Matplotlib

Matplotlib bietet eine Vielzahl von Diagrammtypen, die für unterschiedliche Zwecke geeignet sind. In diesem Kapitel werden die wichtigsten Diagrammtypen vorgestellt und ihre Anwendungsfälle erklärt.

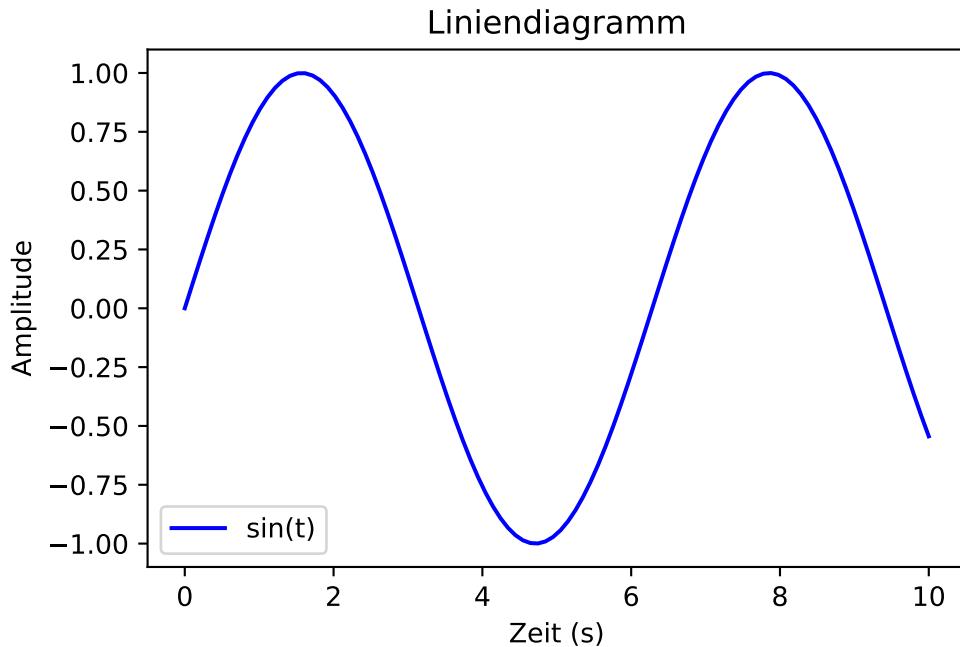
## 10.1 1. Liniendiagramme (`plt.plot()`)

Liniendiagramme eignen sich hervorragend zur Darstellung von Trends über Zeit.

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

t = np.linspace(0, 10, 100)
y = np.sin(t)

plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title('Liniendiagramm')
plt.legend()
plt.show()
```

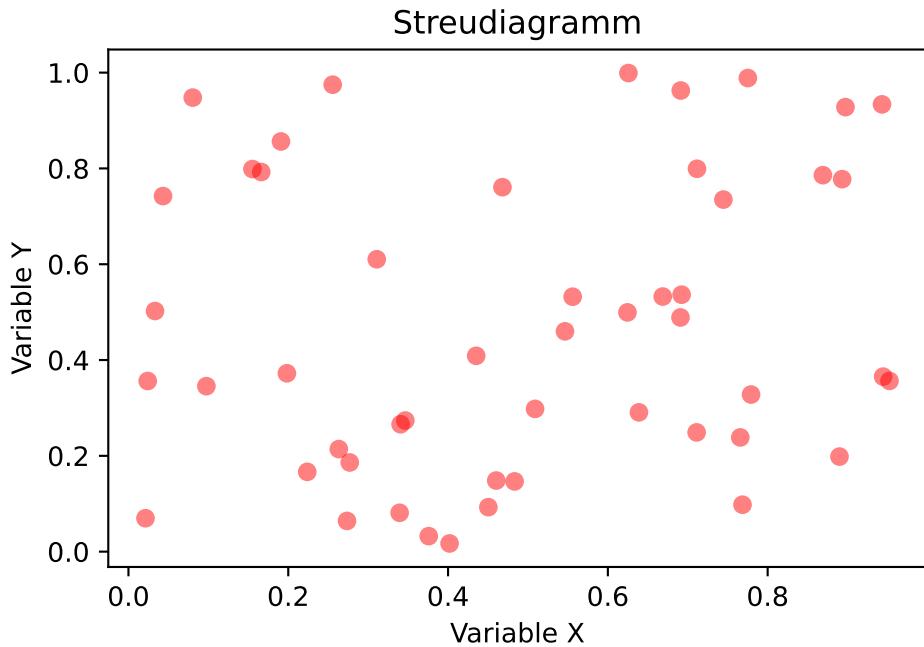


## 10.2 2. Streudiagramme (`plt.scatter()`)

Streudiagramme werden verwendet, um Zusammenhänge zwischen zwei Variablen darzustellen.

```
x = np.random.rand(50)
y = np.random.rand(50)

plt.scatter(x, y, color='r', alpha=0.5)
plt.xlabel('Variable X')
plt.ylabel('Variable Y')
plt.title('Streudiagramm')
plt.show()
```

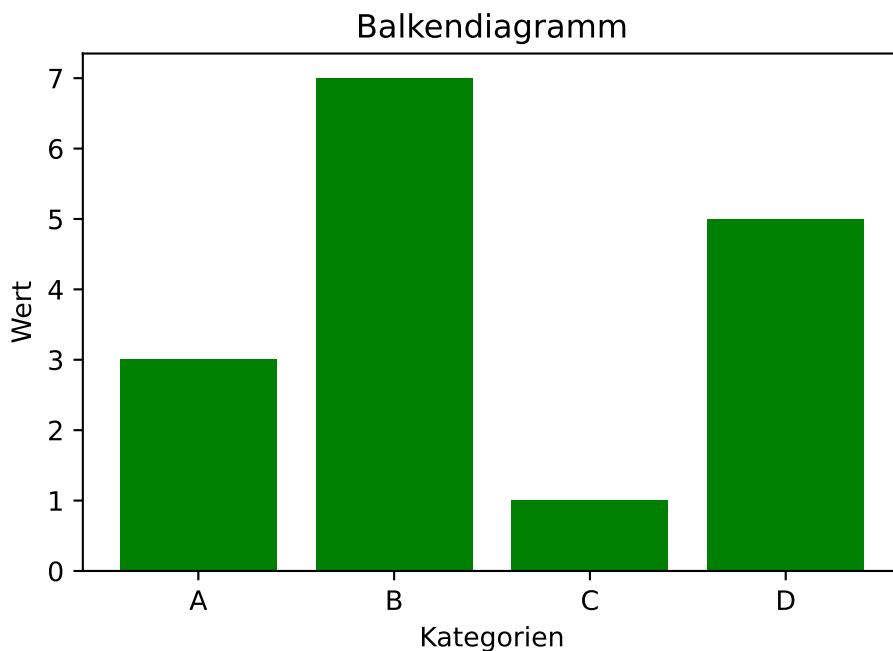


### 10.3 3. Balkendiagramme (plt.bar())

Balkendiagramme eignen sich zur Darstellung kategorialer Daten.

```
kategorien = ['A', 'B', 'C', 'D']
werte = [3, 7, 1, 5]

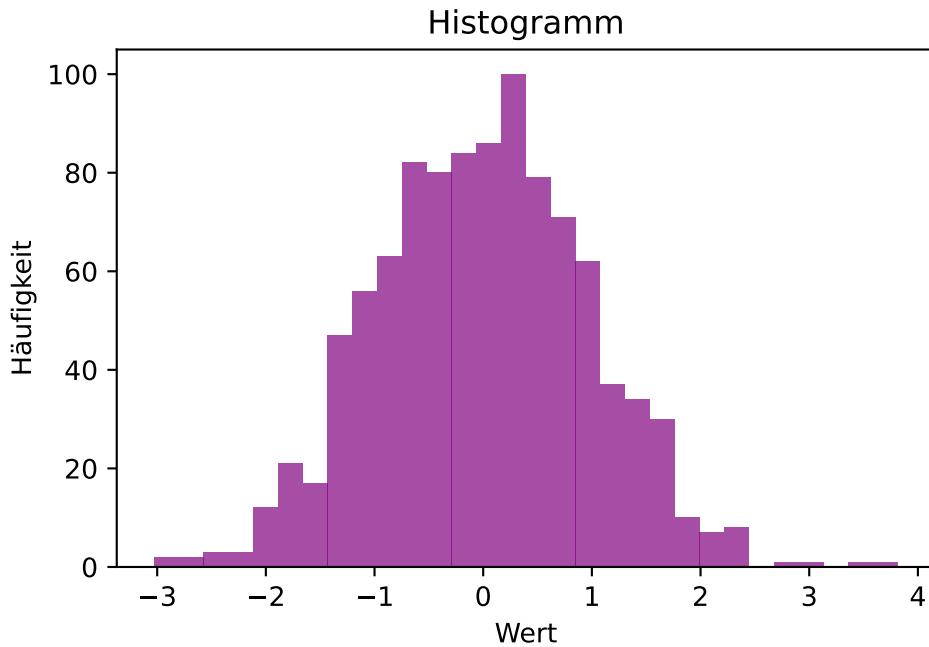
plt.bar(kategorien, werte, color='g')
plt.xlabel('Kategorien')
plt.ylabel('Wert')
plt.title('Balkendiagramm')
plt.show()
```



## 10.4 4. Histogramme (plt.hist())

Histogramme zeigen die Verteilung numerischer Daten.

```
daten = np.random.randn(1000)
plt.hist(daten, bins=30, color='purple', alpha=0.7)
plt.xlabel('Wert')
plt.ylabel('Häufigkeit')
plt.title('Histogramm')
plt.show()
```

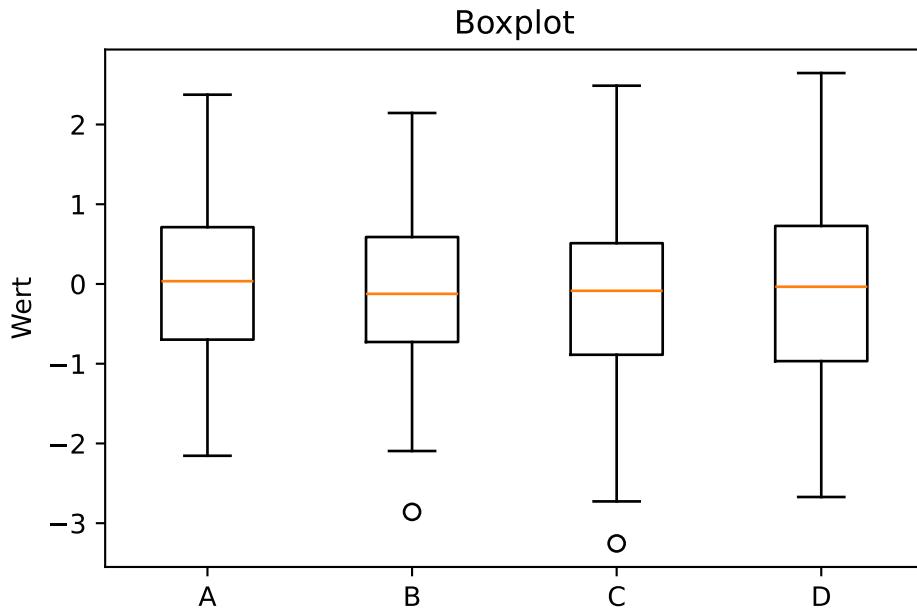


## 10.5 5. Boxplots (plt.boxplot())

Boxplots helfen, Ausreißer und die Verteilung von Daten zu visualisieren.

```
daten = [np.random.randn(100) for _ in range(4)]
plt.boxplot(daten, labels=['A', 'B', 'C', 'D'])
plt.ylabel('Wert')
plt.title('Boxplot')
plt.show()
```

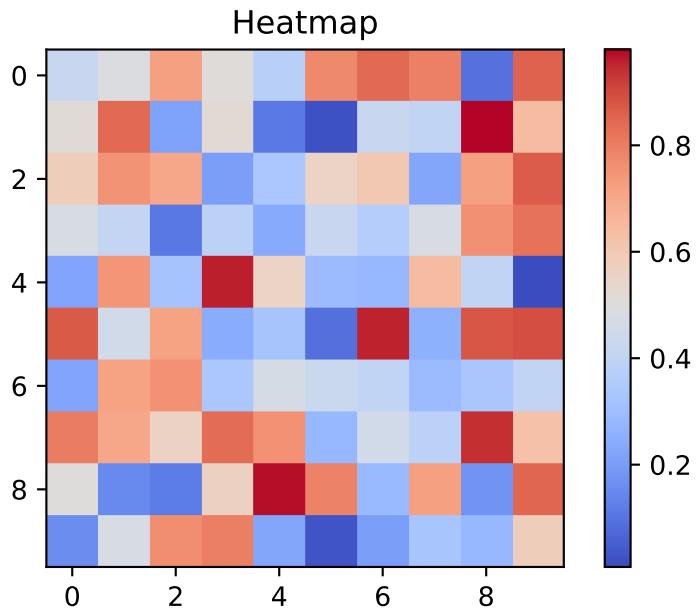
```
/var/folders/p_ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_20347/2728911591.py:2: Matplotlib
```



## 10.6 6. Heatmaps (plt.imshow())

Heatmaps eignen sich zur Darstellung von 2D-Daten.

```
daten = np.random.rand(10, 10)
plt.imshow(daten, cmap='coolwarm', interpolation='nearest')
plt.colorbar()
plt.title('Heatmap')
plt.show()
```



## 10.7 Fazit

Die Wahl des richtigen Diagrammtyps hängt von der Art der Daten und der gewünschten Darstellung ab. Im nächsten Kapitel werden wir uns mit der Anpassung und Gestaltung von Plots beschäftigen.

# 11 Anpassung und Gestaltung von Plots in Matplotlib

Ein gut gestaltetes Diagramm verbessert die Lesbarkeit und Verständlichkeit der dargestellten Daten. In diesem Kapitel werden wir verschiedene Möglichkeiten zur Anpassung und Gestaltung von Plots in Matplotlib erkunden.

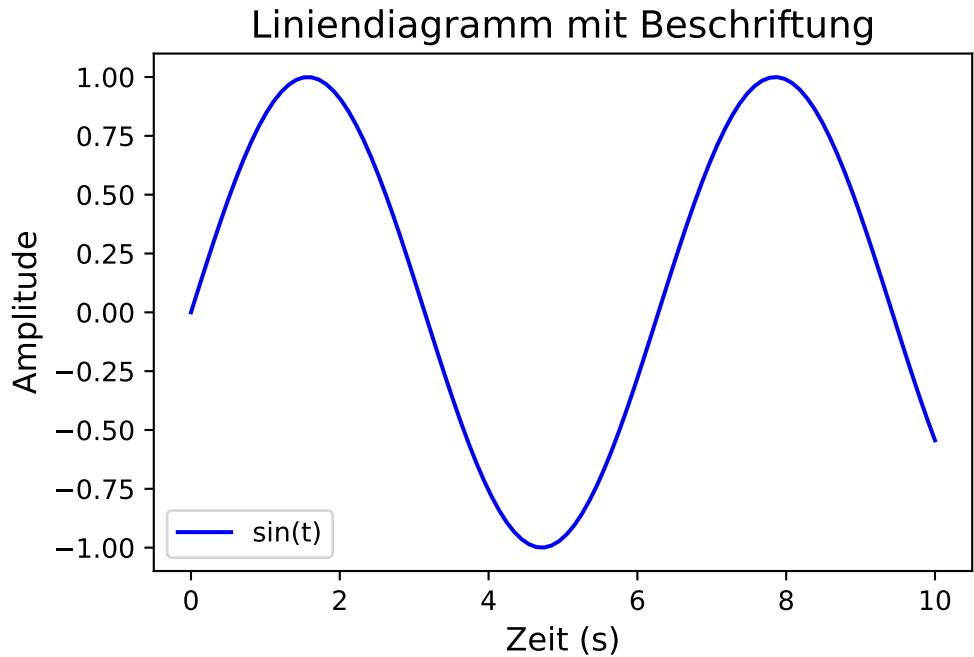
## 11.1 1. Achsentitel und Diagrammtitel

Klare Achsen- und Diagrammtitel sind essenziell für die Verständlichkeit eines Plots.

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

t = np.linspace(0, 10, 100)
y = np.sin(t)

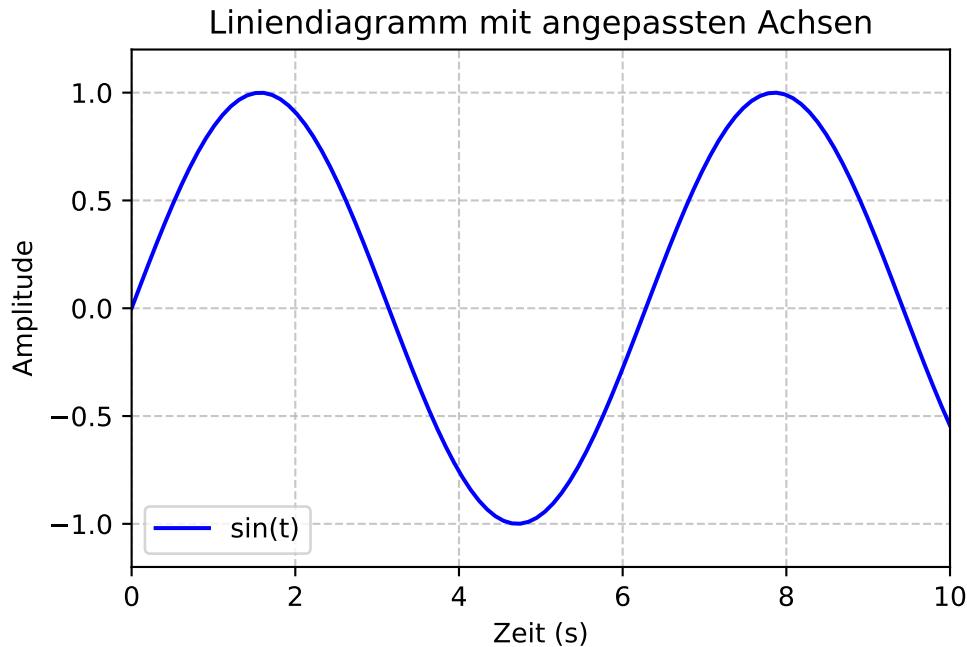
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)', fontsize=12)
plt.ylabel('Amplitude', fontsize=12)
plt.title('Liniendiagramm mit Beschriftung', fontsize=14)
plt.legend()
plt.show()
```



## 11.2 2. Anpassung der Achsen

Die Skalierung der Achsen sollte sinnvoll gewählt werden, um die Daten bestmöglich darzustellen.

```
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.xlim(0, 10)
plt.ylim(-1.2, 1.2)
plt.grid(True, linestyle='--', alpha=0.7)
plt.title('Liniendiagramm mit angepassten Achsen')
plt.legend()
plt.show()
```



### 11.3 3. Farben und Linienstile

Farben und Linienstile helfen dabei, wichtige Informationen im Plot hervorzuheben.

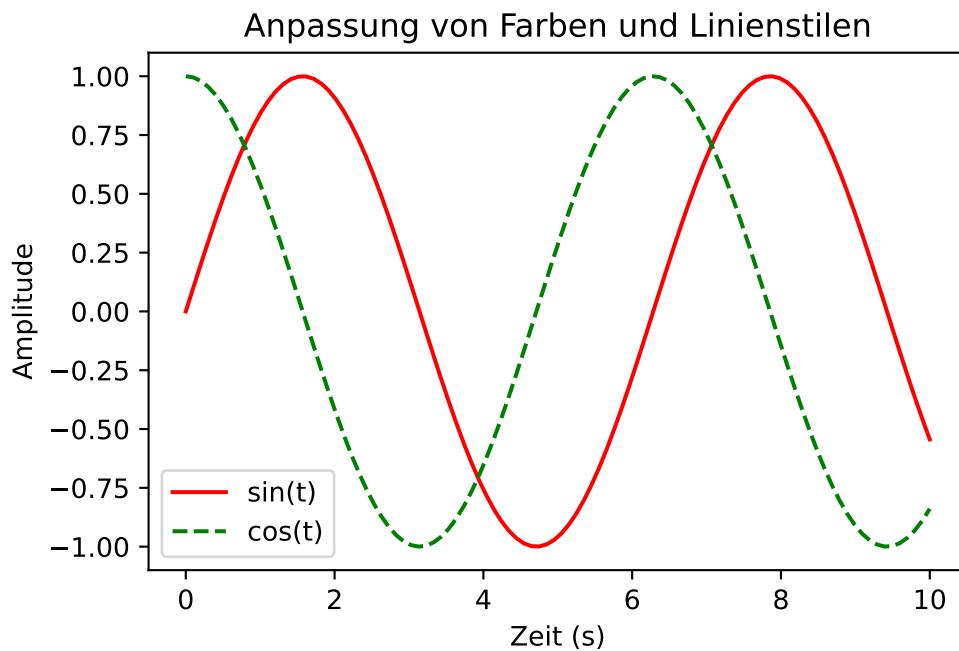
#### 11.3.1 Wichtige Farben (Standardfarben in Matplotlib)

Farbe	Kürzel	Beschreibung
Blau	'b'	blue
Grün	'g'	green
Rot	'r'	red
Cyan	'c'	cyan
Magenta	'm'	magenta
Gelb	'y'	yellow
Schwarz	'k'	black
Weiß	'w'	white

#### 11.3.2 Wichtige Linienstile

Linienstil	Kürzel	Beschreibung
Durchgezogen	'-'	Standardlinie
Gestrichelt	'--'	lange Striche
Gepunktet	'.'	nur Punkte
Strich-Punkt	'-.'	abwechselnd Strich-Punkt

```
plt.plot(t, np.sin(t), linestyle='-', color='r', label='sin(t)')
plt.plot(t, np.cos(t), linestyle='--', color='g', label='cos(t)')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title('Anpassung von Farben und Linienstilen')
plt.legend()
plt.show()
```

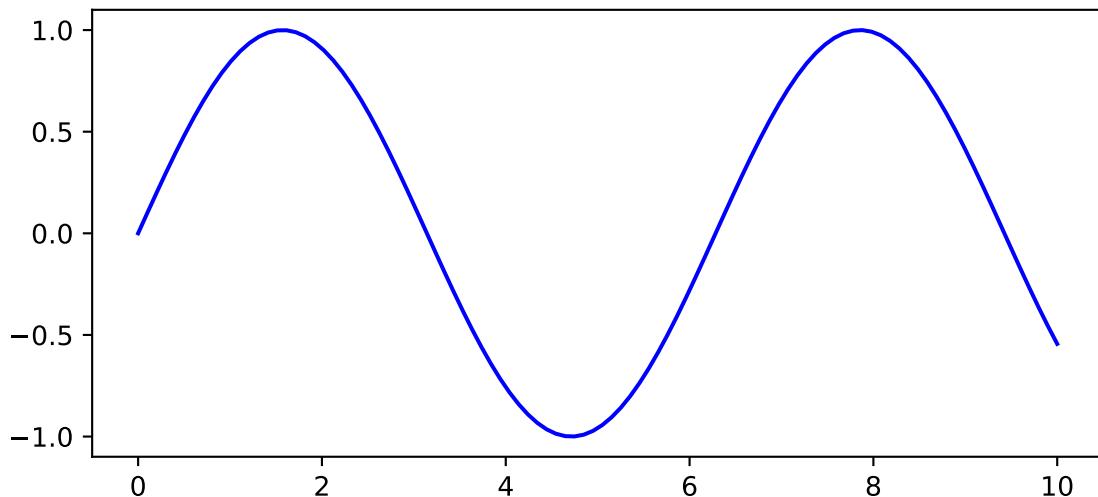


## 11.4 4. Mehrere Plots mit Subplots

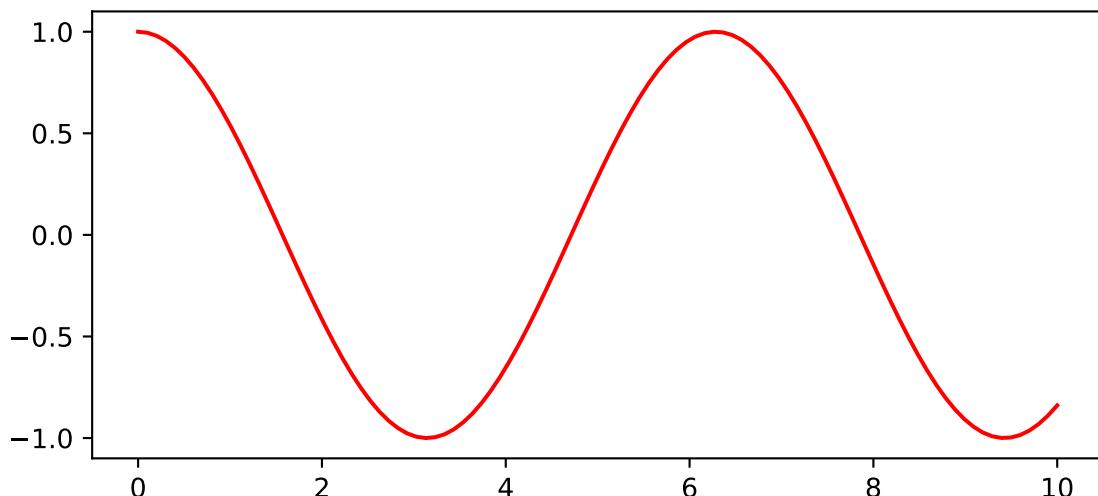
Manchmal ist es sinnvoll, mehrere Diagramme in einer Abbildung darzustellen.

```
fig, axs = plt.subplots(2, 1, figsize=(6, 6))
axs[0].plot(t, np.sin(t), color='b')
axs[0].set_title('Sinusfunktion')
axs[1].plot(t, np.cos(t), color='r')
axs[1].set_title('Kosinusfunktion')
plt.tight_layout()
plt.show()
```

Sinusfunktion



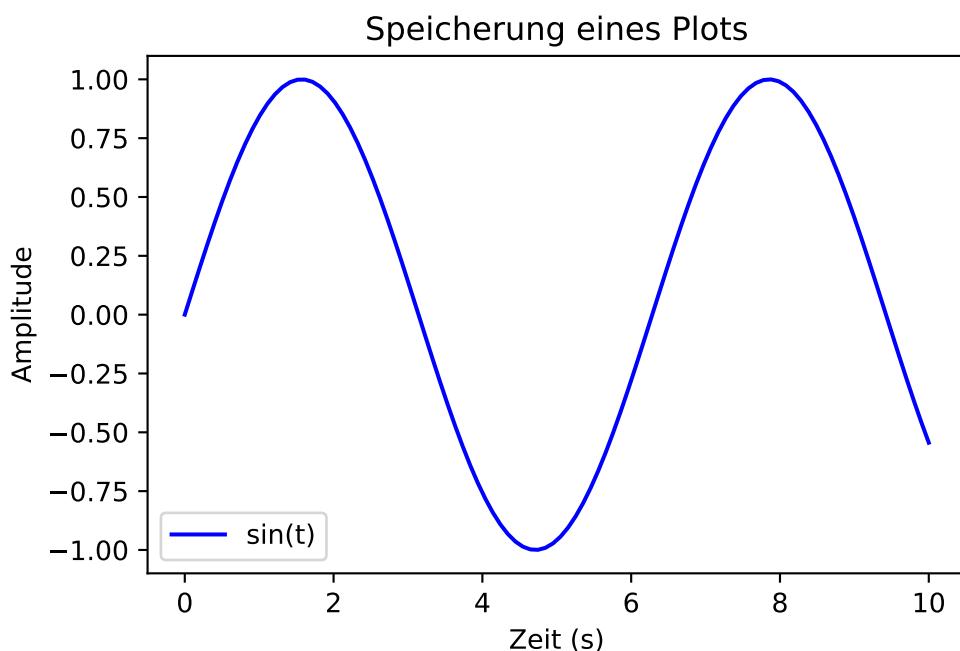
Kosinusfunktion



## 11.5 5. Speichern von Plots

Man kann Diagramme in verschiedenen Formaten speichern.

```
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title('Speicherung eines Plots')
plt.legend()
plt.savefig('mein_plot.png', dpi=300)
plt.show()
```



## 11.6 Fazit

Durch geschickte Anpassungen lassen sich wissenschaftliche Plots deutlich verbessern. Im nächsten Kapitel werden wir uns mit erweiterten Techniken wie logarithmischen Skalen und Annotationen beschäftigen.

# 12 Erweiterte Techniken in Matplotlib

In diesem Kapitel betrachten wir einige fortgeschrittene Funktionen von Matplotlib, die für die wissenschaftliche Datenvisualisierung besonders nützlich sind.

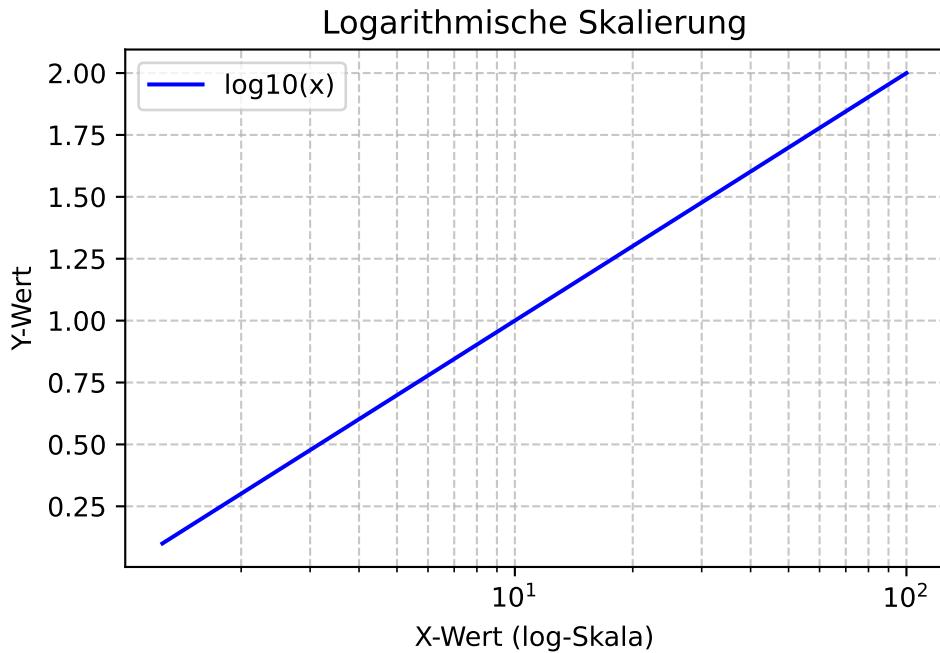
## 12.1 1. Logarithmische Skalen

Logarithmische Skalen werden oft verwendet, wenn Werte große Größenordnungen umfassen.

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

x = np.logspace(0.1, 2, 100)
y = np.log10(x)

plt.plot(x, y, label='log10(x)', color='b')
plt.xscale('log')
plt.xlabel('X-Wert (log-Skala)')
plt.ylabel('Y-Wert')
plt.title('Logarithmische Skalierung')
plt.legend()
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.show()
```



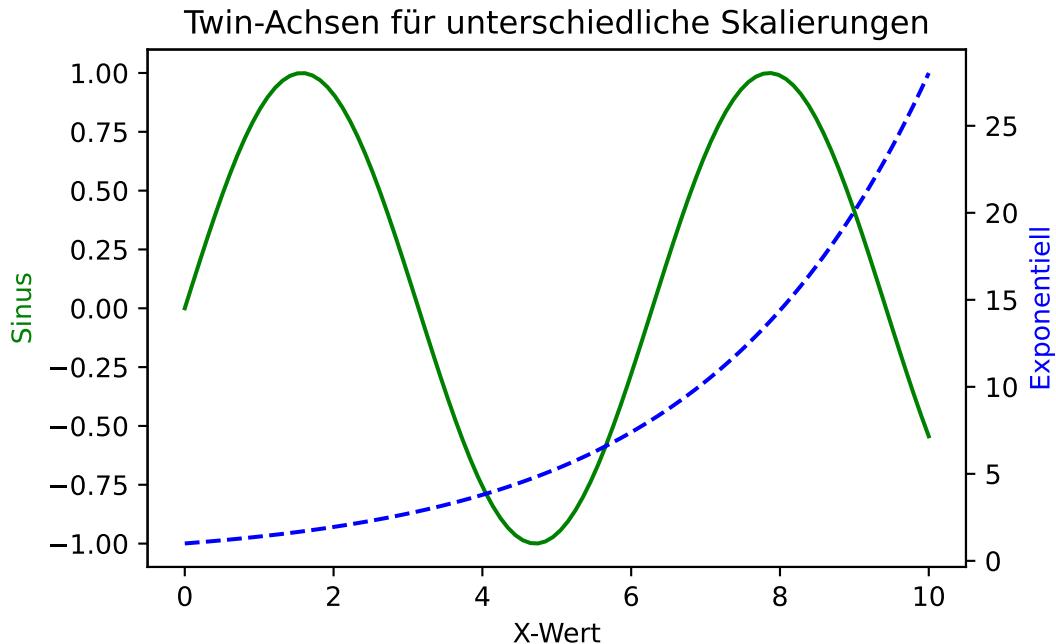
## 12.2 2. Twin-Achsen für verschiedene Skalierungen

Manchmal möchte man zwei verschiedene y-Achsen in einem Plot darstellen.

```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.exp(x / 3)

fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.plot(x, y1, 'g-', label='sin(x)')
ax2.plot(x, y2, 'b--', label='exp(x/3)')

ax1.set_xlabel('X-Wert')
ax1.set_ylabel('Sinus', color='g')
ax2.set_ylabel('Exponentiell', color='b')
ax1.set_title('Twin-Achsen für unterschiedliche Skalierungen')
plt.show()
```

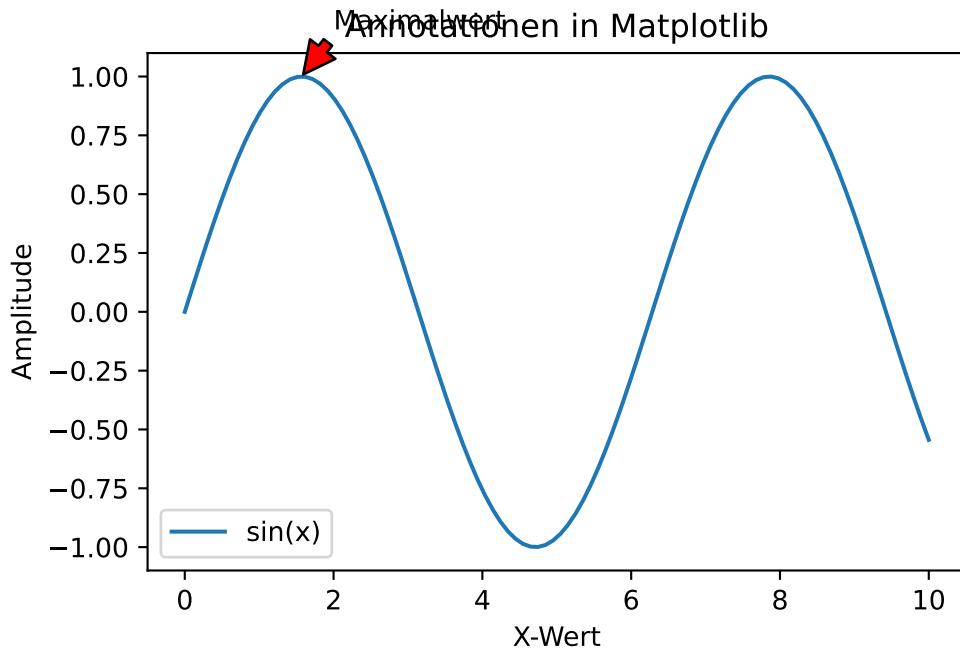


### 12.3 3. Annotationen in Diagrammen

Wichtige Punkte oder Werte in einem Diagramm können mit Annotationen hervorgehoben werden.

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y, label='sin(x)')
plt.xlabel('X-Wert')
plt.ylabel('Amplitude')
plt.title('Annotationen in Matplotlib')
plt.annotate('Maximalwert', xy=(np.pi/2, 1), xytext=(2, 1.2),
            arrowprops=dict(facecolor='red', shrink=0.05))
plt.legend()
plt.show()
```



## 12.4 Fazit

Diese erweiterten Funktionen helfen dabei, wissenschaftliche Plots noch informativer zu gestalten. Im nächsten Kapitel werden wir Best Practices und typische Fehler in der wissenschaftlichen Visualisierung betrachten.

# 13 Best Practices in Matplotlib: Fehler und Verbesserungen

In diesem Kapitel zeigen wir für häufige Problemstellungen jeweils ein schlechtes und ein verbessertes Beispiel.

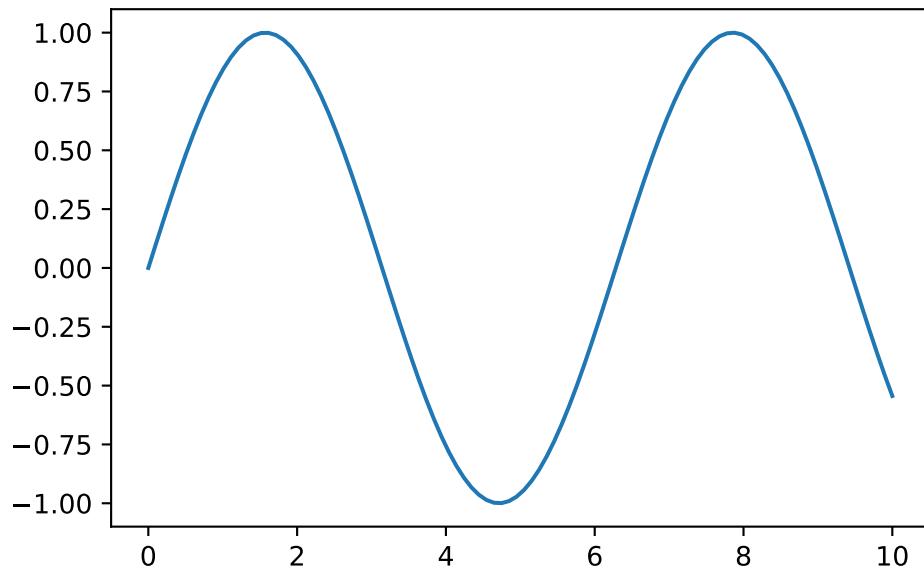
## 13.1 1. Fehlende Beschriftungen

### 13.1.1 Schlechtes Beispiel

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

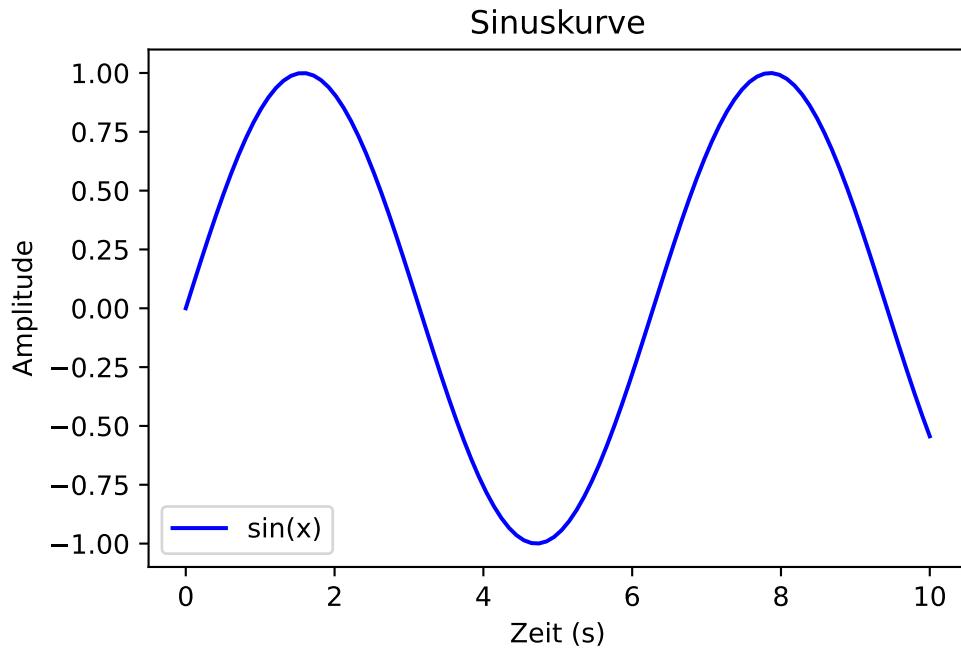
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.show()
```



### 13.1.2 Besseres Beispiel

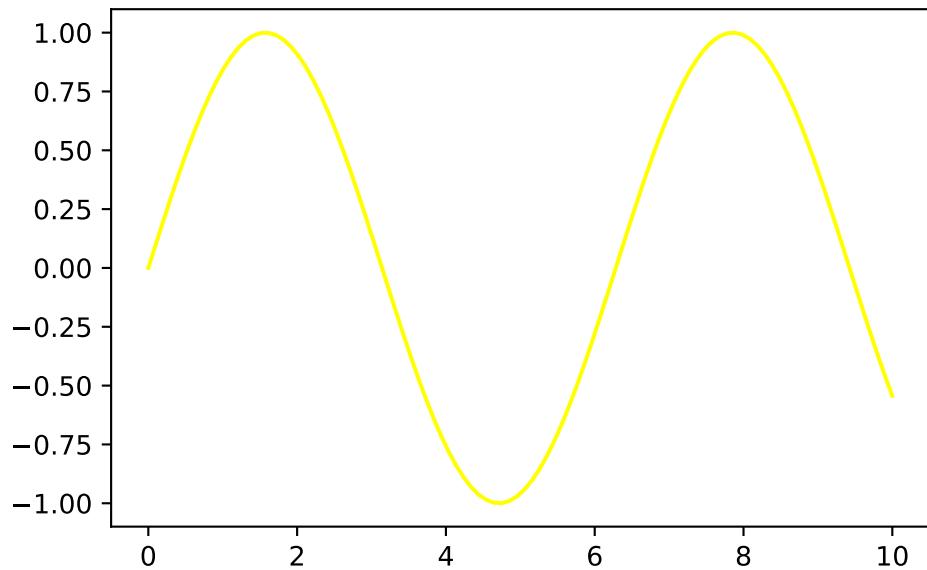
```
plt.plot(x, y, label='sin(x)', color='b')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title('Sinuskurve')
plt.legend()
plt.show()
```



## 13.2 2. Ungünstige Farbwahl

### 13.2.1 Schlechtes Beispiel

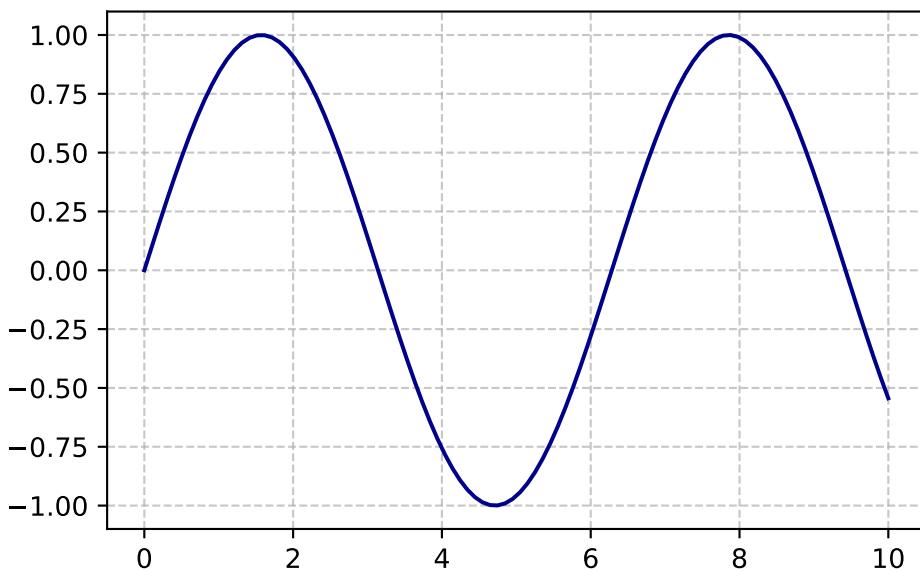
```
plt.plot(x, y, color='yellow')
plt.show()
```



### 13.2.2 Besseres Beispiel

```
plt.plot(x, y, color='darkblue')
plt.grid(True, linestyle='--', alpha=0.7)
plt.title('Gute Kontraste für bessere Lesbarkeit')
plt.show()
```

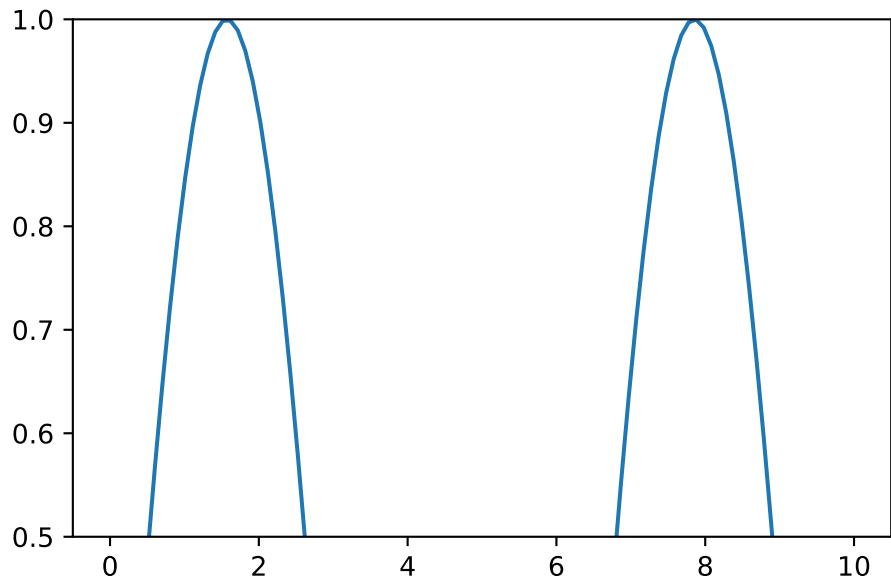
Gute Kontraste für bessere Lesbarkeit



### 13.3 3. Keine sinnvolle Achsen Skalierung

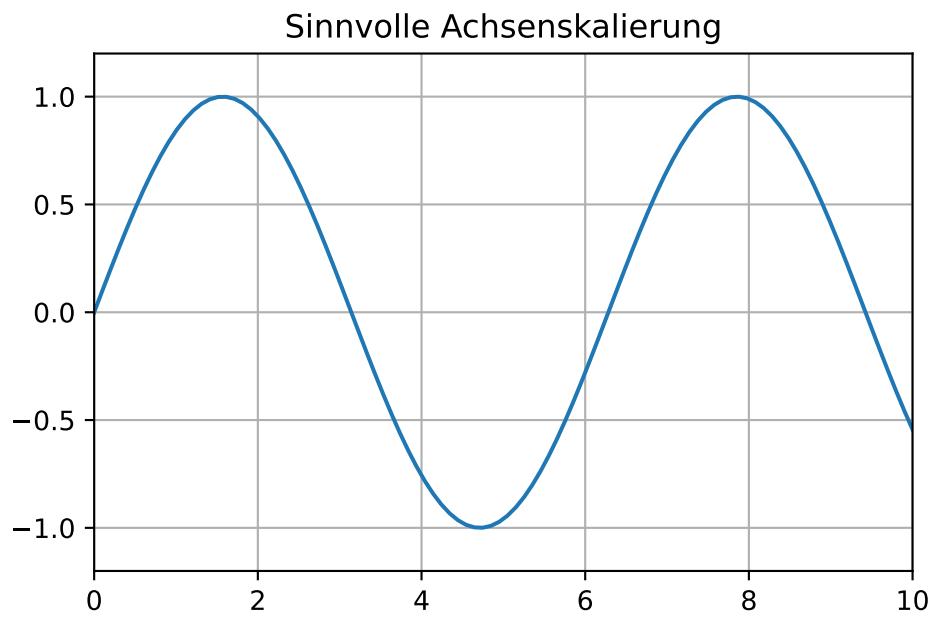
#### 13.3.1 Schlechtes Beispiel

```
plt.plot(x, y)
plt.ylim(0.5, 1)
plt.show()
```



### 13.3.2 Besseres Beispiel

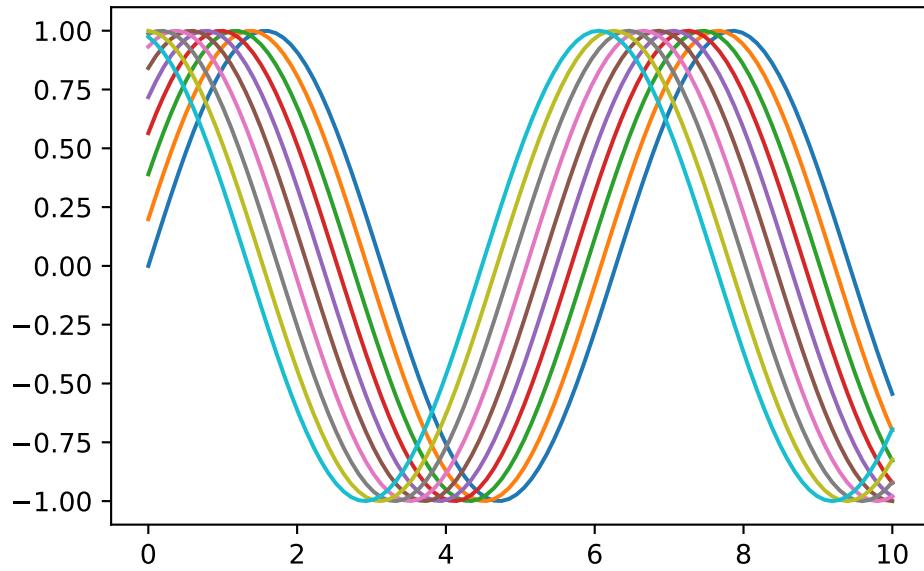
```
plt.plot(x, y)
plt.ylim(-1.2, 1.2)
plt.xlim(0, 10)
plt.grid(True)
plt.title('Sinnvolle Achsenkalierung')
plt.show()
```



## 13.4 4. Überladung durch zu viele Linien

### 13.4.1 Schlechtes Beispiel

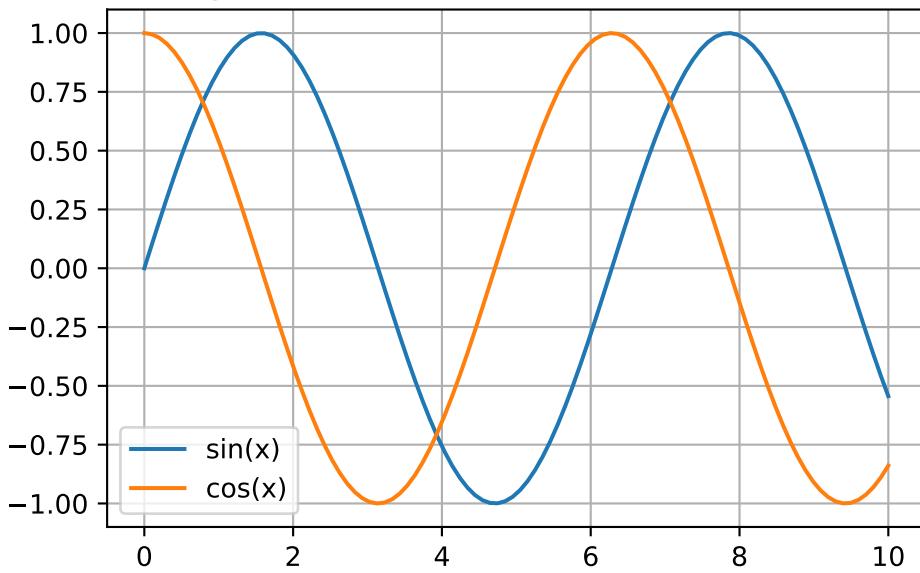
```
for i in range(10):
    plt.plot(x, np.sin(x + i * 0.2))
plt.show()
```



### 13.4.2 Besseres Beispiel

```
plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.legend()
plt.title('Weniger ist mehr: Reduzierte Informationsdichte')
plt.grid(True)
plt.show()
```

### Weniger ist mehr: Reduzierte Informationsdichte



## 13.5 Fazit

Gute Plots zeichnen sich durch klare Beschriftungen, gute Lesbarkeit und eine sinnvolle Informationsdichte aus.

## **Part IV**

# **Numpy**



Bausteine Computergestützter Datenanalyse. "Numpy Grundlagen" von Lukas Arnold, Simone Arnold, Florian Bagemihl, Matthias Baitsch, Marc Fehr, Maik Poetzsch und Sebastian Seipel 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. 2024

## 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}}
```

## **Part V**

### **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

# 14 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.

## 14.1 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.

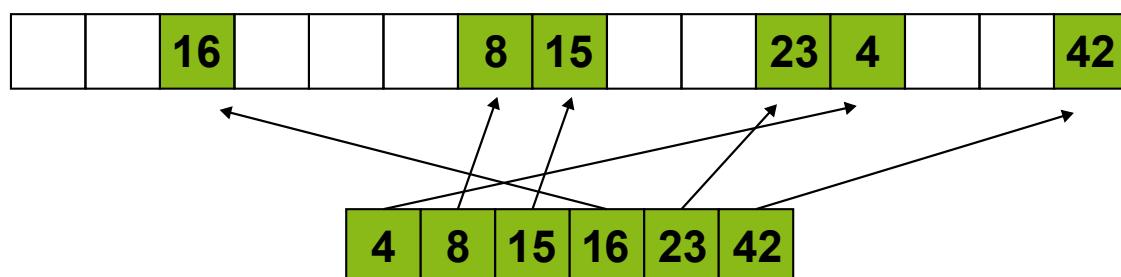


Figure 14.1: Speicherung von Daten in nativem Python

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

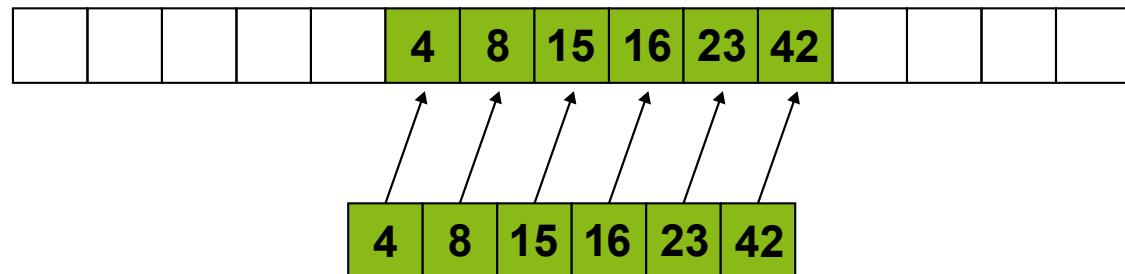


Figure 14.2: Speicherung von Daten bei Numpy

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.

## 14.2 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
```

## 14.3 Referenzen

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

# 15 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 Kommae 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 gefüllt sind erstellen?

Der Trick besteht hierbei 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,Stop,#Anzahl Werte)` und `np.arange(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 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 4 4 4 4]]
```

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

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

# 16 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 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. 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 beinhalten. Im folgenden haben wir drei verschiedene Arrays mit einem jeweils anderen 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.

Table 16.1: 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"]

### 💡 Zwischenübung: Arrayinformationen 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 Typ ist der Inhalt dieser Matrix?

Überprüfen Sie daraufhin Ihre Ergebnisse in dem 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
```

# 17 Rechnen mit Arrays

## 17.1 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])
```

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 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()`.

## 17.2 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
```

## 17.3 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 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)
```

# 18 Slicing

## 18.1 Normales Slicing mit Zahlenwerten

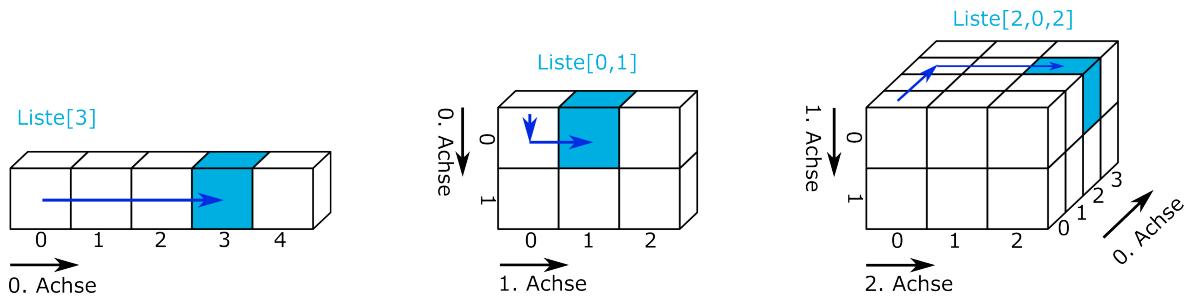


Figure 18.1: Ansprechen der einzelnen Achsen für den ein-, zwei- und dreidimensionalen Fall inkl. jeweiligem Beispiel

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

1. [a] wobei ein einzelner Wert an Position a ausgegeben wird
2. [a:b] wobei alle Werte von Position a bis Position b-1 ausgegeben werden
3. [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 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]

np.int64(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]

np.int64(9)
```

## 18.2 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 `True` oder `False` Werten 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]
```

### ⚠ Warning

Das Verwenden von booleschen Arrays ist nur im numpy-Modul möglich. Es ist nicht Möglich dieses Vorgehen auf native Python Listen anzuwenden. Hier muss durch die Liste iteriert werden.

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

```
[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

```

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]

# 19 Array Manipulation

## 19.1 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.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 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 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]
```

## 19.2 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]
```

## 19.3 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 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: Arraymanipulation

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)
```

# 20 Lesen und Schreiben von Dateien

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

## 20.1 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.

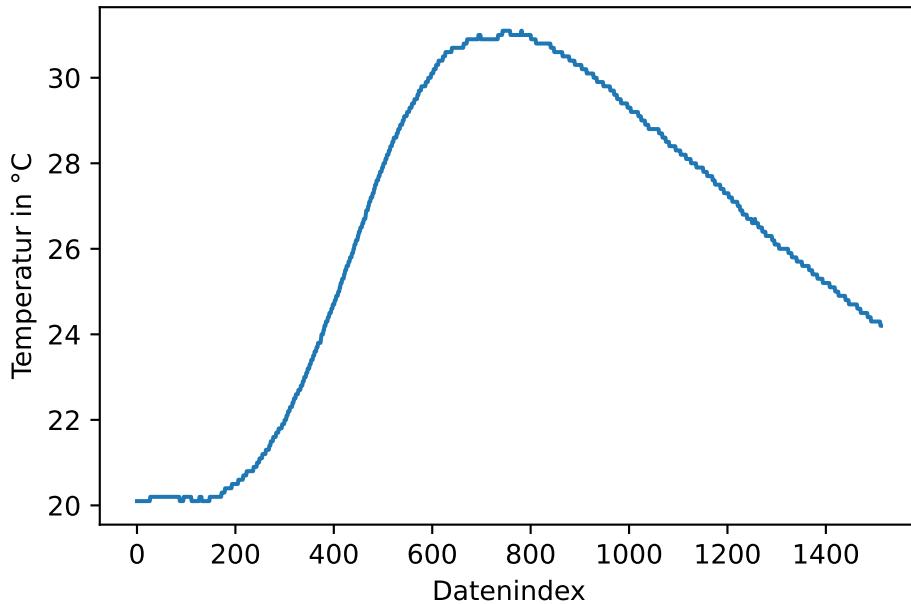
Im folgenden Beispiel wird die Datei `TC01.csv` eingelesen und deren Inhalt graphisch dargestellt. Die erste Zeile der Datei wird dabei ignoriert, da sie als Kommentar – eingeleitet durch das `#`-Zeichen – interpretiert wird.

```
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');
```



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 = ";")`

Beginnt die Datei mit den Daten mit Zeilen bezüglich zusätzlichen Informationen wie Einheiten oder Experimentdaten, können diese mit der Option `skiprows= #Reihenübersprünge` werden.

## 20.2 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.

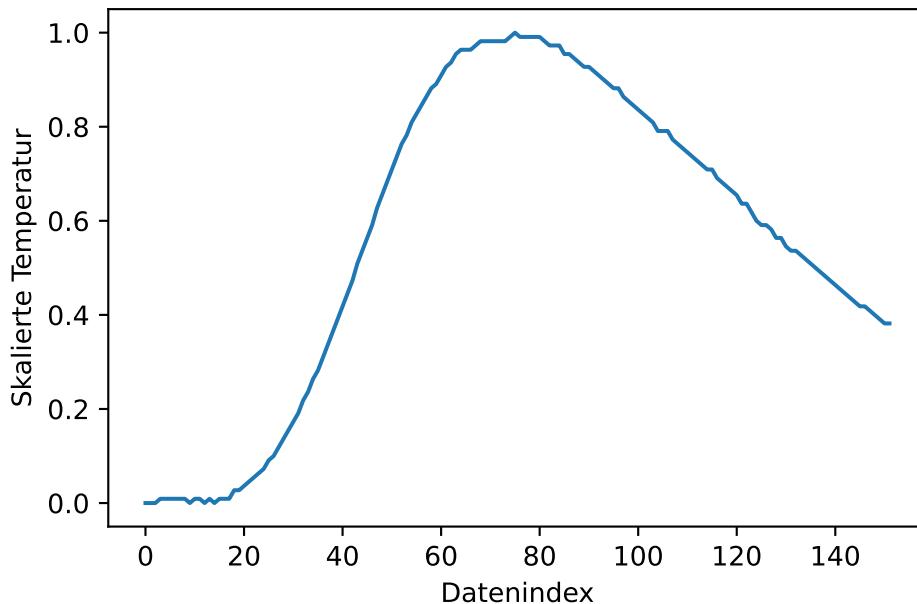
Folgendes Beispiel skaliert die oben eingelesenen Daten und schreibt jeden zehnten Wert in eine Datei. Dabei wird auch ein Kommentar (`header`-Argument) am Anfang der Datei erzeugt. Das Ausabeformat der Zahlen kann mit dem `fmt`-Argument angegeben werden. Das Format ähnelt der Darstellungsweise, welche 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 \noriginale 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='%.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
```

# 21 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.

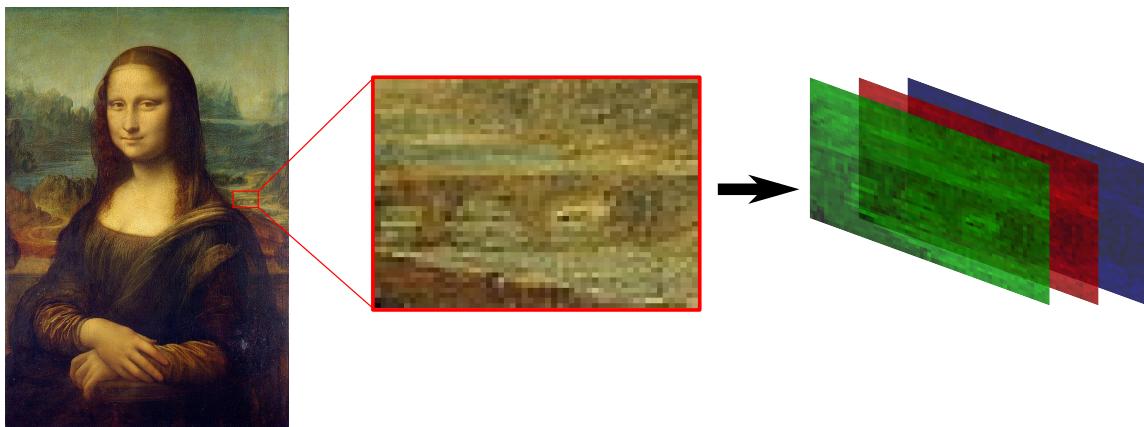


Figure 21.1: Ein hochauflöste Bild besteht aus sehr vielen Pixeln. Jedes Pixel enthält 3 Farbwerte, einen für die Farbe Grün, einen für Blau und einen für Rot.

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.

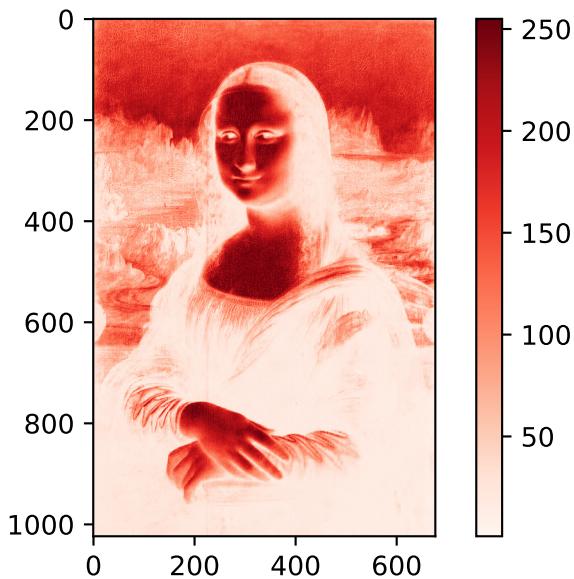
```
plt.imshow(data)
```



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

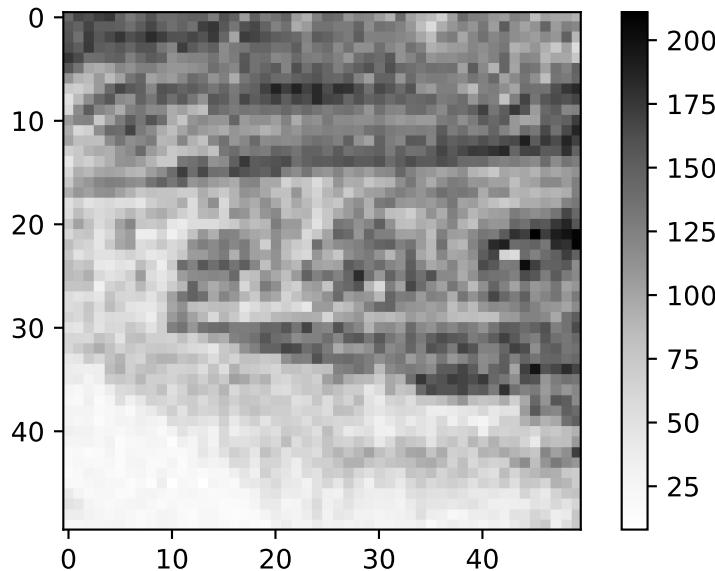
```
# Als Farbskale wird die Rotskala  
# verwendet 'Reds'
```

```
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()
```



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  $\phi$  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 ein: [Bild](#). 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)

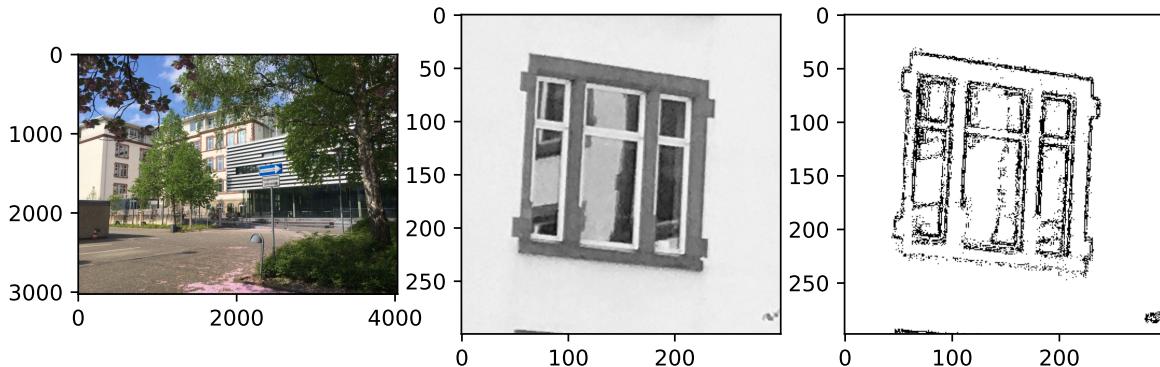
lap = img_lap(bereich)

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

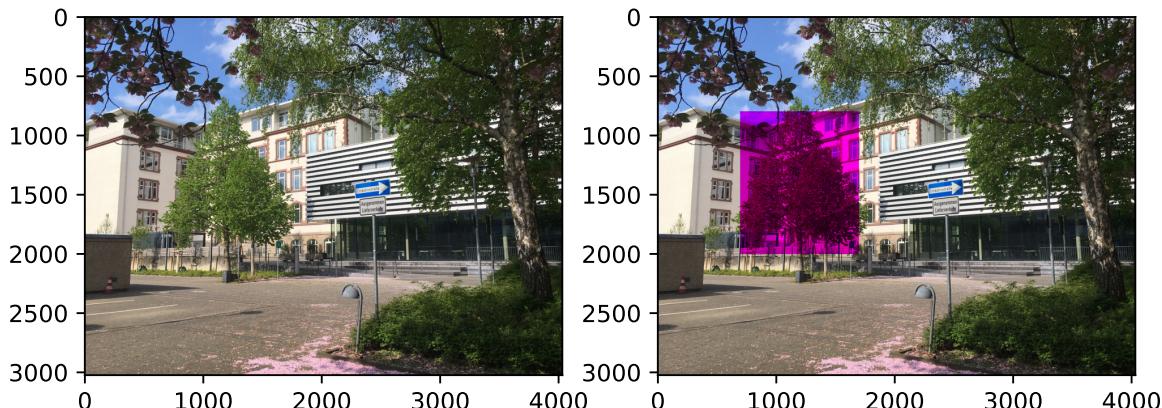
zusammengesetzt = np.dstack((rot, gruen_neu, blau)).astype('uint8')

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 ausgeben, so wie einen beliebigen Bildausschnitt.

#### 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)

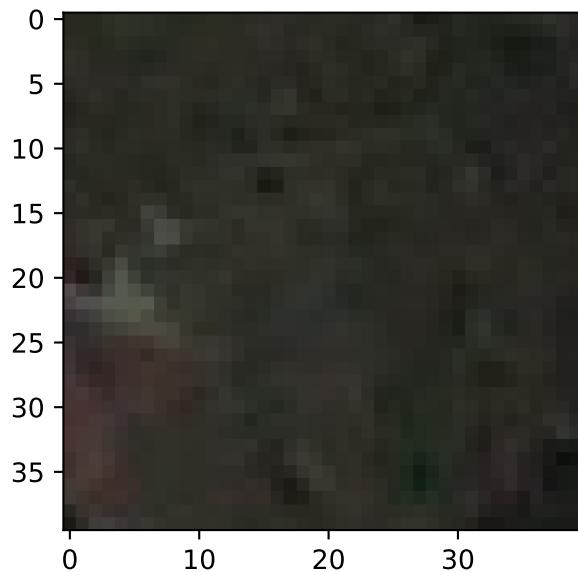
blau = data[:, :, 2]
plt.imshow(blau, cmap='Blues')

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]



# 22 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 Frage 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  $3 \times 3 \times 3$ , 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 besitzt folgendes Array? Mit welcher Funktion kann ich den Datentypen 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 = [5, 1, 3, 6, 4] und b = [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 = [9, 2, 3, 1, 3]

1. Bestimmen Sie Mittelwert und Standardabweichung für das Array a
2. Bestimmen Sie Minimum und Maximum der Liste

## 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  $4 \times 5$ .
2. Ändern Sie die Form des Arrays in eine zweidimensionale Matrix der Form  $5 \times 4$ .
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  $4 \times 54 \times 5$ -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 `float.64`.

```
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.
print(matrix[:, -1])

# 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]]
```

## 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 wie Daten in der Matrix `data` gespeichert sind mit Slicing eine Dimension auswählen: `data[:, :, 0]`, wobei die Zahl 0-2 für die jeweilige Farbe steht.

# 23 Übung

## 23.1 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 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_hoehste_bewertung = np.argmax(bewertungen[:,2])  
  
print(bewertungen[index_hoehste_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 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]]
```

## 23.2 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 23.1: 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.

Zunächst wollen wir eine einfache Caesar-Chiffre anwenden. Dabei werden alle Buchstaben um eine gewisse Anzahl verschoben. Ist Beispielsweise 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 = "abracadabra"
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 = "abracadabra"
versatz = 3

umgewandelter_text = []
verschlüsselte_zahl = []
verschlüsselter_text= []

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

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

for zahl in verschlüsselte_zahl:
    verschlüsselter_text.append(ascii_zu_buchstabe(zahl))
print(verschlüsselter_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']
```

## **Part VI**

# **Datenanalyse und Modellierung**

## Einleitung

In dieser Einheit lernen Sie, wie man reale Messdaten – etwa aus Experimenten oder Ingenieurprojekten – mit NumPy und Matplotlib verarbeitet und analysiert. Der Fokus liegt dabei auf einem praxisnahen Umgang mit Daten im CSV-Format.

## Lernziele dieses Kapitels

Sie lernen in dieser Einheit:

- wie Sie strukturierte CSV-Daten in NumPy-Arrays überführen,
- wie Sie fehlende Werte erkennen und ersetzen,
- wie Sie typische statistische Kennzahlen berechnen,
- wie Sie Daten mit Matplotlib visualisieren,
- wie Sie Daten interpolieren und Trends glätten,
- und wie Sie reale Anwendungen – z. B. aus dem Bauwesen – analysieren.

## CSV-Dateien: Ein typisches Format für Messdaten

CSV-Dateien („Comma Separated Values“) sind weit verbreitet – etwa für:

- Temperaturverläufe,
- Messreihen aus Experimenten,
- Logdaten von Sensoren.

Zu Beginn wird ein Beispiel betrachtet: Temperatur, Luftfeuchtigkeit und CO -Werte. Diese Datei enthält auch einige **fehlende Werte**, wie sie in realen Daten oft vorkommen.

```
import numpy as np
data = np.genfromtxt("beispiel.csv", delimiter=",", skip_header=1)
print(data)
```

```
[[ 21.1  45.  400. ]
 [ 22.5   nan 420. ]
 [  nan  50.  410. ]
 [ 20.   48.    nan]
 [ 23.3  47.  430. ]]
```

## Fehlende Werte erkennen und bereinigen

Fehlende Werte werden beim Einlesen als `np.nan` (Not a Number) codiert. Zunächst wird gezählt, wie viele Werte fehlen:

```
print(np.isnan(data).sum(axis=0))
```

```
[1 1 1]
```

Um die Analyse nicht zu verfälschen, werden sie ersetzt – z.B. durch den Mittelwert der Spalte:

```
for i in range(data.shape[1]):  
    mean = np.nanmean(data[:, i])  
    data[:, i] = np.where(np.isnan(data[:, i]), mean, data[:, i])
```

## Statistische Kennzahlen berechnen

Typische Kennwerte zur Beschreibung von Daten:

- **Mittelwert:** Durchschnitt
- **Standardabweichung:** Streuung
- **Minimum und Maximum**

```
print("Mittelwerte:", np.mean(data, axis=0))  
print("Standardabweichung:", np.std(data, axis=0))
```

```
Mittelwerte: [ 21.725  47.5   415. ]  
Standardabweichung: [ 1.13556154  1.61245155 10. ]
```

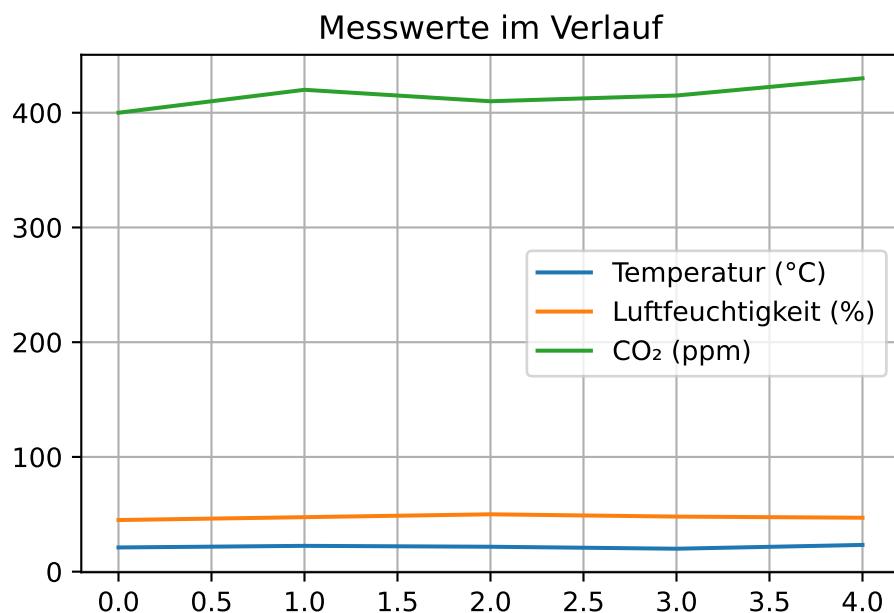
## Daten visualisieren

Mit Matplotlib lassen sich Daten übersichtlich darstellen. Es werden z.B. Linien- und Histogrammplots genutzt.

```
import matplotlib.pyplot as plt

labels = ["Temperatur (°C)", "Luftfeuchtigkeit (%)", "CO2 (ppm)"]

for i in range(data.shape[1]):
    plt.plot(data[:, i], label=labels[i])
plt.legend()
plt.title("Messwerte im Verlauf")
plt.grid(True)
plt.show()
```



# 24 Interpolation – Lücken schließen

Was tun, wenn Werte fehlen? In vielen Datensätzen gibt es Lücken – zum Beispiel, weil Messungen nur an bestimmten Punkten vorgenommen wurden. Interpolation ist eine Methode, mit der wir Zwischenwerte schätzen können, also Werte innerhalb eines bekannten Wertebereichs.

Im Gegensatz dazu versucht Extrapolation, Werte außerhalb des bekannten Bereichs vorherzusagen – was in der Regel mit größerer Unsicherheit verbunden ist.

Bei der Interpolation wird eine Modellfunktion gesucht, welche die Messdaten exakt abbildet.

## Theorie - Modellierung

Die Modellierung von Daten hat das Ziel eine Menge von Daten durch einen funktionalen Zusammenhang abzubilden. Beispielsweise können Daten aus Experimenten oder Simulationen stark verfälscht und so für eine Weiterverarbeitung nicht geeignet sein. Eine mittelnde Funktion kann den Datensatz stark vereinfachen. Oder es existieren nur wenige Datenpunkte und die Zwischenstellen müssen durch eine Funktion bestimmt werden. Generell kann die Modellierung von Daten auf folgendes Problem verallgemeinert werden:

1. Gegeben sind  $n$  Messpunktspaare  $(x_i, y_i)$  mit  $x_i, y_i \in \mathbb{R}$
2. Gesucht ist eine Modellfunktion  $y(x)$ , welche die Messpunktspaare approximiert

Ein möglicher Ansatz ist die Darstellung der Modellfunktion als Summe von  $m$  Basisfunktionen  $\phi_i(x)$  mit den Koeffizienten  $\beta_i$ :

$$y(x) = \sum_{i=1}^m \beta_i \cdot \phi_i(x) = \beta_1 \cdot \phi_1(x) + \dots + \beta_m \cdot \phi_m(x)$$

Die Koeffizienten  $\beta_i$  müssen dabei so bestimmt werden, dass  $y(x)$  so gut wie möglich – oder gar exakt – die Messpunkte approximieren.

Als Abstandmaß zwischen einer Modellfunktion und den Messpunkten kann die [L2-Norm](#) verwendet werden. Diese ist definiert als

$$\|y(x) - (x_i, y_i)\|_2 = \sqrt{\sum_{i=1}^n (y(x_i) - y_i)^2} .$$

Eine solche Norm gibt ein Maß für die Qualität einer Approximation: je kleiner der Abstand, desto besser die Qualität. Dies ermöglicht das Finden optimaler Koeffizienten

und wird beispielsweise in der Methode der kleinsten Quadrate genutzt, in der ein Satz an Koeffizienten gesucht wird, der die L2-Norm minimiert.

## 24.1 Übersicht

In vielen praktischen Anwendungen werden Polynome als Basisfunktionen der Modellfunktion angenommen. Vorteile von Polynomen:

- Polynome sind leicht zu differenzieren und integrieren
- Annäherung von beliebigen Funktionen durch Polynome möglich, siehe [Taylor-Entwicklung](#)
- Auswertung ist sehr einfach und dadurch schnell, d.h. sie benötigt nur wenige schnelle arithmetische Operationen (Addition und Multiplikation)

Ein Beispiel für eine Basis aus Polynomen:

$$\phi_1(x) = 1, \quad \phi_2(x) = x, \quad \phi_3(x) = x^2, \quad \dots, \quad \phi_m = x^{m-1}$$

## 24.2 Polynome

Polynome  $P(x)$  sind Funktionen in Form einer Summe von Potenzfunktionen mit natürlichen Exponenten ( $x^i, i \in \mathbb{N}$ ) mit den entsprechenden Koeffizienten  $a_i$ :

$$P(x) = \sum_{i=0}^n a_i x^i = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0, \quad i, n \in \mathbb{N}, a_i \in \mathbb{R}$$

- Als Grad eines Polynoms wird der Term mit dem höchsten Exponenten und nicht verschwindenden Koeffizienten (der sogenannte Leitkoeffizient) bezeichnet.
- Ein Polynom mit Grad  $n$  hat  $n$ , teilweise [komplexe](#), Nullstellen.

In Python, d.h. im numpy-Modul, werden Polynome durch ihre Koeffizienten representiert. Im Allgemeinen wird ein Polynom mit dem Grad  $n$  durch folgendes Array dargestellt

```
[an, ..., a2, a1, a0]
```

So z.B. für  $P(x) = x^3 + 5x^2 - 2x + 3$ :

```
P = np.array([1, 5, -2, 3])
print(P)
```

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

Die Auswertung des Polynoms an einem Punkt oder einem Array erfolgt mit der `np.polyval`-Funktion.

```
x = 1
y = np.polyval(P, x)
print(f"P(x={x}) = {y}")
```

```
P(x=1) = 7
```

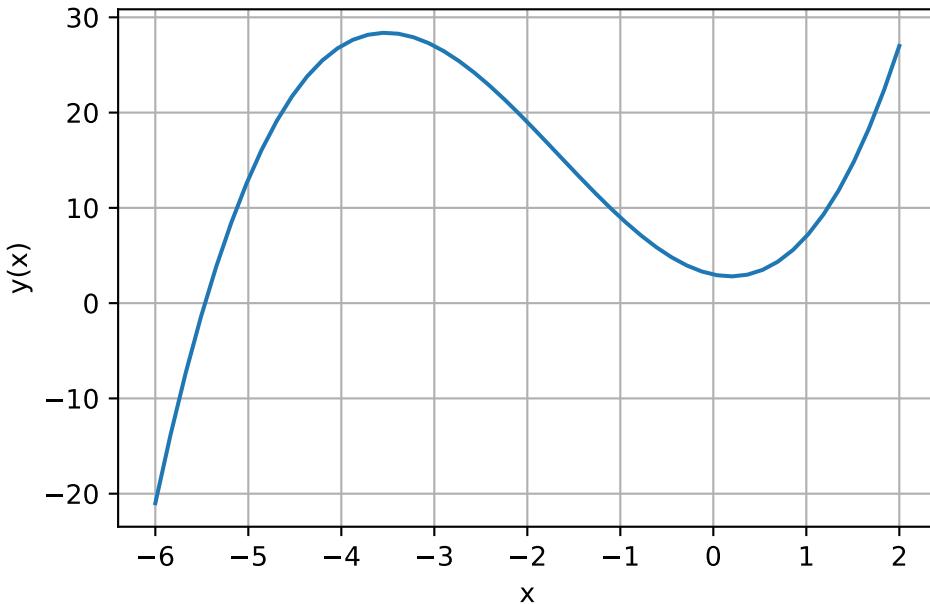
```
x = np.array([-1, 0, 1])
y = np.polyval(P, x)
print(f"P(x={x}) = {y}")
```

```
P(x=[-1  0  1]) = [9 3 7]
```

Für die graphische Darstellung im Bereich  $x \in [-6, 2]$  können die bekannten numpy und matplotlib Funktionen verwendet werden.

```
x = np.linspace(-6, 2, 50)
y = np.polyval(P, x)

plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('y(x)')
plt.grid()
```



Um die Nullstellen eines Polynoms zu finden, kann die numpy-Funktion `np.roots` genutzt werden. Für das obige Polynom können folgende Nullstellen bestimmt werden.

```
nststellen = np.roots(P)

# direkte Ausgabe des Arrays
print("Nullstellen: ")
print(nststellen)
```

```
Nullstellen:
[-5.46628038+0.j         0.23314019+0.703182j  0.23314019-0.703182j]
```

```
print("Nullstellen: ")
# schönere Ausgabe des Arrays
for i, z in enumerate(nststellen):
    if z.imag == 0:
        print(f"  x_{i+1} = {z.real:.2}")
    else:
        print(f"  x_{i+1} = {z.real:.2} {z.imag:+.2}i")
```

```
Nullstellen:
x_1 = -5.5
x_2 = 0.23 +0.7i
x_3 = 0.23 -0.7i
```

In diesem Beispiel sind zwei der Nullstellen komplex. Eine komplexe Zahl  $z$  wird in Python als Summe des Realteils (`Re(z)`) und Imaginarteils (`Im(z)`). Letzterer wird durch ein nachfolgendes `j`, die imaginäre Einheit, gekennzeichnet.

$$z = Re(z) + Im(z)j$$

Die Nullstellen können auch zur alternativen Darstellung des Polynoms verwendet werden. Sind  $x_i$  die  $n$  Nullstellen, so ist das Polynom  $n$ -ten Grades durch folgendes Produkt beschrieben:

$$P(x) = \prod_{i=1}^n (x - x_i) = (x - x_1) \cdot (x - x_2) \cdot \dots \cdot (x - x_n)$$

Seien beispielsweise 1 und 2 die Nullstellen eines Polynoms, so lautet dieses:

$$P(x) = (x - 1)(x - 2) = x^2 - 3x + 2$$

Die numpy-Funktion `np.poly` kann aus den Nullstellen die Polynomkoeffizienten bestimmen. Anhand des obigen Beispiels lautet der Funktionsaufruf:

```
nstellen = [1, 2]
koeffizienten = np.poly(nstellen)

print("Nullstellen:", nstellen)
print("Koeffizienten:", koeffizienten)
```

```
Nullstellen: [1, 2]
Koeffizienten: [ 1. -3.  2.]
```

Das Modul numpy stellt viele praktische Funktionen zum Umgang mit Polynomen zur Verfügung. So existieren Funktionen um Polynome auszuwerten, die Nullstellen zu finden, zu addieren, zu multiplizieren, abzuleiten oder zu integrieren. Eine Übersicht ist in der [numpy-Dokumentation](#) gegeben.

## 24.3 Interpolation

Interpolation ist eine Methode, um Datenpunkte zwischen gegebenen Messpunkten zu konstruieren. Dazu wird eine Funktion gesucht, die alle Messpunkte exakt abbildet, was gleichbedeutend damit ist, dass die L2-Norm zwischen Funktion und Punkten Null ist.

Zwei Punkte können z.B. mit einer Geraden interpoliert werden. D.h. für zwei Messpunktspaare  $(x_1, y_1)$  und  $(x_2, y_2)$  mit  $x_1 \neq x_2$  existiert ein Koeffizientensatz, sodass die L2-Norm zwischen den Messpunkten und der Modellfunktion verschwindet.

$$y(x) = \beta_1 x + \beta_0$$

verschwindet.

```
# Beispieldaten aus y(x) = -x + 2

N = 50
dx = 0.25

def fnk(x):
    return -x + 2

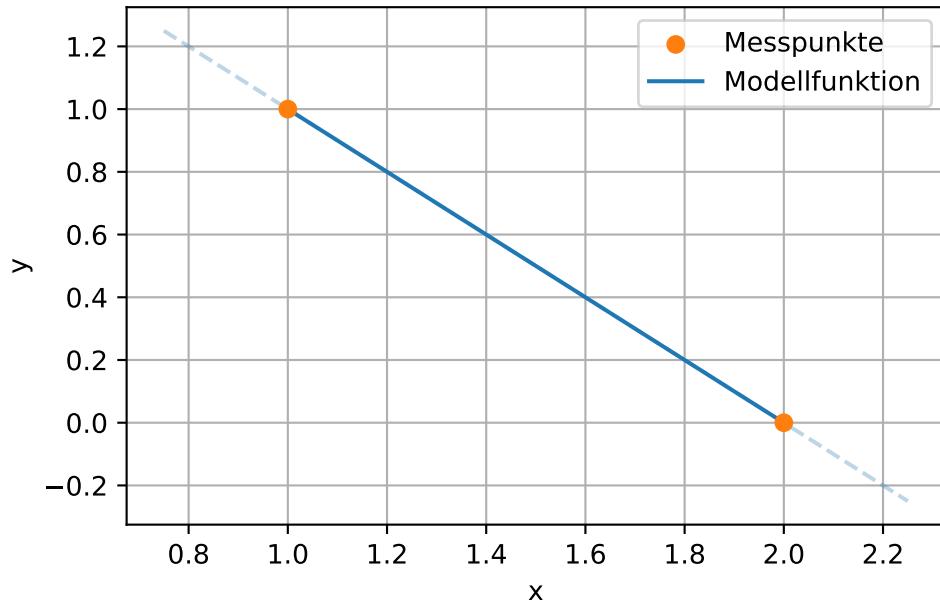
x = np.array([1, 2])
y = fnk(x)

plt.scatter(x, y, color='C1', label="Messpunkte", zorder=3)

x_modell = np.linspace(np.min(x), np.max(x), N)
plt.plot(x_modell, fnk(x_modell), color='C0', label="Modellfunktion")

x_linie = np.linspace(np.min(x)-dx, np.max(x)+dx, N)
plt.plot(x_linie, fnk(x_linie), '--', alpha=0.3, color='C0')

plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()
```



Für drei Messpunkte muss ein Polynom zweiten Grades verwendet werden, um die Punkte exakt zu erfassen.

$$y(x) = \beta_2 x^2 + \beta_1 x + \beta_0$$

```
# Beispieldaten aus y(x) = 3x^2 -4x - 1

N = 50
dx = 0.25

def fnk(x):
    return 3*x**2-4*x - 1

x = np.array([-1, 2, 3])
y = fnk(x)

plt.scatter(x, y, color='C1', label="Messpunkte", zorder=3)

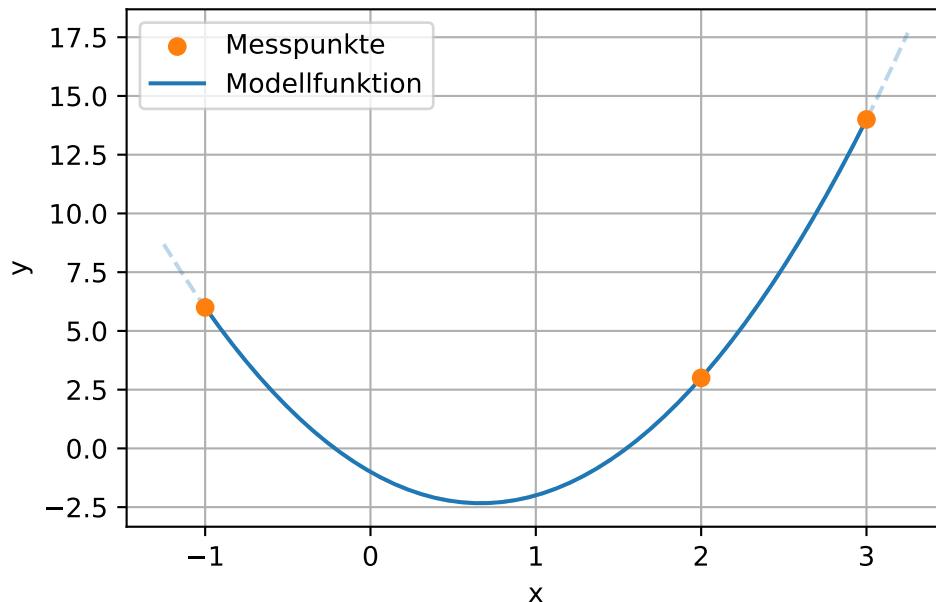
x_modell = np.linspace(np.min(x), np.max(x), N)
plt.plot(x_modell, fnk(x_modell), color='C0', label="Modellfunktion")

x_linie = np.linspace(np.min(x)-dx, np.max(x)+dx, N)
plt.plot(x_linie, fnk(x_linie), '--', alpha=0.3, color='C0')
```

```

plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid()

```



Dies kann verallgemeinert werden:  $n$  Messpunkte können exakt mit einem Polynom ( $\$ n \$$ -ten Grades abgebildet werden. Die Suche nach den passenden Koeffizienten ist das Lagrangesche Interpolationsproblem. Für das gesuchte Polynom  $P(x)$  gilt:

$$P(x_i) = y_i \quad i \in 1, \dots, n$$

Die Existenz und Eindeutigkeit eines solchen Polynoms kann gezeigt werden. Das gesuchte Polynom lautet:

$$P(x) = \sum_{i=1}^n y_i I_i(x)$$

mit  $I_i(x) = \prod_{j=1, j \neq i}^n \frac{x - x_j}{x_i - x_j}$

Alternativ kann auch ein Gleichungssystem, welches durch die Polynomialbasis  $\phi_i(x)$  gegeben ist, gelöst werden. Für die  $n$  Punktpaare gilt jeweils:

$$y(x_i) = \sum_{i=1}^m \beta_i \cdot \phi_i(x_i) = \beta_1 \cdot \phi_1(x_i) + \cdots + \beta_m \cdot \phi_m(x_i) = y_i$$

Das allgemeine Gleichungssystem lautet

$$\begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_m(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_m(x_n) \end{pmatrix} \cdot \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

und mit der Polynomialbasis

$$\underbrace{\begin{pmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^{n-1} \end{pmatrix}}_V \cdot \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Die Matrix  $V$  heisst **Vandermonde-Matrix** und kann exakt gelöst werden, für  $m = n$  und wenn für alle  $i, j, i \neq j$  gilt  $x_i \neq x_j$ .

In Python kann das Interpolationsproblem mit der **Funktion np.polyfit** gelöst werden. Das folgende Beispiel demonstriert deren Anwendung.

Die Messstellen folgen in dem Beispiel der Funktion  $f(x)$ , welche nur zur Generierung der Datenpunkte verwendet wird.

$$f(x) = \frac{1}{2} + \frac{1}{1+x^2}$$

Zunächst werden die Messpunkte generiert.

```
def fnk(x):
    return 0.5 + 1/(1+x**2)
```

```
xmin = -5
xmax = 5
x = np.linspace(xmin, xmax, 100)
y = fnk(x)
```

```

n = 5
xi = np.linspace(xmin, xmax, n)
yi = fnk(xi)

```

Nun folgt die Interpolation für 5 und 15 Messpunkte.

```

P = np.polyfit(xi, yi, n-1)
print("Interpolationskoeffizienten:")
print(P)

```

Interpolationskoeffizienten:

```

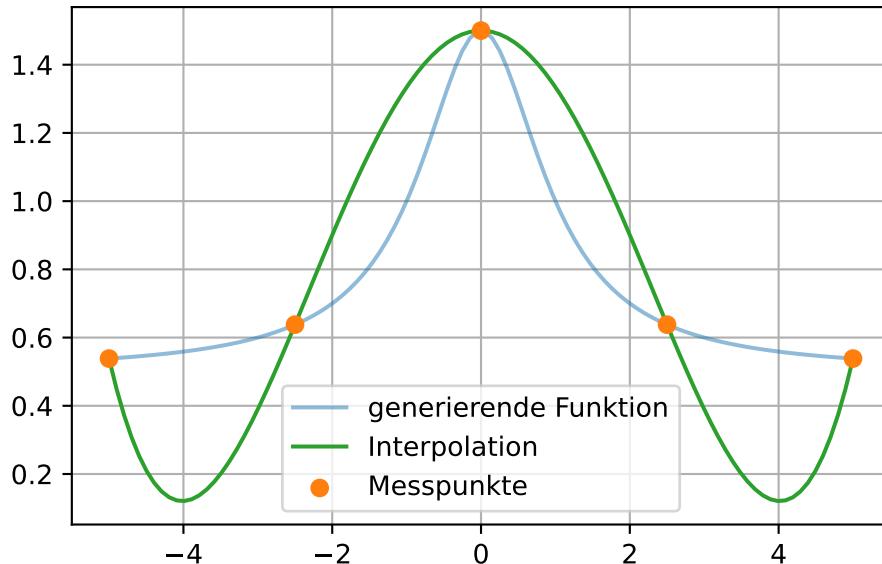
[ 5.30503979e-03  4.23767299e-17 -1.71087533e-01  7.45353051e-16
 1.50000000e+00]

```

```

plt.plot(x, y, color='C0', alpha=0.5, label='generierende Funktion')
plt.plot(x, np.polyval(P, x), color='C2', label='Interpolation')
plt.scatter(xi, yi, color='C1', label='Messpunkte', zorder=3)
plt.legend()
plt.grid()

```



```

n = 15
xi = np.linspace(xmin, xmax, n)
yi = fnk(xi)

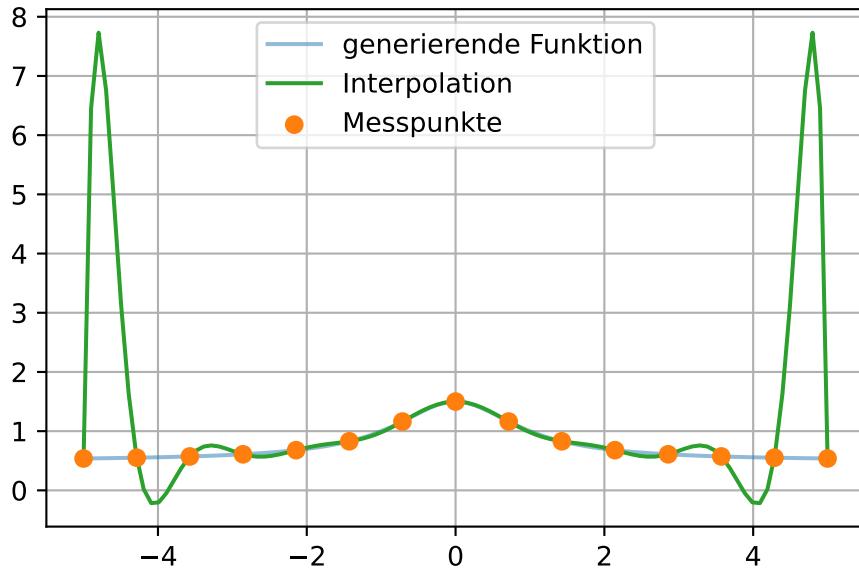
```

```

P = np.polyfit(xi, yi, n-1)

plt.plot(x, y, color='C0', alpha=0.5, label='generierende Funktion')
plt.plot(x, np.polyval(P, x), color='C2', label='Interpolation')
plt.scatter(xi, yi, color='C1', label='Messpunkte', zorder=3)
plt.legend()
plt.grid()

```



Die Interpolation erfüllt immer die geforderte Bedingung  $y(x_i) = y_i$ . Jedoch führen Polynome mit einem hohen Grad oft zu nicht sinnvollen Ergebnissen. Es entstehen starke Überschwinger, welche mit zunehmendem Grad immer stärker werden. Eine alternative Interpolationsmethode stellen Splines dar, welche mehrere, niedrige Polynome zur Interpolation vieler Punkte verwenden.

## 25 Fitting

Beim Fitting wird eine Modellfunktion gesucht, welche die Messdaten nicht unbedingt exakt abbildet. Wird ein Polynom verwendet, so hat es eine Grad, welcher deutlich kleiner ist, als die Anzahl der Messpunkte. Lineare Regression ist ein Beispiel für ein Fitting durch ein Polynom mit dem Grad Eins.

Zum Fitten durch ein Polynom kann die Funktion `np.polyfit` verwendet werden, genauso wie bei der Polynominterpolation. Diesmal jedoch mit einem kleineren Polynomgrad.

Im folgenden Beispiel werden zunächst Modelldaten generiert und dann mit entsprechenden Polynomen gefittet.

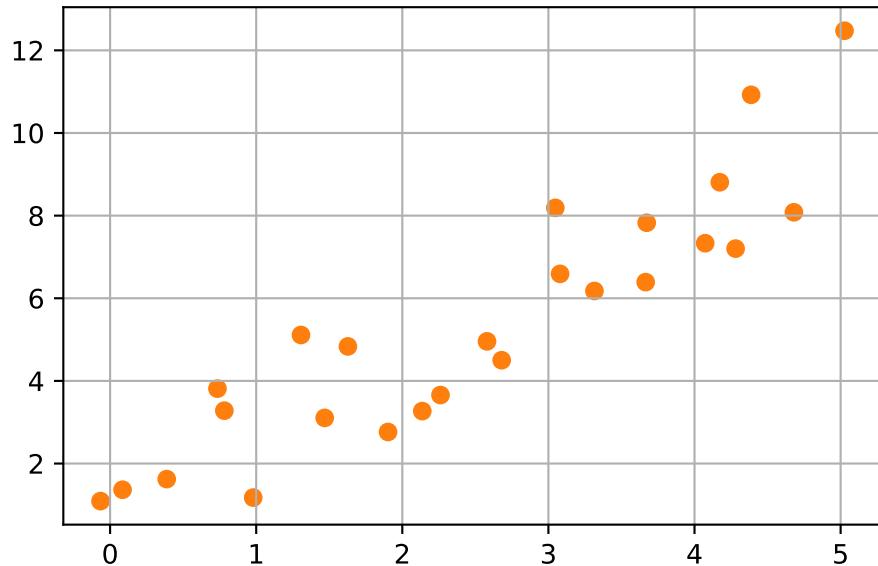
```
xmin = 0
xmax = 5
x = np.linspace(xmin, xmax, 100)

ni = 25

# x-Werte mit leichtem Rauschen
xi = np.linspace(xmin, xmax, ni) + 0.2*(2 * np.random.random(ni) -1)

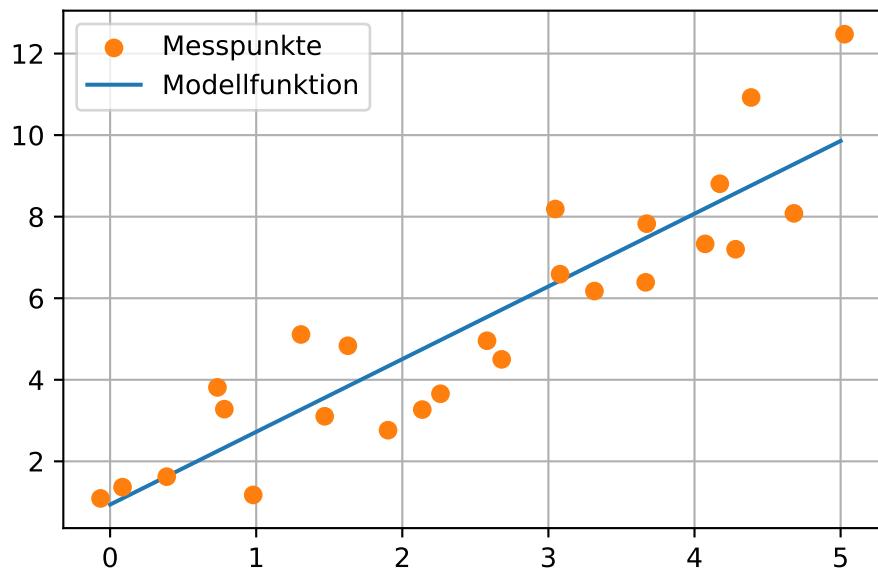
# y(x) = 2x+0.5 mit leichtem Rauschen
yi = 2*xi + 0.5 + 2*(2 * np.random.random(ni) -1)

plt.scatter(xi, yi, color='C1')
plt.grid()
```



```
P1 = np.polyfit(xi, yi, 1)

plt.scatter(xi, yi, color='C1', zorder=3, label='Messpunkte')
plt.plot(x, np.polyval(P1, x), color='C0', label="Modellfunktion")
plt.grid()
plt.legend()
```



```

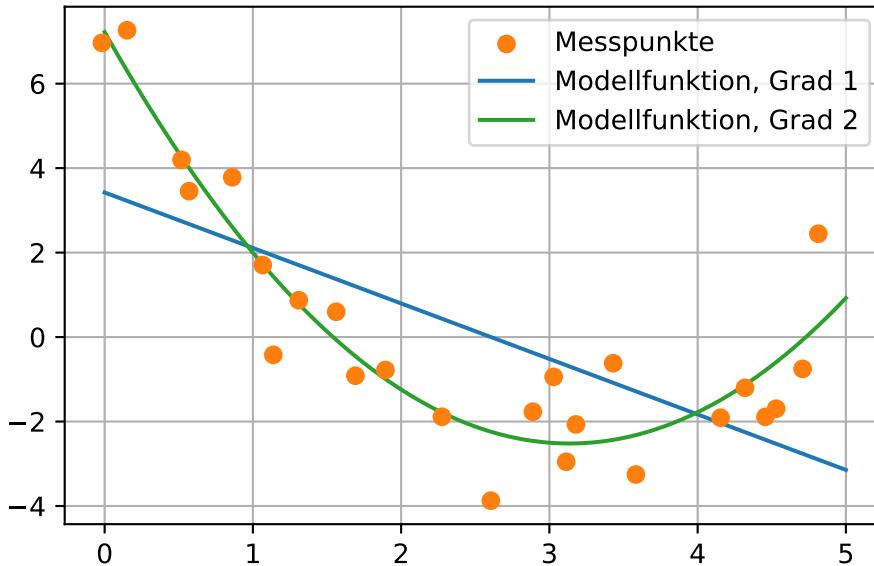
# x-Werte mit leichtem Rauschen
xi = np.linspace(xmin, xmax, ni) + 0.2*(2 * np.random.random(ni) -1)

# y(x) = 2x+0.5 mit leichtem Rauschen
yi = (xi - 2)**2 -2*xi + 2.5 + 2*(2 * np.random.random(ni) -1)

P1 = np.polyfit(xi, yi, 1)
P2 = np.polyfit(xi, yi, 2)

plt.scatter(xi, yi, color='C1', zorder=3, label='Messpunkte')
plt.plot(x, np.polyval(P1, x), color='C0', label="Modellfunktion, Grad 1")
plt.plot(x, np.polyval(P2, x), color='C2', label="Modellfunktion, Grad 2")
plt.grid()
plt.legend()
plt.show()

```



# 26 Splines

Polynominterpolation versucht eine globale Modellfunktion zu finden. Jedoch eignen sich Polynome mit hohen Graden im Allgemeinen nicht für eine Interpolation vieler Punkte. Einen anderen Ansatz verfolgen Splines. Diese sind Polynomzüge, welche die einzelnen Messpunkte verbinden und deren Grad klein – typischerweise zwischen eins und drei – ist.

## 26.1 Definition

Für  $n + 1$  Messpunkte  $(x_i, y_i)$  kann eine Splinefunktion  $s_k$ , hier ein Polynomspline, wie folgt definiert werden.

- Vorausgesetzt ist, dass die Messpunkte sortiert sind, d.h.  $x_0 < x_1 < \dots < x_n$
- für jedes  $i = 0 \dots n - 1$  ist  $s_k$  ein Polynom vom Grad  $k$  auf dem Intervall  $[x_i, x_{i+1}]$
- $s_k$  ist auf  $[x_0, x_n]$  ( $k - 1$ )-mal stetig differenzierbar

Beispiele: \*  $k = 1$ : Polygonzug \*  $k = 3$ : kubische Polynomsplines (B-Splines)

## 26.2 Kubische Splines

Die in der Praxis häufig eingesetzten kubischen Polynomsplines  $s_3$  ( $k = 3$ ) haben folgende Eigenschaften: \*  $s_3|_{[x_i, x_{i+1}]} = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$  \*  $s_3$  ist zweimal stetig differenzierbar auf  $[x_0, x_n]$ , also insbesondere an den Stützpunkten  $x_i$  der Messpunkte

Die Koeffizienten  $\beta_i$  werden wie folgt bestimmt \* aus den  $n + 1$  Messpunkten ergeben sich  $n$  Intervalle, d.h. mit jeweils vier Koeffizienten sind es insgesamt  $4n$  Koeffizienten \* Exakte Darstellung der Messpunkte ( $n + 1$  Gleichungen), d.h.:  $s_3(x_i) = y_i$  \* Glattheitsbedingungen an den inneren Messpunkten ( $i = 1 \dots n - 1$ ), mit jeweils ( $n - 1$  Gleichungen):

$$s'_3(x_i)_- = s'_3(x_i)_+$$

$$s''_3(x_i)_- = s''_3(x_i)_+$$

$$s'''_3(x_i)_- = s'''_3(x_i)_+$$

- Damit sind es  $4n - 2$  Gleichungen für  $4n$  Koeffizienten

Um die beiden fehlenden Gleichungen zu finden bzw. zu bestimmen werden Randbedingungen oder Abschlussbedingungen benötigt. Die gängigsten Bedingungen sind:

- \* natürliche Splines: die Krümmung am Rand verschwindet, d.h.:

$$s_3''(x_0) = s_3''(x_n) = 0$$

- \* periodische Splines: die Steigung und Krümmung ist an beiden Rändern gleich

$$s_3'(x_0) = s_3'(x_n)$$

$$s_3''(x_0) = s_3''(x_n)$$

- \* Hermite Splines: die Steigungen am Rand werden explizit vorgegeben (hier durch  $u$  und  $v$ )

$$s_3'(x_0) = u$$

$$s_3'(x_n) = v$$

## 26.3 Anwendung

Im Folgenden werden zwei Beispiele,  $s_1$  und  $s_3$ , für die Erstellung von Splines mit Python vorgestellt.

```
# Erzeugung von Messpunkten
n = 7
xi = np.linspace(0, np.pi, n)
yi = np.sin(xi)
```

Für die  $s_1$  Splines, kann die Funktion `np.interp` verwendet werden. Sie führt eine lineare Interpolation zwischen gegebenen Wertepaaren durch.

```
# Wertebereich für die Visualisierung der Interpolation
x = np.linspace(0, np.pi, n*6)
y = np.sin(x)
```

```
# Interpolation
y_s1 = np.interp(x, xi, yi)
```

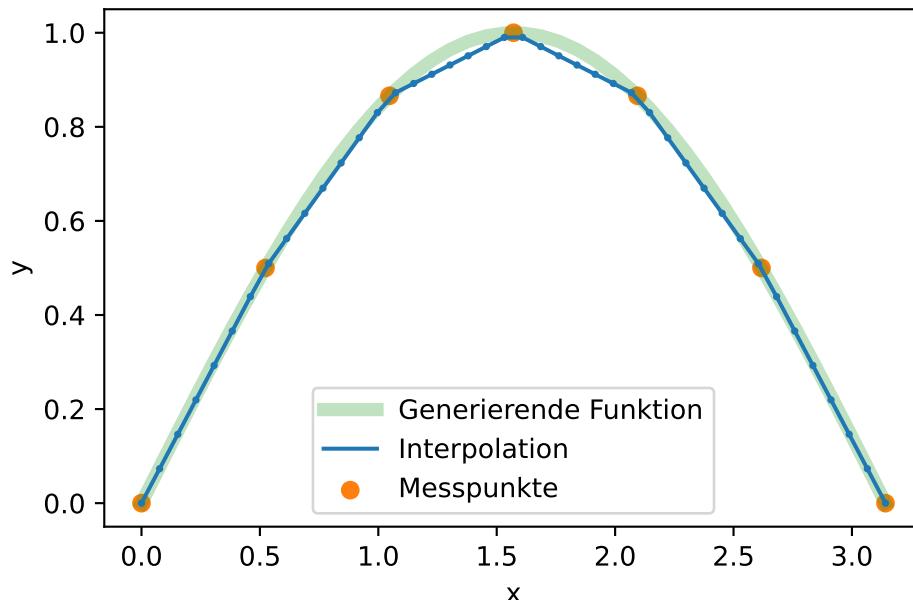
```
plt.plot(x,y, alpha=0.3, color='C2', lw=5,
          label='Generierende Funktion')
plt.plot(x, y_s1, color='C0', label='Interpolation')
plt.scatter(x, y_s1, s=3, zorder=3, color='C0')
```

```

plt.scatter(xi, yi, color='C1', label='Messpunkte')

plt.xlabel('x')
plt.ylabel('y')
plt.legend();

```



Die  $s_3$  Splines können mit Funktionen aus dem `scipy`-Modul berechnet werden. Dazu werden zunächst die Koeffizienten bestimmt (`scipy.interpolate.splrep`) und diese ermöglichen die gewünschte Auswertung, welche mit der Funktion `scipy.interpolate.splev` vorgenommen werden kann.

```
import scipy.interpolate as si
```

```

s3 = si.splrep(xi, yi)
y_s3 = si.splev(x, s3)

```

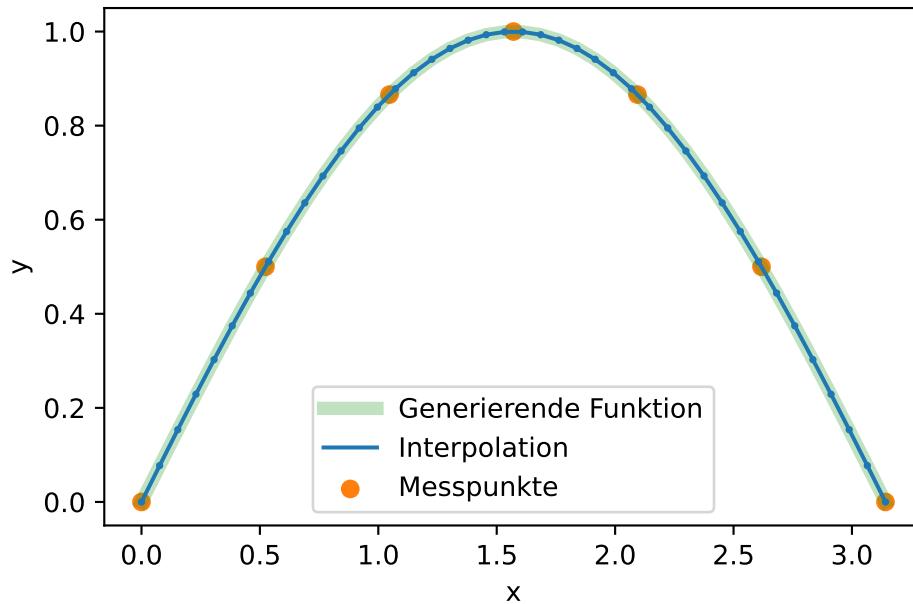
```

plt.plot(x,y, alpha=0.3, color='C2', lw=5,
          label='Generierende Funktion')
plt.plot(x, y_s3, color='C0', label='Interpolation')
plt.scatter(x, y_s3, s=3, zorder=3, color='C0')
plt.scatter(xi, yi, color='C1', label='Messpunkte')

plt.xlabel('x')

```

```
plt.ylabel('y')
plt.legend();
```



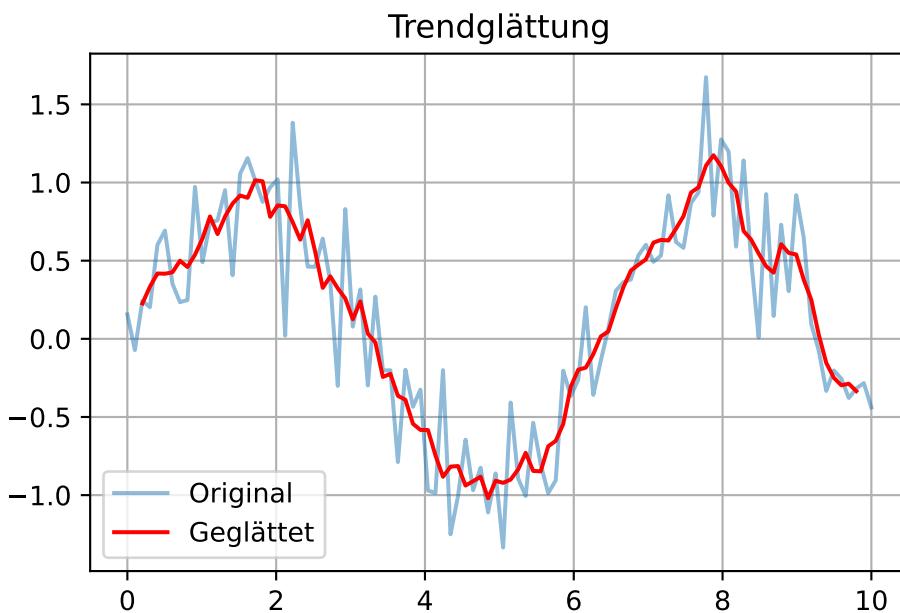
## 27 Trendglättung – Rauschen reduzieren

Verrauschte Daten? Ein **gleitender Mittelwert** glättet Kurven:

```
data = np.genfromtxt("trenddaten_mit_rauschen.csv", delimiter=",", skip_header=1)
x = data[:, 0]
y = data[:, 1]

window = 5
weights = np.ones(window) / window
y_smooth = np.convolve(y, weights, mode='valid')

plt.plot(x, y, label="Original", alpha=0.5)
plt.plot(x[(window-1)//2:-((window//2)]], y_smooth, label="Geglättet", color='red')
plt.legend()
plt.grid(True)
plt.title("Trendglättung")
plt.show()
```



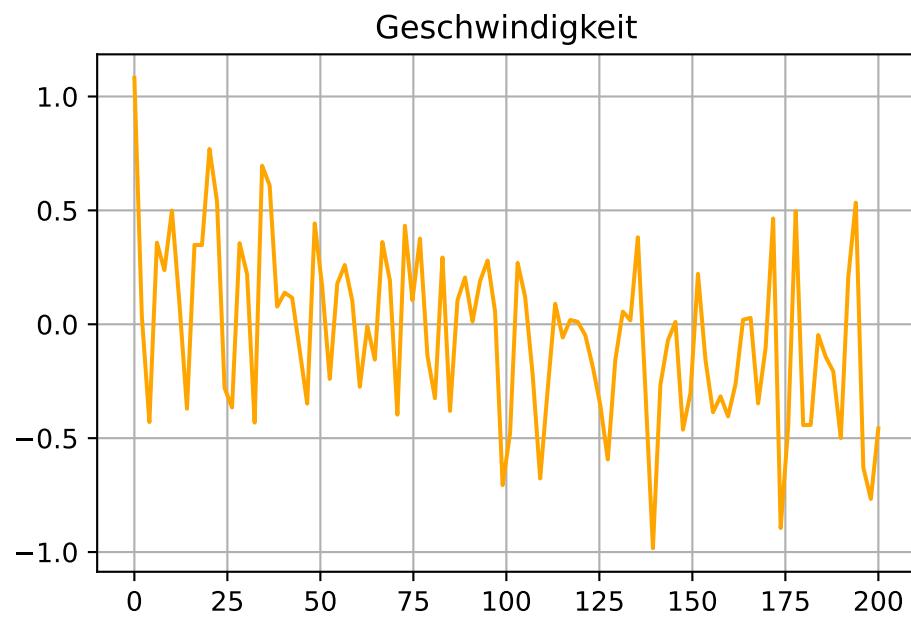
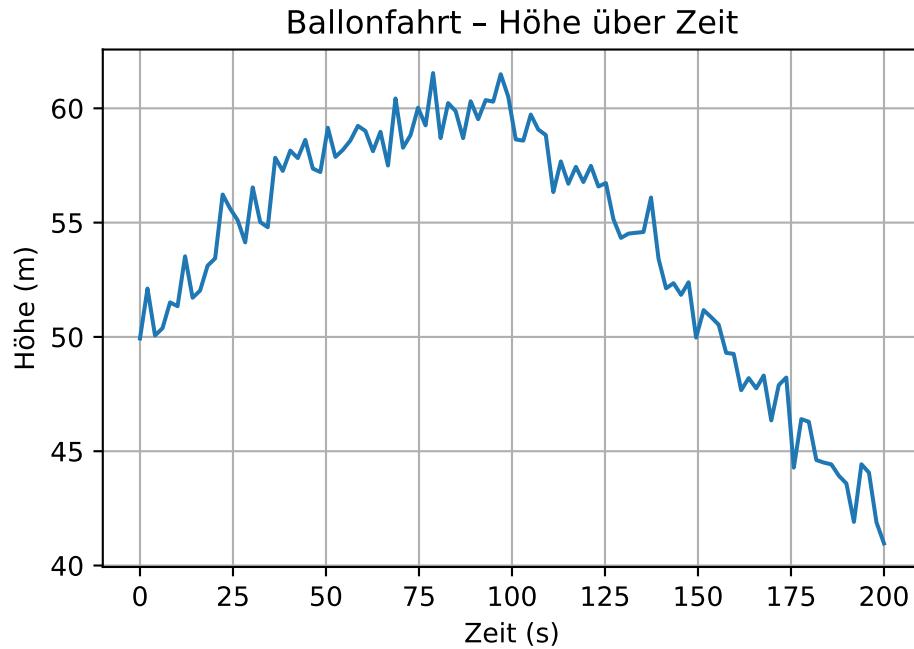
# 28 Anwendungsbeispiele

## 28.1 Anwendung: Ballonfahrt-Daten analysieren

```
ballon = np.genfromtxt("messdaten_ballonfahrt.txt", delimiter=",", skip_header=1)
zeit = ballon[:, 0]
hoehe = ballon[:, 1]

plt.plot(zeit, hoehe)
plt.title("Ballonfahrt - Höhe über Zeit")
plt.xlabel("Zeit (s)")
plt.ylabel("Höhe (m)")
plt.grid(True)
plt.show()

geschwindigkeit = np.gradient(hoehe, zeit)
plt.plot(zeit, geschwindigkeit, color="orange")
plt.title("Geschwindigkeit")
plt.grid(True)
plt.show()
```



## 28.2 Anwendung: Balkenverformung im Bauingenieurwesen

Ein Träger wird in der Mitte belastet. Die Durchbiegung wird an 50 Punkten gemessen:

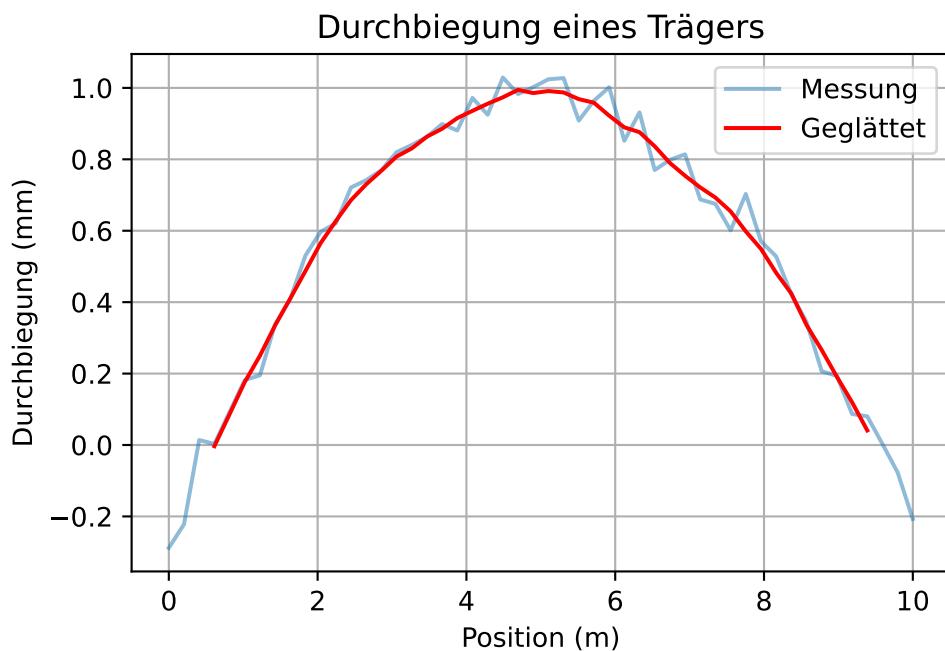
```

balken = np.genfromtxt("balken_durchbiegung.csv", delimiter=",", skip_header=1)
x = balken[:, 0]
y = balken[:, 1]

window = 7
weights = np.ones(window) / window
y_smooth = np.convolve(y, weights, mode='valid')

plt.plot(x, y, label="Messung", alpha=0.5)
plt.plot(x[(window-1)//2:-((window//2))], y_smooth, label="Geglättet", color='red')
plt.title("Durchbiegung eines Trägers")
plt.xlabel("Position (m)")
plt.ylabel("Durchbiegung (mm)")
plt.legend()
plt.grid(True)
plt.show()

```



## 28.3 Zusammenfassung

In dieser Einheit haben Sie gelernt:

- wie Daten eingelesen und bereinigt werden,
- wie man sie analysiert und visualisiert,
- wie Interpolation und Glättung funktionieren,
- wie reale Datensätze aus Technik und Naturwissenschaft ausgewertet werden können.

Diese Fähigkeiten sind grundlegend für jede datengetriebene Analyse im Ingenieurbereich.

# **Part VII**

# **Algorithmen**

# Einführung

## Definition

Ein Algorithmus ist eine formale Vorschrift darüber, wie die Lösung einer Fragestellung gefunden werden kann. Dabei handelt es sich meist um eine Folge von einfachen Anweisungen, welche zur Lösung komplexer Probleme führen kann.



Figure 28.1: Algorithmus

Algorithmen sollten so formuliert sein, dass sie nicht nur für einzelne explizite Fragestellungen, sondern auch im Allgemeinen anwendbar sind. Das wird am Beispiel des schriftlichen Dividierens oder anhand eines Kuchenrezepts deutlich. Beide bestehen aus einfachen Anweisungen und lösen ein komplexeres Problem. Allerdings kann die Rechenvorschrift für beliebige Divisionsaufgaben eingesetzt werden, während das Kuchenrezept nur zur Herstellung eines speziellen Kuchens führt.

Die obigen Beispiele für einen Algorithmus sind zwei von vielen, welche von Menschen eingesetzt werden (können):

- Schriftliches Rechnen
- Lösen von linearen Gleichungssystemen
- Bestimmung des Durchschnitts
- Lösen eines Zauberwürfels

Viele Algorithmen aus unserem Alltag sind aus sehr elementaren Anweisungen aufgebaut. Trotz der Einfachheit der Anweisungen, können sie von Menschen nicht eingesetzt werden, da die erforderliche Anzahl von Operationen sehr hoch sein kann. An dieser Stelle kommen Computer zum Einsatz. Wie in diesem Kapitel gezeigt wird, können mit den Grundrechenarten und Logischen Verknüpfungen komplexe Probleme gelöst werden.

## Beispiele

Beispiele für Algorithmen aus dem Alltag bzw. Ingenieurwesen, welche auf Computer zurückgreifen:

- Numerische Lösung von Differentialgleichungen (z.B. Strukturmechanik, Wärmetransport)
- Suchmaschinen im Internet

- Vorschläge beim online Einkaufen oder Medienkonsum
- Autonavigation

## 29 Von der Idee zum Code

Ein einfacher Algorithmus zur Bestimmung des maximalen Werts einer beliebig großen Menge von Zahlen ist wie folgt definiert:

1. Eingabe: Menge A von n Zahlen, hier durchnumerierte Werteliste  $A = A_0, \dots, A_{n-1}$ .
2. Setzte Hilfswert (Variable) m auf das erste Element der Liste, d.h.  $m = A_0$ .
3. Gehe alle Elemente von A durch, wobei das aktuelle Element als a bezeichnet wird:
4. Falls das aktuelle Element a größer ist als m: \* setze  $m = a$  \*, dann mache weiter mit dem nächsten Element in Schritt 3
5. Falls nicht: mache weiter mit dem nächsten Element in Schritt 3
6. Nachdem alle Elemente aus A in Schritt 3 durchlaufen wurden, enthält m den maximalen Wert der Liste A.

Dieser Algorithmus mag Ihnen auf den ersten Blick kompliziert. Gehen wir nochmal einen Schritt zurück und schauen uns einen Algorithmus an den wir bereits kennengelernt haben: Prüfen ob eine Zahl gerade ist.

1. Eingabe: Eine Zahl X
2. Berechne  $X \bmod 2$
3. Prüfe ob das Ergebnis gleich Null ist
4. Falls ja ist die Zahl gerade und wir geben den Text "die Zahl ist gerade" aus.
5. Falls nein ist die Zahl ungerade und wir geben den Text "die Zahl ist ungerade" aus.

Diesen Auflistung können wir uns besser visualisieren. Dazu benutzen wir sogenannte Flussdiagramme.

### Flussdiagramme - Visuelle Darstellung von Abläufen

Ein **Flussdiagramm** (engl. *flowchart*) ist eine grafische Methode zur Darstellung von Algorithmen. Es zeigt den Ablauf eines Programms oder Prozesses durch standardisierte Symbole und Pfeile. So lassen sich komplexe Abläufe leicht nachvollziehen und logisch überprüfen.

#### Typische Symbole:

- **Prozess (Anweisung):** Rechteck – z. B. „Berechne Fläche“
- **Entscheidung:** Raute – z. B. „Ist  $x > 0?$ “
- **Start/Ende:** Ellipse – z. B. „für alle a in A“

- **Pfeile:** Zeigen den Ablauf von einem Schritt zum nächsten

**Beispiel:** Der Algorithmus zur Bestimmung, ob eine Zahl gerade ist:

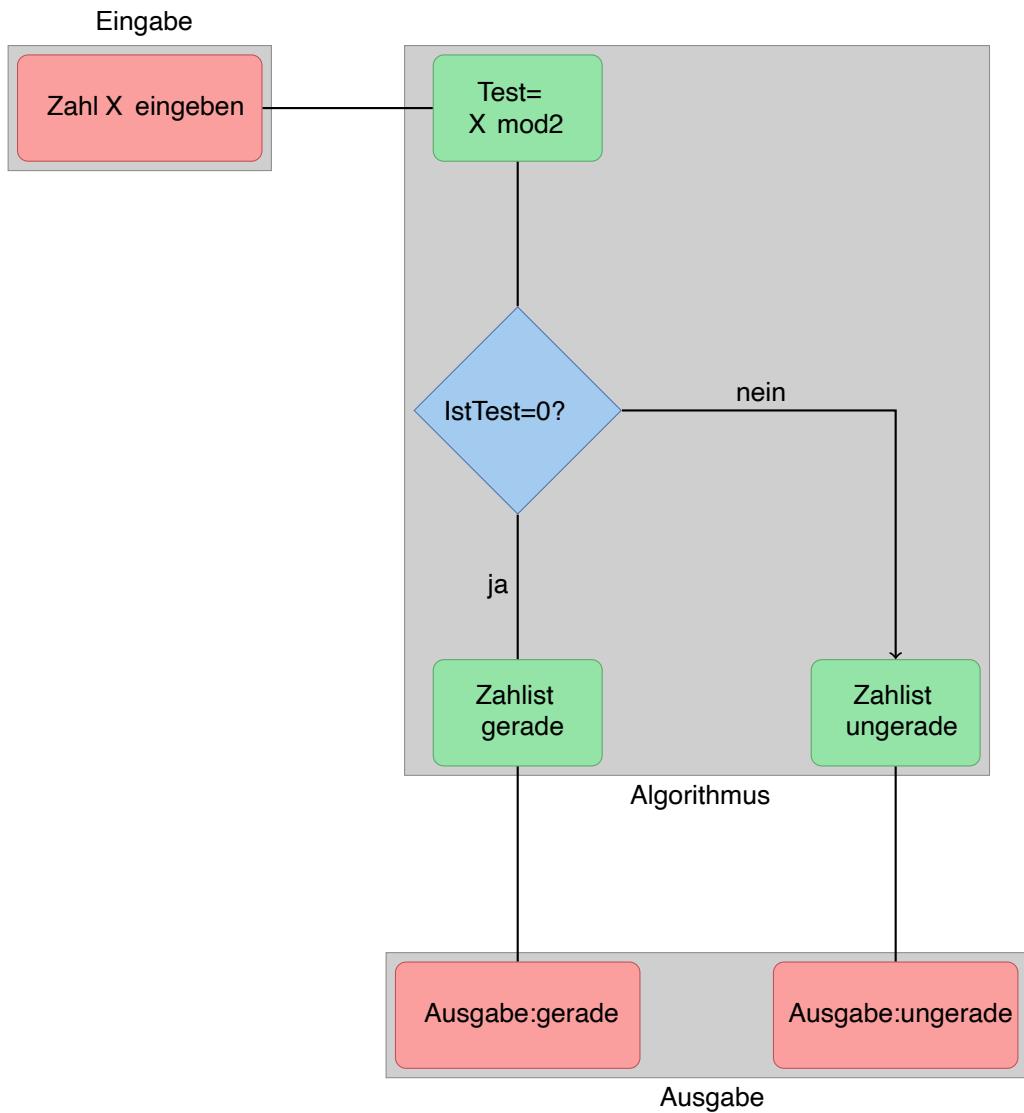


Figure 29.1: Flussdiagramm

Flussdiagramme sind nicht der einzige Weg, um komplexere Algorithmen verständlicher aufzuschreiben. Eine weitere Möglichkeit bietet hier sogenannter Pseudocode:

## Pseudocode – Vom Gedanken zum Programm

**Pseudocode** ist eine formalisierte Beschreibung eines Algorithmus in einfacher, strukturierter Sprache – eine Mischung aus natürlicher Sprache und Programmierlogik. Er ist **sprachunabhängig** und dient zur Planung, nicht zur direkten Ausführung.

**Typische Merkmale:** - Klare **Schritt-für-Schritt-Struktur** - Verwendung von **Kontrollstrukturen** wie **wenn**, **solange**, **wiederhole** - Keine konkrete Syntax einer Programmiersprache

**Beispiel:**

```
BEGIN
    Lese Zahl x ein
    WENN x mod 2 = 0 DANN
        Gib "x ist gerade" aus
    SONST
        Gib "x ist ungerade" aus
ENDE
```

Kommen wir nun zu dem “komplexeren” Algorithmus zurück, der das Maximum in einer Liste von Zahlen finden soll:

1. Eingabe: Menge A von n Zahlen, hier durchnumerierte Werteliste  $A = A_0, \dots, A_{n-1}$ .
2. Setze Hilfswert (Variable) m auf das erste Element der Liste, d.h.  $m = A_0$ .
3. Gehe alle Elemente von A durch, wobei das aktuelle Element als a bezeichnet wird:
4. Falls das aktuelle Element a größer ist als m: \* setze  $m = a$  \*, dann mache weiter mit dem nächsten Element in Schritt 3
5. Falls nicht: mache weiter mit dem nächsten Element in Schritt 3
6. Nachdem alle Elemente aus A in Schritt 3 durchlaufen wurden, enthält m den maximalen Wert der Liste A.

Testen Sie einmal selbst, ob Sie das passende Flussdiagramm erstellen können. In der folgenden Box finden Sie die Musterlösung. Sie brauchen hierfür auch die Verzweigung für Schleifen: **Verzweigung** (blau): Abfrage einer Bedingung, welche entscheidet welche folgenden Elemente ausgeführt werden, hier wird geprüft ob  $a > m$

## Flussdiagramm: Maximumsuche

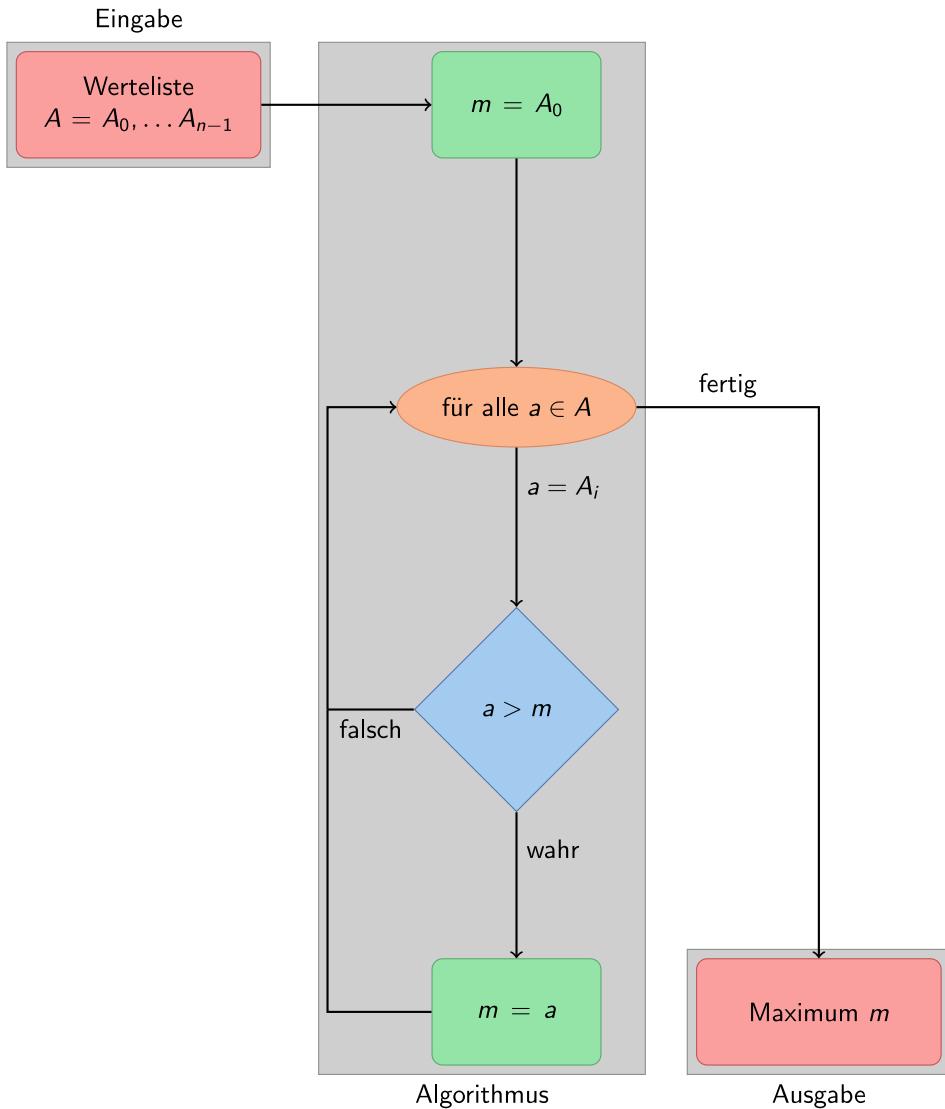


Figure 29.2: Algorithmus zur Bestimmung des Maximums einer Zahlenliste

Wie oben beschrieben, können wir aber nicht nur Flussdiagramme zur Darstellung von Algorithmen nutzen, sondern auch sogenannten Pseudocode. Versuchen Sie einmal selbst den Algorithmus zur Maximumsuche in Pseudocode zu formulieren.

### 🔥 Pseudocode: Maximumssuche

```
Eingabe: Liste von Zahlen L (z. B. L = [5, 8, 2, 9, 3])  
  
1. Setze max_wert := L[0]           // Initialisiere Hilfswert mit dem ersten Element der Liste  
  
2. Für jedes Element x in L:  
    a. Falls x > max_wert:  
        i. Setze max_wert := x  
    b. Andernfalls:  
        i. Tue nichts (fortfahren)  
  
3. Ausgabe: max_wert           // max_wert enthält nun den größten Wert der Liste
```

Ein Beispiel für den Ablauf des Algorithmus für eine Liste von 20 Zahlen ist:

```
import numpy as np  
np.set_printoptions(linewidth=50)  
# A = np.random.randint(0, high=1000, size=20)  
A = np.array([203, 433, 504, 602, 567, 762, 183, 482, 471, 741, 854, 486, 350, 550, 885, 395  
  
print('Schritt 1:')  
print('=====')  
print('A = ', np.array2string(A, separator=', '))  
  
print()  
print('Schritt 2:')  
print('=====')  
m = A[0]  
print('m = A[0] =', m)  
  
print()  
print('Schritt 3:')  
print('=====')  
for a in A:  
    print('a = {:3d}, m = {:3d}'.format(a, m), end=' ')  
    if a > m:  
        m = a  
        print(', da a > m ist, setzte m auf m=', m)  
    else:  
        print()
```

```
print()
print('Schritt 4:')
print('=====')
print('Maximaler Wert in A: m =', m)
```

Schritt 1:

```
=====
A = [203, 433, 504, 602, 567, 762, 183, 482, 471, 741,
     854, 486, 350, 550, 885, 395, 203, 288, 909, 644]
```

Schritt 2:

```
=====
m = A[0] = 203
```

Schritt 3:

```
=====
a = 203, m = 203
a = 433, m = 203, da a > m ist, setzte m auf m= 433
a = 504, m = 433, da a > m ist, setzte m auf m= 504
a = 602, m = 504, da a > m ist, setzte m auf m= 602
a = 567, m = 602
a = 762, m = 602, da a > m ist, setzte m auf m= 762
a = 183, m = 762
a = 482, m = 762
a = 471, m = 762
a = 741, m = 762
a = 854, m = 762, da a > m ist, setzte m auf m= 854
a = 486, m = 854
a = 350, m = 854
a = 550, m = 854
a = 885, m = 854, da a > m ist, setzte m auf m= 885
a = 395, m = 885
a = 203, m = 885
a = 288, m = 885
a = 909, m = 885, da a > m ist, setzte m auf m= 909
a = 644, m = 909
```

Schritt 4:

```
=====
Maximaler Wert in A: m = 909
```

## **29.1 Kapitelübersicht**

In diesem Kapitel werden folgende Themen behandelt:

- Sortieralgorithmen
- Eigenschaften von Algorithmen
- Numerische Algorithmen

# 30 Sortieralgorithmen

Sortieralgorithmen werden genutzt um Listen von Werten der Größe nach zu sortieren. Anwendung finden diese Algorithmen bei Datenbanken oder Suchvorgängen. Insbesondere bei langen Listen mit Millionen oder Milliarden Einträgen ist es wichtig, dass der Algorithmus mit möglichst wenigen Operationen pro Element auskommt. Diese, als Komplexität bezeichnete Eigenschaft, wird im nächsten Kapitel genauer erläutert.

Zunächst werden zwei einfache Sortieralgorithmen

- Selectionsort
- Bubblesort

vorgestellt. Diese werden in der Praxis kaum noch eingesetzt, da es eine Vielzahl anderer **Sortierverfahren** gibt, welche effektiver arbeiten. Jedoch eignen sich diese beiden besonders gut, um die Grundideen zu verdeutlichen.

## 30.1 Selectionsort

Folgende Grundidee liegt dem Selectionsort zugrunde: Es wird der minimale Wert der Liste gesucht, dann der zweit-kleinste und so weiter bis die ganze Liste sortiert ist. Dies kann als Abfolge dieser Schritte für eine Liste mit  $n$  Elementen formalisiert werden.

1. Wiederhole die Schritte 2 bis 4  $n$  Mal. Setzte die Hilfsvariable  $i$  initial auf Null.
2. Suche den minimalen Wert der Liste ab dem  $i$ -ten Element.
3. Tausche dieses Element mit dem  $i$ -ten Element.
4. Erhöhe den Wert von  $i$  um Eins.
5. Die Vertauschungen der Elemente haben zu einer sortierten Liste geführt.

Der Selectionsort kann auch als folgendes Flussdiagramm dargestellt werden.

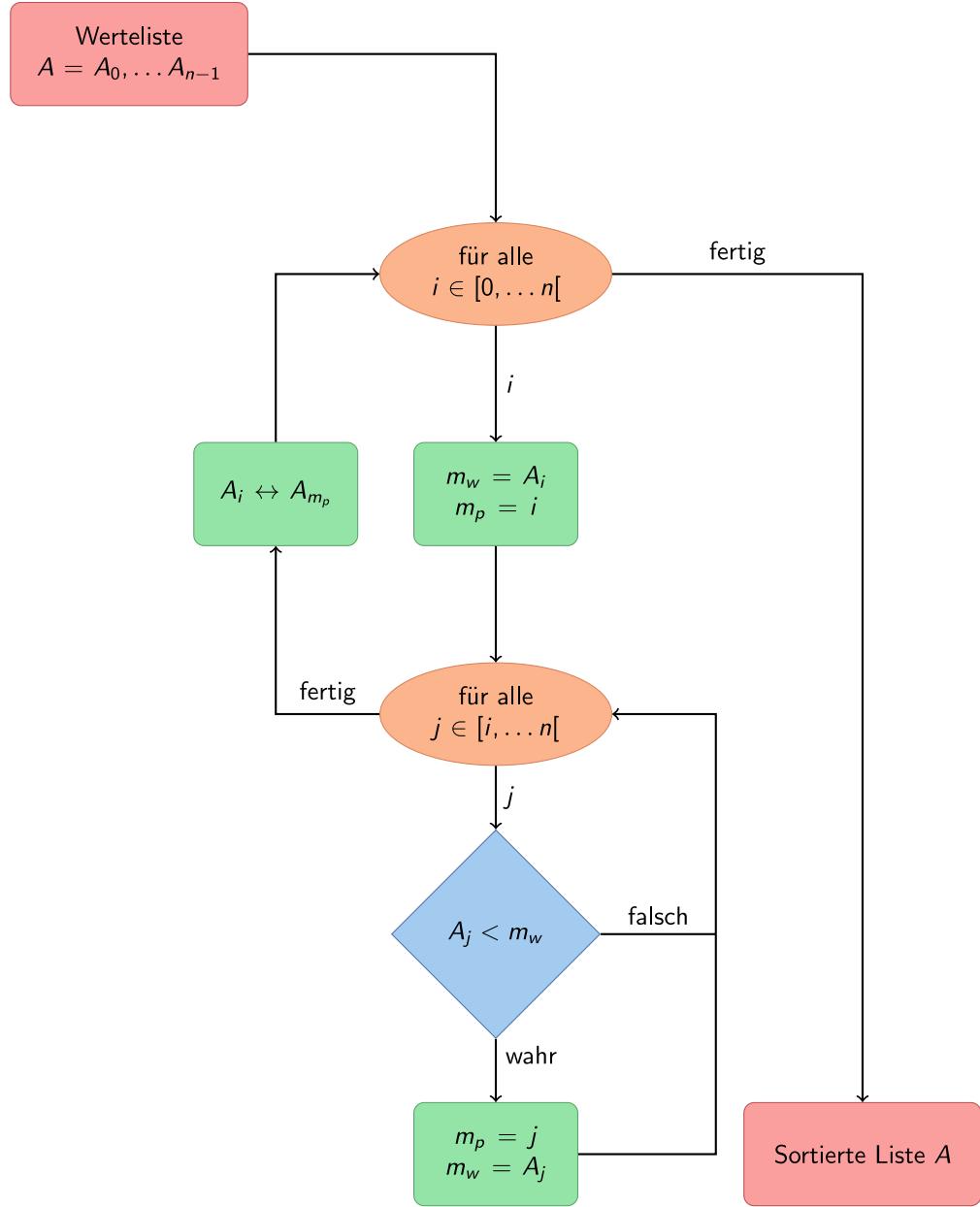


Figure 30.1: Flussdiagramm des Selectionsort

Als Zahlenbeispiel wird die Liste mit den Elementen 42, 6, 22, 11, 54, 12, 31 mit dem Selectionsort sortiert.

```
A = [42, 6, 22, 11, 54, 12, 31]
```

```

print('Zu sortierende Werteliste ', A)
print()

n = len(A)
for i in range(n):
    mv = A[i]
    mi = i
    for j in range(i, n):
        if A[j] < mv:
            mv = A[j]
            mi = j

    print("Iteration {:2d}: ".format(i+1))
    print("  Minimum von ", A[i:n], "ist", mv)
    A[mi] = A[i]
    A[i] = mv
    print("  Sortiert / Unsortiert: ", A[:i+1], "/", A[i+1:])
    print()

```

Zu sortierende Werteliste [42, 6, 22, 11, 54, 12, 31]

Iteration 1:

Minimum von [42, 6, 22, 11, 54, 12, 31] ist 6  
Sortiert / Unsortiert: [6] / [42, 22, 11, 54, 12, 31]

Iteration 2:

Minimum von [42, 22, 11, 54, 12, 31] ist 11  
Sortiert / Unsortiert: [6, 11] / [22, 42, 54, 12, 31]

Iteration 3:

Minimum von [22, 42, 54, 12, 31] ist 12  
Sortiert / Unsortiert: [6, 11, 12] / [42, 54, 22, 31]

Iteration 4:

Minimum von [42, 54, 22, 31] ist 22  
Sortiert / Unsortiert: [6, 11, 12, 22] / [54, 42, 31]

Iteration 5:

Minimum von [54, 42, 31] ist 31  
Sortiert / Unsortiert: [6, 11, 12, 22, 31] / [42, 54]

Iteration 6:

```
Minimum von [42, 54] ist 42
Sortiert / Unsortiert: [6, 11, 12, 22, 31, 42] / [54]
```

Iteration 7:

```
Minimum von [54] ist 54
Sortiert / Unsortiert: [6, 11, 12, 22, 31, 42, 54] / []
```

## 30.2 Bubblesort

Im Gegensatz zum Selectionsort beruht die Idee des Bubblesort auf rein lokalen Operationen. D.h. hier wird nicht nach den maximalen Werten gesucht, sondern durch Vertauschungen eine Sortierung erzielt. Das Verfahren für eine Liste mit  $n$  Elementen ist durch folgende Vorschrift gegeben.

1. Die Schritte 2 bis 4 werden  $n$  Mal durchgeführt. Die Hilfsvariable  $i$  wird initial auf Null gesetzt.
2. Starte beim  $i$ -ten Element und iteriere bis zum Ende der Liste. Falls das aktuell betrachtete Element größer ist als das Folgende, tausche beide Elemente.
3. Falls in Schritt 2 keine Vertauschungen durchgeführt wurden, gehe zu Schritt 5.
4. Addiere Eins zum Wert der Variable  $i$ .
5. Die Liste ist sortiert und der Algorithmus ist fertig.

Das nachfolgende Flussdiagramm verdeutlicht den Ablauf des Bubblesort Algorithmus. Bevor Sie sich das Diagramm anschauen, versuchen Sie es einmal selbst zu erstellen.

🔥 Flussdiagramm: Bubblesort

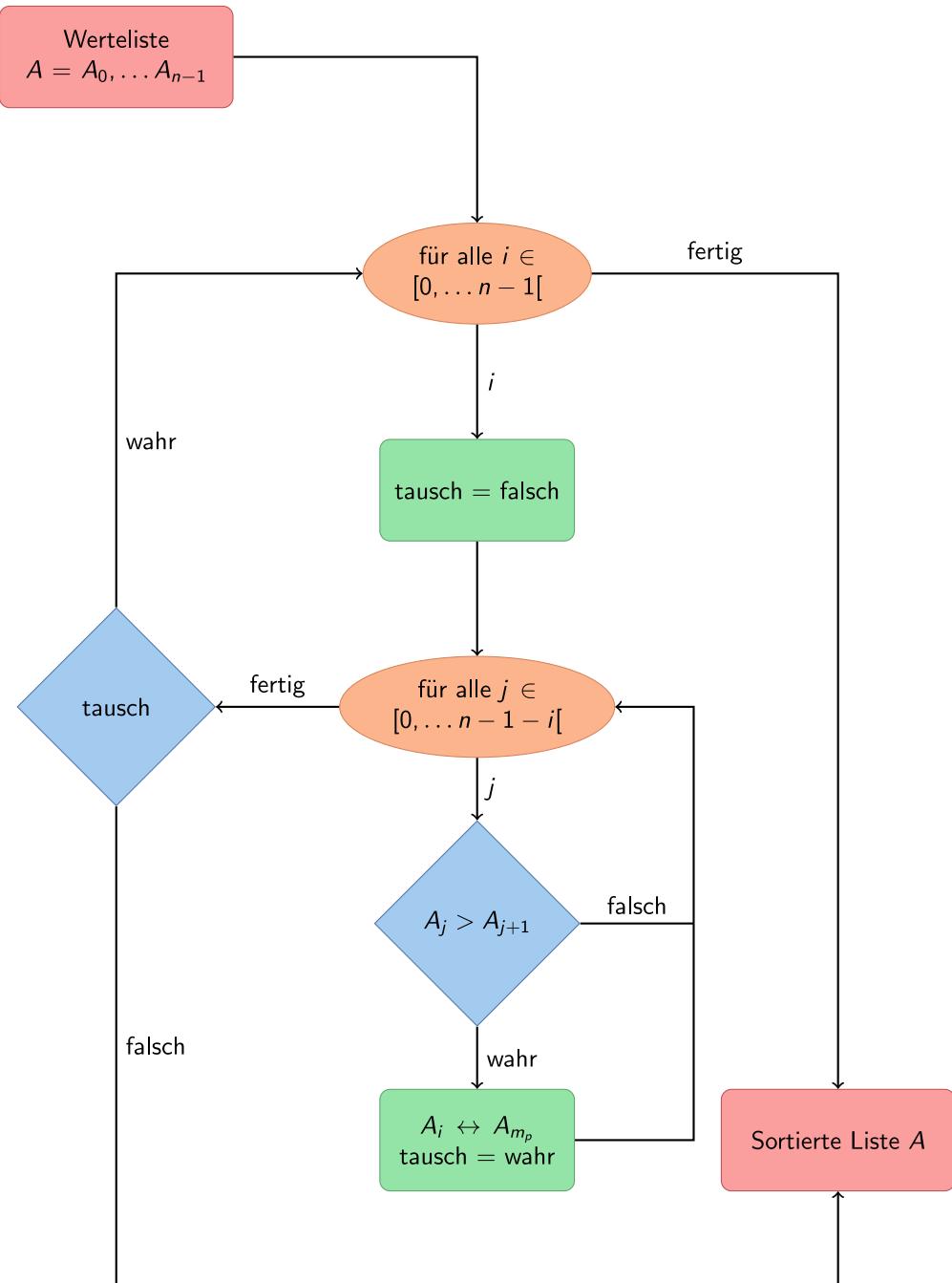


Figure 30.2: Flussdiagramm des Bubblesort

Der Ablauf des Bubblesort wird beispielhaft für die Liste 42, 6, 22, 11, 54, 12, 31 (gleich der im obigen Beispiel) vorgeführt.

```
A = [42, 6, 22, 11, 54, 12, 31]

print('Zu sortierende Werteliste ', A)
print()

n = len(A)
swapped = True
i = 0
while swapped:
    swapped = False
    print("Iteration {:2d}: ".format(len(A) - n + 1))
    print("  Liste zu Beginn der Iteration: ", A)
    for j in range(n-1):
        if A[j+1] < A[j]:
            print("    Tausche: ", A[j], "und", A[j+1])
            mv = A[j]
            A[j] = A[j+1]
            A[j+1] = mv
            print("    Liste nach Tausch: ", A)
            swapped = True
        else:
            print("    Elemente ", A[j], "und", A[j+1], " müssen nicht getauscht werden")
    n -= 1
if not swapped:
    print("  kein Tausch mehr notwendig, Liste ist nun sortiert")
print()
```

Zu sortierende Werteliste [42, 6, 22, 11, 54, 12, 31]

Iteration 1:

```
  Liste zu Beginn der Iteration: [42, 6, 22, 11, 54, 12, 31]
  Tausche: 42 und 6
  Liste nach Tausch: [6, 42, 22, 11, 54, 12, 31]
  Tausche: 42 und 22
  Liste nach Tausch: [6, 22, 42, 11, 54, 12, 31]
  Tausche: 42 und 11
  Liste nach Tausch: [6, 22, 11, 42, 54, 12, 31]
  Elemente 42 und 54 müssen nicht getauscht werden
  Tausche: 54 und 12
```

Liste nach Tausch: [6, 22, 11, 42, 12, 54, 31]

Tausche: 54 und 31

Liste nach Tausch: [6, 22, 11, 42, 12, 31, 54]

Iteration 2:

Liste zu Beginn der Iteration: [6, 22, 11, 42, 12, 31, 54]

Elemente 6 und 22 müssen nicht getauscht werden

Tausche: 22 und 11

Liste nach Tausch: [6, 11, 22, 42, 12, 31, 54]

Elemente 22 und 42 müssen nicht getauscht werden

Tausche: 42 und 12

Liste nach Tausch: [6, 11, 22, 12, 42, 31, 54]

Tausche: 42 und 31

Liste nach Tausch: [6, 11, 22, 12, 31, 42, 54]

Iteration 3:

Liste zu Beginn der Iteration: [6, 11, 22, 12, 31, 42, 54]

Elemente 6 und 11 müssen nicht getauscht werden

Elemente 11 und 22 müssen nicht getauscht werden

Tausche: 22 und 12

Liste nach Tausch: [6, 11, 12, 22, 31, 42, 54]

Elemente 22 und 31 müssen nicht getauscht werden

Iteration 4:

Liste zu Beginn der Iteration: [6, 11, 12, 22, 31, 42, 54]

Elemente 6 und 11 müssen nicht getauscht werden

Elemente 11 und 12 müssen nicht getauscht werden

Elemente 12 und 22 müssen nicht getauscht werden

kein Tausch mehr notwendig, Liste ist nun sortiert

# 31 Eigenschaften

## 31.1 Terminiertheit

Terminiertheit bedeutet, dass ein Algorithmus nach endlich vielen Schritten anhält, oder er bricht kontrolliert ab. Einfache Beispiele:

- Addition zweier Dezimalzahlen
- Summe der ersten N natürlichen Zahlen

Allerdings kann die Terminiertheit nicht für alle Algorithmen gezeigt werden. Das [Halteproblem](#) besagt, dass es gibt keinen Verfahren gibt, welches immer zutreffend sagen kann, ob der Algorithmus für die Eingabe terminiert. Hierzu kann das [Collatz-Problem](#) als Beispiel herangezogen werden.

Die Zahlenfolge wird wie folgt konstruiert:

- beginne mit irgendeiner natürlichen Zahl  $n_0 > 0$
- ist  $n_i$  gerade so ist  $n_{i+1} = n_i/2$
- ist  $n_i$  ungerade so ist  $n_{i+1} = 3n_i + 1$
- endet bei  $n_i = 1$

Collatz-Vermutung: Jede so konstruierte Zahlenfolge mündet in den Zyklus 4, 2, 1, egal, mit welcher natürlichen Zahl man beginnt. Bisher unbewiesen.

## 31.2 Determiniertheit

Ein deterministischer Algorithmus ist ein Algorithmus, bei dem nur definierte und reproduzierbare Zustände auftreten. Die Ergebnisse des Algorithmus sind somit immer reproduzierbar. Beispiele hierfür:

- Addition ganzer Zahlen
- Selectionsort
- Collatz-Sequenz

### 31.3 Effizienz

Die Effizienz eines Algorithmus ist nicht strikt definiert und kann folgende Aspekte umfassen:

- Laufzeit
- Speicheraufwand
- Energieverbrauch

Bei bestimmten Anwendungen sind entsprechende Eigenschaften notwendig:

- Speicheraufwand bei *Big Data*, also riesige Datenmengen, z.B. in der Bioinformatik
- Laufzeit bei Echtzeitanwendung, z.B. Flugzeugsteuerung, Fußgängerdynamik

### 31.4 Komplexität

Bei der Analyse von Algorithmen, gilt es die Komplexität zu bestimmen, welche ein Maß für den Aufwand darstellt. Dabei wird nach einer Aufwandfunktion  $f(n)$  gesucht, welche von der Problemgröße  $n$  abhängt. Beispiel für eine Problemgröße:

- Anzahl der Summanden bei einer Summe
- Anzahl der zu sortierenden Zahlen

Meist wird dabei die Bestimmung auf eine asymptotische Analyse, d.h. eine einfache Vergleichsfunktion  $g(n)$  mit  $n \rightarrow \infty$ , reduziert. Dabei beschränkt  $g(n)$  das Wachstum von  $f(n)$ .

Die Funktion  $g(n)$  wird oft durch ein  $\mathcal{O}$  gekennzeichnet und gibt so eine möglichst einfache Vergleichsfunktion an. Beispiele:

- $f_1(n) = n^4 + 5n^2 - 10 \approx \mathcal{O}(n^4) = g_1(n)$
- $f_2(n) = 2^{n+1} \approx \mathcal{O}(2^n) = g_2(n)$

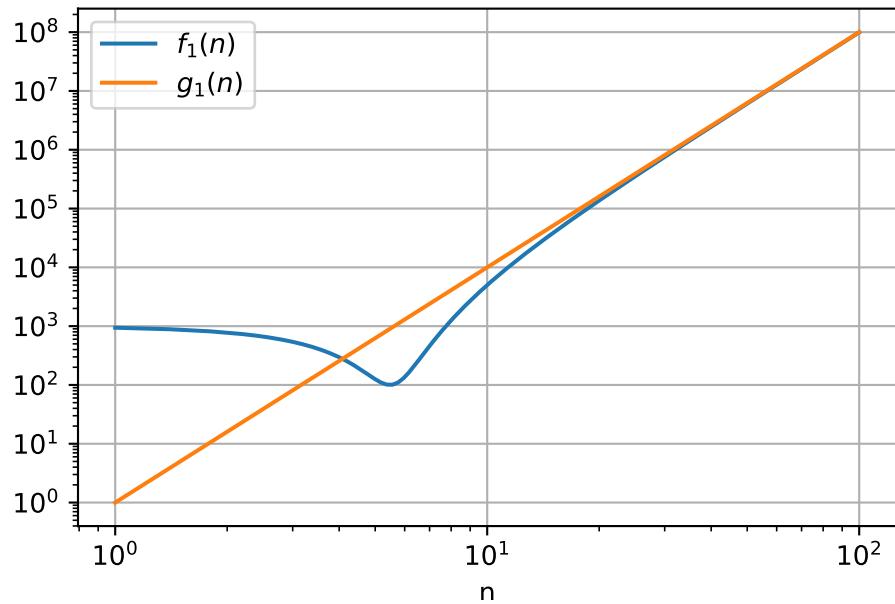


Figure 31.1: Komplexität eines Algorithmus durch Vergleich einer Aufwandfunktion mit einer Vergleichsfunktion

Um sich ein besseres Bild zu den Auswirkungen hoher Komplexitäten zu machen, sei folgendes Beispiel gegeben.

- ein Berechnungsschritt (unabhängig von der Problemgröße  $n$ ) sei z.B. 1 s lang
- das  $n$  sei beispielsweise 1000

Damit ergeben sich folgende (asymptotische) Abschätzungen der Laufzeit:

- $\mathcal{O}(n)$ : 103 s 1 h
- $\mathcal{O}(n^2)$ : 106 s 11 d
- $\mathcal{O}(n^3)$ : 109 s 31 a
- $\mathcal{O}(2^n)$ : 21000 s ...

### 31.4.1 Komplexität Selectionsort

Die Komplexität dieses Verfahrens kann leicht abgeschätzt werden. Bei jedem Durchlauf wir das Minimum / Maximum gesucht, was anfangs  $n$  Operationen benötigt. Beim nächsten Durchlauf sind es nur noch  $n - 1$  Operationen und so weiter. In der Summe sind es also

$$f(n) = \sum_{i=0}^n i = \frac{n(n-1)}{2} \approx \mathcal{O}(n^2)$$

Damit hat der Selectionsort eine Komplexität von  $\mathcal{O}(n^2)$ . Die folgende Abbildung verdeutlicht dies nochmals.

10	41.901
20	180.802
30	419.715
40	757.951
50	1192.501
60	1733.146
70	2367.245
80	3109.177
90	3941.419
100	4881.606

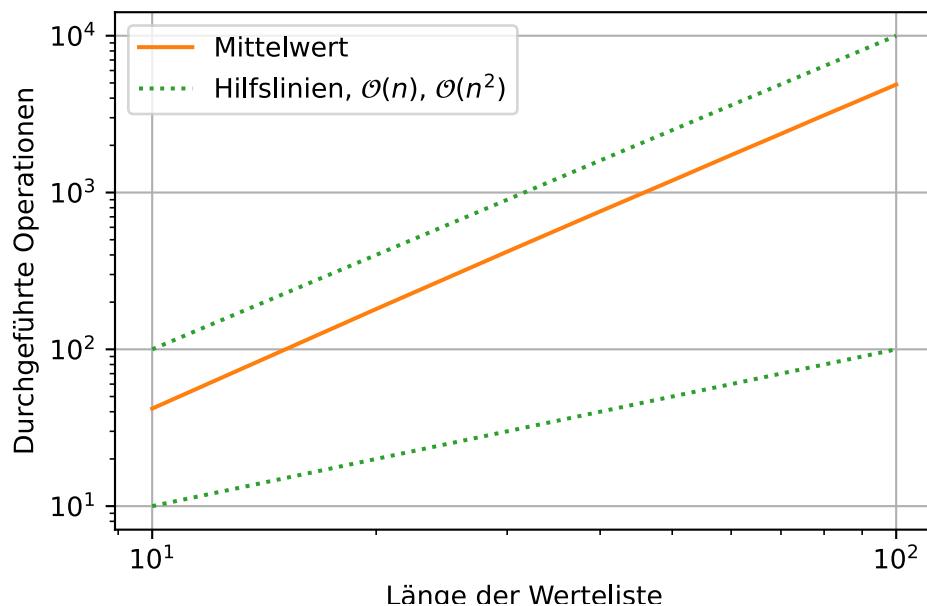


Figure 31.2: Abschätzung der Komplexität des Selectionsort-Algorithmus

### 31.4.2 Komplexität Bubblesort

Die Komplexität des Bubblesort muss unterschieden werden in den günstigsten Fall (best case), den ungünstigsten Fall (worst case) und einem durchschnittlichen Fall (average case):

- best case:  $\mathcal{O}(n)$
- worst case:  $\mathcal{O}(n^2)$
- average case:  $\mathcal{O}(n^2)$

Der best case ergibt sich zum Beispiel, falls die Eingabeliste bereits sortiert ist, da der Algorithmus nur einmal durch die Liste gehen muss, entsprechend  $n$ -Mal. Folgende Abbildung verdeutlicht die Anzahl der durchgeführten Operationen im Falle einer vollständig zufälligen Liste und einer, bei welcher 95% der Werte bereits sortiert ist. Dabei wurden für jedes  $n$  jeweils 10000 Listen sortiert. Es ist der Mittelwert und die minimalen und maximalen Operationen dargestellt.

10	41.9846
20	180.8287
30	418.8286
40	756.7754
50	1194.2935
60	1731.6382
70	2367.9949
80	3105.2446
90	3942.3673
100	4879.6782

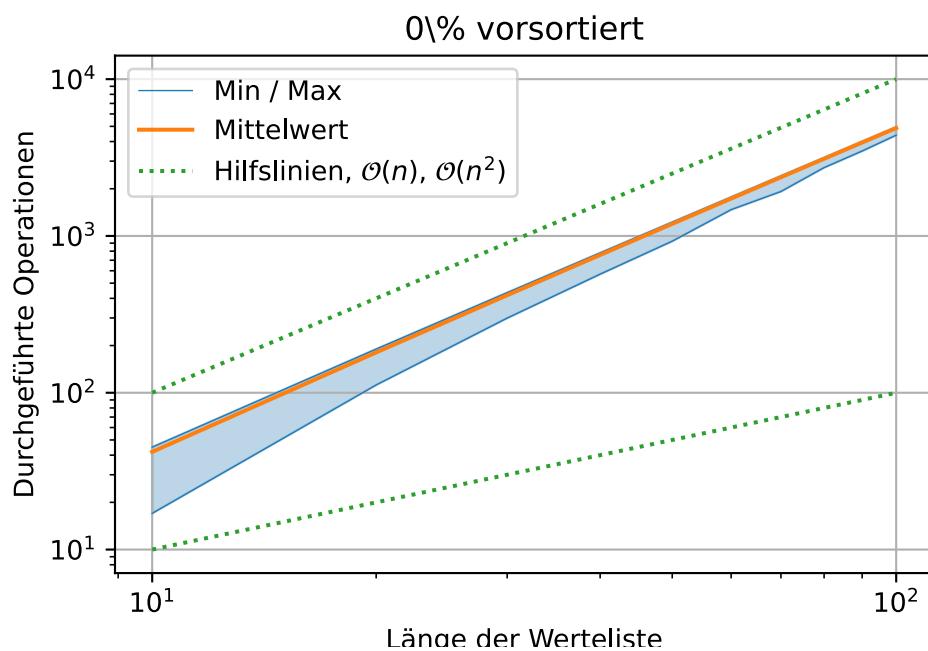


Figure 31.3: Abschätzung der Komplexität des Bubblesort-Algorithmus ohne Vorsortierung

10	29.9801
20	126.7374
30	304.3454
40	541.1145
50	908.5235
60	1307.368
70	1859.6947
80	2439.4462
90	3184.0767
100	3957.0259

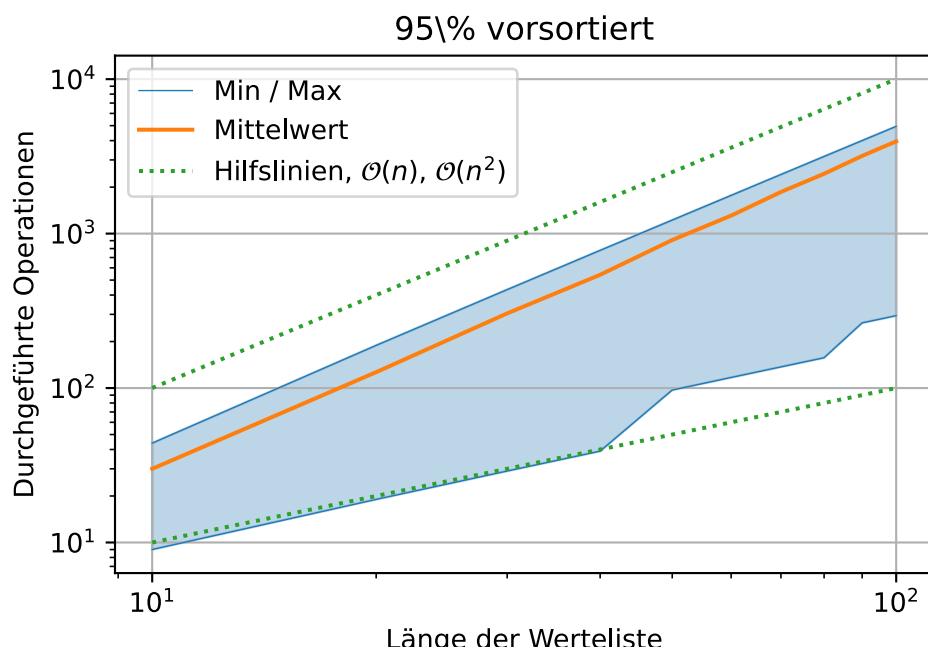


Figure 31.4: Abschätzung der Komplexität des Bubblesort-Algorithmus mit einer 95%-igen Vorsortierung

# 32 Numerische Algorithmen

## 32.1 Newton-Raphson-Verfahren

Eines der einfachsten und auch ältesten Verfahren zur Suche von Nullstellen von Funktionen ist das [Newton-Raphson-Verfahren](#), welches bereits im 17-ten Jahrhundert entwickelt und eingestetzt wurde.

### 32.1.1 Anwendungen

Das Finden von Nullstellen ist die Grundlage für viele Verfahren, welche z.B. für

- das Lösen von nicht-linearen Gleichungen,
- das Finden von Extremwerten, oder
- Optimierungsverfahren

eingesetzt werden kann.

### 32.1.2 Grundidee

Die Grundidee beruht auf einer iterativen Suche der Nullstelle  $x_{ns}$  einer stetig differenzierbaren Funktion  $f(x)$  mit Hilfe der ersten Ableitung  $f'(x)$ . Durch das Anlegen von Tangenten an die aktuelle Näherung der Nullstelle  $x_i$  kann die nächste Näherung bestimmt werden.

Bei gegebenen Startwert,  $x_0$  für den ersten Iterationsschritt ( $i = 0$ ), können die folgenden Näherungen durch

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

berechnet werden. Dabei bestimmt die Wahl des Startwerts, welche der ggf. mehreren Nullstellen gefunden wird.

### 32.1.3 Beispiel 1

Gegeben ist die Funktion  $f(x) = x^2 - 1$ . Die Ableitung ist gegeben durch  $f'(x) = 2x$  und die Nullstellen lauten  $x_{ns} = \{-1, 1\}$ .

Bei einem Startwert von  $x_0 = 0.3$  führt zu folgender Iteration:

Startwert  $x_0 = 0.3000$

```
Iterationsschritt i = 1, x_i = 0.3000
f(x_i) = -0.9100
fp(x_i) = 0.6000
x_(i+1) = 1.8167
```

```
Iterationsschritt i = 2, x_i = 1.8167
f(x_i) = 2.3003
fp(x_i) = 3.6333
x_(i+1) = 1.1836
```

```
Iterationsschritt i = 3, x_i = 1.1836
f(x_i) = 0.4008
fp(x_i) = 2.3671
x_(i+1) = 1.0142
```

```
Iterationsschritt i = 4, x_i = 1.0142
f(x_i) = 0.0287
fp(x_i) = 2.0285
x_(i+1) = 1.0001
```

Endergebnis nach 5 Iterationen:  $x_{(ns)} = 1.0001$

<Figure size 1650x1050 with 0 Axes>

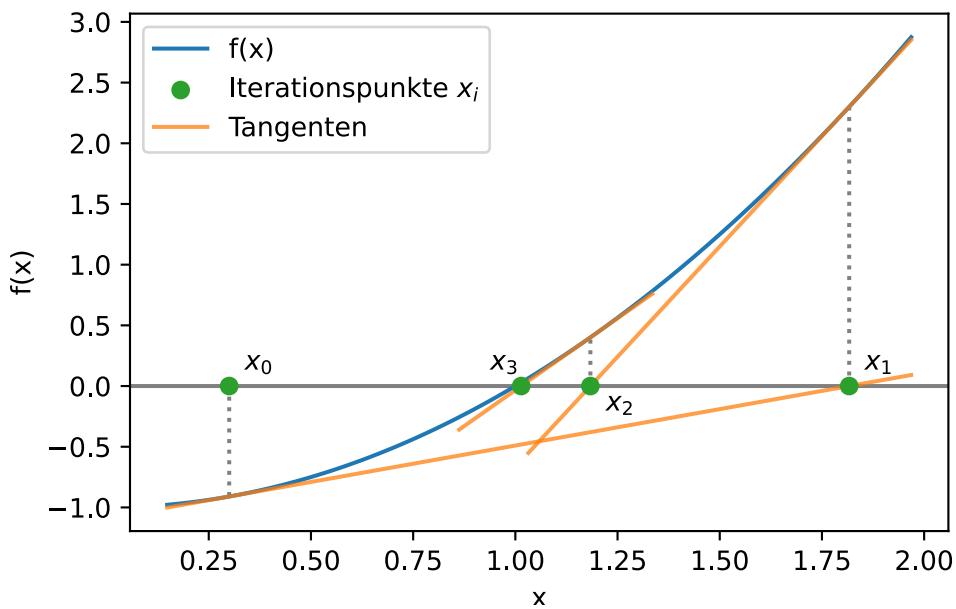
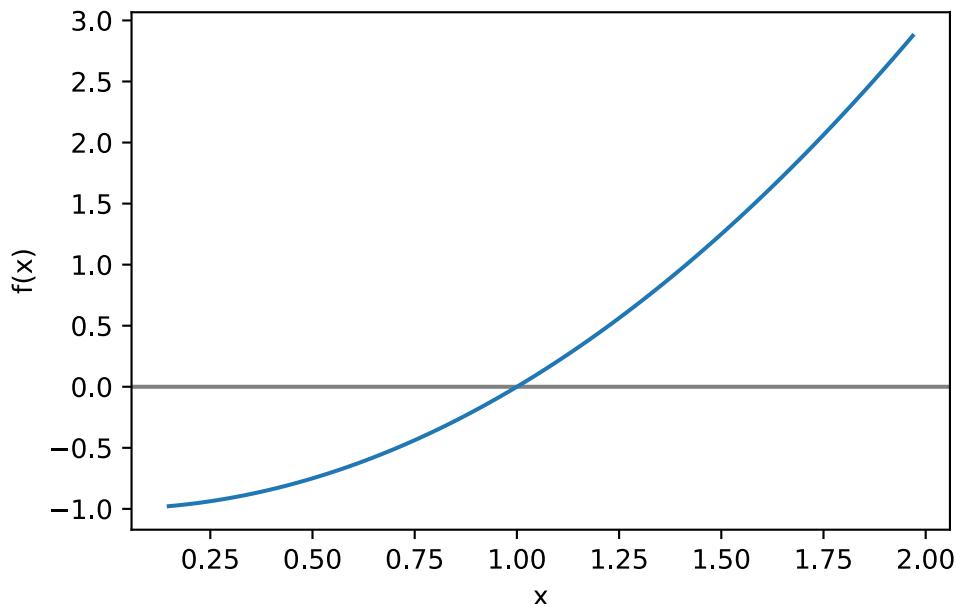
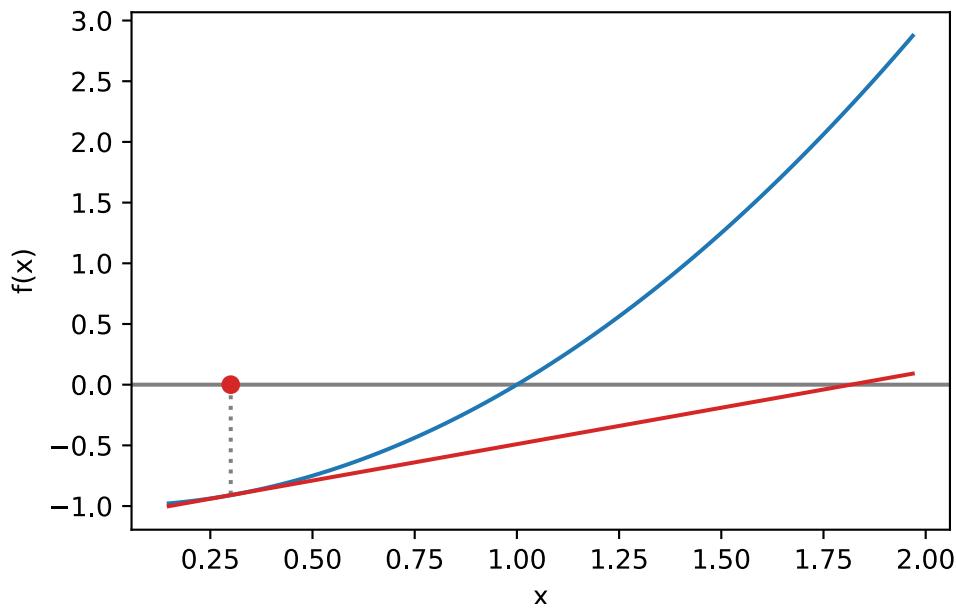


Figure 32.1: Newton-Verfahren, Beispiel 1

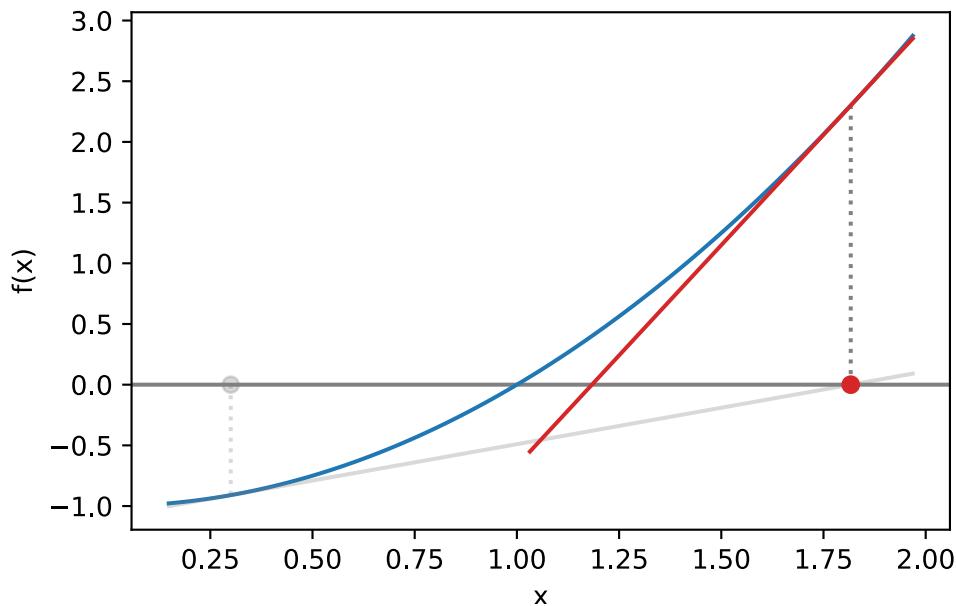
## 32.2 Schritt 0



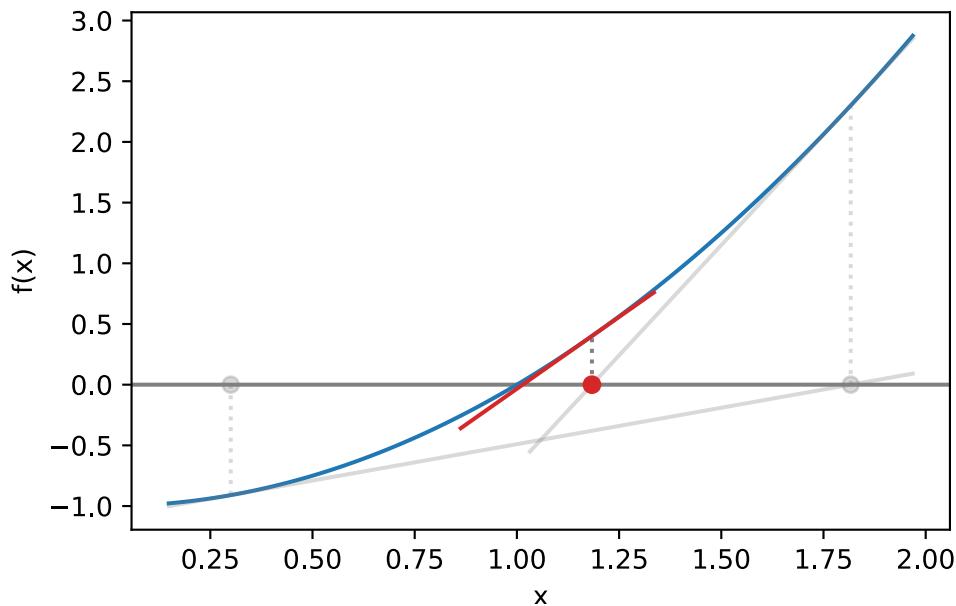
### 32.3 Schritt 1



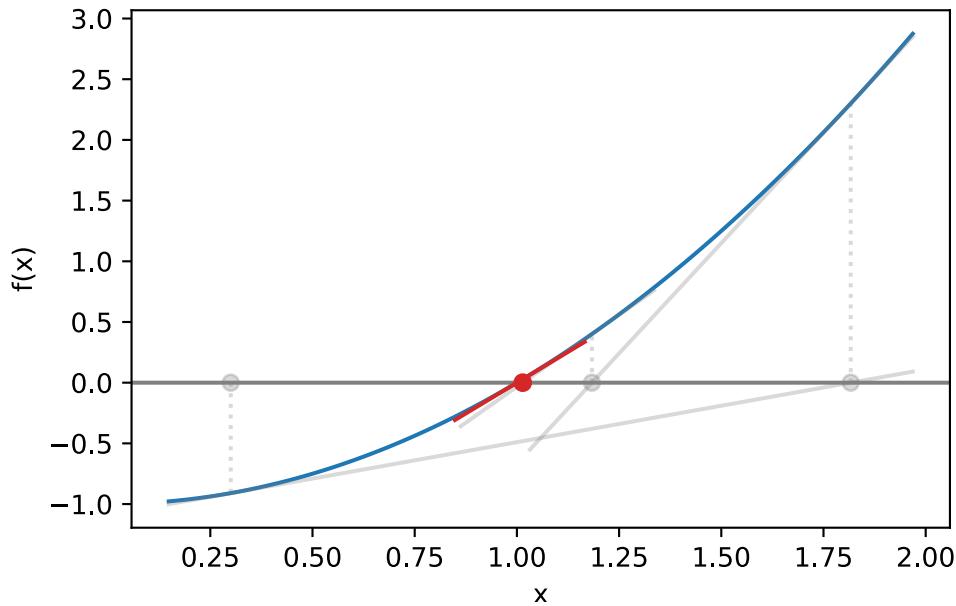
### 32.4 Schritt 2



### 32.5 Schritt 3



## 32.6 Schritt 4



### 32.6.1 Beispiel 2

Gegeben ist die Funktion  $f(x) = \sin(x) - 0.5$  mit der Ableitung  $f'(x) = \cos(x)$ .

```
def f(x):
    return np.sin(x) -0.5
def fp(x):
    return np.cos(x)

x0 = 1.3

print('Startwert x_0 = {:.4f}'.format(x0))
print()

n = 5
xi = [x0]
for i in range(1,n):
    xp = xi[i-1]
    xn = xp - (f(xp)/fp(xp))
```

```

print('Iterationsschritt i = {:.2d}, x_i = {:.4f}'.format(i, xp))
print('    f(x_i) = {:.4f}'.format(f(xp)))
print('    fp(x_i) = {:.4f}'.format(fp(xp)))
print('    x_(i+1) = {:.4f}'.format(xn))
print()

xi.append(xn)

print()
print('Endergebnis nach {} Iterationen: x_(ns) = {:.4f}'.format(n, xi[-1]))

```

Startwert x\_0 = 1.3000

Iterationsschritt i = 1, x\_i = 1.3000

f(x\_i) = 0.4636  
fp(x\_i) = 0.2675  
x\_(i+1) = -0.4329

Iterationsschritt i = 2, x\_i = -0.4329

f(x\_i) = -0.9195  
fp(x\_i) = 0.9077  
x\_(i+1) = 0.5801

Iterationsschritt i = 3, x\_i = 0.5801

f(x\_i) = 0.0481  
fp(x\_i) = 0.8364  
x\_(i+1) = 0.5226

Iterationsschritt i = 4, x\_i = 0.5226

f(x\_i) = -0.0009  
fp(x\_i) = 0.8665  
x\_(i+1) = 0.5236

Endergebnis nach 5 Iterationen: x\_(ns) = 0.5236

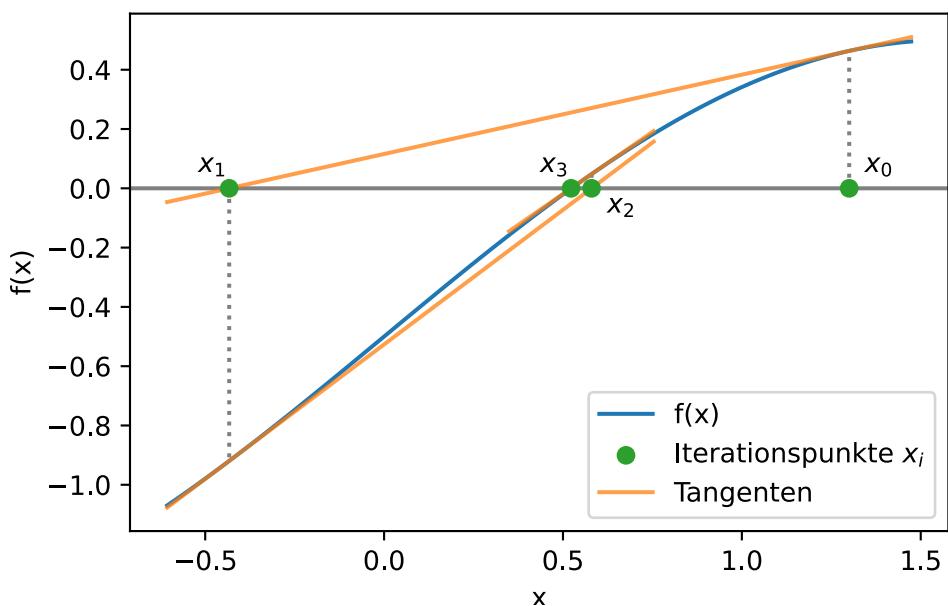
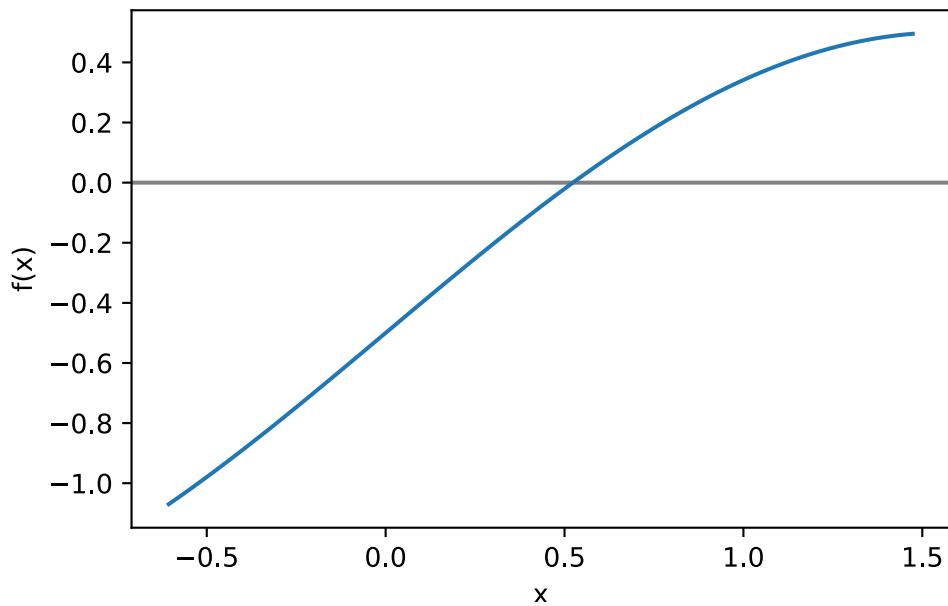
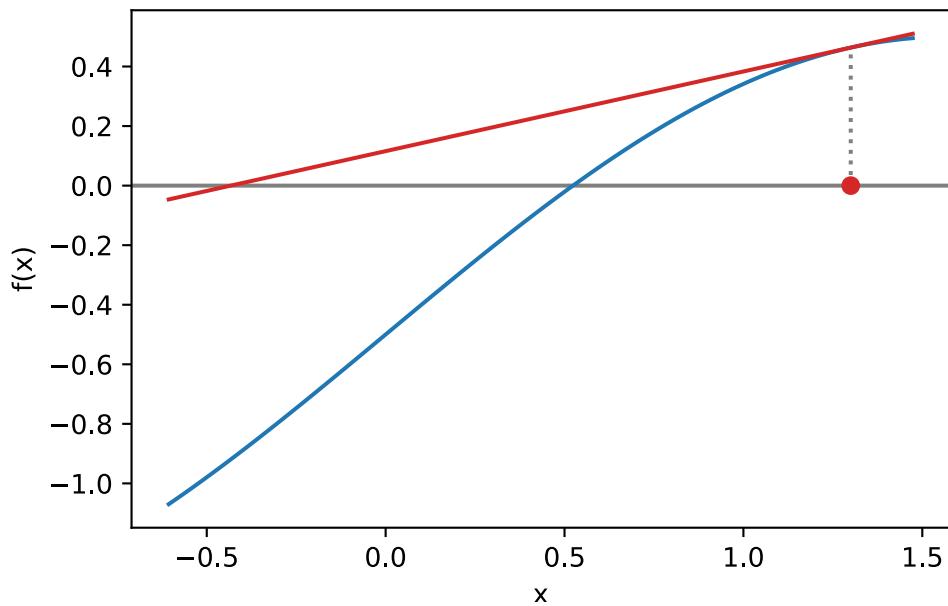


Figure 32.2: Newton-Verfahren, Beispiel 2

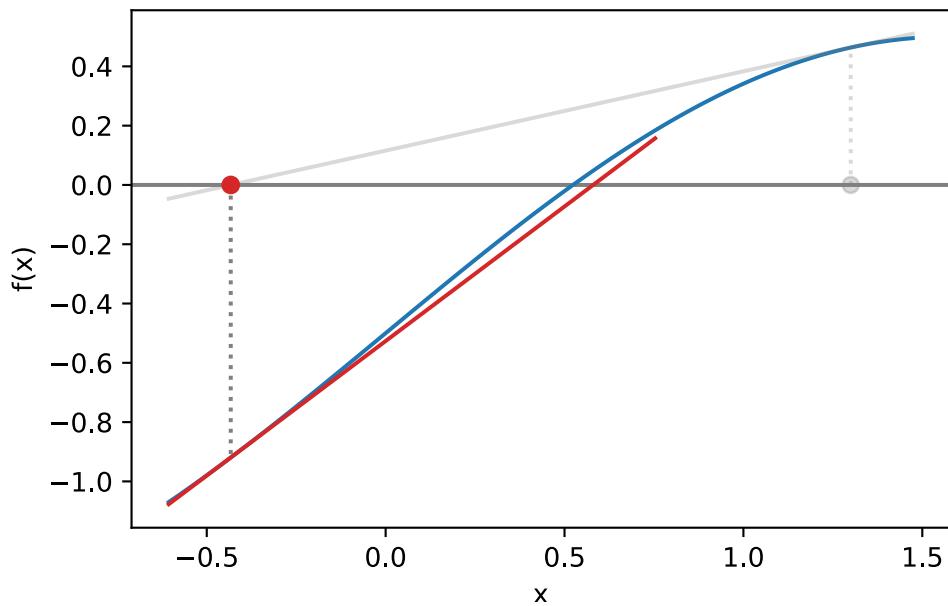
### 32.7 Schritt 0



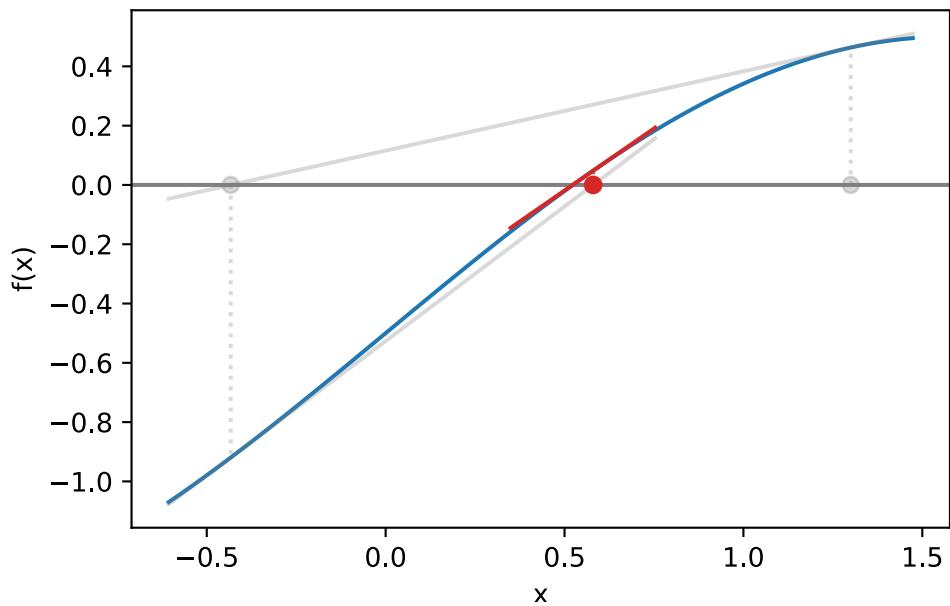
## 32.8 Schritt 1



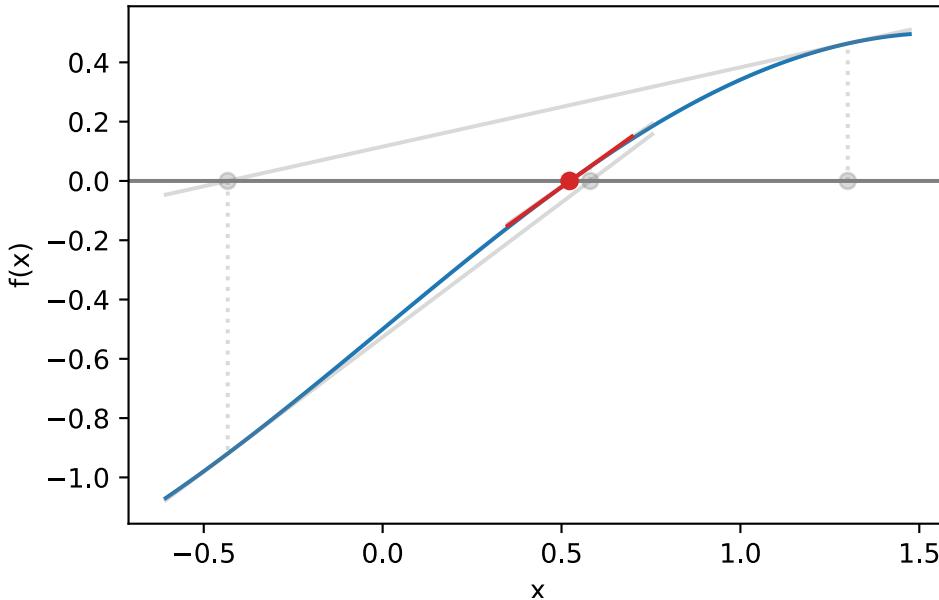
### 32.9 Schritt 2



### 32.10 Schritt 3



## 32.11 Schritt 4



## 32.12 Euler-Verfahren

Das explizite [Euler-Verfahren](#) ist ein einfacher Algorithmus zur Bestimmung von Näherungslösungen von gewöhnlichen Differentialgleichungen, insbesondere Anfangswertprobleme. Das Verfahren wird hier anhand einer linearen Differentialgleichung 1. Ordnung demonstriert, hier ist  $y = y(t)$  eine Funktion der Zeit  $t$ . Die Differentialgleichung lautet

$$\dot{y}(t) + a(t)y(t) + b(t) = 0$$

Mit einem vorgegebenen Anfangswert  $y_0 = y(t_0)$  kann die Näherungslösung iterativ bis zur gewünschten Endzeit  $t_e$  bestimmt werden. Dazu muss das betrachtete Zeitintervall  $[t_0, t_e]$  in  $n_t$  Teilintervalle aufgeteilt werden. Die Länge eines Teilintervalls ist

$$t = \frac{t_e - t_0}{n_t} .$$

Das iterative Verfahren beschreibt die Bestimmung der Lösung im nächsten Zeitintervall  $t_{i+1}$

$$y(t_{i+1}) = y(t_i) - \Delta t(a(t_i)y(t_i) + b(t_i)) .$$

### **32.12.1 Beispiel 1**

Mit  $a(t) = 1$ ,  $b(t) = 0$  und einem Anfangswert von  $y_0 = 1$ .

# 33 Exkurs: Interne Darstellung von Zahlen und Zeichen

Die Grundlage für alle modernen Computer ist die [Digitalisierung](#). Diese ermöglicht es reale Informationen, Kommunikationsformen oder Anweisungen als eine Folge von zwei Zuständen 1/0 darzustellen. Computersysteme nutzen diese Reduktion bzw. Vereinfachung auf nur zwei Zustände zum Speichern, Übertragen und Verarbeiten von Daten.

## 33.1 Analoge und digitale Signale

In der Technik unterscheidet man grundsätzlich zwischen **analogen** und **digitalen Signalen**, wenn Informationen dargestellt, verarbeitet oder übertragen werden.

### 33.1.1 Analoge Signale

Ein analoges Signal ist **kontinuierlich** in Zeit und Ausprägung. Es kann unendlich viele Werte innerhalb eines Bereichs annehmen – wie z. B. die Spannung eines Mikrofonsignals, die mit der Lautstärke variiert, oder die Temperatur, die sich stetig verändert.

Analoge Signale sind gut geeignet, um natürliche Phänomene direkt abzubilden, sind aber anfällig für Störungen und schwer exakt zu speichern oder weiterzuverarbeiten.

### 33.1.2 Digitale Signale

Ein digitales Signal besteht aus **diskreten Werten** – meist zwei: 0 und 1. Es wird also in einzelnen Schritten dargestellt und ist damit für Computer besonders gut geeignet. Die kontinuierlichen Werte der realen Welt müssen dafür zunächst in digitale Werte **umgewandelt** werden (Analog-Digital-Wandlung).

Digitale Signale lassen sich fehlerfrei speichern, übertragen und beliebig oft kopieren, ohne dass die Information an Qualität verliert.

### 33.1.3 Vergleich: Analoge vs. digitale Signale

Merkmal	Analoge Signale	Digitale Signale
Signalverlauf	Stetig, kontinuierlich	Diskret, stufenweise
Wertebereich	Unendlich viele Werte innerhalb eines Bereichs	Endliche Anzahl (z. B. 0 und 1)
Störanfälligkeit	Hoch – kleine Störungen wirken sich direkt aus	Gering – durch Fehlerkorrektur ausgleichbar
Speicherung	Schwierig, da kontinuierlich	Einfach und verlustfrei möglich
Verarbeitung	Aufwendig, da kontinuierlich	Effizient durch digitale Logik
Beispiel	Plattenspieler, Thermometer mit Zeiger	MP3-Datei, Digitalkamera

### 33.1.4 Umwandlung analoger zu digitaler Signale

Um aus analogen Werten, z.B. aus einem Experiment, digitale Werte für die Auswertung zu gewinnen, werden Analog-Digital-Wandler (ADC) genutzt. Einfach gesagt, tastet ein ADC ein Signal mit einer vorgegebenen (endlichen) Abtastrate ab. Dabei wird der Signalwert einem der (endlich vielen) vorgegebenen Wertintervalle zugeordnet. Folgende Abbildung zeigt ein Beispiel für die Umwandlung eines analogen Signals (blaue Kurve) in ein digitales (orangene Punkte). Hier ist das Abtastintervall im Zeit- und Wertbereich jeweils Eins, in der Abbildung durch das graue Gitter veranschaulicht. Damit kann die vom ADC ermittelte Folge von Werten nur Punkte auf dem Gitter enthalten.

## 33.2 Digitale Zahlendarstellung

In den vorherigen Abschnitten haben wir gesehen, wie analoge Informationen in digitale Signale umgewandelt werden können. Damit ein Computer solche digitalen Informationen verarbeiten kann, müssen sie intern als **Zahlen** dargestellt werden - und zwar in einem für Computer verständlichen Format: dem **Binärsystem**.

Wir beschäftigen uns in diesem Abschnitt daher mit der Frage, wie Zahlen **intern gespeichert und dargestellt** werden. Dabei lernen wir unter anderem:

- Wie Zahlen in **verschiedenen Zahlensystemen** dargestellt werden können (z. B. Binär, Dezimal, Hexadezimal),
- wie man zwischen diesen Systemen **umrechnet**,
- und wie man solche Umrechnungen auch **algorithmisch** beschreiben und in Code umsetzen kann.

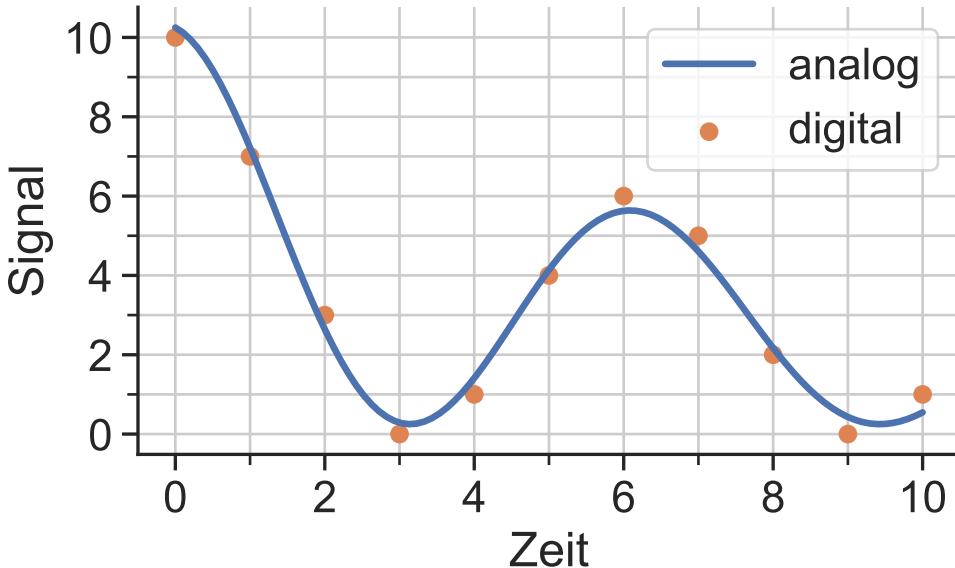


Figure 33.1: Umwandlung eines digitalen (blau) zu einem analogen Signal (orange)

### 33.2.1 Dualsystem

Da in der digitalen Elektronik nur mit zwei Zuständen gerechnet wird, bietet sich das [Dualsystem](#), auch genannt Binärsystem, zur Zahlendarstellung an. Beispiele für Zahlendarstellungen zur Basis 2, wobei der Index die Basis angibt:

- $5_{10} = 101_2$
- $107_{10} = 1101011_2$
- $2635_{10} = 101001001011_2$

Damit lassen sich Zahlen als eine Reihe bzw. Abfolge von 0/1-Zuständen darstellen.

### 33.2.2 Hexadezimalsystem

Bei Zahlen zur Basis 16 müssen auch Stellen, welche größer als 9 sind, abgebildet werden. Hierzu werden Buchstaben eingesetzt, um die Ziffern ‘10’, dargestellt durch A, bis ‘15’ (F) abzubilden. Eine oft verwendete Schreibweise für Zahlen im Hexadezimalsystem ist das Vorstellen von 0x vor die Zahl, wie im folgenden Beispiel gezeigt:

- $5_{10} = 5_{16} = 0x5$
- $107_{10} = 6B_{16} = 0x6B$
- $2635_{10} = A4B_{16} = 0xA4B$

### 33.3 Binäre Maßeinheiten

Da sich in der digitalen Welt alles um Potenzen von 2 dreht, haben sich aus technischen Gründen folgende Einheiten ergeben:

- 1 Bit = eine Ziffer im Binärsystem, Wertebereich: 0 und 1
- 1 Byte = acht Ziffern im Binärsystem, Wertebereich: 0 bis 255

Um größere Datenmengen praktischer anzugeben, werden folgende Einheiten genutzt:

- 1 KB = 1 kiloByte =  $10^3$  Byte
- 1 MB = 1 megaByte =  $10^6$  Byte
- 1 GB = 1 gigaByte =  $10^9$  Byte
- 1 TB = 1 teraByte =  $10^{12}$  Byte
- 1 PB = 1 petaByte =  $10^{15}$  Byte

### 33.4 Geschwindigkeit der Datenübertragung

Die Geschwindigkeit mit der Daten übertragen werden können wird als Datenmenge pro Zeit angegeben. Hierbei wird die Zeit meist auf eine Sekunde bezogen. Beispielhaft sind hier einige Datenübertragungsraten beim Zugriff auf eine [magnetische Festplatte \(HDD\)](#) und auf ein [Halbleiterlaufwerk \(SSD\)](#) aufgeführt.

- Lesen / Schreiben HDD: ~200 MB/s
- Lesen / Schreiben SSD: ~500 MB/s

Als weiteres Beispiel können maximale Übertragungsraten in verschiedenen Netzwerken genannt werden:

- über das Mobilfunknetz, z.B. [3G](#): 384 kbit/s
- über ein Netzwerkkabel, z.B. [Fast Ethernet](#): ~100 Mbit/s

### 33.5 Darstellung ganzer Zahlen

Die Grundidee bei der digitalen Darstellung von Zahlen, hier ganze Zahlen, ist die Verwendung einer festen Anzahl von Bits. Diese bilden dann eine entsprechende Anzahl von Stellen im Dualsystem ab. Dieser Idee folgend, kann eine ganze Zahl mit Vorzeichen wie folgt als 8-Bit-Zahl dargestellt werden:

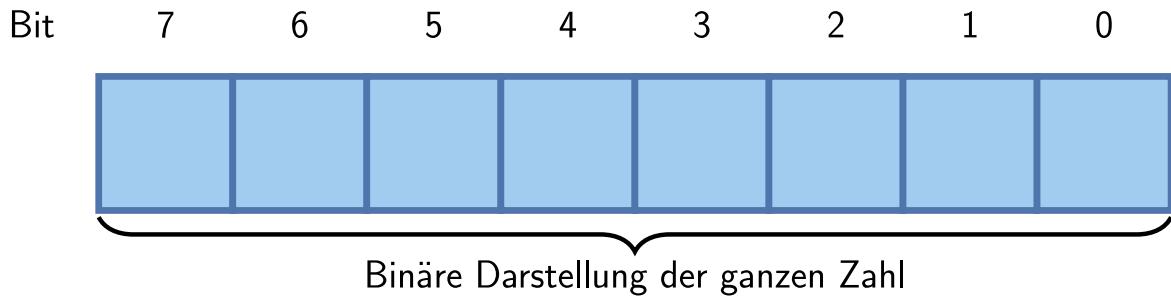


Figure 33.2: Bitzuordnung bei der Darstellung einer ganzen Zahl mit 8 Bit.

Für zwei Zahlen aus dem obigen Beispiel für die Zahldarstellung im Dualsystem könnte die Bitzuweisung wie folgt aussehen.

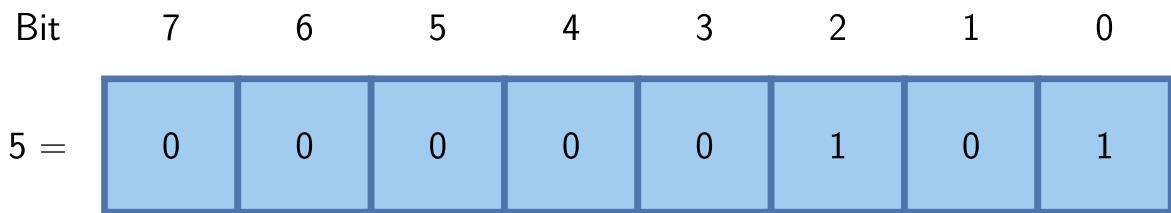


Figure 33.3: Beispiel der Bitzuordnung für die Zahl 5 mit 8 Bit.

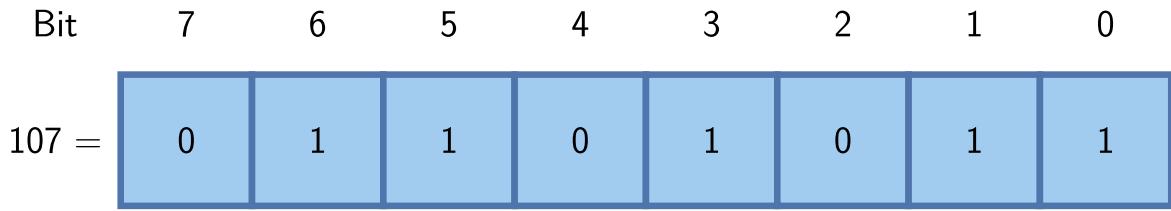


Figure 33.4: Beispiel der Bitzuordnung für die Zahl 107 mit 8 Bit.

Durch die fixe Vorgabe der Stellen im Dualsystem, also hier der Bits, ergibt sich der Zahlenbereich, welcher mit diesen Bits abgebildet werden kann. Für die Darstellung von ganzen Zahlen mit 8 Bit, also mit einem Byte, ergibt sich somit

- kleinste Zahl:  $02 = 0$
- größte Zahl:  $11111112 = 2^8 - 1 = 255$ .

Natürlich können auch längere Bitfolgen für einen größeren Zahlenbereich genutzt werden. Zusätzlich kann eines der Bits auch genutzt werden, um das Vorzeichen darzustellen. Folgende Abbildung zeigt die Darstellung einer vorzeichenbehafteten ganzen Zahl mit 32 Bit.

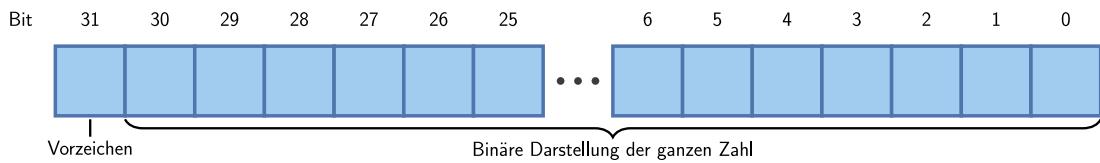


Figure 33.5: Bitzuordnung bei der Darstellung einer ganzen Zahl samt Vorzeichen mit 32 Bit.

Der Wertebereich ist in diesem Fall gegeben durch:

- kleinste Zahl =  $-2^{31} = -2,147,483,648$
- größte Zahl =  $2^{31} - 1 = 2,147,483,647$ .

In der Informatik wird solch eine Darstellung von ganzen Zahlen als [Integer Datentyp](#) bezeichnet. Im Englischen wird dieser als *integer* bezeichnet.

## 33.6 Umrechnung zwischen Dezimal- und Binärzahlen

Um zwischen **Dezimalzahlen (zur Basis 10)** und **Binärzahlen (zur Basis 2)** umzuwandeln, gibt es jeweils einfache Verfahren. Diese lassen sich auch leicht als Algorithmus in Programmiersprachen umsetzen.

### 33.6.1 Von Binär nach Dezimal

Eine Binärzahl besteht aus einzelnen Stellen (Bits), die jeweils eine Potenz von 2 repräsentieren. Zur Umrechnung summiert man die Produkte der Ziffern mit ihrer jeweiligen Stellenwertigkeit:

$$z = \sum_{i=0}^n b_i \cdot 2^i$$

Dabei ist: -  $b_i \in \{0, 1\}$  das i-te Bit (von rechts gezählt), -  $2^i$  der Stellenwert der Position, -  $z$  die Dezimalzahl.

**Beispiel:**

Umwandlung von  $1011_2$  nach Dezimal:

\$\$

$$1011_2 \&= 1 \ 2^3 + 0 \ 2^2 + 1 \ 2^1 + 1 \ 2^0 \ \&= 8 + 0 + 2 + 1 = 11_{10}$$

\$\$

### 33.6.2 Von Dezimal nach Binär

Zur Umwandlung einer Dezimalzahl in eine Binärzahl verwendet man die **ganzzahlige Division durch 2**. Der jeweilige Rest (0 oder 1) ergibt die Binärziffer. Man wiederholt diesen Vorgang so lange, bis der Quotient 0 ist, und liest die Reste **von unten nach oben**.

**Beispiel:**

Umwandlung von  $11_{10}$  nach Binär:

$$\begin{array}{rcl} 11 \div 2 & = 5 & \text{Rest 1} \\ 5 \div 2 & = 2 & \text{Rest 1} \\ 2 \div 2 & = 1 & \text{Rest 0} \\ 1 \div 2 & = 0 & \text{Rest 1} \end{array}$$

→ Von unten gelesen ergibt das:  $1011_2$

**i** Merke

Die Umrechnung funktioniert immer, egal wie groß die Zahl ist – sie ist eine systematische Anwendung der Stellenwertsysteme. Computer arbeiten intern genau auf diese Weise, nur in Hardware.

## 33.7 Darstellung reeller Zahlen

Reelle Zahlen können nur angenährt als eine **Gleitkommazahl** digital dargestellt werden. Dazu wird die zur Verfügung stehende Menge an Bits auf folgende Zuordnungen aufgeteilt: Vorzeichen  $s$ , Exponent  $e$  und Mantisse  $m$ . Jedem dieser Bereiche wird eine feste Anzahl von Bits zugeordnet wodurch sich der Wertebereich und Genauigkeit der Darstellung ergibt. Im Allgemeinen kann somit eine Gleitkommazahl dargestellt werden als

$$z = (-1)^s \cdot m \cdot 2^e.$$

Es existieren mehrere Ansätze für die Abbildung von Gleitkommazahlen. Insbesondere im **IEEE754 Standard** wird folgende Aufteilung definiert: Vorzeichen (1 bit), Exponent (11 bit) und Mantisse (52 bit):

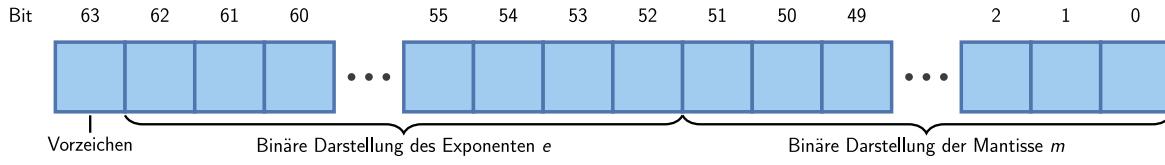


Figure 33.6: Bitzuordnung bei der Darstellung einer reellen Zahl mit 64 bit.

Aus der obigen Festlegung der Bitzuweisung, ergeben sich die Größenordnung für den Wertebereich, welcher durch den Exponenten vorgegeben ist. Um auch Zahlen kleiner 1 darstellen zu können, kann der Exponent  $e$  auch negative Werte annehmen.

Für den Exponenten  $e$  gilt

- kleinster Wert in Etwa:  $-(2^{10} - 1) = -1023$
- größter Wert in Etwa:  $\sim 2^{10} - 1 = 1023$ .

Ohne Beachtung der Mantisse und des Vorzeichens, ergibt sich mit den obigen Werten dieser Bereich für die Größenordnungen:

- kleinste Größenordnung:  $2^{-1023} \sim 10^{-308}$
- größte Größenordnung:  $2^{1023} \sim 10^{308}$

Die Genauigkeit, d.h. die kleinste darstellbare Differenz zwischen zwei Gleitkommazahlen, ergibt sich aus der Mantisse  $\$ m \$$ . Eine grobe Abschätzung der Genauigkeit kann wie folgt durchgeführt werden. Per Definition deckt die Mantisse einen Zahlenbereich von 0 bis etwa 10 ab. Dieser Bereich wird in obiger Festlegung mit 52 Bit dargestellt. Hieraus ergibt sich dann der kleinste Unterschied zu

- kleinster Unterschied zwischen zwei Gleitkommazahlen:  $10 / 2^{52} \sim 2 \cdot 10^{-15}$

Betrachtet man nun Dezimalzahlen, so entspricht das etwa der 15-ten Nachkommastelle.

Der Datentyp, welcher für die Darstellung von Gleitkommazahlen verwendet wird, wird generell als *float* (engl. *floating point number*) bezeichnet. Im [IEEE754 Standard](#) werden viele verschiedene Darstellungen definiert.

## 33.8 Zeichendarstellung

Neben Zahlen können auch Zeichen, z.B. für die Darstellung von Text, abgebildet werden. Die Grundidee ist dabei, dass die Zeichen als vorzeichenlose ganze Zahlen gespeichert und dann anhand einer Tabelle interpretiert werden. Ein Beispiel für eine solche Tabelle, welche den Zahlenwerte Zeichen zuordnet, ist die [ASCII Tabelle](#). In dieser werden 7-Bit-Zahlen, d.h. 128 Zeichen, kodiert. In der 1963 erstellten – und bis heute genutzten – Tabelle, sind sowohl

nicht-druckbare Zeichen (z.B. Zeilenvorschub, Tabulatorzeichen) als auf folgende druckbare Zeichen enthalten:

```
!"#$%&'()*+,./0123456789:;=>?  
@ABCDEFGHIJKLMNPQRSTUVWXYZ[\]^_  
`abcdefghijklmnopqrstuvwxyz{|}~
```

Wobei das erste Zeichen das Leerzeichen ist.

# 34 Algorithmische Umrechnung von Zahlen: Dezimal Binär

In diesem Kapitel betrachten wir die Umrechnung zwischen Dezimal- und Binärzahlen **algorithmisch**. Dabei analysieren wir den Ablauf der Rechenschritte, beschreiben sie als **Pseudocode**, visualisieren sie mit **Flussdiagrammen** und setzen sie anschließend **manuell** in **Python um**.

## 34.1 1. Umrechnung: Dezimal → Binär

### 34.1.1 Grundidee

Wir wiederholen die ganzzahlige Division durch 2 und merken uns den Rest. Die **Binärziffern** ergeben sich aus den **Resten** — von unten nach oben gelesen.

### 34.1.2 Flussdiagramm

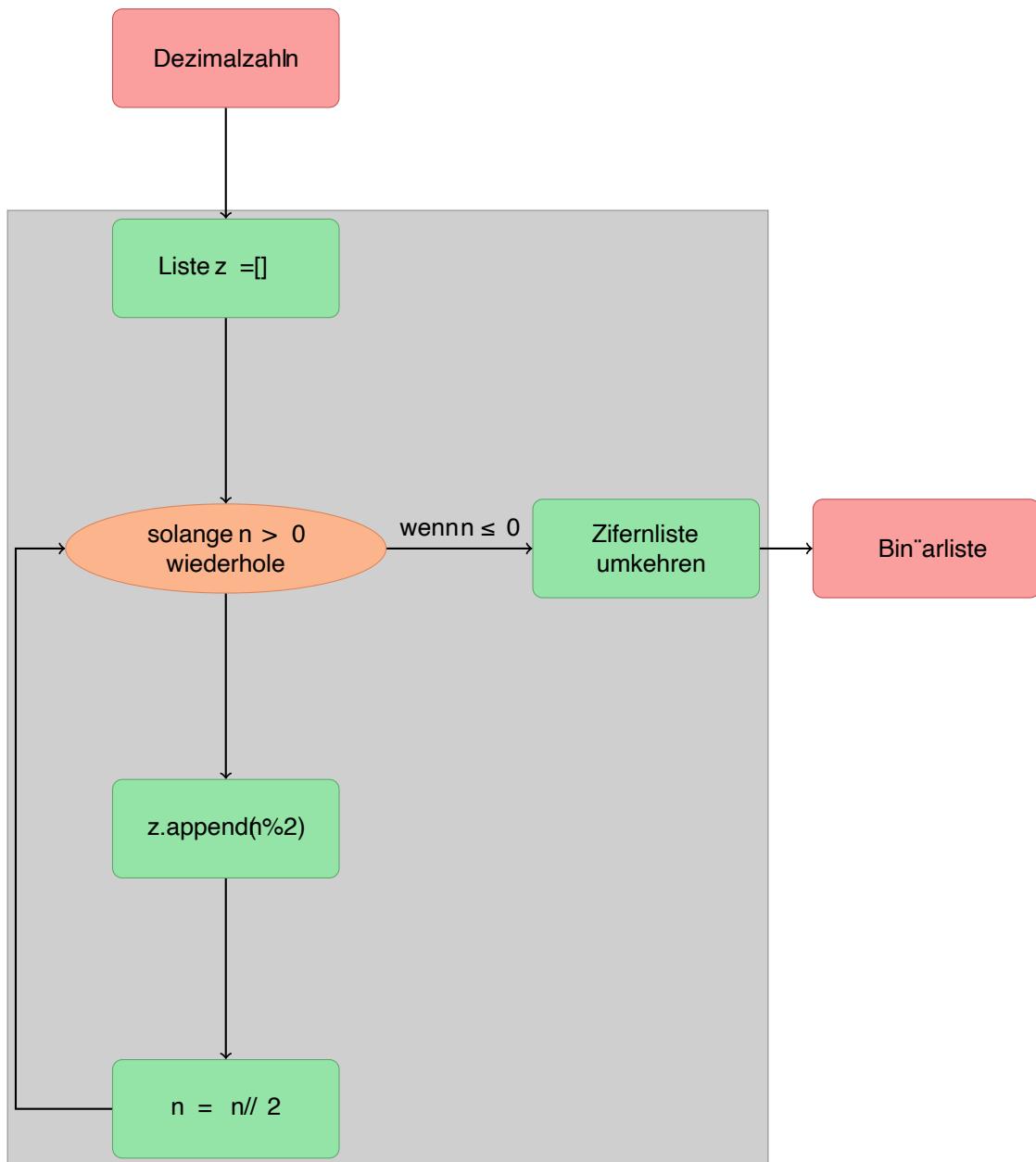


Figure 34.1: Umrechnug von Dezimal zu Binär

### 34.1.3 Pseudocode

```
Eingabe: Dezimalzahl n
Initialisiere leere Liste ziffern
Solange n > 0:
    rest ← n mod 2
    ziffern an rest anhängen
    n ← n ganzzahlig geteilt durch 2
Ausgabe: ziffern in umgekehrter Reihenfolge
```

### 34.1.4 Python-Implementierung

```
def dezimal_zu_binaer(n):
    ziffern = []
    while n > 0:
        ziffern.append(n % 2)
        n //= 2
    return ziffern[::-1]

dezimal_zu_binaer(23) # Beispiel: 23 → 10111
```

```
[1, 0, 1, 1, 1]
```

## 34.2 2. Umrechnung: Binär → Dezimal

### 34.2.1 Grundidee

Jede Stelle repräsentiert eine Zweierpotenz. Wir addieren die Produkte der Ziffern mit ihrer Potenz.

### 34.2.2 Flussdiagramm

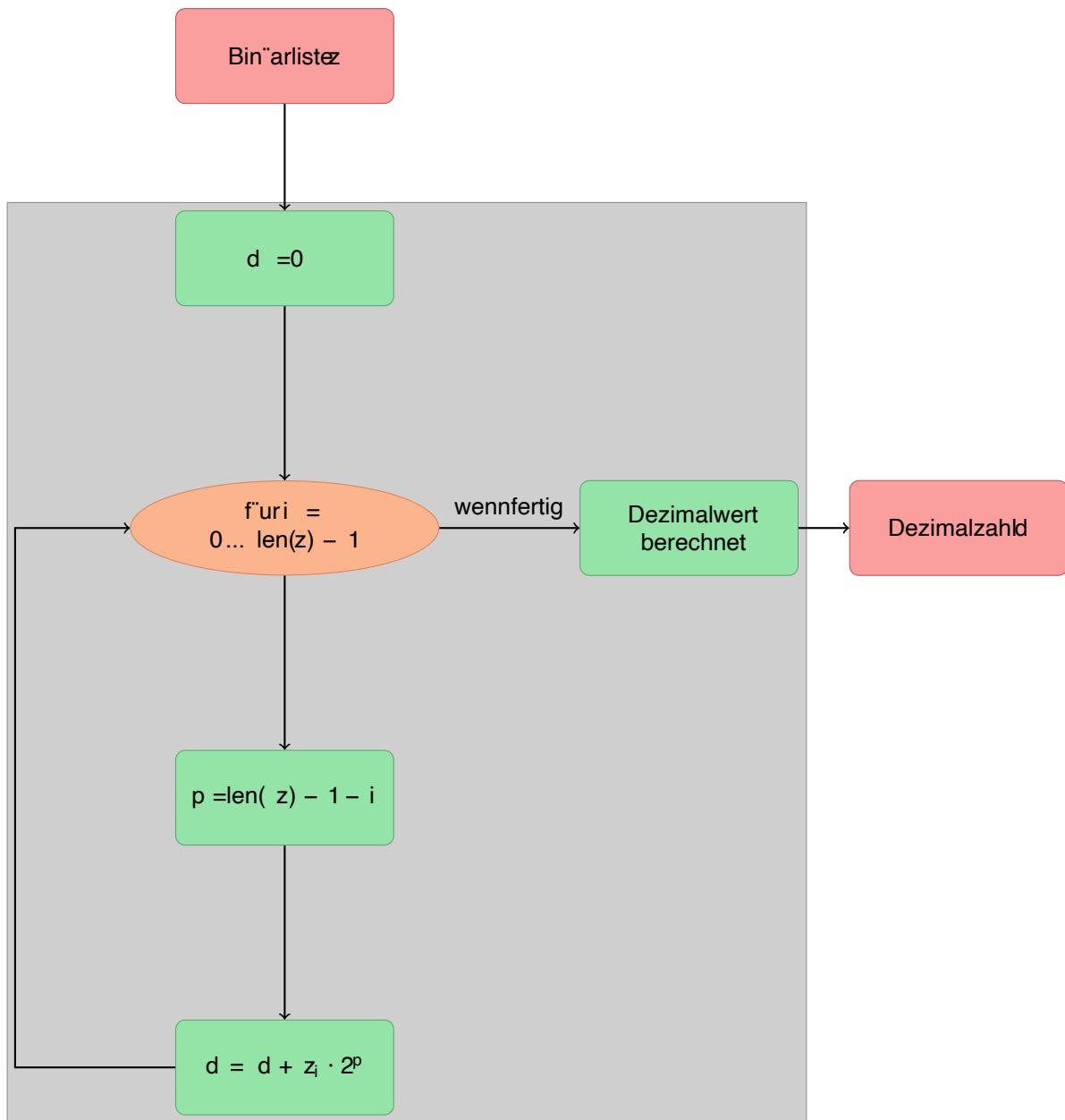


Figure 34.2: Umrechnug von Binär zu Dezimal

### 34.2.3 Pseudocode

```
Eingabe: Liste binärer Ziffern (z. B. [1, 0, 1, 1])
Initialisiere dezimalwert ← 0
Für jede Stelle i von rechts nach links:
    dezimalwert ← dezimalwert + ziffer * 2^position
Ausgabe: dezimalwert
```

### 34.2.4 Python-Implementierung

```
def binaer_zu_dezimal(ziffern):
    dezimalwert = 0
    for i in range(len(ziffern)):
        potenz = len(ziffern) - i - 1
        dezimalwert += ziffern[i] * (2 ** potenz)
    return dezimalwert

binaer_zu_dezimal([1, 0, 1, 1]) # Ergebnis: 11
```

11

## 34.3 3. Mathematische Komplexität

Beide Algorithmen haben eine **logarithmische Laufzeit** bezogen auf die Eingabegröße  $n$ , denn:

- Die Umrechnung Dezimal → Binär wiederholt die Division durch 2, bis  $n = 0$ . Das sind  $\log_2(n)$  Schritte.
- Die Umrechnung Binär → Dezimal summiert über  $\log_2(n)$  Stellen.

Daher gehören beide zur Klasse der **logarithmischen Algorithmen**.

**i** Hinweis

Diese Verfahren sind nicht nur theoretisch interessant – genau so arbeiten Computer intern mit Bitfolgen!

# **Part VIII**

# **Numerik**

Die **Numerik** bzw. numerische Mathematik ist eine Disziplin der Mathematik in der Verfahren für Lösungen bzw. Näherungslösungen von Problemstellungen gesucht werden, welche nicht analytisch lösbar sind.

Dazu gehören die Teilgebiete der Integration und Differentiation als auch der Bereich der Differentialgleichungen. Viele Simulationsprogramme aus dem Ingenieurwesen verwenden numerische Verfahren zur Berechnung komplexer Sachverhalte. Insbesondere die Lösung von gewöhnlichen und partiellen Differentialgleichungen ist ohne die Numerik kaum möglich.

Zu den Anwendungsfeldern gehören beispielsweise:

- Verformung von Körpern unter Kraftausübung
- Schwingungen von Gebäuden
- Zeitliche Entwicklung von Stoffkonzentrationen
- Strömungsdynamik: Wasserbau, Umströmung von Gebäuden, Lüftungstechnik, Rauchausbreitung
- Wärmetransport in Bauteilen

# 35 Integration

Die Bildung von Integralen findet beispielsweise bei der Bestimmung von Flächeninhalten oder von Gesamtkräften Anwendung. Formal wird das bestimmte Integral  $I$  der Funktion  $f(x)$  auf dem Intervall  $x \in [a, b]$  wie folgt dargestellt.

$$I = \int_a^b f(x) \, dx$$

Im Allgemeinen kann das Integral nicht analytisch gelöst werden, da die Stammfunktion  $F(x)$  nicht leicht zu bestimmen ist. In solchen Fällen können numerische Verfahren eingesetzt werden um den Integralwert zu approximieren. Die numerische Integration wird oft auch als numerische Quadratur bezeichnet.

Dieses Kapitel bietet eine kurze Übersicht von numerischen Integrationsmethoden:

- Ober- und Untersumme
- Quadratur
- Monte-Carlo

## 35.0.1 Ober- und Untersumme

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

Eine der grundlegendsten Arten Integrale von Funktionen zu bestimmen sind die [Ober- und Untersumme](#). Sie nähern den Integralwert durch eine Abschätzung nach oben bzw. unten an. Mit einer steigenden Anzahl von Stützstellen, d.h. Positionen an welchen die Funktion ausgewertet wird, konvergieren beide Abschätzungen gegen den Integralwert.

## 35.1 Definition

Für die Bildung der Ober- und Untersumme, werden gleichmäßig verteilte Stützstellen auf dem Intervall  $[a, b]$  benötigt. Werden  $n + 1$  Stützstellen gewählt, so gilt:

$$a = x_0 < x_1 < \dots < x_n = b$$

Der Abstand der Stützstellen beträgt  $\Delta x = (b - a)/(n - 1)$ . Auf jedem der  $n$  Teilintervalle  $[x_{i-1}, x_i]$  wird nun der maximale bzw. minimale Wert der Funktion  $f(x)$  bestimmt und als  $O_i$  bzw.  $U_i$  definiert.

$$O_i = \max(f(x) | x \in [x_{i-1}, x_i])$$
$$U_i = \min(f(x) | x \in [x_{i-1}, x_i])$$

Die gesuchte Approximation des Integrals ist die Summe der  $O_i$  bzw.  $U_i$  mal der Breite des Teilintervalls, hier  $\Delta x$ :

$$\sum_{i=1}^n \Delta x U_i \lesssim I \lesssim \sum_{i=1}^n \Delta x O_i$$

## 35.2 Beispiel

Beispielhaft soll folgendes Integral bestimmt werden

$$I = \int_0^2 \sin(3x) + 2x \, dx$$

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung
I_exakt = (-1/3*np.cos(3*2) + 2**2) - (-1/3)
```

Als erstes werden die Stützstellen gleichmäßig im Intervall  $[0, 2]$  verteilt.

```

n = 5

xi = np.linspace(0, 2, n)
yi = fkt(xi)

```

Die beiden Summen benötigen die Extremwerte der zu integrierenden Funktion in den Teilintervallen. Diese werden mit Hilfe einer Funktionsauswertung auf dem Teilintervall bestimmt. Für die nachfolgende Visualisierung hat die Menge der Summen ebenfalls  $n$  Elemente.

```

oben = np.zeros(n)
unten = np.zeros(n)

for i in range(len(oben)-1):
    cx = np.linspace(xi[i], xi[i+1], 50)
    cy = fkt(cx)
    oben[i+1] = np.max(cy)
    unten[i+1] = np.min(cy)

```

Die ersten Elemente der beiden Summenlisten werden auf die ersten Funktionswerte gesetzt, dies dient nur der folgenden Darstellung.

```

oben[0] = yi[0]
unten[0] = yi[0]

```

Visualisierung der einzelnen Funktionen.

```

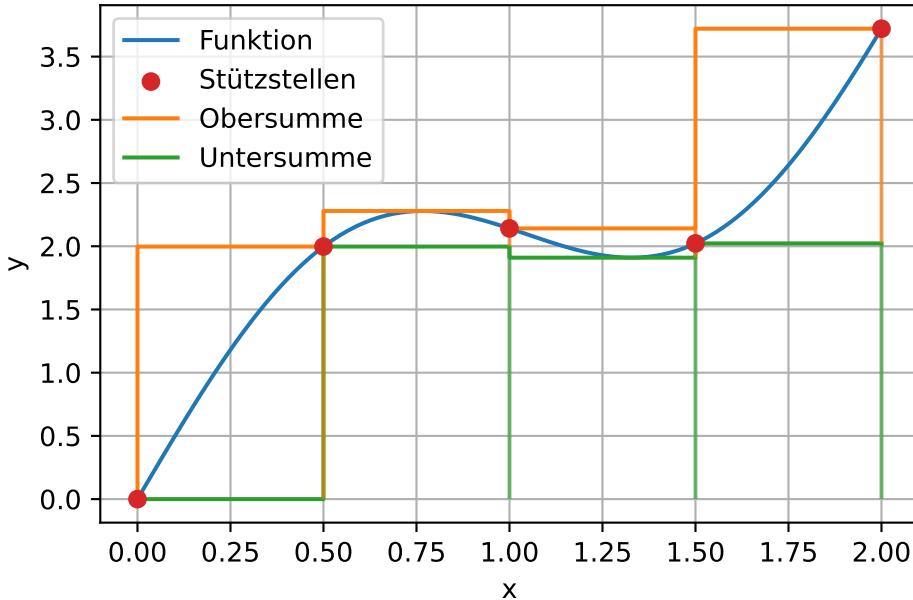
plt.plot(x, y, label='Funktion')
plt.scatter(xi, yi, label='Stützstellen', c='C3', zorder=3)
plt.plot(xi, oben, drawstyle='steps-pre', label='Obersumme')
plt.plot(xi, unten, drawstyle='steps-pre', label='Untersumme')

plt.vlines(xi, ymin=unten, ymax=oben, color='C1', alpha=0.6)
plt.vlines(xi, ymin=0, ymax=unten, color='C2', alpha=0.6)

plt.xlabel('x')
plt.ylabel('y')

plt.grid()
plt.legend();

```



Das obige Verfahren kann nun in einer Funktion zusammengefasst werden, welche die Summen der beiden Folgen zurückgibt.

```
def ou_summe(n, a=0, b=2):
    xi = np.linspace(a, b, n)
    yi = fkt(xi)
    dx = xi[1] - xi[0]

    sum_oben = 0
    sum_unten = 0

    for i in range(n-1):
        cx = np.linspace(xi[i], xi[i+1], 50)
        cy = fkt(cx)
        oben = np.max(cy)
        unten = np.min(cy)
        sum_oben += dx * oben
        sum_unten += dx * unten

    return sum_oben, sum_unten
```

Für eine systematische Untersuchung des Konvergenzverhaltens, wird die Integrationsfunktion für verschiedene Anzahlen von Stützstellen aufgerufen.

```

n_max = 100
ns = np.arange(2, n_max, 1, dtype=int)
os = np.zeros(len(ns))
us = np.zeros(len(ns))

for i, n in enumerate(ns):
    o, u = ou_summe(n)
    os[i] = o
    us[i] = u

```

Die graphische Darstellung der beiden Summen zeigt eine kontinuierliche Annäherung dieser.

```

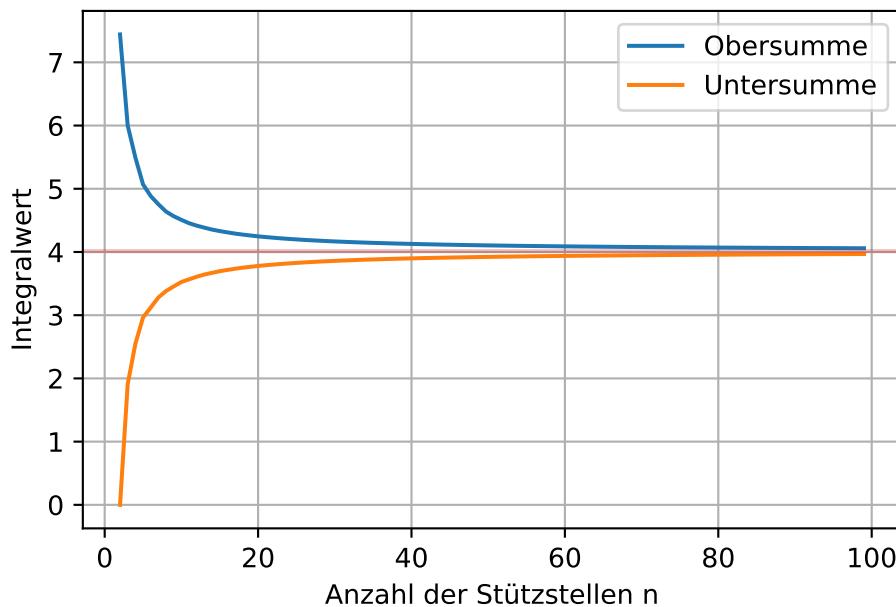
plt.plot(ns, os, label='Obersumme')
plt.plot(ns, us, label='Untersumme')

plt.axhline(y=I_exakt, color='C3', alpha=0.3)

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Integralwert')

plt.grid()
plt.legend();

```



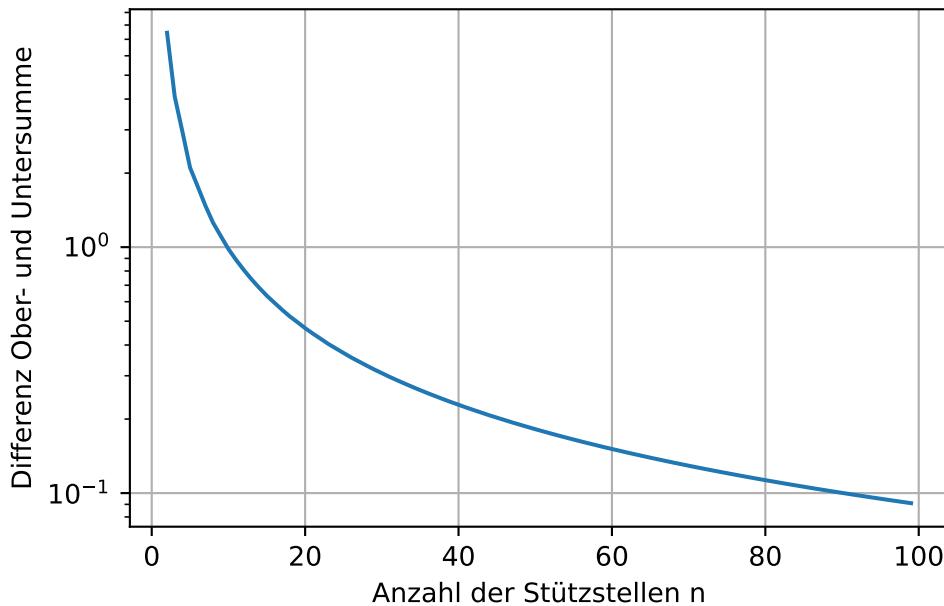
Dies wird insbesondere deutlich, wenn die Differenz der beiden Summen aufgetragen wird. Mit einer logarithmischen Darstellung kann die kontinuierliche Annäherung auch quantitativ abgelesen werden.

```
plt.plot(ns, os-us)

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz Ober- und Untersumme')

# plt.xscale('log')
# plt.yscale('log')

plt.grid();
```



### 35.3 Interpolation

Bei der Bildung der Ober- und Untersumme wurde die zu integrierende Funktion durch einen konstanten Wert in den Teilintervallen zwischen den Stützstellen angenähert. Eine genauere Berechnung des Integrals kann durch eine bessere Interpolation erfolgen. Dazu eignen sich Polynome, da diese leicht zu Integrieren sind.

### 35.3.1 Trapezregel

Die Trapezregel beruht auf der Annäherung der zu integrierenden Funktion durch Geraden, d.h. Polynome vom Grad 1, auf den Teilintervallen. Die Approximation des Integralwertes ergibt sich entsprechend aus den Flächeninhalten der so entstandenen Trapeze.

Wie im vorhergehenden Kapitel wird das Verfahren anhand folgender Funktion demonstriert

$$I = \int_0^2 \sin(3x) + 2x \, dx$$

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung
I_exakt = (-1/3*np.cos(3*2) + 2**2) - (-1/3)
```

Bildung der Stützpunkte:

```
n = 5

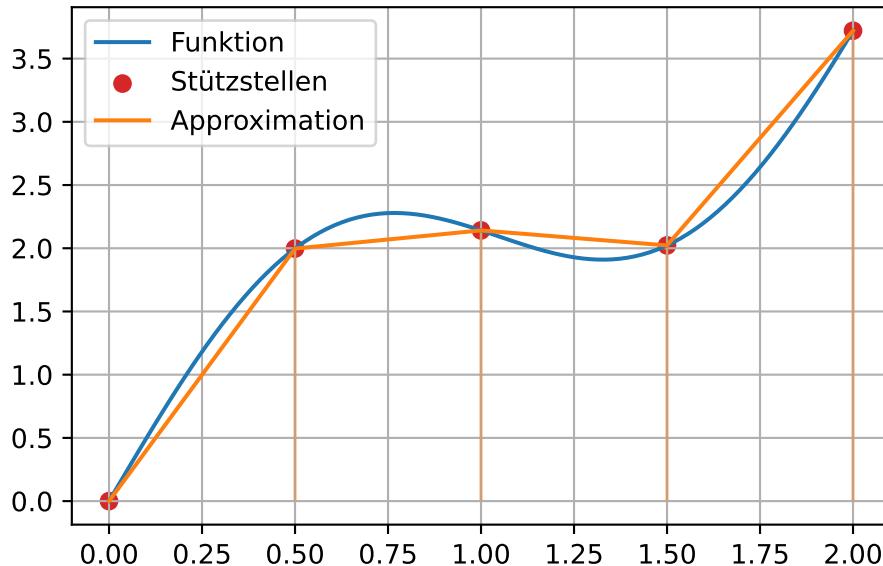
xi = np.linspace(0, 2, n)
yi = fkt(xi)
```

Zunächst erfolgt noch die Visualisierung des Verfahrens.

```
plt.plot(x, y, label='Funktion')
plt.scatter(xi, yi, label='Stützstellen', c='C3')
plt.plot(xi, yi, label='Approximation', c='C1')

plt.vlines(xi, ymin=0, ymax=yi, color='C1', alpha=0.3)

plt.grid()
plt.legend();
```



Die Integration selbst kann mittels der [Funktion `scipy.integrate.trapz`](#) ausgeführt werden.

```
res = scipy.integrate.trapz(yi, xi)
print(f"Integralwert mit {n} Stützstellen: {res:.4f}")
```

Integralwert mit 5 Stützstellen: 4.0107

```
/var/folders/p/_ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_26454/2607466481.py:1: DeprecationWarning:
res = scipy.integrate.trapz(yi, xi)
```

Der so ermittelte Wert nähert sich dem exakten Wert mit zunehmender Anzahl der Stützstellen.

```
n_max = 50
ns = np.arange(2, n_max, 1, dtype=int)
tr = np.zeros(len(ns))

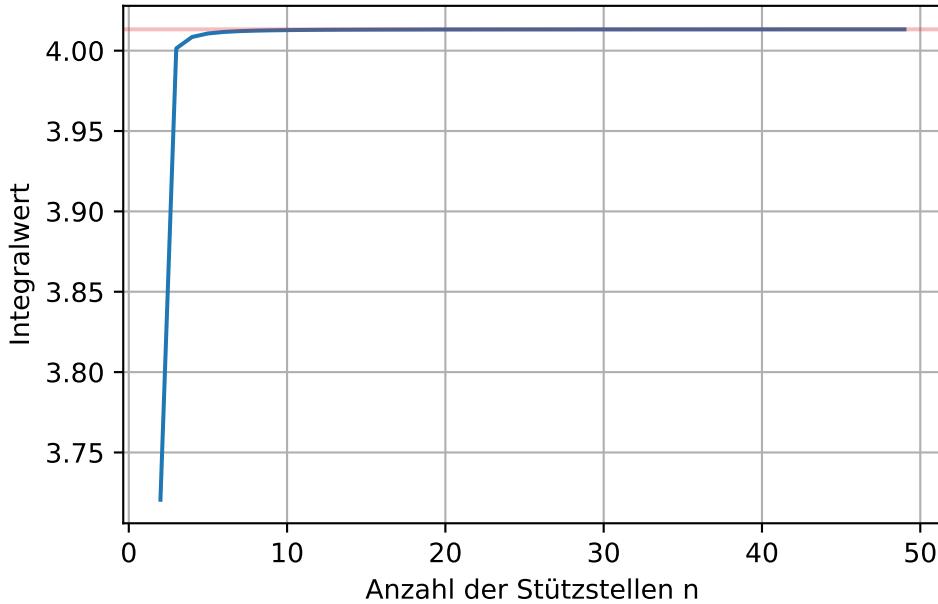
for i, n in enumerate(ns):
    xi = np.linspace(0, 2, n)
    yi = fkt(xi)
    tr[i] = scipy.integrate.trapz(yi, xi)
```

```
/var/folders/p_/_ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_26454/3326232058.py:8: DeprecationWarning
  tr[i] = scipy.integrate.trapz(yi, xi)

plt.plot(ns, tr)
plt.axhline(y=I_exakt, color='C3', alpha=0.3)

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Integralwert')

plt.grid();
```

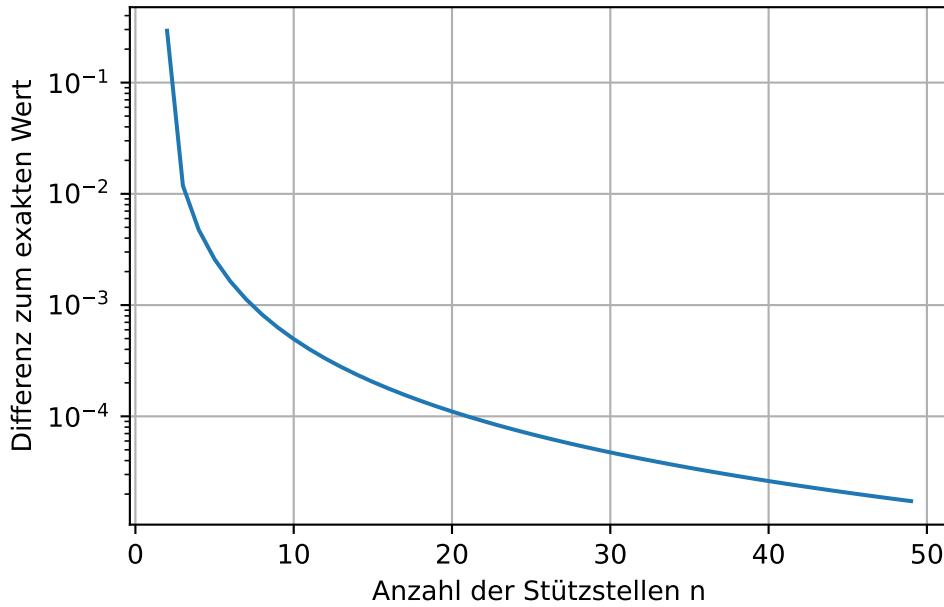


```
plt.plot(ns, np.abs(tr-I_exakt))

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
# plt.yscale('log')

plt.grid();
```



### 35.3.2 Simpsonregel

Die Verwendung eines Polynoms vom zweiten Grad führt zur Simpsonregel. Hierzu wird die Funktion an einem Zwischenwert, mittig im Teilintervall, ausgewertet und zusammen mit den Werten an den Stützstellen zur Bestimmung der Polynomkoeffizienten verwendet.

Anhand des obigen Beispiels wird die Simpsonregel visuell demonstriert.

```

n = 5

xi = np.linspace(0, 2, n)
yi = fkt(xi)

plt.plot(x, y, label='Funktion')
plt.scatter(xi, yi, label='Stützstellen', c='C3')

# Bestimmung und Plotten der Polynome
for i in range(n-1):
    dx = xi[i+1] - xi[i]
    cx = (xi[i] + xi[i+1]) / 2
    cy = fkt(cx)

    P = np.polyfit([xi[i], cx, xi[i+1]], [yi[i], cy, yi[i+1]], 2)

```

```

Px = np.linspace(xi[i], xi[i+1], 20)
Py = np.polyval(P, Px)

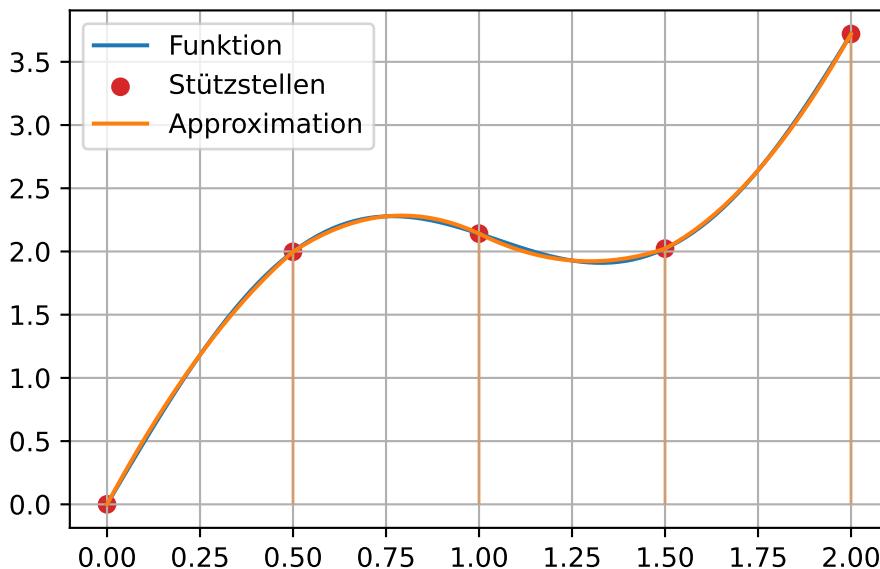
label=None
if i==0:
    label='Approximation'

plt.plot(Px, Py, color='C1', label=label)

plt.vlines(xi, ymin=0, ymax=yi, color='C1', alpha=0.3)

plt.grid()
plt.legend();

```



Die Simpsonregel ist bereits in der [Funktion `scipy.integrate.simps`](#) implementiert. Im Folgenden wird nur die Differenz zur Trapezregel demonstriert.

```

n_max = 50
ns = np.arange(3, n_max, 2, dtype=int)
si = np.zeros(len(ns))
tr = np.zeros(len(ns))

for i, n in enumerate(ns):
    xi = np.linspace(0, 2, n)

```

```

yi = fkt(xi)
si[i] = scipy.integrate.simps(yi, xi)
tr[i] = scipy.integrate.trapz(yi, xi)

```

```

/var/folders/p_/_ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_26454/1089908967.py:9: DeprecationWarning: 
    si[i] = scipy.integrate.simps(yi, xi)
/var/folders/p_/_ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_26454/1089908967.py:10: DeprecationWarning: 
    tr[i] = scipy.integrate.trapz(yi, xi)

```

```

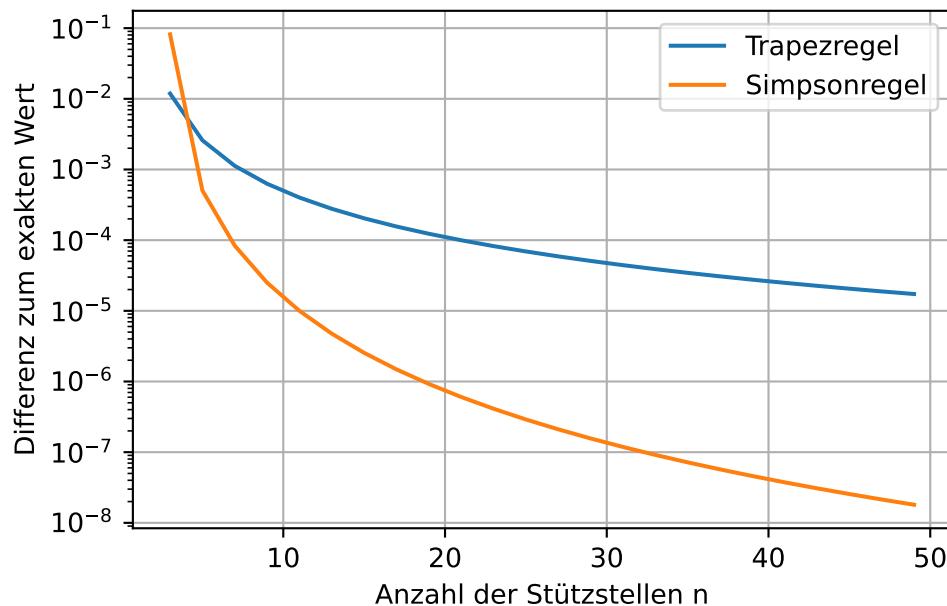
plt.plot(ns, np.abs(tr-I_exakt), label='Trapezregel')
plt.plot(ns, np.abs(si-I_exakt), label='Simpsonregel')

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
plt.yscale('log')

plt.legend()
plt.grid();

```



## 35.4 Monte-Carlo

Ein ganz anderer Ansatz zur Integration wird mit dem **Monte-Carlo-Ansatz** verfolgt. Hierbei werden Zufallspunkte  $x_i$  innerhalb der gesuchten Integralbereichs generiert. Der Mittelwert der dazugehörigen Summe der Funktionswerte  $f(x_i)$  nähert das Integral an. Insbesondere für eine kleine Anzahl von Zufallswerten kann das Ergebnis deutlich vom exakten Wert abweichen. Der Vorteil des Verfahrens wird bei hochdimensionalen Integralen deutlich.

Für  $n \gg 1$  zufällige Stützstellen  $x_i \in [a, b]$  gilt folgende Näherung

$$I = \int_a^b f(x) dx \approx \frac{b-a}{n} \sum_{i=1}^n f(x_i)$$

Für das Beispiel aus den vorhergehenden Kapiteln gilt

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung
I_exakt = (-1/3*np.cos(3*2) + 2**2) - (-1/3)
```

```
n = 2000
xi = np.random.random(n) * 2
yi = fkt(xi)
I = 2 * 1/n * np.sum(yi)
print(f"Integralwert für {n} Stützstellen: {I:.4f}")
```

Integralwert für 2000 Stützstellen: 3.9988

```
n_max = 50000
dn = 250
ns = np.arange(dn, n_max, dn, dtype=int)
mc = np.zeros(len(ns))

xi = np.zeros(n_max)

for i, n in enumerate(ns):
```

```

xi[n-dn:n] = np.random.random(dn) * 2
yi = fkt(xi[:n])
mc[i] = 2 * 1/n * np.sum(yi)

```

```

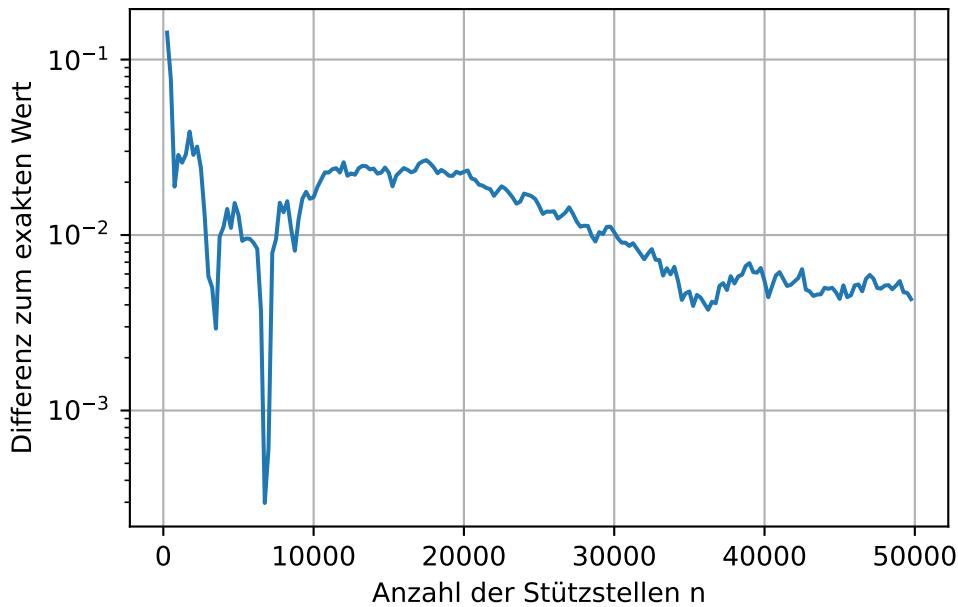
plt.plot(ns, np.abs(mc-I_exakt))

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
# plt.yscale('log')

plt.grid();

```



Alternativ kann auch das Flächenverhältnis zwischen der zu integrierenden Funktion und einer Referenzfläche  $A_r$  gebildet werden. Hierzu werden  $n$  Zufallszahlenpaare  $(x_i, y_i)$  generiert und gezählt wieviele davon in der gesuchten Fläche liegen. Die Annahme ist, dass sich beide Verhältnisse für große  $n$  annähern. Im einfachsten Fall, wenn  $f(x) \geq 0$ , gilt folgende Abschätzung

$$I \approx \frac{A_r \cdot |\{y_i \mid y_i < f(x_i)\}|}{n}$$

Im obigen Beispiel kann die Fläche  $[0, 2] \times [0, 4] = 8$  als Referenzfläche verwendet werden.

```

n = 2000
xi = np.random.random(n) * 2
yi = np.random.random(n) * 4

z = np.sum(yi < fkt(xi))

I = z / n * 8
print(f"Integralwert für {n} Stützstellen: {I}")

```

Integralwert für 2000 Stützstellen: 4.092

```

n_max = 50000
dn = 250
ns = np.arange(dn, n_max, dn, dtype=int)
mc = np.zeros(len(ns))

xi = np.zeros(n_max)
yi = np.zeros(n_max)

for i, n in enumerate(ns):
    xi[n-dn:n] = np.random.random(dn) * 2
    yi[n-dn:n] = np.random.random(dn) * 4
    z = np.sum(yi[:n] < fkt(xi[:n]))
    mc[i] = z / n * 8

```

```

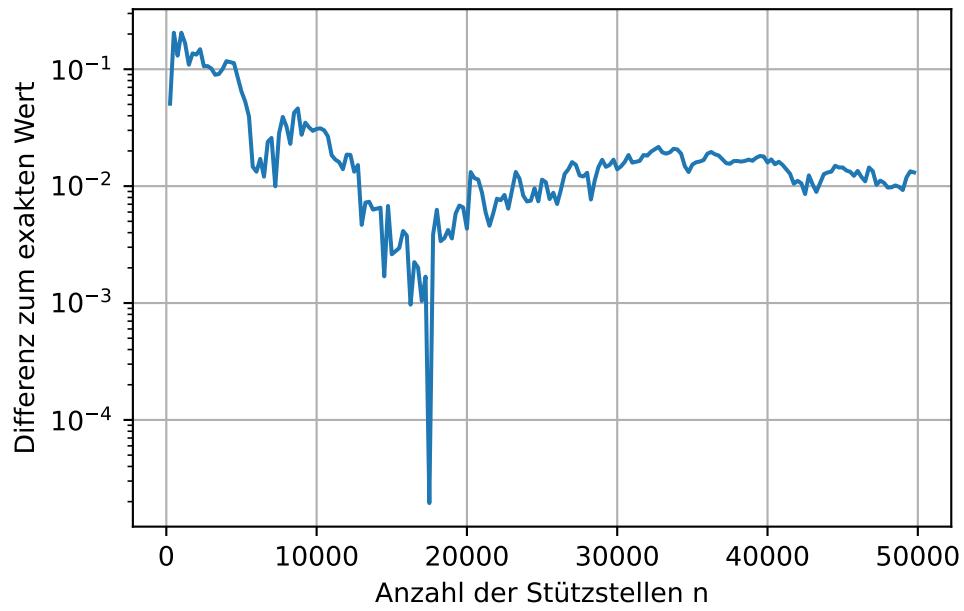
plt.plot(ns, np.abs(mc-I_exakt))

plt.xlabel('Anzahl der Stützstellen n')
plt.ylabel('Differenz zum exakten Wert')

# plt.xscale('log')
# plt.yscale('log')

plt.grid();

```



# 36 Differentiation

Die numerische Bestimmung von Ableitungen wird hier anhand von zwei Ansätzen demonstriert. Zum Einen als Differenzenquotienten und zum Anderen über das Polynomfitting. Angewendet werden diese Verfahren z.B. beim Suchen von Extrema in Experimental- oder Simulationsdaten, beim Lösen von Differentialgleichungen oder bei Optimierungsverfahren.

Obwohl die analytische Bildung einer Ableitung oft viel einfacher ist als die Integration, ist dies in den oben genannten Fällen nicht direkt möglich. Gesucht ist hierbei immer die Ableitung  $f'(x)$  einer Funktion  $f(x)$  oder einer diskreten Punktmenge  $(x_i, y_i)$  an einer bestimmten Stelle  $x = x_0$  oder auf einem Intervall.

Die Grundidee bei den hier vorgestellten Differenzenquotienten bzw. Differenzenformeln ist die Annäherung der abzuleitenden Funktion mit einer Taylor-Entwicklung an mehreren Stellen. Damit kann nach der gesuchte Ableitung an der entsprechenden Entwicklungsstelle aufgelöst werden.

## 36.1 Taylor-Entwicklung

Mittels der [Taylor-Entwicklung](#) kann jede beliebig oft stetig differenzierbare Funktion  $f(x)$  um einem Entwicklungspunkt  $x_0$  beliebig genau angenähert werden. Die funktionale Abhängigkeit bezieht sich nun auf die Variable  $h$ , welche nur in direkter Umgebung um  $x_0$  betrachtet wird. Die Taylor-Entwicklung lautet:

$$\begin{aligned} f(x_0 + h) &= \sum_{i=0}^{\infty} \frac{1}{i!} f^{(i)}(x_0) \cdot h^i \\ &= f(x_0) + f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \frac{1}{6} f'''(x_0) \cdot h^3 + \dots \end{aligned}$$

Diese Entwicklung kann auch nur bis zu einer vorgegebenen Ordnung betrachtet werden. So nimmt die Entwicklung bis zur Ordnung  $\mathcal{O}(h^3)$  folgende Form an:

$$f(x_0 + h) = f(x_0) + f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \mathcal{O}(h^3)$$

Hierbei deutet das Landau-Symbol  $\mathcal{O}$  die Ordnung an, welche die vernachlässigten Terme, hier ab  $h^3$ , als Approximationsfehler zusammenfasst. Die Ordnung gibt an wie schnell bzw. mit welchem funktionalem Zusammenhang der Approximationsfehler gegen Null läuft für  $h \rightarrow 0$ .

Eine graphische Darstellung der ersten Elemente der Reihe verdeutlichen nochmals die Grundeidee. Das folgende Beispiel entwickelt die Funktion

$$f(x) = \sin(3x) + 2x$$

am Punkt  $x_0 = 0.85$ .

```
def fkt(x, p=0):
    if p==0:
        return np.sin(3*x) + 2*x
    if p==1:
        return 3*np.cos(3*x) + 2
    if p==2:
        return -9*np.sin(3*x)
    if p==3:
        return -27*np.cos(3*x)
    return None

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x, p=0)
```

```
x0 = 0.85

# Taylor-Elemente
te = []
te.append(0*(x-x0) + fkt(x0, p=0))
te.append((x-x0) * fkt(x0, p=1))
te.append((x-x0)**2 * fkt(x0, p=2) * 1/2)
te.append((x-x0)**3 * fkt(x0, p=3) * 1/6)
```

```
plt.plot(x, y, color='Grey', lw=3, label="Funktion")
plt.plot(x, te[0], label="$\mathcal{O}(1)$")
plt.plot(x, te[0] + te[1], label="$\mathcal{O}(h)$")
plt.plot(x, te[0] + te[1] + te[2], label="$\mathcal{O}(h^2)$")
plt.plot(x, te[0] + te[1] + te[2] + te[3], label="$\mathcal{O}(h^3)$")

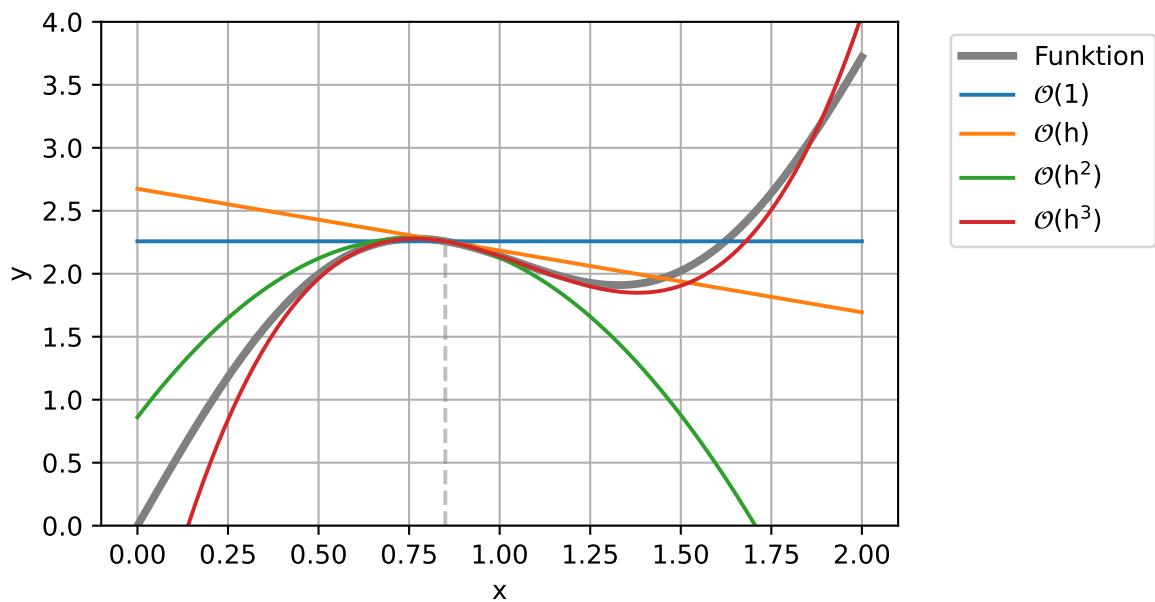
plt.vlines(x0, ymin=0, ymax=fkt(x0), color='Grey', ls='--', alpha=0.5)
```

```

plt.ylim([0,4])

plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
plt.grid()
plt.xlabel('x')
plt.ylabel('y');

```



## 36.2 Differenzenformeln

In diesem Abschnitt werden Berechnungsformeln für die Approximation von Ableitungen durch Bildung von Funktionswertdifferenzen vorgestellt. Diese beruhen alle auf der Taylor-Entwicklung und können für beliebige Ableitungen und Ordnungen formuliert werden. Die einfachsten davon werden hier vorgestellt.

### 36.2.1 Erste Ableitung erster Ordnung

Die einfachste Differenzenformel ergibt sich aus der Taylor-Reihe bis  $\mathcal{O}(h^2)$ . Hier kann die Reihe direkt nach der gesuchten Ableitung an der Stelle  $x_0$  umgeformt werden.

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \mathcal{O}(h^2)$$

$$\Rightarrow f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \mathcal{O}(h)$$

Dies ist die vorwärtsgerichtete Differenzformel erster Ordnung für die erste Ableitung. Erste Ordnung bedeutet hierbei, dass im Grenzwert  $h \rightarrow 0$  der Approximationsfehler linear mit der Schrittweite abnimmt.

Nach dieser Formel muss die abzuleitende Funktion an zwei Stellen  $f(x_0)$  und  $f(x_0 + h)$  ausgewertet werden, um die Ableitung numerisch zu bestimmen. Im Grenzwert für eine beliebig kleine Schrittweite, d.h.  $h \rightarrow 0$ , nähert sich dieser Quotient der exakten Ableitung an der Stelle  $x_0$  an.

Das folgende Beispiel demonstriert die Näherung anhand der Funktion

$$f(x) = \sin(3x) + 2x$$

Die Ableitung wird an der Stelle  $x_0 = 0.85$  angenähert.

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung bei x=0.85
fp_exakt = 3*np.cos(3*0.85) + 2

# Entwicklungspunkt und Schrittweite
h = 0.25
x0 = 0.85

# Auswertung an den beiden Stellen
f0 = fkt(x0)
fh = fkt(x0 + h)

# Bestimmung der Ableitungsnäherung
fp = (fh - f0) / h

print(f"Die numerische Näherung der Ableitung an der Stelle {x0:.2f}:")
print(f"Näherung mit Schrittweite {h:.2f}: {fp:.2f}")
print(f"Exakter Wert: {fp_exakt:.2f}")
```

Die numerische Näherung der Ableitung an der Stelle 0.85:  
 Näherung mit Schrittweite 0.25: -0.86  
 Exakter Wert: -0.49

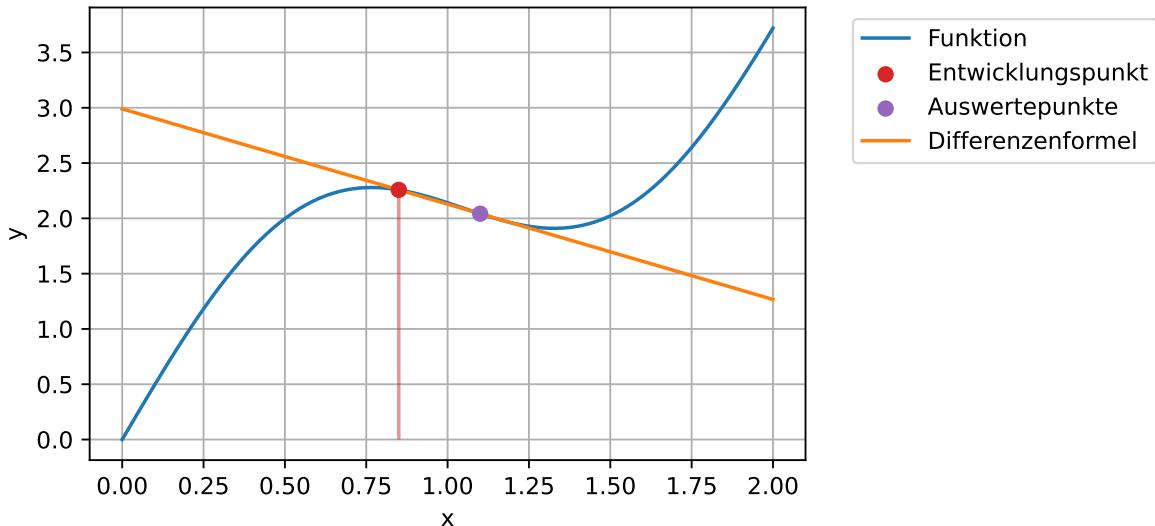
Die Methode kann auch graphisch dargestellt werden. Die gesuchte Steigung ist die Steigung der eingezeichneten Geraden.

```
plt.plot(x, y, label="Funktion")
plt.scatter([x0], [f0], color='C3', label='Entwicklungs punkt', zorder=3)
plt.scatter([x0+h], [fh], color='C4', label='Auswertepunkte', zorder=3)

plt.vlines(x0, ymin=0, ymax=f0, color='C3', alpha=0.5)

plt.plot(x, f0 + fp*(x-x0), label='Differenzenformel')

plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left');
```



### 36.2.2 Erste Ableitung zweiter Ordnung

Mit dem gleichen Ansatz kann auch eine Differenzenformel zweiter Ordnung gefunden werden. Dazu wird die Funktion an den Stellen  $x_0 - h$  und  $x_0 + h$  mit der Taylor-Reihe bis zur Ordnung  $\mathcal{O}(h^3)$  approximiert.

$$f(x_0 + h) = f(x_0) + f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \mathcal{O}(h^3)$$

$$f(x_0 - h) = f(x_0) - f'(x_0) \cdot h + \frac{1}{2} f''(x_0) \cdot h^2 + \mathcal{O}(h^3)$$

Die Differenz dieser beiden Gleichungen führt zu

$$f(x_0 + h) - f(x_0 - h) = 2f'(x_0) \cdot h + \mathcal{O}(h^3)$$

Und die Umformung nach der gesuchten Ableitung an der Stelle  $x_0$  ergibt

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + \mathcal{O}(h^2)$$

Dies ist die zentrale Differenzenformel für die erste Ableitung zweiter Ordnung. Wie bei der vorwärtsgerichteten Formel muss hier die Funktion an zwei Stellen ausgewertet werden, jedoch nicht mehr am Entwicklungspunkt selbst. Durch diese Symmetrie bzgl. des Entwicklungspunkts ergibt sich ein besseres, hier quadratisches, Konvergenzverhalten.

```
# Auswertung an den beiden Stellen
fnh = fkt(x0 - h)
fph = fkt(x0 + h)

# Bestimmung der Ableitungsnäherung
fp = (fph - fnh) / (2*h)

print(f"Die numerische Näherung der Ableitung an der Stelle {x0:.2f}:")
print(f"Näherung mit Schrittweite {h:.2f}: {fp:.2f}")
print(f"Exakter Wert: {fp_exakt:.2f}")
```

Die numerische Näherung der Ableitung an der Stelle 0.85:  
Näherung mit Schrittweite 0.25: -0.26  
Exakter Wert: -0.49

Die Methode kann auch graphisch dargestellt werden. Die gesuchte Steigung ist die Steigung der eingezeichneten Geraden.

```

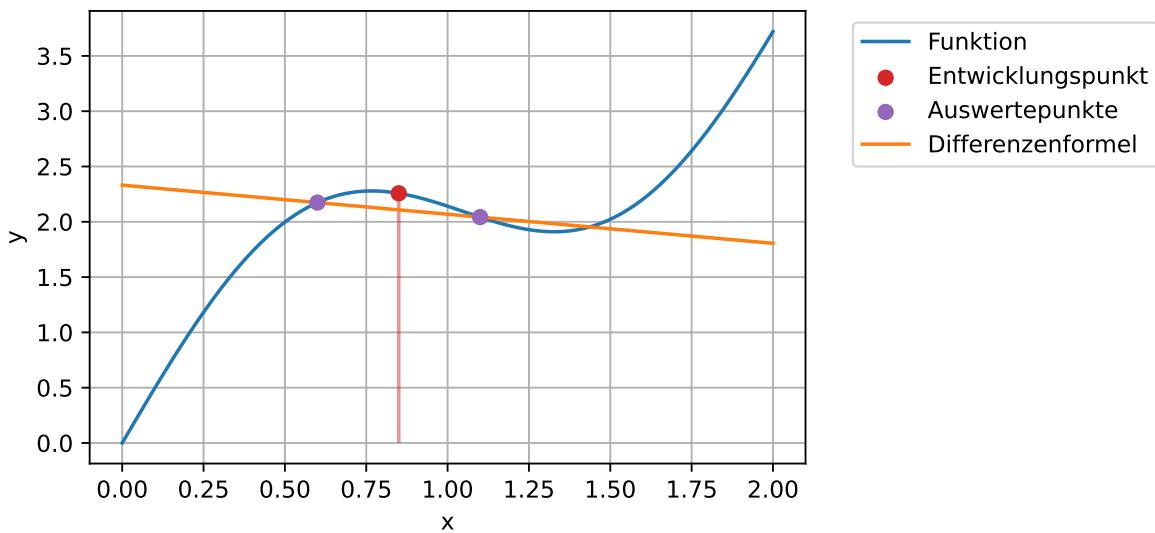
plt.plot(x, y, label="Funktion")
plt.scatter([x0], [f0], color='C3', label='Entwicklungs punkt', zorder=3)
plt.scatter([x0-h, x0+h], [fnh, fph], color='C4', label='Auswertepunkte', zorder=3)

plt.vlines(x0, ymin=0, ymax=f0, color='C3', alpha=0.5)

plt.plot(x, fnh + fp*(x-x0+h), label='Differenzenformel')

plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left');

```



### 36.3 Zweite Ableitung zweiter Ordnung

Mit dem gleichen Schema wie oben, kann auch die Differenzenformel für die zweite Ableitung bestimmt werden. Diese lautet

$$f''(x_0) = \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} + \mathcal{O}(h^2)$$

## 36.4 Fehlerbetrachtung

In diesem Abschnitt werden die Approximationsfehler, d.h. Fehler aus der Differenzenformeln, und Rundungsfehler, d.h. Fehler durch die endliche Genauigkeit der digitalen Darstellung von Zahlen, betrachtet.

### 36.4.1 Approximationsfehler

Die Ordnung des Verfahrens kann durch die Betrachtung des Fehlers, hier zum bekannten exakten Wert, bestimmt werden. Dazu wird die Schrittweite kontinuierlich verkleinert.

```
def fkt(x):
    return np.sin(3*x) + 2*x

# Daten für die Visualisierung
x = np.linspace(0, 2, 100)
y = fkt(x)

# Exakte Lösung bei x=1
fp_exakt = 3*np.cos(3*0.85) + 2

x0 = 0.85

hs = []
fpfs = []
fpcos = []

h0 = 1
for i in range(18):
    h = h0 / 2**i

    f0 = fkt(x0)
    fnh = fkt(x0 - h)
    fph = fkt(x0 + h)

    fpf = (fph - f0) / h
    fpc = (fph - fnh) / (2*h)

    hs.append(h)
    fpfs.append(fpf)
    fpcos.append(fpc)
```

```

plt.plot(hs, np.abs(fpfs - fp_exakt), label='vorwärts')
plt.plot(hs, np.abs(fpcs - fp_exakt), label='zentral')

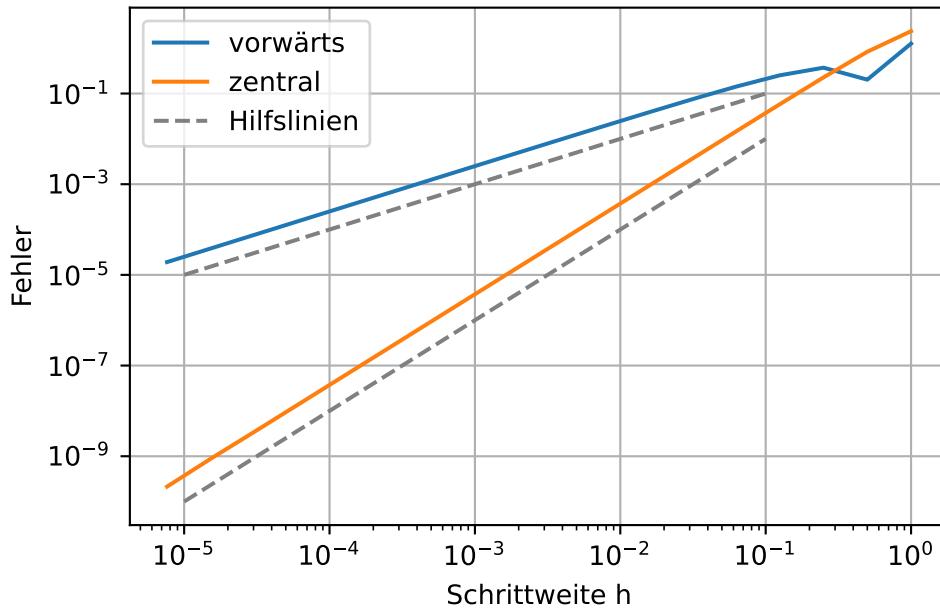
plt.plot([1e-5, 1e-1], [1e-5, 1e-1], '--', color='grey', label='Hilfslinien')
plt.plot([1e-5, 1e-1], [1e-10, 1e-2], '--', color='grey')

plt.xlabel('Schrittweite h')
plt.ylabel('Fehler')

plt.xscale('log')
plt.yscale('log')

plt.legend()
plt.grid();

```



In der logarithmischen Darstellung beider Achsen werden Potenzfunktionen zu Graden mit dem Potenzgrad als Steigung. Das bedeutet, dass der Fehler im obigen Plot sich wie eine Potenzfunktion mit dem Grad eins bzw. zwei verhält. Die eingezeichneten Hilfslinien haben eine Steigung von eins bzw. zwei. Dies entspricht auch der Ordnung  $\mathcal{O}(h)$  bzw.  $\mathcal{O}(h^2)$  aus der Differenzenformel.

### 36.4.2 Rundungsfehler

Wird nun die Schrittweite noch weiter verkleinert, wirkt sich die Genauigkeit der Darstellung von Zahlen bzw. Rundungsfehler auf die Approximation aus.

```
x0 = 0.85

hs = []
fpfs = []
fpcs = []

h0 = 1
for i in range(35):
    h = h0 / 2**i

    f0 = fkt(x0)
    fnh = fkt(x0 - h)
    fph = fkt(x0 + h)

    fpf = (fph - f0) / h
    fpc = (fph - fnh) / (2*h)

    hs.append(h)
    fpfs.append(fpf)
    fpcs.append(fpc)

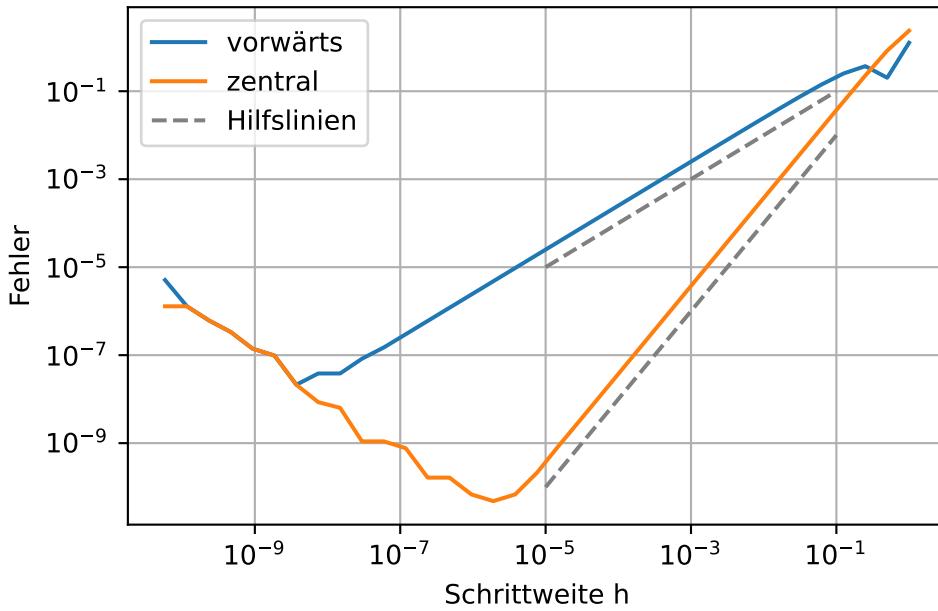
plt.plot(hs, np.abs(fpfs - fp_exakt), label='vorwärts')
plt.plot(hs, np.abs(fpcs - fp_exakt), label='zentral')

plt.plot([1e-5, 1e-1], [1e-5, 1e-1], '--', color='grey', label='Hilfslinien')
plt.plot([1e-5, 1e-1], [1e-10, 1e-2], '--', color='grey')

plt.xlabel('Schrittweite h')
plt.ylabel('Fehler')

plt.xscale('log')
plt.yscale('log')

plt.legend()
plt.grid();
```



Wie bereits vorgestellt, können 64-Bit-Zahlen nur mit einer Genauigkeit von etwa  $\approx 10^{-16}$  dargestellt werden. Das bedeutet, dass z.B. die Differenz von zwei Zahlen nicht genauer als berechnet werden kann. Dies ist der sogenannte Rundungsfehler.

Im konkreten Fall der Vorwärtssubtraktionsformel bedeutet dies:

$$\begin{aligned}
 f'(x_0) &= \frac{f(x_0 + h) - f(x_0)}{h} + \mathcal{O}(h) \\
 \xrightarrow{\text{Rundungsfehler}} \quad &\frac{f(x_0 + h) - f(x_0) + \mathcal{O}(\epsilon)}{h} + \mathcal{O}(h) \\
 &= \frac{f(x_0 + h) - f(x_0)}{h} + \mathcal{O}\left(\frac{\epsilon}{h}\right) + \mathcal{O}(h)
 \end{aligned}$$

Damit macht eine Verkleinerung von  $h$  nur Sinn, solange der Rundungsfehler klein gegenüber  $h$  ist. Genauer:

$$\begin{aligned}
 \frac{\epsilon}{h} &\leq h \\
 \Rightarrow h &\geq \sqrt{\epsilon}
 \end{aligned}$$

Mit  $\approx 10^{-16}$  ist für diese Differenzenformel ein  $h$  nur bis etwa  $10^{-8}$  angemessen.