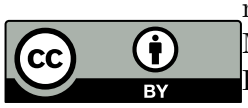
Bausteine Computergestützter Datenanalyse

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

2025-10-10

Inhaltsverzeichnis

Werkzeugbaustein Python



Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Franca Hollmann, 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 Hollmann,
 year={2025},
 url={https://github.com/bausteine-der-datenanalyse/w-python}}
```

Voraussetzungen

Keine Voraussetzungen

Lernziele

In diesem Bausteine 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

Abbildung 1.1: Logo der Programmiersprache Python

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:

```
Hallo 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 + "!")
```

```
Hallo Ihr Name hier!
```

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

```
Hallo 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:

```python
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`

```python
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'>
```

> Wichtig
>
> 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: 3.3333333333333335
Ganzzahlige Division: 3
Division mit Rest: 1
```

> **Hinweis**
>
> `//` 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
`int()`	Text/Zahl → ganze Zahl	`int("10")` → `10`
`float()`	Text/Zahl → Kommazahl	`float("3.14")` → `3.14`

```
# 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:

```python
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`

```python
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
```

> **Wichtig**
>
> 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 (exklusive)
- 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]
```

> Hinweis
>
> 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']
```

> **Hinweis**
>
> 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**:

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

print(sortiert)
```

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

> **Wichtig**
>
> Die Original-Liste bleibt **unverändert**.
> Wenn Sie die Liste direkt verändern möchten, geht das mit:
>
> ```python
> 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()
```

```
Hallo und willkommen!
```

6.3 Parameter übergeben

Funktionen können Eingabewerte (Parameter) erhalten:

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

begruessung("Alex")
```

```
Hallo Alex!
```

6.4 Rückgabewerte mit `return`

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

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

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

```
25
```

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))
```

```
68.0
```

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!
```

i `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.
```

> **Hinweis**
>
> `.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 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**!

7.5 Zeilenweise schreiben mit Schleife

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

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

! Wichtig

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")
```

7.6 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.']
```

> **!** Wichtig
>
> Jede Zeile endet mit `\n`, deshalb kann eine Nachbearbeitung mit `.strip()` sinnvoll sein:
>
> ```
> for zeile in zeilen:
>     print(zeile.strip())
> ```
>
> ```
> Dies ist ein Test.
> ```

7.7 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()
```

> **!** Wichtig
>
> `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.