Die Notebooks zum Lesen

Informatik 10 · MINT-Zweig · Gymnasium Ottweiler

Alle Notebooks mit den Ausgaben der Beispiele, ohne Lösungen. Zum Bearbeiten lädst du das Notebook (.ipynb) herunter.

1. DS 1 – Einstieg: Variablen, Eingabe und Ausgabe
2. DS 1 – Weitere Übungen
3. DS 2 – Operatoren: rechnen, vergleichen, verknüpfen
4. DS 2 – Weitere Übungen
5. DS 3 – Verzweigungen: Programme treffen Entscheidungen
6. DS 3 – Weitere Übungen
7. DS 4 – Übung: Verzweigungen lesen, testen und erweitern
8. DS 4 – Weitere Übungen
9. DS 5 – Die for-Schleife: Anweisungen wiederholen
10. DS 5 – Weitere Übungen
11. DS 6 – Die while-Schleife: Wiederholen, solange eine Bedingung gilt
12. DS 6 – Weitere Übungen
13. DS 7 – Übung: for- oder while-Schleife?
14. DS 7 – Weitere Übungen
15. DS 8 – Funktionen: Programme aus Bausteinen
16. DS 8 – Weitere Übungen
17. DS 9 – Übungsstunde: Funktionen lesen, testen und erweitern
18. DS 9 – Weitere Übungen
19. DS 10 – Zeichenketten: Texte Zeichen für Zeichen
20. DS 10 – Weitere Übungen
21. DS 11 – Listen: viele Werte unter einem Namen
22. DS 11 – Weitere Übungen
23. DS 12 – Algorithmen auf Listen: Summe, Minimum und Suche
24. DS 12 – Weitere Übungen
25. DS 13 – Teamprojekt: Verschlüsselung mit der Caesar-Chiffre (Teil 1)
26. DS 13 – Weitere Übungen
27. DS 14 – Teamprojekt: Verschlüsselung (Teil 2) und Berufe in der Softwareentwicklung
28. DS 14 – Weitere Übungen
29. DS 15 – Grundgatter AND, OR, XOR, NOT und Belegungstabellen
30. DS 16 – Entwurf logischer Schaltungen
31. DS 17 – Addition im Binärsystem und Halbaddierer
32. DS 18 – Volladdierer und Carry-Ripple-Addierer
33. DS 19 – Rückkopplung als Speichertechnik: Das Flipflop
34. DS 20 – Klassische Aussagenlogik: Syntax und Termbäume
35. DS 21 – Belegungstabellen, Äquivalenz und Disjunktive Normalform (DNF)
36. DS 22 – Karnaugh-Veitch-Diagramme und Minimierung
37. DS 23 – Problem, Probleminstanz und abstrahierte Darstellung
38. DS 24 – Optimale Lösung und Brute-Force-Ansatz
39. DS 25 – Heuristiken: Der Greedy-Ansatz
40. DS 26/27 – Vertiefung am Beispiel: Das Rucksackproblem
41. DS 28 – Aufwand verschiedener Lösungsansätze vergleichen
42. DS 29 – Bewertung von Heuristiken & Abschluss
43. DS 30 – Sprachmodelle: Regelbasierte Systeme vs. n-Gramm-Modelle
44. DS 31 – n-Gramm-Analyse von Textkorpora (von Hand)
45. DS 32 – Rechnergestützte Textkörperanalyse
46. DS 33 – Textgenerierung und Sampling-Strategien
47. DS 34 – Halluzination, n-Gramm-Modelle vs. moderne LLMs & Berufsorientierung

DS 1 – Einstieg: Variablen, Eingabe und Ausgabe

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- mit `print()` Texte und Zahlen auf dem Bildschirm ausgeben,
- Variablen anlegen, ihnen Werte zuweisen und mit ihnen rechnen,
- die Datentypen `int`, `float`, `str` und `bool` unterscheiden,
- mit `input()` Eingaben einlesen und mit `int()` und `float()` in Zahlen umwandeln,
- eine Fehlermeldung lesen und die Ursache des Fehlers finden.

🗣️ Sprachhilfe

- *„alter ist 15."* → „Der Variablen `alter` wird der Wert 15 zugewiesen."
- *„Das Programm fragt nach dem Namen."* → „Die Eingabeanweisung liest den Namen ein und speichert ihn als Zeichenkette in der Variablen `name`."
- *„Da kommt 14 raus."* → „Zur Eingabe 7 gibt das Programm den Wert 14 aus."

Worum es geht

Ein Programm ist eine Liste von Anweisungen, die der Computer der Reihe nach ausführt, von oben nach unten. In der Blockprogrammierung hast du Bausteine zusammengeschoben, in Python schreibst du Text. Heute lernst du die drei wichtigsten Dinge, die fast jedes Programm braucht: Es **gibt etwas aus**, es **merkt sich Werte** und es **liest Eingaben** ein.

1 Das erste Programm: Ausgaben mit `print`

`print` heißt „drucken", gemeint ist: auf dem Bildschirm anzeigen. Was angezeigt werden soll, steht in den **runden Klammern**, ein Text in **Anführungszeichen**. Python führt die Zeilen **nacheinander von oben nach unten** aus.

```
1 print("Hallo Welt!")
2 print("Guten Morgen!")
3 print("Heute lernen wir Python.")
```

```
Hallo Welt!
Guten Morgen!
Heute lernen wir Python.
```

🧪 **Probiere es aus:** Ändere die Texte. Was passiert, wenn du die letzte Zeile an den Anfang verschiebst?

Text oder Zahl?

🔮 **Vorhersage:** Was geben die beiden ersten Zeilen aus?

```python
print(3 + 4)
print("3 + 4")
print("Ich bin", 15, "Jahre alt.")
print("In fünf Jahren bin ich", 15 + 5, "Jahre alt.")
```

```
7
3 + 4
Ich bin 15 Jahre alt.
In fünf Jahren bin ich 20 Jahre alt.
```

- **Zeile 1:** Ohne Anführungszeichen ist `3 + 4` eine Rechnung, Python gibt `7` aus.
- **Zeile 2:** In Anführungszeichen ist es ein Text, der genau so ausgegeben wird.
- **Zeilen 3–4:** Mehrere Dinge trennt man durch **Kommas**. Python setzt je ein Leerzeichen dazwischen.

> 📌 **Merksatz:** Mit `print(...)` gibt man Werte auf dem Bildschirm aus. Texte stehen in Anführungszeichen, Zahlen und Rechnungen ohne. Mehrere Werte trennt man durch Kommas.

⚠️ Wenn etwas schiefgeht

Beim Abtippen passieren Fehler, auch Profis jeden Tag. Die Fehlermeldung ist ein Hinweis, wo das Problem liegt. Führe beide Zellen aus.

```python
print("Hallo)
```

```
  File "<Zelle 3>", line 1
    print("Hallo)
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

```python
print(Hallo)
```

```
Traceback (most recent call last):
  File "<Zelle 4>", line 1, in <module>
NameError: name 'Hallo' is not defined
```

1. `SyntaxError: unterminated string literal (detected at line 1)`: Die Schreibweise stimmt nicht, der Text ist nicht abgeschlossen. Python nennt sogar die Zeile.
2. `NameError: name 'Hallo' is not defined`: Ohne Anführungszeichen hält Python `Hallo` für einen **Namen**, und den gibt es nicht.

Häufige Tippfehler am Anfang: vergessenes Anführungszeichen oder Klammer, `Print` statt `print` (Groß- und Kleinschreibung zählt!) und deutsche Anführungszeichen „…“ statt `"…"`. Schau bei einer Fehlermeldung zuerst die genannte Zeile an und die Zeile davor.

2 Variablen: Werte speichern

Programme müssen sich Dinge merken: einen Namen, einen Punktestand, einen Preis. Stell dir eine **Variable** wie eine **beschriftete Schachtel** vor: Außen steht der Name, z. B. `alter`, innen liegt ein Wert, z. B. `15`.

> 📌 **Definition:** Eine **Variable** ist ein benannter Speicherplatz im Arbeitsspeicher, in dem ein Wert abgelegt wird. Über ihren Namen kann das Programm den Wert lesen und verändern.

```python
alter = 15
print(alter)
print("Ich bin", alter, "Jahre alt.")
```

```
15
Ich bin 15 Jahre alt.
```

- **Zeile 1:** `alter` bekommt den Wert `15`. Auf dem Bildschirm passiert nichts.
- **Zeile 2:** `alter` steht ohne Anführungszeichen, Python schaut in die Schachtel und gibt `15` aus.

Das Gleichheitszeichen ist ein Zuweisungszeichen

`=` heißt in Python **nicht** „ist gleich“, sondern **„bekommt den Wert“**. Zuerst wird die rechte Seite ausgerechnet, dann kommt das Ergebnis in die Variable links. Darum ist `alter = alter + 1` sinnvoll, obwohl es in Mathe falsch wäre.

```
1 alter = 15
2 print(alter)
3 alter = alter + 1
4 print(alter)
```

```
15
16
```

Das **Ablaufprotokoll** zeigt nach jeder Zeile den Wert der Variablen:

Schritt	Zeile	ausgeführte Anweisung	`alter` danach
1	1	`alter = 15`	`15`
2	2	`print(alter)`	`15`
3	3	`alter = alter + 1`	`16`
4	4	`print(alter)`	`16`

Der alte Wert `15` ist nach Zeile 3 weg, er wurde **überschrieben**.

> 📌 **Merksatz:** Mit dem **Zuweisungsoperator** `=` wird einer Variablen ein Wert zugewiesen, z. B. `alter = 15`. Zuerst wird die rechte Seite ausgewertet, dann wird das Ergebnis in der Variablen links gespeichert.

Gute Namen für Variablen

- Erlaubt sind Buchstaben, Ziffern und `_`. Ein Name darf **nicht mit einer Ziffer beginnen** und **keine Leerzeichen** enthalten.
- `alter` und `Alter` sind zwei verschiedene Namen.
- Wähle **sprechende Namen**: `preis` statt `p`, `anzahl_artikel` statt `aa`. Verzichte auf Umlaute: `groesse` statt `größe`.

> ⚠️ **Stolperstein:** Vertippst du dich beim Namen, meldet Python `NameError: name 'Alter' is not defined. Did you mean: 'alter'?`

🔮 Übung 1 – Vorhersage

Lies das Programm, **ohne** es auszuführen. Schreibe auf, was es ausgibt. Überprüfe danach.

```
1 punkte = 10
2 print(punkte)
3 punkte = punkte + 5
4 punkte = punkte * 2
5 print("Endstand:", punkte)
```

```
10
Endstand: 30
```

Meine Vorhersage:

…

3 Datentypen: Zahlen, Texte und Wahrheitswerte

Die Art eines Wertes heißt **Datentyp**. Davon hängt ab, was man mit dem Wert machen kann: Mit Zahlen kann man rechnen, mit Texten nicht.

Datentyp	Bedeutung	Beispiele
`int`	ganze Zahl (*integer*)	`15`, `-3`, `0`, `2026`
`float`	Kommazahl (*floating point number*)	`1.68`, `-0.5`, `3.0`
`str`	Zeichenkette, also Text (*string*)	`"Alex"`, `"Hallo Welt!"`, `"15"`
`bool`	Wahrheitswert (nach George Boole)	`True`, `False`

- Kommazahlen schreibt man mit **Punkt**: `1.68`, nicht `1,68`.
- Alles in Anführungszeichen ist Text, auch `"15"`.
- `True` und `False` schreibt man groß und ohne Anführungszeichen.

Mit `type(...)` fragst du Python nach dem Datentyp:

```
name = "Alex"
alter = 15
groesse = 1.68
volljaehrig = False
print(type(name))
print(type(alter))
print(type(groesse))
print(type(volljaehrig))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

> 📌 **Merksatz:** Python unterscheidet u. a. die Datentypen `int` (Ganzzahl), `float` (Kommazahl), `str` (Zeichenkette) und `bool` (Wahrheitswert).

Rechnen mit Zahlen

`+`, `-`, `*` (mal) und `/` (geteilt). Es gilt **Punkt vor Strich**, Klammern zuerst. Das Ergebnis von `/` ist **immer** ein `float`, auch wenn die Division aufgeht.

```
a = 7
b = 2
print(a + b)
print(a - b)
print(a * b)
print(a / b)
print(2 + 3 * 4)
print((2 + 3) * 4)
```

```
9
5
14
3.5
14
20
```

⚠️ Stolperstein: Komma statt Punkt

🔮 **Vorhersage:** Was gibt die Zelle aus?

```
preis = 2,50
print(preis)
```

```
(2, 50)
```

Keine Fehlermeldung, aber Python hat zwei Zahlen gespeichert, `2` und `50`. Richtig ist `preis = 2.50`.

✏️ Übung 2 – Datentypen bestimmen

Gib zu jedem Wert den Datentyp an. Überprüfe danach mit `type(...)` in der Codezelle.

Wert	42	"42"	4.2	True	"True"	-7	7 / 7	"Hallo"
Datentyp								

```
print(type(42))
# TODO: die übrigen sieben Werte prüfen
```

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

4 Eingaben mit `input`

Mit `input(...)` kann die Nutzerin oder der Nutzer etwas eingeben. `input` zeigt zuerst den Text in den Klammern an (die **Eingabeaufforderung**) und **wartet**, bis jemand etwas eintippt und die Eingabetaste drückt.

```
name = input("Wie heißt du? ")
print("Hallo", name)
```

```
Wie heißt du? Alex
Hallo Alex
```

> 💡 **Tipp:** Setze ans Ende der Eingabeaufforderung ein Leerzeichen, sonst klebt die Eingabe am Fragezeichen.

Die Eingabe ist immer ein Text

🔮 **Vorhersage:** Gib `7` ein. Was erwartest du?

```
1 zahl = input("Gib eine Zahl ein: ")
2 print(zahl + zahl)
3 print(type(zahl))
```

```
Gib eine Zahl ein: 15
1515
<class 'str'>
```

Python rechnet nicht `7 + 7 = 14`, sondern gibt `77` aus. Was jemand eintippt, ist für Python **immer eine Zeichenkette**, und `+` hängt zwei Zeichenketten aneinander: `"7" + "7"` ergibt `"77"`.

> 📌 **Merksatz:** Mit `input(...)` liest man eine Eingabe ein. **Die Eingabe ist immer vom Typ `str`!**

5 Typumwandlung: aus Text wird eine Zahl

Befehl	wandelt um in	Beispiel	Ergebnis
`int(...)`	ganze Zahl	`int("7")`	`7`
`float(...)`	Kommazahl	`float("2.5")`	`2.5`
`str(...)`	Zeichenkette	`str(15)`	`"15"`

```python
lieblingszahl = input("Gib deine Lieblingszahl ein: ")
print("Deine Eingabe war:", lieblingszahl, "- Typ:", type(lieblingszahl))

lieblingszahl_int = int(lieblingszahl)
print("Deine Zahl mal 2 ist:", lieblingszahl_int * 2)
```

```
Gib deine Lieblingszahl ein: 15
Deine Eingabe war: 15 - Typ: <class 'str'>
Deine Zahl mal 2 ist: 30
```

Eingabe und Umwandlung in einer Zeile

Meistens schreibt man beides in **eine** Zeile. Lies sie **von innen nach außen**: Zuerst liefert `input(...)` den Text `"15"`, dann macht `int(...)` daraus die Zahl `15`. Am Ende stehen **zwei** schließende Klammern. Für Kommazahlen nimmst du `float`.

```python
alter = int(input("Wie alt bist du? "))
print("Nächstes Jahr bist du", alter + 1)

groesse = float(input("Wie groß bist du in Metern? "))
print("In Zentimetern sind das", groesse * 100)
```

```
Wie alt bist du? 15
Nächstes Jahr bist du 16
Wie groß bist du in Metern? 1.72
In Zentimetern sind das 172.0
```

📌 **Merksatz:** Soll eine Eingabe als Zahl weiterverarbeitet werden, muss sie umgewandelt werden: `int(...)` bzw. `float(...)` (**Typumwandlung**).

⚠️ **Stolperstein:** Gibt jemand `sieben` ein, meldet `int(...)` einen `ValueError: invalid literal for int() with base 10: 'sieben'`. Ebenso bei `int("2.5")` und `float("2,5")`. Schau dann nach, **was eingegeben wurde**.

6 Texte zusammensetzen

Mit Kommas setzt `print` immer ein Leerzeichen dazwischen. Soll direkt nach einem Namen ein Komma oder Punkt folgen, hängst du Texte mit `+` **aneinander** (verketten). Leerzeichen musst du dann selbst in die Anführungszeichen schreiben.

```python
name = "Alex"
print("Hallo, " + name + "!")
print("Hallo,", name, "!")
```

```
Hallo, Alex!
Hallo, Alex !
```

⚠️ Stolperstein: Text plus Zahl

Mit `+` lassen sich nur **Texte mit Texten** verbinden. Führe die Zelle aus.

```python
alter = 15
print("Du bist " + alter + " Jahre alt.")
```

```
Traceback (most recent call last):
  File "<Zelle 17>", line 2, in <module>
TypeError: can only concatenate str (not "int") to str
```

Lies die Fehlermeldung **von unten nach oben**:

1. **Letzte Zeile:** `TypeError: can only concatenate str (not "int") to str`, also „kann nur `str` (nicht `int`) an `str` anhängen“.
2. **Darüber:** die Zeile, in der der Fehler passiert ist, und Zeichen `^`, die auf die Stelle zeigen.

Richtig ist `print("Du bist " + str(alter) + " Jahre alt.")`.

Fehler	Bedeutung	typische Ursache
`SyntaxError`	Schreibweise falsch	Anführungszeichen oder Klammer fehlt
`NameError`	Name unbekannt	Tippfehler, Variable noch nicht angelegt, Anführungszeichen vergessen
`TypeError`	Datentyp passt nicht	Zahl und Text mit `+` verbunden
`ValueError`	Wert passt nicht	`int("sieben")`, `float("2,5")`

📝 Übungen: Schritt für Schritt selbstständig

Die Übungen werden schrittweise schwieriger. Das **Musterbeispiel** ist ganz vorgemacht. Übung 3 ist eine Fehlersuche, bei Übung 4 füllst du Lücken `___` aus, bei Übung 5 steht nur noch ein Gerüst aus Kommentaren, die Zusatzaufgabe schreibst du allein. Gehe jedes Mal in drei Schritten vor:

Schritt	Frage
1	**Eingabe:** Was muss das Programm abfragen? Text oder Zahl (`int`/`float`)?
2	**Verarbeitung:** Was muss gerechnet werden?
3	**Ausgabe:** Was soll auf dem Bildschirm stehen?

🧩 Musterbeispiel (vorgemacht) – Kinobesuch

Aufgabe: Das Programm fragt nach dem Namen und nach der Anzahl der Kinokarten. Eine Karte kostet 9.50 €. Es gibt den Gesamtpreis in einem Satz aus.

1. **Eingabe:** Der Name ist ein Text, also nur `input(...)`. Die Anzahl ist eine ganze Zahl, also `int(input(...))`.
2. **Verarbeitung:** `gesamt = anzahl * 9.50`.
3. **Ausgabe:** Der Name soll direkt vor einem Komma stehen, deshalb `+`. Die Zahl `gesamt` muss dafür mit `str(...)` umgewandelt werden.

Lies den Code Zeile für Zeile und führe ihn aus, z. B. mit `Alex` und `3`.

```
# Schritt 1: Eingabe
name = input("Wie heißt du? ")
anzahl = int(input("Wie viele Karten? "))

# Schritt 2: Verarbeitung
gesamt = anzahl * 9.50

# Schritt 3: Ausgabe
print("Danke, " + name + "! Das macht " + str(gesamt) + " Euro.")
```

```
Wie heißt du? Alex
Wie viele Karten? 4
Danke, Alex! Das macht 38.0 Euro.
```

🐞 Übung 3 – Fehler finden

Das Programm soll nach zwei Zahlen fragen und ihre Summe ausgeben. Es enthält **drei Fehler**. Finde sie, erkläre jeweils, welche Fehlermeldung oder welches falsche Verhalten sie auslösen, und schreibe das Programm in der zweiten Zelle richtig auf. Tipp: Ein Fehler erzeugt keine Fehlermeldung, sondern ein falsches Ergebnis.

```
# Fehlerhafter Code – erst lesen, dann korrigieren
# zahl1 = input("Erste Zahl: )
# zahl2 = input("Zweite Zahl: ")
# print("Die Summe ist", Zahl1 + zahl2)
```

Die drei Fehler:

…

📝 Übung 4 – Steckbrief (Lücken füllen)

Das Programm fragt Name, Alter (als Zahl!) und Lieblingsfach ab und gibt einen vollständigen Satz aus. In der letzten Zeile steht das **doppelte Alter**: Daran siehst du, ob das Alter wirklich als Zahl gespeichert ist. Ersetze jedes `___`.

```
Wie heißt du? Alex
Wie alt bist du? 15
Was ist dein Lieblingsfach? Informatik
Hallo, ich bin Alex, 15 Jahre alt und mein Lieblingsfach ist Informatik.
Doppeltes Alter: 30
```

Lückencode

```
# Schritt 1: Eingabe
name = input("Wie heißt du? ")
alter = ___(input("Wie alt bist du? "))
fach = input("Was ist dein Lieblingsfach? ")

# Schritt 3: Ausgabe
print("Hallo, ich bin " + name + ", " + ___(alter)
      + " Jahre alt und mein Lieblingsfach ist " + ___ + ".")
print("Doppeltes Alter:", ___)
```

📝 Übung 5 – Einfacher Rechner (mit Gerüst)

Das Programm liest zwei Zahlen ein und gibt **Summe, Differenz und Produkt** aus. Ersetze die Kommentare durch Programmzeilen. Kontrolle: 7.5 und 2 ergeben 9.5, 5.5 und 15.0.

```
# Schritt 1: erste Zahl einlesen und in eine Kommazahl umwandeln
# Schritt 1: zweite Zahl einlesen und umwandeln
# Schritt 2 und 3: Summe ausgeben
#                  Differenz ausgeben
#                  Produkt ausgeben
```

🚀 Zusatzaufgabe – Gesamtpreis (selbstständig)

Frage einen Preis in Euro (als Kommazahl) und eine Anzahl an Artikeln (als ganze Zahl) ab und berechne den Gesamtpreis. Runde die Ausgabe auf zwei Stellen. Kontrolle: 1.99 € und 3 Artikel ergeben 5.97 Euro.

> 🧰 **Werkzeug:** `round(wert, stellen)` rundet auf eine bestimmte Anzahl Nachkommastellen: `round(3.14159, 2)` ergibt `3.14`.

📌 Das Wichtigste in Kürze

- `print(...)` gibt Werte aus. Texte stehen in Anführungszeichen, mehrere Werte trennt man durch Kommas.
- Eine **Variable** ist ein benannter Speicherplatz. `=` heißt „bekommt den Wert“: erst rechts rechnen, dann links speichern.
- Datentypen: `int`, `float` (mit Punkt!), `str`, `bool`. `/` liefert immer einen `float`.
- `input(...)` liefert **immer** einen Text. Zum Rechnen umwandeln mit `int(...)` oder `float(...)`.
- Texte verkettet man mit `+`, Zahlen vorher mit `str(...)` umwandeln.

- Fehlermeldungen von unten nach oben lesen: `SyntaxError`, `NameError`, `TypeError`, `ValueError`.

Zum Weiterüben: `DS01_Weitere_Uebungen.ipynb`.

DS 1 – Weitere Übungen

Variablen, Eingabe und Ausgabe

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Mit oder ohne Anführungszeichen? ●○○

Lies das Programm, **ohne** es auszuführen. Schreibe auf, was es ausgibt. Überprüfe deine Vorhersage danach am Computer.

```
name = "Mia"
alter = 14
print("Name:", name)
print("alter")
print(alter)
print("In drei Jahren bist du", alter + 3)
```

```
Name: Mia
alter
14
In drei Jahren bist du 17
```

Deine Antwort:

…

Aufgabe 2: Welcher Datentyp? ●○○

Das Programm wird mit den Eingaben `12`, `12` und `1.5` ausgeführt. Gib zu jeder Variablen an, welchen Datentyp ihr Wert hat (`int`, `float`, `str` oder `bool`). Überprüfe deine Antworten anschließend am Computer mit `print(type(...))`.

```
a = input("Gib eine Zahl ein: ")
b = int(input("Gib eine Zahl ein: "))
c = float(input("Gib eine Zahl ein: "))
d = "3.5"
e = 3.5
f = False
```

```
Gib eine Zahl ein: 15
Gib eine Zahl ein: 15
Gib eine Zahl ein: 15
```

Variable	a	b	c	d	e	f
Datentyp						

Aufgabe 3: Kontostand verfolgen ●○○

Fülle das Ablaufprotokoll aus. Trage nach jedem Schritt die Werte der beiden Variablen ein (– bedeutet: noch kein Wert). Gib dann an, was das Programm ausgibt.

```
kontostand = 20
kontostand = kontostand - 5
taschengeld = 15
kontostand = kontostand + taschengeld
print("Kontostand:", kontostand)
```

```
Kontostand: 30
```

Schritt	Zeile	kontostand	taschengeld
1	1		
2	2		
3	3		
4	4		
5	5		

Deine Antwort:

…

Aufgabe 4: Handyvertrag ●○○

Das Programm soll den monatlichen Preis eines Handyvertrags (Kommazahl) und die Laufzeit in Monaten (ganze Zahl) einlesen und die Gesamtkosten ausgeben, auf Cent gerundet. Ergänze die drei Lücken `___` und teste dein Programm mit den Eingaben `9.99` und `24`.

Lückencode

```
preis = ___(input("Preis pro Monat in Euro: "))
monate = ___(input("Laufzeit in Monaten: "))
gesamt = preis * monate
print("Gesamtkosten:", ___, "Euro")
```

Aufgabe 5: Fehler finden: Temperatur umrechnen ●●○

In den USA misst man Temperaturen in Grad Fahrenheit. Man rechnet so um: Grad Fahrenheit = Grad Celsius · 1,8 + 32. Das Programm soll eine Temperatur in Grad Celsius einlesen und in Grad Fahrenheit umrechnen. Es enthält **drei Fehler**.

```
celsius = input("Temperatur in Grad Celsius: ")
fahrenheit = celsius * 1.8 + 32
print("Das sind", Fahrenheit, "Grad Fahrenheit.)
```

```
  File "<Zelle 5>", line 3
    print("Das sind", Fahrenheit, "Grad Fahrenheit.)
                                  ^
SyntaxError: unterminated string literal (detected at line 3)
```

1. Finde die drei Fehler. Notiere jeweils die Zeile und was falsch ist.
2. Schreibe das Programm richtig auf und teste es mit `20` (richtig ist 68 Grad Fahrenheit).

Deine Antwort:

…

Aufgabe 6: Taschengeld ●●○

Schreibe ein Programm, das das wöchentliche Taschengeld (Kommazahl) und eine Anzahl von Wochen (ganze Zahl) einliest. Es soll ausgeben, wie viel Geld in dieser Zeit zusammenkommt. Danach soll es den Preis eines Wunschartikels einlesen und ausgeben, wie viel Geld nach dem Kauf noch übrig ist. Runde die Beträge auf Cent.

So könnte es aussehen:

Ausgabe:

```
Taschengeld pro Woche in Euro: «7.5»
Anzahl der Wochen: «6»
In 6 Wochen bekommst du 45.0 Euro.
Preis des Wunschartikels in Euro: «29.99»
Nach dem Kauf bleiben 15.01 Euro übrig.
```

Aufgabe 7: Pfannkuchen für alle ●●○

Für **eine** Person braucht man für Pfannkuchen 62,5 g Mehl, 0,5 Eier und 125 ml Milch.

1. Schreibe ein Programm, das die Anzahl der Personen einliest und ausgibt, wie viel man von jeder Zutat braucht.
2. Begründe, warum du die Anzahl der Personen mit `int` und nicht mit `float` umwandelst.

Deine Antwort:

…

Aufgabe 8: Busfahrt oder Monatskarte? ●●●

Eine Einzelfahrt mit dem Bus kostet 2,10 Euro. Eine Monatskarte kostet 58 Euro.

1. Schreibe ein Programm, das einliest, wie viele Busfahrten man an einem Schultag macht und an wie vielen Schultagen man im Monat fährt. Es soll ausgeben, was die Einzelfahrten im Monat zusammen kosten. Danach soll es ausgeben, wie viel man mit der Monatskarte spart, also die Kosten der Einzelfahrten minus 58 Euro.
2. Teste dein Programm mit `2` Fahrten an `20` Tagen und mit `2` Fahrten an `10` Tagen. Erkläre, was ein **negatives** Ergebnis für die Ersparnis bedeutet.

Deine Antwort:

…

DS 2 – Operatoren: rechnen, vergleichen, verknüpfen

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- die Rechenoperatoren `//`, `%` und `**` benutzen und erklären, was sie berechnen,
- mit `//` und `%` Minuten in Stunden und Minuten umrechnen und prüfen, ob eine Zahl gerade ist,
- zwei Werte mit `==`, `!=`, `<`, `>`, `<=` und `>=` vergleichen,
- Wahrheitswerte mit `and`, `or` und `not` verknüpfen und das Ergebnis vorhersagen,
- die Zuweisung `=` und den Vergleich `==` sicher unterscheiden.

🗣️ Sprachhilfe

- *„Da kommt True raus."* → „Für `zahl = 7` liefert der Vergleich `zahl > 4` den Wahrheitswert `True`, weil 7 größer als 4 ist."
- *„Ich bekomme raus, ob die Zahl größer als 4 ist."* → „Die Anweisung prüft, ob eine Zahl größer als 4 ist, und gibt das Ergebnis als Wahrheitswert aus."
- *„Prozent rechnet den Rest."* → „Der Modulo-Operator `%` liefert den Rest der Ganzzahldivision."

Worum es geht

Ein Film dauert 135 Minuten. Wie viele Stunden und Minuten sind das? Ist eine Zahl gerade? Darf eine 15-Jährige ohne Ausweis ins Kino? Für solche Fragen muss ein Programm **rechnen**, **vergleichen** und Aussagen **verknüpfen**. Die Zeichen dafür heißen **Operatoren**.

> 📌 **Definition:** Ein **Operator** ist ein Zeichen oder Wort, das mit einem oder mehreren Werten (den **Operanden**) einen neuen Wert berechnet. In `7 + 3` ist `+` der Operator, `7` und `3` sind die Operanden. Eine solche Verbindung heißt **Ausdruck**.

1 Arithmetische Operatoren

Operator	Name	Beispiel	Ergebnis
`+`	Addition	`7 + 3`	`10`
`-`	Subtraktion	`7 - 3`	`4`
`*`	Multiplikation	`7 * 3`	`21`
`/`	Division	`7 / 3`	`2.3333333333333335`
`//`	Ganzzahldivision	`7 // 3`	`2`
`%`	Modulo (Rest)	`7 % 3`	`1`
`**`	Potenz	`7 ** 3`	`343`

Ganzzahldivision `//` und Rest `%`

17 Kekse werden gerecht an 5 Kinder verteilt. Jedes Kind bekommt 3 Kekse, 2 bleiben übrig: 17 : 5 = 3 Rest 2.

```
1 print(17 // 5)
2 print(17 % 5)
3 print(17 / 5)
```

```
3
2
3.4
```

- **Zeile 1:** `//` gibt an, wie oft die 5 **ganz** in die 17 passt: 3-mal.
- **Zeile 2:** `%` (**Modulo**) liefert den **Rest**: 17 − 3 · 5 = 2.
- **Zeile 3:** Die normale Division `/` rechnet mit Nachkommastellen.

Probe: 3 · 5 + 2 = 17. Das Zeichen `%` hat in Python **nichts mit Prozent** zu tun.

Anwendung: Minuten in Stunden und Minuten

```
1 minuten_gesamt = int(input("Wie viele Minuten dauert der Film? "))
2 stunden = minuten_gesamt // 60
3 minuten = minuten_gesamt % 60
4 print(minuten_gesamt, "Minuten sind", stunden, "Stunden und",
5       minuten, "Minuten.")
```

```
Wie viele Minuten dauert der Film? 135
135 Minuten sind 2 Stunden und 15 Minuten.
```

- **Zeile 2:** `135 // 60` ergibt `2`, die 60 passt zweimal ganz in die 135.
- **Zeile 3:** `135 % 60` ergibt `15`, denn 135 − 2 · 60 = 15.
- **Zeilen 4–5:** Innerhalb der Klammern von `print` darf man in der nächsten Zeile weiterschreiben.

🧪 **Probiere es aus:** Starte mit `60`, `59` und `600`. Was passiert bei `0`?

Anwendung: Ist eine Zahl gerade?

Eine Zahl ist **gerade**, wenn beim Teilen durch 2 der Rest `0` bleibt.

```
print(14 % 2, 15 % 2, 2024 % 4, 2026 % 4)
```

```
0 1 0 2
```

> 📌 **Merksatz:** Eine Zahl `a` ist genau dann durch `b` teilbar, wenn `a % b` den Wert `0` ergibt. Eine Zahl ist gerade, wenn `zahl % 2` gleich `0` ist.

Potenzieren mit `**`

`2 ** 3` bedeutet $2^3 = 2 \cdot 2 \cdot 2 = 8$. Die erste Zahl ist die Basis, die zweite der Exponent.

```
print(2 ** 3, 10 ** 6, 5 ** 2)
```

```
8 1000000 25
```

⚠️ Stolperstein: `^` ist nicht „hoch“

In Tabellenkalkulationen schreibt man „hoch“ mit `^`. In Python gibt das **keine** Fehlermeldung, aber ein falsches Ergebnis. 🔮 **Vorhersage:** Was kommt heraus?

```
print(2 ^ 3)
```

```
1
```

Für Potenzen brauchst du in Python immer `**`.

> 📌 **Merksatz:** Arithmetische Operatoren: `+`, `-`, `*`, `/` (Ergebnis immer `float`), `//` (Ganzzahldivision), `%` (Rest) und `**` (Potenz).

Was wird zuerst gerechnet?

Klammern, dann `**`, dann Punktrechnung (`*`, `/`, `//`, `%`), zuletzt `+` und `-`.

```
print(2 + 3 ** 2)
print((2 + 3) ** 2)
print(10 - 7 % 4)
```

```
11
25
7
```

- **Zeile 1:** Erst $3^2 = 9$, dann $2 + 9 = 11$.
- **Zeile 2:** Erst die Klammer $2 + 3 = 5$, dann $5^2 = 25$.
- **Zeile 3:** Erst `7 % 4` = 3, dann $10 - 3 = 7$.

Wenn du unsicher bist, setze Klammern. Sie schaden nie.

🔮 Übung 1 – Rechenausdrücke

Bestimme jedes Ergebnis **zuerst im Kopf**, dann überprüfe in der Codezelle.

Ausdruck	`23 // 4`	`23 % 4`	`23 / 4`	`2 ** 5`	`10 % 2`	`7 % 10`	`3 + 2 ** 3`	`(3 + 2) ** 2`
Ergebnis								

```
print(23 // 4)
# TODO: die übrigen sieben Ausdrücke prüfen
```

```
5
```

2 Vergleichsoperatoren: wahr oder falsch?

Viele Fragen beantwortet man mit Ja oder Nein. Python antwortet mit einem **Wahrheitswert**: `True` oder `False` (Datentyp `bool`).

Operator	Bedeutung	Beispiel	Ergebnis
`==`	ist gleich	`7 == 3`	`False`
`!=`	ist ungleich	`7 != 3`	`True`
`<`	ist kleiner als	`7 < 3`	`False`
`>`	ist größer als	`7 > 3`	`True`
`<=`	ist kleiner oder gleich	`7 <= 7`	`True`
`>=`	ist größer oder gleich	`3 >= 7`	`False`

`!=` liest man „ungleich“. Bei `<=` und `>=` steht das Gleichheitszeichen immer **rechts**.

```
a = 7
b = 3
print(a > b)
print(a == b)
print(a != b)
print(a <= 7)
```

```
True
False
True
True
```

Ergebnisse von Vergleichen speichern

Ein Wahrheitswert ist ein ganz normaler Wert und kann in einer Variablen gespeichert werden.

```
alter = int(input("Wie alt bist du? "))
ist_volljaehrig = alter >= 18
print("Volljährig:", ist_volljaehrig)
print(type(ist_volljaehrig))
```

```
Wie alt bist du? 15
Volljährig: False
<class 'bool'>
```

- **Zeile 2:** Zuerst wird die rechte Seite ausgewertet: Ist 15 ≥ 18? Nein, also `False`. Dann wird der Wert gespeichert.

Für solche Variablen wählt man Namen wie eine Ja-Nein-Frage: `ist_volljaehrig`, `hat_ausweis`, `ist_gerade`.

Texte vergleichen

Es kommt auf jedes Zeichen an, auch auf Groß- und Kleinschreibung und Leerzeichen.

```
print("ja" == "ja", "Ja" == "ja", "ja " == "ja")
```

```
True False False
```

⚠️ Stolperstein: Eingabe nicht umgewandelt

🔮 **Vorhersage:** Gib `7` ein. Was gibt das Programm aus?

```
zahl = input("Gib eine Zahl ein: ")
print(zahl == 7)
```

```
Gib eine Zahl ein: 15
False
```

Keine Fehlermeldung, aber immer `False`: Der Text `"7"` ist nicht die Zahl `7`. Abhilfe: `zahl = int(input(...))`.

⚠️ Zuweisung `=` oder Vergleich `==`?

- `x = 5` ist eine **Zuweisung**: „`x` bekommt den Wert 5.“ Es wird nichts geprüft.
- `x == 5` ist ein **Vergleich**: „Ist `x` gleich 5?“ Es wird nichts verändert.

Führe die Zelle aus und lies die Fehlermeldung.

```
print(7 = 7)
```

```
  File "<Zelle 12>", line 1
    print(7 = 7)
          ^^^
SyntaxError: expression cannot contain assignment, perhaps you meant "=="?
```

`SyntaxError: expression cannot contain assignment, perhaps you meant "=="?` heißt: „Ein Ausdruck darf keine Zuweisung enthalten. Meintest du `==`?“

> 📌 **Merksatz:** Vergleichsoperatoren liefern immer einen Wahrheitswert (`True` oder `False`): `==`, `!=`, `<`, `>`, `<=`, `>=`. Verwechsle nicht `=` (Zuweisung) und `==` (Vergleich)!

3 Logische Operatoren: `and`, `or` und `not`

Ein Film ist ab 12, und an der Kasse wird ein Ausweis verlangt. Du darfst nur hinein, wenn **beide** Bedingungen erfüllt sind.

- `and` („und“): nur `True`, wenn **beide** Seiten `True` sind.
- `or` („oder“): `True`, wenn **mindestens eine** Seite `True` ist, also **auch, wenn beide** wahr sind. Anders als „entweder … oder“ im Alltag!
- `not` („nicht“): steht **vor** einem Wahrheitswert und kehrt ihn um.

```
1 print(True and False)
2 print(True or False)
3 print(not True)
4 print(not 7 > 3)
```

```
False
True
False
False
```

- **Zeile 4:** Zuerst der Vergleich: `7 > 3` ist `True`. Dann kehrt `not` das um.

A	B	A and B	A or B	not A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

> 📌 **Merksatz:** `and`: beide Bedingungen müssen wahr sein. `or`: mindestens eine muss wahr sein. `not`: kehrt einen Wahrheitswert um.

Vergleiche verknüpfen

Ist eine Person jugendlich, also mindestens 13 und jünger als 18?

```
1 alter = int(input("Wie alt bist du? "))
2 ist_jugendlich = alter >= 13 and alter < 18
3 print("Jugendlich:", ist_jugendlich)
```

```
Wie alt bist du? 15
Jugendlich: True
```

Python wertet Zeile 2 so aus:

1. `15 >= 13` ergibt `True`.
2. `15 < 18` ergibt `True`.
3. `True and True` ergibt `True`.

Rechnungen und Vergleiche kommen **vor** den logischen Operatoren. Unter ihnen gilt: erst `not`, dann `and`, zuletzt `or`.

⚠️ Stolperstein: halber Vergleich

Im Deutschen sagt man „älter als 12 und jünger als 18“. Python braucht auf **beiden** Seiten von `and` einen vollständigen Vergleich.

```
alter = 15
print(alter > 12 and < 18)
```

```
  File "<Zelle 15>", line 2
    print(alter > 12 and < 18)
                         ^
SyntaxError: invalid syntax
```

Richtig ist `alter > 12 and alter < 18`.

🔮 Übung 2 – Wahr oder falsch?

Es gilt `x = 5` und `y = 10`. Bestimme **zuerst im Kopf**, ob jeder Ausdruck `True` oder `False` ergibt.

Ausdruck	Ergebnis
`x < y`	
`x == 5 and y == 5`	
`x == 5 or y == 5`	
`not x > y`	
`x % 2 == 0`	
`x + y >= 15 and not y == 10`	

```
x = 5
y = 10
print(x < y)
# TODO: die übrigen fünf Ausdrücke prüfen
```

```
True
```

4 Klammern entscheiden: Darf ich zocken?

- `A` = Die Hausaufgaben sind fertig.
- `B` = Das Zimmer ist aufgeräumt.
- `C` = Es ist Wochenende.

Zwei Eltern formulieren ihre Regel unterschiedlich:

- **Regel 1:** `(A and B) or C`: Hausaufgaben **und** Zimmer, **oder** Wochenende. Am Wochenende reicht `C` allein.
- **Regel 2:** `A and (B or C)`: Hausaufgaben **immer**, dazu Zimmer **oder** Wochenende.

🔮 **Vorhersage:** Hausaufgaben nicht fertig, Zimmer nicht aufgeräumt, aber Wochenende. Wer darf zocken?

```
A = False   # Hausaufgaben sind fertig
B = False   # Zimmer ist aufgeräumt
C = True    # Es ist Wochenende

ausdruck_1 = (A and B) or C
ausdruck_2 = A and (B or C)

print("(A and B) or C =", ausdruck_1)
print("A and (B or C) =", ausdruck_2)
```

```
(A and B) or C = True
A and (B or C) = False
```

- **Zeile 5:** Erst die Klammer: `False and False` ergibt `False`. Dann `False or True`, also `True`.
- **Zeile 6:** Erst die Klammer: `False or True` ergibt `True`. Dann `False and True`, also `False`.

Die Tabelle zeigt alle acht Möglichkeiten:

A	B	C	A and B	B or C	(A and B) or C	A and (B or C)
False	False	False	False	False	False	False
False	False	True	False	True	True	False
False	True	False	False	True	False	False
False	True	True	False	True	True	False
True	False	False	False	False	False	False
True	False	True	False	True	True	True
True	True	False	True	True	True	True
True	True	True	True	True	True	True

In vier Zeilen unterscheiden sich die beiden letzten Spalten: Die Ausdrücke sind **nicht äquivalent**.

> 📌 **Merksatz:** Klammern bestimmen, welcher Teilausdruck zuerst ausgewertet wird. Setze immer Klammern, wenn du `and` und `or` mischst.

Bedingungen verneinen: die Regeln von De Morgan

Ist eine Person **nicht** jugendlich?

```
1 alter = 20
2 print(not (alter >= 13 and alter < 18))
3 print(alter < 13 or alter >= 18)
```

```
True
True
```

Beide Zeilen liefern für **jedes** Alter dasselbe. Probiere es mit `alter = 15` und `alter = 9`. Beim Umformen wurde jeder Vergleich verneint (`>=` wird `<`, `<` wird `>=`), und aus `and` wurde `or`.

> 📌 **Regeln von De Morgan:** `not (A and B)` ist dasselbe wie `not A or not B`. `not (A or B)` ist dasselbe wie `not A and not B`. Verneint man eine verknüpfte Bedingung, wird **jede** Teilbedingung verneint, und aus `and` wird `or` (und umgekehrt).

Vergleich	verneint
`a == b`	`a != b`
`a < b`	`a >= b`
`a > b`	`a <= b`

> ⚠️ **Stolperstein:** `alter < 13 and alter >= 18` ist **immer** `False`, denn niemand ist gleichzeitig jünger als 13 und mindestens 18. Beim Verneinen wird aus `and` ein `or`!

✏️ Übung 3 – Bedingungen verneinen

Schreibe zu jeder Bedingung eine Bedingung **ohne** `not`, die das Gegenteil prüft. Teste in der Codezelle: Beide Werte einer Zeile müssen gleich sein.

1. `zahl >= 1 and zahl <= 6` (eine Würfelzahl)
2. `note == 1 or note == 2`

3. `not (temperatur > 30 or regen)` (`regen` ist ein Wahrheitswert, „nicht `regen`“ darf als `not regen` stehen bleiben)

Lückencode

```
zahl = 7
print(not (zahl >= 1 and zahl <= 6), ___)
note = 2
print(not (note == 1 or note == 2), ___)
temperatur = 25
regen = False
print(not (temperatur > 30 or regen), ___)
```

📝 Übungen: Schritt für Schritt selbstständig

Das **Musterbeispiel** ist ganz vorgemacht. Bei Übung 4 füllst du Lücken `___`, bei Übung 5 steht nur noch ein Gerüst aus Kommentaren, die Zusatzaufgabe schreibst du allein.

Schritt	Frage
1	**Eingabe:** Was wird eingelesen? Als welcher Datentyp?
2	**Verarbeitung:** Welcher Operator beantwortet die Frage?
3	**Ausgabe:** Was soll auf dem Bildschirm stehen?

🧩 Musterbeispiel (vorgemacht) – Durch 3 teilbar?

Aufgabe: Das Programm liest eine Ganzzahl ein und gibt als Wahrheitswert aus, ob sie durch 3 teilbar ist.

1. **Eingabe:** eine ganze Zahl, also `int(input(...))`.
2. **Verarbeitung:** „durch 3 teilbar“ heißt „Rest beim Teilen durch 3 ist 0“: `zahl % 3 == 0`. Python rechnet erst `%`, dann vergleicht es mit `==`.
3. **Ausgabe:** Text und Wahrheitswert, mit Komma getrennt.

```
# Schritt 1: Eingabe
zahl = int(input("Ganzzahl eingeben: "))

# Schritt 2: Verarbeitung
ist_teilbar = zahl % 3 == 0

# Schritt 3: Ausgabe
print("Durch 3 teilbar:", ist_teilbar)
```

```
Ganzzahl eingeben: 15
Durch 3 teilbar: True
```

Teste mit 12 (`True`), 13 (`False`) und 0 (`True`).

📝 Übung 4 – Gerade oder ungerade? (Lücken füllen)

Das Programm liest eine Ganzzahl ein und gibt als Wahrheitswert aus, ob sie gerade ist. Ersetze jedes `___`.
Kontrolle: 14 → `Ist gerade: True`, 7 → `Ist gerade: False`.

Lückencode

```
zahl = ___(input("Ganzzahl eingeben: "))
ist_gerade = zahl ___ 2 ___ 0
print("Ist gerade:", ist_gerade)
```

📝 Übung 5 – Schaltjahr-Check (mit Gerüst)

Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, aber nicht durch 100, **außer** es ist auch durch 400 teilbar. Du brauchst nur Operatoren. Ersetze die Kommentare durch Programmzeilen.

Kontrolle: 2024 → `True`, 2023 → `False`, 1900 → `False`, 2000 → `True`.

```
# Jahreszahl als Ganzzahl einlesen
# A = durch 4 teilbar?
# B = durch 100 teilbar?
# C = durch 400 teilbar?
# schaltjahr = durch 4 und nicht durch 100, oder durch 400 (Klammern setzen!)
# alle Zwischenergebnisse und das Ergebnis ausgeben
```

🚀 Zusatzaufgabe – Sekunden umrechnen (selbstständig)

Lies eine Anzahl von Sekunden ein und gib aus, wie viele volle Stunden, Minuten und Sekunden das sind. Kontrolle: 3725 Sekunden sind 1 Stunden, 2 Minuten und 5 Sekunden. Tipp: Eine Stunde hat 3600 Sekunden.

📌 Das Wichtigste in Kürze

- `//` Ganzzahldivision, `%` Rest, `**` Potenz. `^` ist **nicht** „hoch“.
- `a % b == 0` heißt: `a` ist durch `b` teilbar.
- Vergleiche `==`, `!=`, `<`, `>`, `<=`, `>=` liefern `True` oder `False`. `=` weist zu, `==` vergleicht.
- `and`: beide wahr. `or`: mindestens eins wahr. `not`: umkehren. Beim Mischen Klammern setzen.
- De Morgan: Beim Verneinen jede Teilbedingung verneinen und `and` ↔ `or` tauschen.

Zum Weiterüben: `DS02_Weitere_Uebungen.ipynb`.

DS 2 – Weitere Übungen

Operatoren: rechnen, vergleichen, verknüpfen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Teilen mit Rest ●○○

Bestimme das Ergebnis jedes Ausdrucks zuerst im Kopf und trage es in die Tabelle ein. Überprüfe dann mit `print(...)` am Computer.

Ausdruck	Ergebnis	Ausdruck	Ergebnis
`29 // 5`		`2 ** 4`	
`29 % 5`		`17 % 2`	
`40 % 8`		`10 - 4 // 2`	
`3 % 8`		`(10 - 4) // 2`	

Aufgabe 2: True oder False? ●○○

Es gilt `a = 4` und `b = 9`. Bestimme für jeden Vergleich, ob er `True` oder `False` ergibt. Überprüfe deine Antworten danach am Computer.

Ausdruck	Ergebnis	Ausdruck	Ergebnis
`a < b`		`a * 2 > b`	
`b != 9`		`b % a == 1`	
`a >= 5`		`a + 5 == b`	

Aufgabe 3: Wahrheitstabelle ausfüllen ●○○

Fülle die Tabelle aus. Bestimme zuerst die Spalte `not B`. Mit ihrer Hilfe kannst du die beiden letzten Spalten ausfüllen.

A	B	not B	A and not B	A or not B
`False`	`False`			
`False`	`True`			
`True`	`False`			
`True`	`True`			

Aufgabe 4: Eier verpacken ●○○

Eier werden in Kartons zu je 6 Stück verpackt. Das Programm soll eine Anzahl von Eiern einlesen und ausgeben, wie viele Kartons voll werden und wie viele Eier übrig bleiben. Ergänze die beiden Lücken `___` mit `//` und `%`. Teste mit `20`, `18` und `5`.

Lückencode

```
eier = int(input("Wie viele Eier? "))
kartons = ___    # volle Kartons
rest = ___       # übrige Eier
print(eier, "Eier ergeben", kartons, "volle Kartons und",
      rest, "übrige Eier.")
```

Aufgabe 5: Bedingungen aufschreiben ●●○

Schreibe zu jeder Aussage einen Python-Ausdruck, der `True` ergibt, wenn die Aussage stimmt, und sonst `False`. Teste jeden Ausdruck am Computer mit zwei passenden Werten.

1. Die Zahl `x` ist durch 3 teilbar.
2. Das Alter `alter` liegt zwischen 10 und 15 (beide Grenzen gehören dazu).
3. Die Punktzahl `punkte` ist kleiner als 0 oder größer als 100.
4. Die Zahl `zahl` ist ungerade.
5. Die Anzahl `stunden` ist nicht 0.

Deine Antwort:

…

Aufgabe 6: Fehler finden: Teilbar durch 5? ●●○

Das Programm soll eine ganze Zahl einlesen und einen Wahrheitswert ausgeben, der angibt, ob die Zahl durch 5 teilbar ist. Es enthält **drei Fehler**.

```
zahl = input("Zahl: ")
durch_fuenf = zahl % 5 = 0
print("Durch 5 teilbar:" durch_fuenf)
```

```
  File "<Zelle 2>", line 2
    durch_fuenf = zahl % 5 = 0
                  ^^^^^^^^
SyntaxError: cannot assign to expression
```

1. Finde die drei Fehler. Notiere jeweils die Zeile und was falsch ist.
2. Schreibe das Programm richtig auf und teste es mit `35` und `42`.

Deine Antwort:

…

Aufgabe 7: Ausleihe in der Bücherei ●●○

In der Stadtbücherei gilt: Man darf ein Buch ausleihen, wenn man einen Ausweis hat **und** entweder keine offenen Gebühren hat **oder** jünger als 18 ist. Im Programm sieht das so aus:

```
hat_ausweis = True
gebuehren = 0
alter = 30
darf_ausleihen = hat_ausweis and (gebuehren == 0 or alter < 18)
print("Darf ausleihen:", darf_ausleihen)
```

```
Darf ausleihen: True
```

1. Bestimme `darf_ausleihen` für die vier Fälle in der Tabelle. Prüfe am Computer, indem du die Werte in den Zeilen 1 bis 3 änderst.
2. Jemand lässt die Klammern weg: `hat_ausweis and gebuehren == 0 or alter < 18`. Bestimme das Ergebnis für den vierten Fall und erkläre den Unterschied.

hat_ausweis	gebuehren	alter	darf_ausleihen
`True`	`0`	`30`	
`True`	`2`	`15`	
`True`	`2`	`40`	
`False`	`0`	`12`	

Deine Antwort:

...

Aufgabe 8: Ziffern zerlegen ●●●

Schreibe ein Programm, das eine dreistellige Zahl einliest und ihre Ziffern einzeln ausgibt: die Hunderterziffer, die Zehnerziffer und die Einerziffer. Zum Schluss soll es die **Quersumme** ausgeben, also die Summe der drei Ziffern. Verwende dafür `//` und `%`.

So könnte es aussehen:

Ausgabe:

```
Dreistellige Zahl: «472»
Hunderter: 4
Zehner: 7
Einer: 2
Quersumme: 13
```

> **Tipp:** Die Einerziffer ist der Rest beim Teilen durch 10. Die Hunderterziffer sagt, wie oft die 100 ganz in die Zahl passt. Für die Zehnerziffer brauchst du zwei Schritte: `472 // 10` ergibt `47`. Welche Rechnung liefert daraus die `7`?

DS 3 – Verzweigungen: Programme treffen Entscheidungen

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- erklären, woraus eine Verzweigung besteht, und die Begriffe *Bedingung*, *dann-Teil* und *sonst-Teil* verwenden,
- ein-, zwei- und mehrseitige Verzweigungen mit `if`, `else` und `elif` schreiben,
- mit der Einrückung festlegen, welche Anweisungen zu einem Block gehören, und Einrückungsfehler beheben,
- den Ablauf eines Programms mit Verzweigung in einem Ablaufprotokoll darstellen,
- Bedingungen mit `and` und `or` formulieren und Verzweigungen verschachteln.

🗣️ Sprachhilfe

- *„Ich bekomme raus, ob die Zahl größer als 4 ist."*
- → „Die Anweisung gibt ‚Die Zahl ist größer als 4.' aus, wenn die Zahl größer als 4 ist, und sonst ‚Die Zahl ist nicht größer als 4.'"
- → „Die Alternativanweisung prüft, ob eine Zahl größer als 4 ist. Ist dies der Fall, wird der dann-Teil ausgeführt, sonst der sonst-Teil."

Nützliche Wendungen: *Die Verzweigung besteht aus … · Es wird geprüft, ob … · Ist die Bedingung erfüllt, wird …*

Worum es geht

Wenn es regnet, *dann* nehme ich einen Schirm mit. *Wenn* der Bus kommt, steige ich ein, *sonst* warte ich. Bisher hat Python jede Zeile ausgeführt, von oben nach unten. Jetzt lernst du, wie ein Programm abhängig von einer **Bedingung** nur bestimmte Anweisungen ausführt. Aus der Blockprogrammierung kennst du „falls … dann" und „falls … dann … sonst".

1 Die einseitige Verzweigung: `if`

```
1 temperatur = 30
2 if temperatur > 25:
3     print("Es ist heiß. Trink genug Wasser!")
4 print("Schönen Tag!")
```

```
Es ist heiß. Trink genug Wasser!
Schönen Tag!
```

- **Zeile 2:** `if` heißt „wenn" oder „falls". Python prüft die Bedingung `temperatur > 25`: 30 > 25 ist `True`. Am Ende steht ein **Doppelpunkt**.
- **Zeile 3:** Diese Zeile ist **eingerückt** (vier Leerzeichen). Sie gehört zur `if`-Anweisung und wird nur ausgeführt, weil die Bedingung wahr ist.
- **Zeile 4:** Nicht eingerückt, wird also in jedem Fall ausgeführt.

🔮 **Vorhersage:** Was gibt das Programm bei `temperatur = 18` aus? Überlege zuerst, dann führe die Zelle aus.

```
1 temperatur = 18
2 if temperatur > 25:
3     print("Es ist heiß. Trink genug Wasser!")
4 print("Schönen Tag!")
```

```
Schönen Tag!
```

Die Bedingung ist `False`, Python **überspringt** Zeile 3. Anweisungen wie `if`, die den Ablauf steuern, heißen **Kontrollstrukturen**.

> 📌 **Definition:** Eine **Bedingung** ist ein Ausdruck, dessen Wert `True` oder `False` ist, z. B. `temperatur > 25`. Eine **Verzweigung** (bedingte Anweisung) ist eine Kontrollstruktur, die abhängig von einer Bedingung festlegt, welche Anweisungen ausgeführt werden.

> 📌 **Merksatz: Einseitige Verzweigung:** `if Bedingung:` – der eingerückte Block wird nur ausgeführt, wenn die Bedingung `True` ist.

2 Einrückung: Was gehört zum Block?

Alle Zeilen direkt unter dem `if`, die gleich weit eingerückt sind, gehören zum Block. 🔮 **Vorhersage:** Die beiden Programme unterscheiden sich nur in Zeile 4. Was gibt jedes aus?

```
1 punkte = 3
2 if punkte >= 10:
3     print("Gewonnen!")
4     print("Du bekommst einen Preis.")
5 print("Spiel beendet.")
```

```
Spiel beendet.
```

```
1 punkte = 3
2 if punkte >= 10:
3     print("Gewonnen!")
4 print("Du bekommst einen Preis.")
5 print("Spiel beendet.")
```

```
Du bekommst einen Preis.
Spiel beendet.
```

Im ersten Programm werden Zeilen 3 und 4 übersprungen. Im zweiten gehört nur Zeile 3 zum Block, und das Programm verteilt einen Preis, obwohl niemand gewonnen hat.

> 📌 **Merksatz:** Die **Einrückung** (4 Leerzeichen) bestimmt, welche Anweisungen zu einem Block gehören. Die Zeile mit der Bedingung endet mit einem **Doppelpunkt** `:`.

⚠️ Stolpersteine: Doppelpunkt und Einrückung

Führe die drei Zellen aus und lies jeweils die letzte Zeile der Fehlermeldung.

```
1 zahl = 5
2 if zahl > 0
3     print("positiv")
```

```
  File "<Zelle 5>", line 2
    if zahl > 0
               ^
SyntaxError: expected ':'
```

```
1 zahl = 5
2 if zahl > 0:
3 print("Die Zahl ist positiv.")
```

```
  File "<Zelle 6>", line 3
    print("Die Zahl ist positiv.")
    ^^^^^
IndentationError: expected an indented block after 'if' statement on line 2
```

```
zahl = 5
    print(zahl)
```

```
  File "<Zelle 7>", line 2
    print(zahl)
IndentationError: unexpected indent
```

1. `SyntaxError: expected ':'`: Der Doppelpunkt fehlt, `^` zeigt auf die Stelle.
2. `IndentationError: expected an indented block after 'if' statement on line 2`: Nach dem `if` fehlt der eingerückte Block.
3. `IndentationError: unexpected indent`: Eine Zeile ist eingerückt, obwohl kein Block mit Doppelpunkt davor steht.

> 💡 **Tipp:** Nach einem Doppelpunkt rückt die Umgebung meist automatisch ein. Achte darauf, dass alle Zeilen eines Blocks **genau gleich weit** eingerückt sind.

3 Die zweiseitige Verzweigung: `if` … `else`

Soll das Programm in **beiden** Fällen etwas tun, gibt es `else` („sonst“).

```
1 zahl = int(input("Gib eine Zahl ein: "))
2 if zahl > 4:
3     print("Die Zahl ist größer als 4.")
4 else:
5     print("Die Zahl ist nicht größer als 4.")
6 print("Fertig.")
```

```
Gib eine Zahl ein: 15
Die Zahl ist größer als 4.
Fertig.
```

- **Zeile 3:** der **dann-Teil**, wird ausgeführt, wenn die Bedingung wahr ist.
- **Zeile 4:** `else:` steht genau unter dem `if`, hat **keine** eigene Bedingung, aber einen Doppelpunkt.
- **Zeile 5:** der **sonst-Teil**, nur wenn die Bedingung falsch ist.

Es wird **immer genau einer** der beiden Teile ausgeführt. Deshalb heißt die zweiseitige Verzweigung auch **Alternativanweisung**.

Das **Ablaufprotokoll** zeigt, welche Zeilen wirklich ausgeführt werden, links für die Eingabe 7, rechts für 2:

Schritt	Zeile	`zahl`		Schritt	Zeile	`zahl`
1	1	7		1	1	2
2	2	7		2	2	2
3	3	7		3	5	2
4	6	7		4	6	2

Die Zeile `else:` selbst tut nichts, deshalb taucht Zeile 4 nicht auf.

📌 **Merksatz: Zweiseitige Verzweigung:** `if … else:` – entweder der **dann-Teil** oder der **sonst-Teil** wird ausgeführt.

🔮 Übung 1 – Glatteis-Warnung

Lies das Programm. Beschreibe in zwei bis drei Sätzen mit Fachbegriffen, was es tut. Gib dann **ohne Ausführen** an, was es bei `-3`, `0` und `5` ausgibt. Überprüfe danach.

```
1 temperatur = int(input("Temperatur in Grad Celsius: "))
2 if temperatur <= 0:
3     print("Achtung, Glatteisgefahr!")
4 else:
5     print("Keine Glatteisgefahr.")
```

```
Temperatur in Grad Celsius: 23
Keine Glatteisgefahr.
```

Beschreibung und Ausgaben:

…

4 Die mehrseitige Verzweigung: `if` … `elif` … `else`

Bei mehr als zwei Möglichkeiten gibt es `elif`, kurz für *else if*, „sonst, wenn“.

```
1 note = int(input("Punktzahl (0-100) eingeben: "))
2 if note >= 90:
3     print("Note: sehr gut")
4 elif note >= 75:
5     print("Note: gut")
6 elif note >= 50:
7     print("Note: befriedigend")
8 else:
9     print("Note: nicht bestanden")
```

```
Punktzahl (0-100) eingeben: 78
Note: gut
```

- **Zeile 2:** Ist 80 ≥ 90? Nein.
- **Zeile 4:** Weil die erste Bedingung falsch war, wird die nächste geprüft: Ist 80 ≥ 75? Ja.
- **Zeile 5:** Der zugehörige Block wird ausgeführt.
- **Zeilen 6–9:** Werden übersprungen. Sobald eine Bedingung zutrifft, prüft Python die weiteren **gar nicht mehr**, obwohl auch `80 >= 50` wahr wäre.

Der `else`-Teil fängt alle übrigen Fälle auf. Er darf auch fehlen.

📌 **Merksatz: Mehrseitige Verzweigung:** `if … elif … else:` – es wird nur der Block der **ersten** zutreffenden Bedingung ausgeführt. Trifft keine zu, wird der `else`-Block ausgeführt.

⚠️ Stolperstein: die Reihenfolge

🔮 **Vorhersage:** Was gibt das Programm bei 95 Punkten aus?

```python
note = int(input("Punktzahl (0-100) eingeben: "))
if note >= 50:
    print("Note: befriedigend")
elif note >= 75:
    print("Note: gut")
elif note >= 90:
    print("Note: sehr gut")
```

```
Punktzahl (0-100) eingeben: 78
Note: befriedigend
```

95 Punkte, aber nur „befriedigend“: `95 >= 50` ist schon wahr, der Rest wird übersprungen. Regel: Bei `>=` die **größte** Grenze zuerst prüfen, bei `<` die **kleinste**.

5 Bedingungen mit `and` und `or`

Eine Bedingung darf alles sein, was `True` oder `False` ergibt, also auch Verknüpfungen aus DS 2.

```python
tag = input("Welcher Tag ist heute? ")
if tag == "Samstag" or tag == "Sonntag":
    print("Wochenende! Ausschlafen.")
else:
    print("Wecker stellen.")
```

```
Welcher Tag ist heute? Montag
Wecker stellen.
```

- **Zeile 1:** Hier wird nicht umgewandelt, denn es soll ein Text verglichen werden.
- **Zeile 2:** Wahr, wenn **mindestens einer** der beiden Vergleiche wahr ist.

⚠️ Stolperstein: abgekürztes `or`

🔮 **Vorhersage:** Was gibt das Programm am Montag aus?

```python
tag = input("Welcher Tag ist heute? ")
if tag == "Samstag" or "Sonntag":
    print("Wochenende! Ausschlafen.")
```

```
Welcher Tag ist heute? Montag
Wochenende! Ausschlafen.
```

Python liest `"Sonntag"` als eigene Bedingung, und ein nicht leerer Text zählt als wahr. Die Bedingung ist **immer** erfüllt. Schreibe auf **beiden** Seiten von `or` und `and` einen vollständigen Vergleich.

> ⚠️ **Stolperstein:** `if zahl = 0:` meldet `SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?` Gemeint ist `==`.

6 Verschachtelte Verzweigungen

Im Block einer Verzweigung darf wieder eine Verzweigung stehen. Nur wenn der Benutzername bekannt ist, wird nach dem Passwort gefragt.

```python
benutzer = input("Benutzername: ")
if benutzer == "alex":
    passwort = input("Passwort: ")
    if passwort == "geheim123":
        print("Willkommen, Alex!")
    else:
        print("Falsches Passwort.")
else:
    print("Unbekannter Benutzer.")
```

```
Benutzername: alex
Passwort: 1234
Falsches Passwort.
```

- **Zeile 2:** Die äußere Verzweigung prüft den Benutzernamen.
- **Zeilen 4–7:** die **innere** Verzweigung. Ihre Blöcke sind acht Leerzeichen eingerückt.
- **Zeile 8:** Das `else` steht genau unter dem äußeren `if` und gehört deshalb dazu.

Ablaufprotokoll für die Eingaben `alex` und `1234` (– heißt: die Variable gibt es noch nicht):

Schritt	Zeile	benutzer	passwort
1	1	`"alex"`	–
2	2	`"alex"`	–
3	3	`"alex"`	`"1234"`
4	4	`"alex"`	`"1234"`
5	7	`"alex"`	`"1234"`

📌 **Merksatz:** Verzweigungen können **verschachtelt** werden. Die Einrückung zeigt, welcher Block zu welcher Bedingung gehört. Ein `else` gehört zu dem `if`, unter dem es genau steht.

📝 Übungen: Schritt für Schritt selbstständig

Das **Musterbeispiel** ist ganz vorgemacht. Bei Übung 2 füllst du Lücken `___`, bei Übung 3 steht nur noch ein Gerüst aus Kommentaren, die Zusatzaufgabe schreibst du allein.

Schritt	Frage
1	**Fälle:** Wie viele Möglichkeiten gibt es? Zwei → `if … else`, mehr → `elif`.
2	**Bedingungen:** Welche Bedingung trennt die Fälle? In welcher Reihenfolge prüfe ich?
3	**Testen:** Welche Eingaben prüfen jeden Fall und jeden **Grenzwert**?

🧩 Musterbeispiel (vorgemacht) – Schwimmbad

Aufgabe: Das Programm fragt nach dem Alter. Unter 18 kostet der Eintritt 3 Euro, sonst 5 Euro.

1. **Fälle:** zwei, also `if … else`.
2. **Bedingung:** `alter < 18`. Ist sie wahr, kostet es 3 Euro.
3. **Testen:** 12 (jugendlich), 30 (erwachsen) und die Grenzwerte 17 und 18.

```python
alter = int(input("Wie alt bist du? "))
if alter < 18:
    print("Eintritt: 3 Euro")
else:
    print("Eintritt: 5 Euro")
```

```
Wie alt bist du? 15
Eintritt: 3 Euro
```

Eingabe	`alter < 18`	Ausgabe
`12`	`True`	`Eintritt: 3 Euro`
`17`	`True`	`Eintritt: 3 Euro`
`18`	`False`	`Eintritt: 5 Euro`
`30`	`False`	`Eintritt: 5 Euro`

📝 Übung 2 – Größte von zwei Zahlen (Lücken füllen)

Das Programm liest zwei Zahlen ein und gibt aus, welche größer ist oder ob beide gleich sind. Das sind drei Fälle. Ersetze jedes `___`.

Kontrolle: 7 und 3 → erste größer, 3 und 7 → zweite größer, 5 und 5 → gleich.

Lückencode

```
a = float(input("Erste Zahl: "))
b = float(input("Zweite Zahl: "))
if a ___ b:
    print("Die erste Zahl ist größer.")
___ b > a:
    print("Die zweite Zahl ist größer.")
___:
    print("Beide Zahlen sind gleich.")
```

📝 Übung 3 – Altersklassen (mit Gerüst)

Das Programm liest ein Alter ein und ordnet es ein:

- unter 6: „Kleinkind“
- 6 bis unter 13: „Kind“
- 13 bis unter 18: „Jugendlicher“
- ab 18: „Erwachsener“

Ersetze die Kommentare durch Programmzeilen. Teste besonders die **Grenzwerte** 5, 6, 12, 13, 17 und 18.

```
# Alter als Ganzzahl einlesen
# if: jünger als 6 → Kleinkind
# elif: jünger als 13 → Kind
# elif: jünger als 18 → Jugendlicher
# else → Erwachsener
```

🚀 Zusatzaufgabe – Achterbahn-Check (selbstständig)

Regeln für eine Achterbahn:

- ab 140 cm: Man darf mitfahren.
- 120 cm bis unter 140 cm: nur in Begleitung eines Erwachsenen.
- unter 120 cm: nicht mitfahren.

Frage die Körpergröße (in cm) und ob eine erwachsene Begleitung dabei ist (`ja`/`nein`). Gib die passende Meldung aus. Nutze eine **verschachtelte** Verzweigung.

Kontrolle: 150/nein → mitfahren, 130/ja → in Begleitung, 130/nein → nicht, 110/ja → nicht.

📌 Das Wichtigste in Kürze

- `if Bedingung:` → Block nur bei `True`. Doppelpunkt nicht vergessen!
- Die **Einrückung** (4 Leerzeichen) bestimmt, was zum Block gehört.
- `if … else:` → genau einer der beiden Teile wird ausgeführt.
- `if … elif … else:` → nur die **erste** zutreffende Bedingung zählt. Reihenfolge beachten!

- Bei `or` und `and` auf beiden Seiten einen vollständigen Vergleich schreiben.
- Immer die **Grenzwerte** testen.

Zum Weiterüben: `DS03_Weitere_Uebungen.ipynb`.

DS 3 – Weitere Übungen

Verzweigungen: Programme treffen Entscheidungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Bestanden? ●○○

Lies das Programm, **ohne** es auszuführen. Gib an, was es bei den Eingaben `70`, `50` und `49` ausgibt. Überprüfe danach am Computer.

```
punkte = int(input("Punkte: "))
if punkte >= 50:
    print("bestanden")
else:
    print("nicht bestanden")
print("Ende")
```

```
Punkte: 65
bestanden
Ende
```

Deine Antwort:

…

Aufgabe 2: Einrückung entscheidet ●○○

Die beiden Programme unterscheiden sich nur in der Einrückung der letzten Zeile. Gib für beide an, was sie ausgeben. Erkläre den Unterschied in einem Satz.

Programm A:

```
x = 3
if x > 5:
    print("groß")
    print("fertig")
```

Programm B:

```
x = 3
if x > 5:
    print("groß")
print("fertig")
```

```
fertig
```

Deine Antwort:

…

Aufgabe 3: Gerade oder ungerade? ●○○

Das Programm soll ausgeben, ob eine eingegebene ganze Zahl gerade oder ungerade ist. Ergänze die beiden Lücken `___` und teste dein Programm mit `8`, `7` und `0`.

Lückencode

```
zahl = int(input("Zahl: "))
if ___:
    print(zahl, "ist gerade.")
___
    print(zahl, "ist ungerade.")
```

Aufgabe 4: Fieber messen ●○○

Das Programm ordnet eine gemessene Körpertemperatur ein. Teste es mit den Eingaben aus der Tabelle. Überlege die Ausgabe zuerst und trage sie ein. Prüfe danach am Computer.

```
temp = float(input("Temperatur in Grad Celsius: "))
if temp < 37.5:
    print("normal")
elif temp < 38.5:
    print("erhöht")
else:
    print("Fieber")
```

```
Temperatur in Grad Celsius: 23
normal
```

Eingabe	36.8	37.4	37.5	38.4	38.5	39.2
Ausgabe						

Aufgabe 5: Fehler finden: Medaillen beim Sportfest ●●○

Beim Sportfest gibt es ab 60 Punkten Gold, ab 45 Punkten Silber und ab 30 Punkten Bronze. Wer weniger hat, bekommt keine Medaille. Das Programm soll die Medaille ausgeben. Es enthält **drei Fehler**. Einer davon erzeugt keine Fehlermeldung, sondern eine falsche Ausgabe.

```
punkte = int(input("Punkte: "))
if punkte >= 30:
    print("Bronze")
elif punkte >= 45:
    print("Silber")
elif punkte >= 60
    print("Gold")
else:
print("keine Medaille")
```

```
  File "<Zelle 6>", line 6
    elif punkte >= 60
                     ^
SyntaxError: expected ':'
```

1. Finde die drei Fehler. Notiere jeweils die Zeile und was falsch ist.
2. Schreibe das Programm richtig auf. Teste es mit den Grenzwerten 29, 30, 44, 45, 59 und 60.

Deine Antwort:

…

Aufgabe 6: Was ziehe ich an? ●●○

Schreibe ein Programm, das die Temperatur (ganze Zahl in Grad Celsius) einliest und eine Kleidungsempfehlung ausgibt:

- unter 10 Grad: „Zieh eine Jacke an.“
- 10 bis unter 20 Grad: „Ein Pulli reicht.“

- ab 20 Grad: „T-Shirt-Wetter!“

Danach soll das Programm fragen, ob es regnet (`ja/nein`). Nur wenn die Antwort `ja` ist, gibt es zusätzlich „Nimm einen Regenschirm mit.“ aus. Teste mit allen Zweigen und den Grenzwerten.

Aufgabe 7: Schaltjahr mit Verzweigung ●●○

Im Abschnitt über Operatoren hast du im Auftrag *Schaltjahr-Check* einen Wahrheitswert berechnet. Schreibe das Programm jetzt so, dass es einen vollständigen Satz ausgibt, zum Beispiel `2024 ist ein Schaltjahr.` oder `2023 ist kein Schaltjahr.` Teste mit 2024, 2023, 1900 und 2000.

Zur Erinnerung: Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, aber nicht durch 100 – **außer** es ist auch durch 400 teilbar.

Aufgabe 8: Welches Dreieck? ●●●

Aus drei Seitenlängen `a`, `b` und `c` kann man nur dann ein Dreieck bauen, wenn je zwei Seiten zusammen länger sind als die dritte: `a + b > c`, `a + c > b` und `b + c > a`.

Schreibe ein Programm, das drei Seitenlängen einliest. Wenn sich kein Dreieck bauen lässt, soll es das ausgeben. Sonst soll es mit einer **verschachtelten** Verzweigung ausgeben, ob das Dreieck

- **gleichseitig** ist (alle drei Seiten gleich lang),
- **gleichschenklig** ist (genau zwei Seiten gleich lang) oder
- **allgemein** ist (alle Seiten verschieden).

Teste mit 3, 4, 5 und mit 5, 5, 8 und mit 5, 5, 5 und mit 1, 2, 5.

DS 4 – Übung: Verzweigungen lesen, testen und erweitern

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- Fehler in Programmen mit Verzweigungen finden, mit Hilfe der Fehlermeldung erklären und beheben,
- ein Programm mit Verzweigungen lesen und seine Ausgabe vorhersagen, ohne es auszuführen,
- zu einem Programm sinnvolle Testfälle auswählen, auch für die Grenzfälle,
- eine Mehrfachauswahl als `CASE OF` beschreiben und in Python mit `match`/`case` umsetzen,
- ein Programm mit einer Mehrfachauswahl und einer verschachtelten Verzweigung schreiben.

🗣️ Sprachhilfe

- *„Da ist ein Fehler in Zeile 2."* → „In Zeile 2 fehlt der Doppelpunkt. Python meldet `SyntaxError: expected ':'`."
- *„Ich hab's mit 5 probiert."* → „Im Testfall mit der Eingabe 5 erwarte ich die Ausgabe ‚positiv'. Das Programm liefert ‚positiv', der Test ist bestanden."
- *„Match guckt, was es ist."* → „Die Mehrfachauswahl vergleicht den Wert von `op` nacheinander mit den festen Werten der Fälle."

Worum es geht

Programmiererinnen und Programmierer **lesen** fremden Code, **suchen Fehler** und **erweitern** Programme. Die Fehlersuche heißt *Debugging*, „Entwanzen", denn einen Programmfehler nennt man *Bug* („Käfer"). Heute übst du das mit Verzweigungen, lernst gut ausgewählte **Testfälle** und eine neue Verzweigung: die **Mehrfachauswahl**.

1 Das brauchst du

```
if Bedingung_1:
    Anweisungen, wenn Bedingung_1 wahr ist
elif Bedingung_2:
    Anweisungen, wenn Bedingung_1 falsch und Bedingung_2 wahr ist
else:
    Anweisungen, wenn keine Bedingung wahr ist
```

📌 **Merksatz:** Nur der Block der **ersten zutreffenden** Bedingung wird ausgeführt. Achte auf **Doppelpunkt** und **Einrückung**. `==` vergleicht, `=` weist zu.

Fehler im Code	Meldung von Python	Abhilfe
Doppelpunkt vergessen: `if zahl > 0`	`SyntaxError: expected ':'`	`:` ans Zeilenende
Block nicht eingerückt	`IndentationError: expected an indented block …`	Block 4 Leerzeichen einrücken
Zeile ohne Grund eingerückt	`IndentationError: unexpected indent`	Einrückung entfernen
`=` statt `==` in der Bedingung	`SyntaxError: invalid syntax. Maybe you meant '==' …`	`==` schreiben
Bedingung hinter `else`: `else zahl < 0:`	`SyntaxError: expected ':'`	`elif` statt `else`
`elif`-Bedingungen in falscher Reihenfolge	keine, aber falsches Ergebnis	Reihenfolge prüfen

2 Programme testen

Ein Programm ohne Fehlermeldung ist noch lange nicht richtig. Man **testet** es: mit verschiedenen Eingaben ausführen und die Ausgabe mit dem vergleichen, was herauskommen **soll**. Eine Eingabe mit erwarteter Ausgabe heißt **Testfall**. Die erwartete Ausgabe überlegst du **vorher**.

1. **Jeder Zweig mindestens einmal.**
2. **Die Grenzfälle**, also die Werte direkt an einer Grenze. Bei `alter < 18` sind das 17 und 18.

Beispiel: Ein Programm soll ausgeben, ob eine Zahl positiv, negativ oder null ist.

Eingabe	erwartete Ausgabe	Warum dieser Test?
`5`	`Die Zahl ist positiv`	Zweig „positiv“
`-3`	`Die Zahl ist negativ`	Zweig „negativ“
`0`	`Die Zahl ist null`	Zweig „null“, zugleich die Grenze
`1` und `-1`	positiv bzw. negativ	direkt neben der Grenze

> 📌 **Merksatz:** Ein **Testfall** besteht aus einer Eingabe und der vorher überlegten erwarteten Ausgabe. Teste jeden Zweig und jeden Grenzfall.

📝 Übungen: Schritt für Schritt selbstständig

Das **Musterbeispiel** zeigt dir das Vorgehen beim Debugging. Danach werden die Übungen schrittweise schwieriger, die Hilfen werden weniger.

🧩 Musterbeispiel (vorgemacht) – Bestanden?

Das Programm soll ab 50 Punkten „bestanden“ ausgeben, sonst „nicht bestanden“. Es enthält **zwei Fehler**. Führe die Zelle aus.

```
1 punkte = int(input("Punkte: "))
2 if punkte >= 50
3     print("bestanden")
4 else punkte < 50:
5     print("nicht bestanden")
```

```
  File "<Zelle 1>", line 2
    if punkte >= 50
                  ^
SyntaxError: expected ':'
```

So gehst du vor:

1. **Letzte Zeile der Meldung lesen:** `SyntaxError: expected ':'` in Zeile 2. Am Ende von `if punkte >= 50` fehlt der Doppelpunkt. Ergänzen, neu starten.
2. **Nächste Meldung:** Wieder `expected ':'`, jetzt in Zeile 4. Hinter `else` darf keine Bedingung stehen. `else` heißt ohnehin „in allen anderen Fällen“, also einfach `else:`.
3. **Testen** mit vorher überlegten Testfällen:

Eingabe	erwartete Ausgabe	Warum?
`80`	`bestanden`	Zweig „bestanden“
`20`	`nicht bestanden`	Zweig „nicht bestanden“
`50`	`bestanden`	Grenzfall, wegen `>=`
`49`	`nicht bestanden`	direkt neben der Grenze

Die korrigierte Fassung:

```
punkte = int(input("Punkte: "))
if punkte >= 50:
    print("bestanden")
else:
    print("nicht bestanden")
```

```
Punkte: 65
bestanden
```

🐞 Übung 1 – Fehler finden (Debugging)

Das Programm soll ausgeben, ob eine Zahl positiv, negativ oder null ist. Es enthält **drei Fehler**.

1. Finde die Fehler **bevor** du es ausführst. Notiere jeweils Zeile und Fehler.
2. Führe es aus. Python meldet immer nur den **ersten** Fehler. Behebe ihn und starte neu, wie im Musterbeispiel.
3. Teste dein korrigiertes Programm mit den Testfällen aus Teil 2.

> 💡 **Tipp:** Zwei der drei Fehler stehen in derselben Zeile.

```
zahl = int(input("Zahl eingeben: "))
if zahl = 0
    print("Die Zahl ist null")
elif zahl > 0:
print("Die Zahl ist positiv")
else:
    print("Die Zahl ist negativ")
```

```
  File "<Zelle 3>", line 2
    if zahl = 0
       ^^^^^^^^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Die Fehler:

…

🔮 Übung 2 – FizzBuzz lesen

Lies das Programm, **ohne** es auszuführen.

1. Was gibt es für `zahl = 17` aus? Begründe mit den Bedingungen.
2. Sage die Ausgabe für `zahl = 9`, `10` und `30` voraus.
3. Warum wird `zahl % 15 == 0` **zuerst** geprüft?

Ändere danach den Wert in Zeile 1 und prüfe deine Vorhersagen.

```
zahl = 17
if zahl % 15 == 0:
    print("FizzBuzz")
elif zahl % 3 == 0:
    print("Fizz")
elif zahl % 5 == 0:
    print("Buzz")
else:
    print(zahl)
```

```
17
```

Meine Vorhersagen und Begründungen:

…

3 Mehrfachauswahl: `CASE OF`

Oft wird **ein und derselbe Wert** mit mehreren festen Möglichkeiten verglichen, zum Beispiel eine Note in Worten. Mit `if` und `elif` steht in jeder Zeile wieder `note == …`:

```
note = int(input("Note (1-6): "))
if note == 1:
    print("sehr gut")
elif note == 2:
    print("gut")
elif note == 3:
    print("befriedigend")
elif note == 4:
    print("ausreichend")
elif note == 5:
    print("mangelhaft")
elif note == 6:
    print("ungenügend")
else:
    print("Das ist keine gültige Note.")
```

```
Note (1-6): 2
gut
```

Unabhängig von einer Programmiersprache beschreibt man eine **Mehrfachauswahl** so:

```
CASE note OF
    1: Ausgabe "sehr gut"
    2: Ausgabe "gut"
    ...
    6: Ausgabe "ungenügend"
    OTHERWISE: Ausgabe "Das ist keine gültige Note."
END CASE
```

`OTHERWISE` („andernfalls“) ist der **Auffangfall**, wenn keiner der Werte passt.

> 📌 **Definition:** Bei einer **Mehrfachauswahl** (`CASE OF`) wird **ein Ausdruck** mit mehreren **festen Werten** verglichen. Ausgeführt wird der Fall, dessen Wert passt. Passt keiner, wird der Auffangfall ausgeführt.

In Python: `match` und `case`

```
note = int(input("Note (1-6): "))
match note:
    case 1:
        print("sehr gut")
    case 2:
        print("gut")
    case 3:
        print("befriedigend")
    case 4:
        print("ausreichend")
    case 5:
        print("mangelhaft")
    case 6:
        print("ungenügend")
    case _:
        print("Das ist keine gültige Note.")
```

```
Note (1-6): 2
gut
```

- **Zeile 2:** `match note:` legt fest, **welcher Wert** verglichen wird. Doppelpunkt, dann einrücken.
- **Zeilen 3–14:** Jedes `case` nennt einen festen Wert. Beim ersten passenden Fall wird der Block darunter ausgeführt, die übrigen werden übersprungen.
- **Zeilen 15–16:** `case _` ist der **Auffangfall**. Der Unterstrich passt zu jedem Wert. Er entspricht `else` und steht als **letzter** Fall.

Das funktioniert genauso mit Texten. 🔮 **Vorhersage:** Was gibt die Zelle aus?

```
operator = "-"
match operator:
    case "+":
        print("Addition")
    case "-":
        print("Subtraktion")
    case _:
        print("Unbekannter Operator")
```

```
Subtraktion
```

> 📌 **Merksatz:** `CASE OF` eignet sich, wenn **ein Ausdruck mit mehreren festen Werten** verglichen wird. In Python heißt das `match`/`case`, `case _` ist der Auffangfall. Für **Bereiche** („unter 6 Jahren“) und Bedingungen mit `and`/`or` bleibt `if`/`elif`/`else` die richtige Wahl.

> ⚠️ **Stolperstein:** `match` gibt es erst seit Python 3.10. Steht `case _` nicht am Ende, meldet Python einen `SyntaxError`, weil die Fälle darunter nie erreicht würden.

📝 Übung 3 – Taschenrechner mit CASE OF (Lücken füllen)

Der Taschenrechner fragt zwei Zahlen und einen Operator (`+`, `-`, `*`, `/`) ab. Bei einer Division durch 0 und bei einem unbekannten Operator gibt er eine verständliche Meldung aus. Ersetze jedes `___`. Die Zelle läuft erst, wenn alle Lücken gefüllt sind.

```
Erste Zahl: 7
Operator (+ - * /): /
Zweite Zahl: 2
Ergebnis: 3.5
```

Teste mit allen vier Operatoren, mit `7 / 0` und mit einem unbekannten Operator wie `^`.

Lückencode

```
a = float(input("Erste Zahl: "))
op = input("Operator (+ - * /): ")      # Text, wird nicht umgewandelt
b = float(input("Zweite Zahl: "))
match ___:
    case "+":
        print("Ergebnis:", a + b)
    case "-":
        print("Ergebnis:", a - b)
    case ___:
        print("Ergebnis:", a * b)
    case "/":
        if ___:
            print("Ergebnis:", a / b)
        else:
            print("Division durch 0 ist nicht erlaubt!")
    case ___:
        print("Unbekannter Operator")
```

📝 Übung 4 – Eintrittspreis (mit Gerüst)

Im Freibad gelten diese Preise:

- Kinder unter 6 Jahren: Eintritt frei
- 6 bis 17 Jahre: 2,50 Euro
- ab 18 Jahren: 4,00 Euro, mit Schüler- oder Studierendenausweis 3,00 Euro

Das Programm fragt zuerst das Alter. **Nur** ab 18 fragt es zusätzlich nach dem Ausweis (`ja`/`nein`).

1. Lege **zuerst** eine Tabelle mit Testfällen an, die alle Zweige und Grenzfälle abdeckt.
2. Ersetze dann die Kommentare durch Programmzeilen und teste mit deiner Tabelle.

Eingaben	erwartete Ausgabe	Warum dieser Test?

```
# Alter einlesen
# if: unter 6 → Eintritt frei
# elif: unter 18 → 2,50 Euro
# else:
#     Ausweis abfragen (nur hier!)
#     innere Verzweigung: ja → 3,00 Euro, sonst → 4,00 Euro
```

🚀 Zusatzaufgabe – Taschenrechner erweitern (selbstständig)

Kopiere deinen Taschenrechner aus Übung 3 und erweitere ihn um die Operatoren `//`, `%` und `**`. Auch bei `//` und `%` darf nicht durch 0 geteilt werden. Lege zuerst die Testfälle fest. Kontrolle: `7 // 2` → 3.0, `7 % 2` → 1.0, `7 ** 2` → 49.0.

📌 Das Wichtigste in Kürze

- Fehlermeldungen von unten lesen, ersten Fehler beheben, neu starten.
- **Testfall** = Eingabe + vorher überlegte erwartete Ausgabe. Jeden Zweig und jeden Grenzfall testen.
- Bei `elif` den **speziellsten** Fall zuerst prüfen (FizzBuzz).
- **Mehrfachauswahl** `CASE OF`: ein Wert, mehrere feste Möglichkeiten. In Python `match`/`case`, Auffangfall `case _` am Ende.
- Vor einer Division prüfen, ob durch 0 geteilt wird.

Zum Weiterüben: `DS04_Weitere_Uebungen.ipynb`.

DS 4 – Weitere Übungen

Übung: Verzweigungen lesen, testen und erweitern

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Welcher Tag? ●○○

Lies das Programm, **ohne** es auszuführen. Gib an, was es für `tag = 3` ausgibt. Gib ebenso die Ausgabe für `tag = 7` und für `tag = 1` an. Prüfe danach am Computer, indem du den Wert in Zeile 1 änderst.

```
1 tag = 3
2 match tag:
3     case 1:
4         print("Montag")
5     case 2:
6         print("Dienstag")
7     case 3:
8         print("Mittwoch")
9     case _:
10        print("anderer Tag")
```

```
Mittwoch
```

Deine Antwort:

…

Aufgabe 2: Ampel mit match ●○○

Schreibe das folgende Programm so um, dass es statt `if/elif/else` eine Mehrfachauswahl mit `match` und `case` verwendet. Das Programm soll danach genau dasselbe ausgeben. Teste mit `rot`, `grün` und `blau`.

```
1 farbe = input("Ampelfarbe: ")
2 if farbe == "rot":
3     print("Stehen bleiben!")
4 elif farbe == "gelb":
5     print("Warten!")
6 elif farbe == "grün":
7     print("Gehen!")
8 else:
9     print("Diese Farbe kennt die Ampel nicht.")
```

```
Ampelfarbe: rot
Stehen bleiben!
```

Aufgabe 3: Wochentage ●○○

Das Programm soll zu einem Kürzel von `Mo` bis `Fr` den ganzen Wochentag ausgeben. Bei allen anderen Eingaben soll es „Das ist kein Schultag.“ ausgeben. Ergänze die Lücken `___` und die fehlenden Fälle für `Mi`, `Do` und `Fr`.

Lückencode

```
kuerzel = input("Kürzel (Mo bis Fr): ")
match ___:
    case "Mo":
        print("Montag")
    case ___:
        print("Dienstag")
    # Hier fehlen die Fälle für Mi, Do und Fr.
    ___:
        print("Das ist kein Schultag.")
```

Aufgabe 4: Testfälle für ein Spiel ●○○

Ein Spiel teilt Spielerinnen und Spieler nach ihrem Level ein:

```
level = int(input("Level: "))
if level < 1:
    print("ungültiges Level")
elif level <= 10:
    print("Anfänger")
elif level <= 30:
    print("Fortgeschritten")
else:
    print("Profi")
```

```
Level: 3
Anfänger
```

Lege eine Tabelle mit Testfällen an: Jeder Zweig soll mindestens einmal vorkommen, außerdem alle Grenzfälle. Trage zu jedem Testfall die **erwartete** Ausgabe ein, **bevor** du das Programm ausführst. Prüfe dann am Computer.

Eingabe	erwartete Ausgabe	Warum dieser Test?

Aufgabe 5: Fehler finden: Pizzagröße ●●○

Das Programm soll zu einer Pizzagröße (`S`, `M` oder `L`) den Preis ausgeben. Es enthält **drei Fehler**.

```
groesse = input("Größe (S, M, L): ")
match groesse
    case "S":
        print("Preis: 6,50 Euro")
    case "M":
    print("Preis: 8,00 Euro")
    case "L":
        print("Preis: 9,50 Euro")
    else:
        print("Diese Größe gibt es nicht.")
```

```
  File "<Zelle 5>", line 2
    match groesse
                 ^
SyntaxError: expected ':'
```

1. Finde die drei Fehler. Notiere jeweils die Zeile und was falsch ist.
2. Schreibe das Programm richtig auf und teste es mit `M` und mit `XL`.

Deine Antwort:

…

Aufgabe 6: match oder if? ●●○

Entscheide für jede Situation, ob eine Mehrfachauswahl mit `match`/`case` oder eine Verzweigung mit `if`/`elif`/`else` besser passt. Begründe kurz.

Situation	`match` oder `if`?
a) Eine Schulnote von 1 bis 6 in Worten ausgeben	
b) Aus einer Temperatur eine Warnstufe bestimmen (unter 0, 0 bis 30, über 30)	
c) Ein Menü mit den Punkten `a`, `b` und `c`	
d) Prüfen, ob jemand mindestens 16 ist **und** eine Einverständniserklärung hat	
e) Zu einer Monatsnummer von 1 bis 12 den Namen des Monats ausgeben	

Deine Antwort:

…

Aufgabe 7: Längen umrechnen ●●○

Schreibe ein Programm, das eine Länge (Kommazahl) und eine Einheit (`km`, `m` oder `cm`) einliest und die Länge in Metern ausgibt. Verwende eine Mehrfachauswahl. Bei einer unbekannten Einheit soll eine verständliche Meldung erscheinen. Teste mit `2.5` und `km`, mit `150` und `cm` und mit `3` und `mm`.

So könnte es aussehen:

Ausgabe:

```
Länge: «2.5»
Einheit (km, m, cm): «km»
Das sind 2500.0 Meter.
```

Aufgabe 8: Taschenrechner erweitern ●●●

Erweitere deinen Taschenrechner aus dem Auftrag *Taschenrechner mit CASE OF* um drei weitere Operatoren: `//` (Ganzzahldivision), `%` (Rest) und `**` (Potenz). Auch bei `//` und `%` darf die zweite Zahl nicht 0 sein. Lege eine Tabelle mit Testfällen an, in der jeder neue Operator und jeder Sonderfall mindestens einmal vorkommt, und teste dein Programm damit.

Deine Antwort:

…

DS 5 – Die for-Schleife: Anweisungen wiederholen

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- erklären, was eine Schleife ist und wann man eine for-Schleife verwendet,
- eine for-Schleife mit Schleifenkopf und eingerücktem Schleifenrumpf schreiben,
- mit `range` Zahlenfolgen erzeugen, auch mit eigenem Start und eigener Schrittweite,
- mit einer Schleife eine Summe berechnen und dabei `+=` verwenden,
- den Ablauf einer Schleife in einem Ablaufprotokoll Schritt für Schritt verfolgen.

🗣️ Sprachhilfe

- *„Das wird fünfmal gemacht."* → „Die Anweisungen im Schleifenrumpf werden fünfmal wiederholt."
- *„i wird immer eins mehr."* → „Der Wert der Zählvariablen `i` wird bei jedem Durchlauf um 1 erhöht."
- *„Da kommt 15 raus."* → „Die Schleife addiert die Zahlen von 1 bis 5 zur Summenvariablen. Nach der Schleife gibt das Programm den Wert 15 aus."

Worum es geht

Dein Programm soll die Zahlen von 1 bis 100 ausgeben. Bisher bräuchtest du 100 `print`-Zeilen. Besser wäre: „Wiederhole das 100-mal." Aus der Blockprogrammierung kennst du „wiederhole 10-mal". In Python heißt das **for-Schleife**.

1 Wiederholen statt abschreiben

```
for i in range(3):
    print("Hallo!")
```

```
Hallo!
Hallo!
Hallo!
```

- **Zeile 1:** der **Schleifenkopf**: „Mache das Folgende dreimal." Er endet mit einem **Doppelpunkt**.
- **Zeile 2: eingerückt**. Alles, was eingerückt unter dem Kopf steht, wird wiederholt. Das ist der **Schleifenrumpf**.

🧪 **Probiere es aus:** Schreibe statt `3` die Zahl `100`.

📌 **Definition:** Eine **Schleife** ist eine Kontrollstruktur, die einen Block von Anweisungen (den **Schleifenrumpf**) mehrfach ausführt. Jede Ausführung heißt **Durchlauf**. Bei einer **for-Schleife** (Zählschleife) steht die Anzahl der Durchläufe schon vor dem Start fest.

📌 **Merksatz:** Die **for-Schleife** verwendet man, wenn die **Anzahl der Wiederholungen vorher bekannt** ist.

2 Aufbau einer for-Schleife

Die Zählvariable

Im Kopf steht eine Variable, hier `i`, die **Zählvariable**. Bei jedem Durchlauf bekommt sie automatisch den nächsten Wert.

```
1 for i in range(1, 4):
2     print("Runde", i)
3 print("Fertig")
```

```
Runde 1
Runde 2
Runde 3
Fertig
```

- **Zeile 1:** `range(1, 4)` liefert der Reihe nach `1`, `2` und `3`.
- **Zeile 2:** Der Rumpf gibt in jedem Durchlauf den aktuellen Wert von `i` aus.
- **Zeile 3:** **Nicht** eingerückt. Sie wird erst ausgeführt, wenn die Schleife fertig ist, und nur einmal.

Stell dir eine Läuferin auf der Bahn vor: Sie läuft ihre Runden und zählt mit. Nach der letzten Runde geht sie zur Bank (Zeile 3).

Schritt	Zeile	Anweisung	i
1	1	`for i in range(1, 4):`	1
2	2	`print("Runde", i)`	1
3	1	`for i in range(1, 4):`	2
4	2	`print("Runde", i)`	2
5	1	`for i in range(1, 4):`	3
6	2	`print("Runde", i)`	3
7	1	`for i in range(1, 4):`	3
8	3	`print("Fertig")`	3

- Nach dem Rumpf springt Python **zurück zum Kopf**. Dort bekommt `i` den nächsten Wert.
- In Schritt 7 liefert `range` keine Zahl mehr. Die Schleife endet, `i` behält den Wert `3`.
- Der Kopf wird viermal ausgeführt, der Rumpf dreimal.

> 📌 **Merksatz:** Die **Zählvariable** bekommt bei jedem Durchlauf automatisch den nächsten Wert. Du musst sie nicht selbst erhöhen.

Einrückung: Was gehört zur Schleife?

🔮 **Vorhersage:** Die beiden Programme unterscheiden sich nur in der letzten Zeile. Was gibt jedes aus?

```
1 for i in range(3):
2     print("Ich übe Python.")
3     print("Das macht Spaß!")
```

```
Ich übe Python.
Das macht Spaß!
Ich übe Python.
Das macht Spaß!
Ich übe Python.
Das macht Spaß!
```

```python
for i in range(3):
    print("Ich übe Python.")
print("Das macht Spaß!")
```

```
Ich übe Python.
Ich übe Python.
Ich übe Python.
Das macht Spaß!
```

⚠️ Stolpersteine: Einrückung und Doppelpunkt

Führe beide Zellen aus und lies die letzte Zeile der Fehlermeldung.

```python
for i in range(3):
print(i)
```

```
  File "<Zelle 5>", line 2
    print(i)
    ^^^^^
IndentationError: expected an indented block after 'for' statement on line 1
```

```python
for i in range(3)
    print(i)
```

```
  File "<Zelle 6>", line 1
    for i in range(3)
                    ^
SyntaxError: expected ':'
```

1. `IndentationError: expected an indented block after 'for' statement on line 1`: Der Rumpf muss eingerückt sein.
2. `SyntaxError: expected ':'`: Am Ende des Kopfes fehlt der Doppelpunkt.

3 Zahlenfolgen mit `range`

`range` („Bereich“) erzeugt eine Folge ganzer Zahlen.

- **`range(stop)`** beginnt bei `0` und endet **vor** `stop`.
- **`range(start, stop)`** beginnt bei `start` und endet **vor** `stop`.
- **`range(start, stop, schritt)`** zählt mit der **Schrittweite** `schritt`, negativ heißt rückwärts.

Aufruf	erzeugt die Zahlen	Anzahl
`range(5)`	0, 1, 2, 3, 4	5
`range(1, 6)`	1, 2, 3, 4, 5	5
`range(3, 8)`	3, 4, 5, 6, 7	5
`range(0, 10, 2)`	0, 2, 4, 6, 8	5
`range(1, 10, 3)`	1, 4, 7	3
`range(10, 0, -1)`	10, 9, 8, 7, 6, 5, 4, 3, 2, 1	10
`range(5, 1)`	keine Zahl (die Schleife läuft gar nicht)	0

📌 **Merksatz:** `range(n)` erzeugt `0, 1, …, n-1`. `range(start, stop)` erzeugt `start` bis `stop-1`. `range(start, stop, schritt)` erlaubt eine eigene Schrittweite.

💡 **Tipp:** `stop` ist wie die Ziellinie: Man zählt bis kurz davor. Bei `range(start, stop)` gibt es genau `stop - start` Zahlen.

Zahlen ausgeben und das Einmaleins

🔮 **Vorhersage:** Überlege zuerst, was das Programm ausgibt.

```
for i in range(1, 6):
    print(i)
print("---")
for i in range(1, 11):
    print("7 *", i, "=", 7 * i)
```

```
1
2
3
4
5
---
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
```

- **Zeilen 1–2:** Für „1 bis 5“ braucht man `range(1, 6)`.
- **Zeile 3:** nicht eingerückt, wird einmal ausgegeben.
- **Zeilen 4–5:** `7 * i` wird in jedem Durchlauf neu ausgerechnet. Beide Schleifen dürfen `i` benutzen.

Rückwärts zählen

```
for i in range(5, 0, -1):
    print(i)
print("Start!")
```

```
5
4
3
2
1
Start!
```

⚠️ **Stolperstein:** `range(1, 10)` erzeugt die Zahlen von 1 bis **9**. Python meldet keinen Fehler. Prüfe immer die erste und die letzte Zahl der Ausgabe. Für „1 bis 10“ schreibst du `range(1, 11)`.

🔮 Übung 1 – range-Detektiv

1. Welche Zahlen erzeugen die Aufrufe? Erst im Kopf, dann in der Codezelle prüfen.
2. Gib zu den Zahlenfolgen einen passenden `range`-Aufruf an.

Aufruf	erzeugt die Zahlen
`range(4)`	
`range(2, 7)`	
`range(5, 30, 5)`	
`range(9, 3, -2)`	

Zahlenfolge	Aufruf
1, 2, 3, 4, 5, 6, 7, 8	
10, 20, 30, 40, 50	
3, 2, 1, 0	

```
# Prüfe hier jeden Aufruf, indem du die Zahl in range(...) änderst
for i in range(4):
    print(i)
```

```
0
1
2
3
```

✏️ Übung 2 – Zahlenreihe

Gib alle geraden Zahlen von 2 bis 20 aus. Tipp: Schrittweite nutzen. Prüfe die erste und die letzte Zahl!

Lückencode

```
for i in range(___, ___, ___):
    print(i)
```

4 Summen bilden

Denk an eine Kasse: Am Anfang steht die Anzeige auf 0. Bei jedem Artikel kommt sein Preis dazu. Am Ende steht der Gesamtbetrag da. Im Programm brauchst du dafür eine **Summenvariable** (Fachwort **Akkumulator**, lateinisch *accumulare* = anhäufen).

```
summe = 0
for i in range(1, 6):
    summe = summe + i
print("Summe:", summe)
```

```
Summe: 15
```

- **Zeile 1:** Startwert `0`, die Kasse ist leer.
- **Zeile 3:** Der aktuelle Wert von `i` kommt zur bisherigen Summe dazu, wie `alter = alter + 1`.
- **Zeile 4:** nicht eingerückt, die Summe wird einmal am Ende ausgegeben.

Lies im Ablaufprotokoll die Spalte `summe` von oben nach unten: 0, 1, 3, 6, 10, 15.

Schritt	Zeile	Anweisung	i	summe
1	1	`summe = 0`	-	0
2	2	`for i in range(1, 6):`	1	0
3	3	`summe = summe + i`	1	1
4	2	`for i in range(1, 6):`	2	1
5	3	`summe = summe + i`	2	3
6	2	`for i in range(1, 6):`	3	3
7	3	`summe = summe + i`	3	6
8	2	`for i in range(1, 6):`	4	6
9	3	`summe = summe + i`	4	10
10	2	`for i in range(1, 6):`	5	10
11	3	`summe = summe + i`	5	15
12	2	`for i in range(1, 6):`	5	15
13	4	`print("Summe:", summe)`	5	15

> 📌 **Merksatz:** Eine Summe berechnet man in drei Schritten: **vor** der Schleife `summe = 0`, **in** der Schleife `summe = summe + zahl`, **nach** der Schleife das Ergebnis ausgeben.

Die Kurzschreibweise `+=`

`summe += i` heißt „`summe` wird um `i` erhöht“ und bewirkt dasselbe wie `summe = summe + i`.

Kurzschreibweise	bedeutet	Beispiel (vorher `x = 10`)
`x += 3`	`x = x + 3`	`x` ist danach `13`
`x -= 3`	`x = x - 3`	`x` ist danach `7`
`x *= 3`	`x = x * 3`	`x` ist danach `30`

⚠️ Stolpersteine beim Summieren

🔮 **Vorhersage:** Was geben die beiden Programme aus? Vergleiche mit dem richtigen Programm oben.

```
1 for i in range(1, 6):
2     summe = 0
3     summe = summe + i
4 print("Summe:", summe)
```

```
Summe: 5
```

```
1 summe = 0
2 for i in range(1, 6):
3     summe = summe + i
4     print("Summe:", summe)
```

```
Summe: 1
Summe: 3
Summe: 6
Summe: 10
Summe: 15
```

1. Der Startwert steht **in** der Schleife: Die Summe wird jedes Mal auf 0 gesetzt, am Ende steht nur die letzte Zahl 5 darin.

2. Die Ausgabe ist eingerückt: Nach jedem Durchlauf kommt eine Zwischensumme. Das ist zum Nachprüfen nützlich, ist aber nur das Ergebnis gefragt, gehört `print` nach der Schleife.

Die Obergrenze eingeben lassen

`range` arbeitet nur mit ganzen Zahlen, deshalb `int(...)`. Damit `n` selbst mitgezählt wird, ist der Stoppwert `n + 1`.

```
n = int(input("Bis zu welcher Zahl soll ich addieren? "))
summe = 0
for i in range(1, n + 1):
    summe += i
print("Die Summe von 1 bis", n, "ist", summe)
```

```
Bis zu welcher Zahl soll ich addieren? 10
Die Summe von 1 bis 10 ist 55
```

⚠️ **Stolperstein:** Ohne `int(...)` ist `n` ein Text: `TypeError: 'str' object cannot be interpreted as an integer`. Mit `float` heißt es `'float'`. `range` braucht **ganze Zahlen**.

🔮 Übung 3 – Ablaufprotokoll

Fülle das Ablaufprotokoll aus, **ohne** das Programm auszuführen (– heißt: noch kein Wert). Gib dann an, was es ausgibt. Prüfe danach.

```
x = 0
for i in range(3):
    x = x + 2 * i
print("x =", x)
```

```
x = 6
```

Schritt	Zeile	Anweisung	i	x
1	1	`x = 0`	–	
2	2	`for i in range(3):`		
3	3	`x = x + 2 * i`		
4	2	`for i in range(3):`		
5	3	`x = x + 2 * i`		
6	2	`for i in range(3):`		
7	3	`x = x + 2 * i`		
8	2	`for i in range(3):`		
9	4	`print("x =", x)`		

Ausgabe: …

📝 Übungen: Schritt für Schritt selbstständig

Das **Musterbeispiel** ist ganz vorgemacht. Bei Übung 4 füllst du Lücken `___`, bei Übung 5 steht nur noch ein Gerüst aus Kommentaren, die Zusatzaufgabe schreibst du allein.

Schritt	Frage
1	**Vor der Schleife:** Welche Variable braucht einen Startwert?
2	**Schleifenkopf:** Welche Zahlen soll `i` annehmen? Erste und letzte Zahl prüfen!
3	**Rumpf:** Was passiert in jedem Durchlauf?
4	**Nach der Schleife:** Was wird einmal am Ende ausgegeben?

🧩 Musterbeispiel (vorgemacht) – Summe der geraden Zahlen

Aufgabe: Berechne die Summe der geraden Zahlen von 2 bis 10, also 2 + 4 + 6 + 8 + 10.

1. **Vor der Schleife:** `summe = 0`.
2. **Schleifenkopf:** `i` soll 2, 4, 6, 8, 10 sein. Start 2, Schrittweite 2, und der Stoppwert muss **größer** als 10 sein: `range(2, 11, 2)`.
3. **Rumpf:** `summe += i`.
4. **Nach der Schleife:** nicht eingerückt `print`.

```
# Schritt 1: vor der Schleife
summe = 0

# Schritt 2 und 3: Schleifenkopf und Rumpf
for i in range(2, 11, 2):
    summe += i

# Schritt 4: nach der Schleife
print("Summe der geraden Zahlen von 2 bis 10:", summe)
```

```
Summe der geraden Zahlen von 2 bis 10: 30
```

Kontrolle von Hand: 2 + 4 + 6 + 8 + 10 = 30.

📝 Übung 4 – Summe von 1 bis 100 (Lücken füllen)

Berechne die Summe aller Zahlen von 1 bis 100, ohne eine fertige Summenfunktion. Ersetze jedes `___`. Kontrolle: Wer `4950` erhält, hat die 100 vergessen.

Lückencode

```
summe = ___
for i in range(___, ___):
    summe ___ i
print("Summe:", ___)
```

📝 Übung 5 – Quadratzahlen (mit Gerüst)

Lies eine Zahl `n` ein. Gib für jede Zahl von 1 bis `n` ihre Quadratzahl aus und am Ende die Summe aller Quadratzahlen. Ersetze die Kommentare durch Programmzeilen.

```
Bis zu welcher Zahl? 4
1 zum Quadrat ist 1
2 zum Quadrat ist 4
3 zum Quadrat ist 9
4 zum Quadrat ist 16
Summe der Quadratzahlen: 30
```

```
# n als Ganzzahl einlesen
# Summenvariable mit Startwert
# Schleife: i von 1 bis n (Stoppwert beachten!)
#     Quadratzahl ausgeben
#     Quadratzahl zur Summe addieren
# nach der Schleife: Summe ausgeben
```

🚀 Zusatzaufgabe – Sternendreieck (selbstständig)

> 🧰 **Werkzeug:** Ein Text mal eine ganze Zahl hängt den Text so oft hintereinander: `"ab" * 3` ergibt `ababab`, `"-" * 20` ergibt 20 Striche. Die Zahl darf auch die Zählvariable sein.

Gib mit einer for-Schleife dieses Muster aus:

```
*
**
***
****
*****
```

Für Schnelle: Gib das Dreieck auch auf dem Kopf aus.

📌 Das Wichtigste in Kürze

- Die **for-Schleife** wiederholt den eingerückten **Rumpf**, wenn die Anzahl der Durchläufe vorher feststeht.
- Die **Zählvariable** bekommt bei jedem Durchlauf automatisch den nächsten Wert.
- `range(stop)`, `range(start, stop)`, `range(start, stop, schritt)`: `stop` gehört **nie** dazu.
- Summe: vor der Schleife `summe = 0`, in der Schleife `summe += zahl`, danach ausgeben.
- `range` braucht ganze Zahlen, Eingaben also mit `int(...)` umwandeln.

Zum Weiterüben: `DS05_Weitere_Uebungen.ipynb`.

DS 5 – Weitere Übungen

Die for-Schleife: Anweisungen wiederholen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Wie oft läuft die Schleife? ●○○

Bestimme für jeden Schleifenkopf, wie oft die Schleife durchlaufen wird und welchen Wert `i` im ersten und im letzten Durchlauf hat. Überprüfe am Computer, indem du in den Rumpf `print(i)` schreibst.

Schleifenkopf	Anzahl Durchläufe	erster Wert	letzter Wert
`for i in range(6):`			
`for i in range(1, 6):`			
`for i in range(3, 10):`			
`for i in range(0, 20, 5):`			
`for i in range(1, 2):`			
`for i in range(4, 4):`			

Aufgabe 2: Runde für Runde ●○○

Lies das Programm, **ohne** es auszuführen. Schreibe auf, was es ausgibt. Überprüfe deine Vorhersage danach am Computer.

```
for i in range(1, 4):
    print("Runde", i)
    print("---")
print("Spiel vorbei")
```

```
Runde 1
---
Runde 2
---
Runde 3
---
Spiel vorbei
```

Deine Antwort:

…

Aufgabe 3: Einmaleins nach Wahl ●○○

Das Programm soll eine Zahl einlesen und ihr Einmaleins von 1 bis 10 ausgeben, zum Beispiel für 9 die Zeilen `9 * 1 = 9` bis `9 * 10 = 90`. Ergänze die drei Lücken `___`.

Lückencode

```
zahl = int(input("Welche Reihe? "))
for i in range(___, ___):
    print(zahl, "*", i, "=", ___)
```

Aufgabe 4: Quadratzahlen ●○○

Schreibe ein Programm, das mit einer for-Schleife die Quadratzahlen von 1 bis 10 ausgibt. Jede Zeile soll so aussehen wie im Beispiel. Verwende den Operator `**`.

Ausgabe:

```
1 hoch 2 ist 1
2 hoch 2 ist 4
3 hoch 2 ist 9
```

Aufgabe 5: Fehler finden: Ungerade Zahlen addieren ●●○

Das Programm soll die Summe der ungeraden Zahlen von 1 bis 19 berechnen (1 + 3 + 5 + … + 19) und **einmal** ausgeben. Richtig wäre `Summe: 100`. Das Programm läuft ohne Fehlermeldung, enthält aber **drei Fehler**.

```
for i in range(1, 19, 2):
    summe = 0
    summe = summe + i
    print("Summe:", summe)
```

```
Summe: 1
Summe: 3
Summe: 5
Summe: 7
Summe: 9
Summe: 11
Summe: 13
Summe: 15
Summe: 17
```

1. Führe das Programm aus und beschreibe, was falsch ist.
2. Finde die drei Fehler und schreibe das Programm richtig auf.

Deine Antwort:

…

Aufgabe 6: Ablaufprotokoll: Abziehen ●●○

Fülle das Ablaufprotokoll zu dem Programm aus. Trage nach jedem Schritt die Werte von `i` und `wert` ein (– bedeutet: noch kein Wert). Gib dann an, was das Programm ausgibt.

```
wert = 10
for i in range(2, 8, 2):
    wert = wert - i
print("wert =", wert)
```

```
wert = -2
```

Schritt	Zeile	Anweisung	`i`	`wert`
1	1	`wert = 10`		
2	2	`for i in range(2, 8, 2):`		
3	3	`wert = wert - i`		
4	2	`for i in range(2, 8, 2):`		
5	3	`wert = wert - i`		
6	2	`for i in range(2, 8, 2):`		
7	3	`wert = wert - i`		
8	2	`for i in range(2, 8, 2):`		
9	4	`print("wert =", wert)`		

Deine Antwort:

...

Aufgabe 7: Sparschwein ●●○

Im Sparschwein liegt schon ein Startbetrag. Jede Woche kommt derselbe Betrag dazu. Schreibe ein Programm, das den Startbetrag, den Betrag pro Woche und die Anzahl der Wochen (alles ganze Zahlen) einliest. Es soll für jede Woche den Stand ausgeben.

So könnte es aussehen:

Ausgabe:

```
Startbetrag in Euro: «10»
Betrag pro Woche in Euro: «3»
Wie viele Wochen? «4»
Nach Woche 1 sind 13 Euro im Sparschwein.
Nach Woche 2 sind 16 Euro im Sparschwein.
Nach Woche 3 sind 19 Euro im Sparschwein.
Nach Woche 4 sind 22 Euro im Sparschwein.
```

Aufgabe 8: Notendurchschnitt ●●●

Schreibe ein Programm, das zuerst einliest, wie viele Noten es gibt. Danach soll es mit einer for-Schleife jede Note einzeln einlesen. Am Ende soll es die Summe der Noten und den Durchschnitt ausgeben, auf zwei Nachkommastellen gerundet.

Teste mit den Noten 2, 3, 1 und 2 (Durchschnitt 2.0) und mit 2, 2 und 1 (Durchschnitt 1.67). Was passiert, wenn du am Anfang `0` eingibst? Verbessere dein Programm so, dass es dann eine Meldung ausgibt, statt abzubrechen.

> **Tipp:** Die Eingabe der einzelnen Noten gehört **in** den Schleifenrumpf. Dann wird sie in jedem Durchlauf neu ausgeführt. Den Durchschnitt berechnest du **nach** der Schleife: Summe geteilt durch Anzahl.

DS 6 – Die while-Schleife: Wiederholen, solange eine Bedingung gilt

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- eine while-Schleife mit Bedingung und eingerücktem Schleifenrumpf schreiben,
- den Ablauf einer while-Schleife in einem Ablaufprotokoll darstellen,
- eine Endlosschleife erkennen, abbrechen und ihre Ursache beheben,
- Eingaben so lange wiederholen lassen, bis sie gültig sind,
- Zufallszahlen mit `random.randint` erzeugen,
- begründet entscheiden, ob eine for- oder eine while-Schleife besser passt.

🗣️ Sprachhilfe

- *„Das läuft, bis man 0 eingibt."* → „Die Anweisungen im Schleifenrumpf werden wiederholt, solange die eingegebene Zahl ungleich 0 ist."
- *„Dann hört es auf."* → „Sobald die Bedingung `zahl != 0` falsch ist, terminiert die Schleife, und das Programm gibt die Summe aus."
- *„Das hängt sich auf."* → „Die Schleife terminiert nicht, weil der Wert der Variablen `i` im Schleifenrumpf nicht erhöht wird."

Worum es geht

Mit der for-Schleife wiederholst du eine feste Anzahl von Malen. Oft weiß man das vorher nicht: Beim Würfelspiel würfelst du, **bis** eine Sechs kommt. Ein Geldautomat fragt die PIN, **solange** sie falsch ist. Dafür gibt es die **while-Schleife**.

1 Wiederholen, solange eine Bedingung wahr ist

while heißt „solange". Die Bedingung kennst du von `if`: ein Ausdruck, der `True` oder `False` ergibt.

```
1 i = 1
2 while i <= 3:
3     print(i)
4     i = i + 1
5 print("Ende")
```

```
1
2
3
Ende
```

- **Zeile 1:** Startwert. Das musst du bei while selbst erledigen.
- **Zeile 2:** `while`, Bedingung, Doppelpunkt. Vor **jedem** Durchlauf prüft Python, ob `i <= 3` wahr ist.
- **Zeilen 3–4:** der Rumpf. Die Erhöhung in Zeile 4 ist wichtig: Ohne sie bliebe `i` immer `1`.
- **Zeile 5:** wird ausgeführt, sobald die Bedingung falsch ist.

Schritt	Zeile	Anweisung	`i`
1	1	`i = 1`	`1`
2	2	`while i <= 3:`	`1`
3	3	`print(i)`	`1`
4	4	`i = i + 1`	`2`
5	2	`while i <= 3:`	`2`
6	3	`print(i)`	`2`
7	4	`i = i + 1`	`3`
8	2	`while i <= 3:`	`3`
9	3	`print(i)`	`3`
10	4	`i = i + 1`	`4`
11	2	`while i <= 3:`	`4`
12	5	`print("Ende")`	`4`

In Schritt 11 ist `4 <= 3` falsch, die Schleife **endet**. Am Ende hat `i` den Wert `4`, obwohl zuletzt `3` ausgegeben wurde.

> 📌 **Definition:** Eine **while-Schleife** (bedingte Schleife) wiederholt ihren Rumpf, **solange** eine Bedingung wahr ist. Die Bedingung wird **vor** jedem Durchlauf geprüft. Ist sie schon beim ersten Mal falsch, wird der Rumpf kein einziges Mal ausgeführt.

> 📌 **Merksatz:** Die **while-Schleife** verwendet man, wenn die **Anzahl der Wiederholungen vorher nicht bekannt** ist. Syntax: `while Bedingung:` und darunter der eingerückte Block.

2 Zahlen addieren, bis 0 eingegeben wird

Hier hilft die for-Schleife nicht: Wie viele Zahlen es sind, entscheidet die Nutzerin beim Eintippen.

```
1 summe = 0
2 zahl = int(input("Zahl eingeben (0 zum Beenden): "))
3 while zahl != 0:
4     summe = summe + zahl
5     zahl = int(input("Zahl eingeben (0 zum Beenden): "))
6 print("Summe aller Eingaben:", summe)
```

```
Zahl eingeben (0 zum Beenden): 4
Zahl eingeben (0 zum Beenden): 7
Zahl eingeben (0 zum Beenden): 0
Summe aller Eingaben: 11
```

- **Zeile 2:** Die **erste** Zahl wird **vor** der Schleife eingelesen. Sonst hätte `zahl` in Zeile 3 noch keinen Wert.
- **Zeile 3:** Solange die Zahl nicht 0 ist, wird der Rumpf ausgeführt.
- **Zeile 5:** Die **nächste** Zahl wird eingelesen. Diese Zeile sorgt dafür, dass sich die Bedingung ändern kann.

Ablaufprotokoll für die Eingaben 4, 7, 2, 0:

Schritt	Zeile	zahl	summe
1	1	-	0
2	2	4	0
3	3	4	0
4	4	4	4
5	5	7	4
6	3	7	4
7	4	7	11
8	5	2	11
9	3	2	11
10	4	2	13
11	5	0	13
12	3	0	13
13	6	0	13

> 💡 **Muster für Schleifen mit Eingaben:** 1. **vor** der Schleife die erste Eingabe einlesen, 2. im Kopf die Eingabe prüfen, 3. **am Ende** des Rumpfs die nächste Eingabe einlesen.

🧪 **Probiere es aus:** Gib gleich als Erstes `0` ein. Warum wird der Rumpf kein einziges Mal ausgeführt?

3 Endlosschleifen

Bei while bist **du** dafür verantwortlich, dass die Bedingung irgendwann falsch wird. Ein Countdown soll von 3 herunterzählen. 🔮 **Findest du den Fehler, bevor du die Zelle startest?**

Führe die Zelle aus und **brich sie ab**: in Jupyter mit der Stopp-Schaltfläche (Quadrat) oben, in VS Code mit „Interrupt“. Links neben der Zelle steht so lange ein Sternchen `[*]`.

```
1 zahl = 3
2 while zahl > 0:
3     print(zahl)
4     zahl = zahl + 1
5 print("Start!")
```

```
3
4
5
6
7
8
9
10
11
12
13
14
…
```

Hier abgebrochen: Die Ausgabe hört nicht auf (Endlosschleife). In Jupyter stoppst du sie mit ■ (Interrupt).

In Zeile 4 wird `zahl` erhöht statt verringert, `zahl > 0` bleibt für immer wahr. Richtig wäre `zahl = zahl - 1`. Beim Abbrechen meldet Python `KeyboardInterrupt`: Das ist kein Fehler im Code, nur die Bestätigung, dass du angehalten hast.

📌 **Definition:** Eine **Endlosschleife** ist eine Schleife, deren Bedingung nie falsch wird. Sie **terminiert nicht**.

📌 **Merksatz:** Die Bedingung muss sich **innerhalb der Schleife verändern** können, sonst entsteht eine Endlosschleife.

So findest du die Ursache:

1. Welche Variablen stehen in der Bedingung?
2. Wird mindestens eine davon **im Rumpf** verändert (`i = i + 1`, neues `input(...)`)?
3. In die **richtige Richtung**?
4. Steht die Änderung wirklich **eingerückt** im Rumpf?

🐞 Übung 1 – Endlosschleife reparieren

Der Code enthält eine Endlosschleife. Ändere **nicht** die Bedingung `i < 10`, sondern behebe den Fehler im Rumpf. Kontrolle: Ausgabe 0 bis 9. (Vorsicht beim Starten der unveränderten Zelle: abbrechen!)

```
1 i = 0
2 while i < 10:
3     print(i)
```

```
0
0
0
0
0
0
0
0
0
0
0
…
```

Hier abgebrochen: Die Ausgabe hört nicht auf (Endlosschleife). In Jupyter stoppst du sie mit ■ (Interrupt).

4 Eingaben prüfen: Wiederholen, bis die Eingabe stimmt

Das Programm akzeptiert nur Schulnoten von 1 bis 6. Wie oft sich jemand vertippt, weiß man vorher nicht.

```
1 note = int(input("Gib deine Note ein (1 bis 6): "))
2 while note < 1 or note > 6:
3     print("Ungültig! Erlaubt sind nur 1 bis 6.")
4     note = int(input("Gib deine Note ein (1 bis 6): "))
5 print("Gespeichert: Note", note)
```

```
Gib deine Note ein (1 bis 6): 2
Gespeichert: Note 2
```

- **Zeile 2:** Die Bedingung beschreibt eine **ungültige** Eingabe. Solange das zutrifft, wird wiederholt.
- **Zeile 5:** Hier kommt das Programm nur an, wenn die Note gültig ist.

💡 **Tipp:** Frage dich: „**Wann muss ich noch einmal fragen?**“ Die Antwort ist die Bedingung der while-Schleife.

5 Zufallszahlen

> 🧰 **Werkzeug:** Das **Modul** `random` ist eine Sammlung von Befehlen rund um den Zufall. Schreibe **einmal am Anfang** `import random`. Dann liefert `random.randint(a, b)` eine zufällige ganze Zahl von `a` bis `b`. Anders als bei `range` gehören **beide** Grenzen dazu: `random.randint(1, 6)` ist ein Würfel.

Das Programm würfelt, bis eine Sechs fällt, und zählt die Würfe. Starte die Zelle mehrmals.

```
1 import random
2 wurf = random.randint(1, 6)
3 anzahl = 1
4 print("Gewürfelt:", wurf)
5 while wurf != 6:
6     wurf = random.randint(1, 6)
7     anzahl += 1
8     print("Gewürfelt:", wurf)
9 print("Endlich eine Sechs! Anzahl der Würfe:", anzahl)
```

```
Gewürfelt: 1
Gewürfelt: 3
Gewürfelt: 5
Gewürfelt: 5
Gewürfelt: 6
Endlich eine Sechs! Anzahl der Würfe: 5
```

- **Zeilen 2–4:** Der erste Wurf wird vor der Schleife ausgeführt, gezählt und ausgegeben.
- **Zeilen 6–8:** Neu würfeln, Zähler erhöhen, ausgeben. Durch das neue Würfeln kann sich die Bedingung ändern.

6 for oder while?

🔮 **Vorhersage:** Geben beide Zellen dasselbe aus?

```
for i in range(1, 4):
    print(i)
```

```
1
2
3
```

```
1 i = 1
2 while i <= 3:
3     print(i)
4     i = i + 1
```

```
1
2
3
```

Ja. Die for-Schleife ist aber kürzer und sicherer: Startwert, Bedingung und Erhöhung stecken alle in `range(1, 4)`. Für „würfle bis zur Sechs“ passt for nicht, weil `range` eine feste Anzahl braucht. Die entscheidende Frage: **Steht beim Start der Schleife fest, wie oft wiederholt wird?** (Das gilt auch, wenn die Anzahl vorher eingegeben wurde.)

Situation	Schleife	Begründung
Zahlen von 1 bis 100 ausgeben	`for`	genau 100 Durchläufe
Summe von 1 bis `n`, `n` wird eingegeben	`for`	beim Start der Schleife steht `n` fest
Zahlen addieren, bis 0 eingegeben wird	`while`	Ende hängt von der Eingabe ab
Eingabe wiederholen, bis sie gültig ist	`while`	Anzahl der Fehlversuche unbekannt
würfeln, bis eine Sechs fällt	`while`	Ende hängt vom Zufall ab

> 📌 **Merksatz:** Anzahl der Durchläufe vorher bekannt → `for`. Ende hängt von einer Bedingung oder Eingabe ab → `while`.

✏️ Übung 2 – for oder while?

Entscheide und begründe kurz. Du musst die Programme nicht schreiben.

Situation	for oder while?	Begründung
a) 20-mal „Hallo“ ausgeben		
b) Wörter einlesen, bis „stopp“ eingegeben wird		
c) das Einmaleins einer eingegebenen Zahl ausgeben		
d) 100 € wachsen jedes Jahr um 5 %: Nach wie vielen Jahren sind es mindestens 200 €?		

📝 Übungen: Schritt für Schritt selbstständig

Das **Musterbeispiel** ist ganz vorgemacht. Bei Übung 3 füllst du Lücken `___`, bei Übung 4 steht nur noch ein Gerüst aus Kommentaren, die Zusatzaufgabe schreibst du allein.

Schritt	Frage
1	**Vor der Schleife:** Welche Variablen brauchen einen Startwert? Welche erste Eingabe?
2	**Bedingung:** Wann muss die Schleife **noch einmal** laufen?
3	**Rumpf:** Was passiert pro Durchlauf? Was ändert die Bedingung (neue Eingabe, Erhöhung)?
4	**Nach der Schleife:** Was wird einmal am Ende ausgegeben?

🧩 Musterbeispiel (vorgemacht) – Wörter zählen

Aufgabe: Das Programm liest Wörter ein, bis „stopp“ eingegeben wird. Am Ende gibt es aus, wie viele Wörter es waren (ohne „stopp“).

1. **Vor der Schleife:** Zähler `anzahl = 0`, erstes Wort einlesen.
2. **Bedingung:** noch einmal, solange das Wort nicht „stopp“ ist: `wort != "stopp"`.
3. **Rumpf:** Zähler um 1 erhöhen, **am Ende** das nächste Wort einlesen.
4. **Nach der Schleife:** die Anzahl ausgeben.

```python
# Schritt 1: vor der Schleife
anzahl = 0
wort = input("Wort (stopp zum Beenden): ")

# Schritt 2: Bedingung
while wort != "stopp":
    # Schritt 3: Rumpf
    anzahl += 1
    wort = input("Wort (stopp zum Beenden): ")

# Schritt 4: nach der Schleife
print("Du hast", anzahl, "Wörter eingegeben.")
```

```
Wort (stopp zum Beenden): Informatik
Wort (stopp zum Beenden): Python
Wort (stopp zum Beenden): stopp
Du hast 2 Wörter eingegeben.
```

Testfälle: Hund, Katze, Maus, stopp → 3 Wörter. Sofort `stopp` → 0 Wörter, der Rumpf läuft kein einziges Mal.

📝 Übung 3 – Zahl erraten (Lücken füllen)

Die geheime Zahl ist im Code festgelegt. Man darf so lange raten, bis sie stimmt. Nach jedem Versuch kommt „zu hoch“ oder „zu niedrig“. Ersetze jedes `___`.

Kontrolle: 50 → zu hoch, 30 → zu niedrig, 42 → Richtig geraten!

Lückencode

```python
geheim = 42
versuch = int(input("Rate die Zahl: "))
while versuch ___ geheim:
    if versuch < geheim:
        print(___)
    else:
        print(___)
    versuch = ___
print("Richtig geraten!")
```

📝 Übung 4 – Quersumme (mit Gerüst)

Berechne mit einer while-Schleife die Quersumme einer eingegebenen Zahl. Beispiel: Die Quersumme von 384 ist 3 + 8 + 4 = 15.

> 💡 **Tipp:** `384 % 10` ergibt `4` (die letzte Ziffer), `384 // 10` ergibt `38` (die letzte Ziffer ist weg). Mit `38` bekommst du `8` und `3`, mit `3` bekommst du `3` und `0`. Bei `0` ist keine Ziffer mehr übrig: Das ist deine Abbruchbedingung.

Ersetze die Kommentare durch Programmzeilen. Weitere Kontrollwerte: 7 → 7, 1000 → 1, 99999 → 45.

```python
# Zahl einlesen
# Quersumme mit Startwert 0
# solange noch Ziffern übrig sind:
#     letzte Ziffer zur Quersumme addieren
#     letzte Ziffer abschneiden
# Quersumme ausgeben
```

🚀 Zusatzaufgabe – Ratespiel mit Zufallszahl (selbstständig)

Erweitere dein Programm aus Übung 3:

1. Die geheime Zahl wird mit `random.randint` zufällig zwischen 1 und 100 gewählt.
2. Das Programm zählt die Versuche und gibt am Ende aus, wie viele es waren.

> 💡 **Tipp zum Testen:** Lass die geheime Zahl vorübergehend mit `print(geheim)` ausgeben und lösche die Zeile danach.

📌 Das Wichtigste in Kürze

- `while Bedingung:` wiederholt den Rumpf, **solange** die Bedingung wahr ist. Sie wird **vor** jedem Durchlauf geprüft.
- Startwerte setzt du selbst, und im Rumpf muss sich die Bedingung ändern können, sonst **Endlosschleife**.
- Muster mit Eingaben: erste Eingabe vor der Schleife, nächste Eingabe am Ende des Rumpfs.
- `import random`, dann `random.randint(a, b)`: Zufallszahl von `a` bis `b`, beide Grenzen inklusive.
- Anzahl bekannt → `for`, Ende hängt von Bedingung ab → `while`.

Zum Weiterüben: `DS06_Weitere_Uebungen.ipynb`.

DS 6 – Weitere Übungen

Die while-Schleife: Wiederholen, solange eine Bedingung gilt

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: In Viererschritten ●○○

Lies das Programm, **ohne** es auszuführen.

1. Schreibe auf, was es ausgibt.
2. Gib an, wie oft der Schleifenrumpf ausgeführt wird und welchen Wert `zahl` am Ende hat.

```
zahl = 20
while zahl > 5:
    print(zahl)
    zahl = zahl - 4
print("fertig")
```

```
20
16
12
8
fertig
```

Deine Antwort:

…

Aufgabe 2: Countdown ●○○

Das Programm soll eine Startzahl einlesen und von dort bis 1 herunterzählen. Danach soll es „Start!“ ausgeben. Ergänze die beiden Lücken `___` und teste mit `5` und mit `0`.

Lückencode

```
zahl = int(input("Starte bei: "))
while ___:
    print(zahl)
    ___
print("Start!")
```

Aufgabe 3: Anzahl der Spieler ●○○

Ein Brettspiel ist für 2 bis 4 Personen. Schreibe ein Programm, das die Anzahl der Spielerinnen und Spieler so lange abfragt, bis eine gültige Zahl eingegeben wird. Bei einer ungültigen Eingabe soll es einen Hinweis ausgeben. Am Ende soll es `Das Spiel startet mit 3 Personen.` ausgeben (mit der eingegebenen Zahl).

Aufgabe 4: Endlosschleife oder nicht? ●●○

Entscheide für jedes Programm, ob die Schleife irgendwann endet oder ob eine Endlosschleife entsteht. Begründe kurz. Führe die Programme **nicht** aus.

a)

```
i = 1
while i < 5:
    i = i + 1
```

b)

```
i = 10
while i > 0:
    i = i + 1
```

Nach 5 Sekunden abgebrochen: Die Schleife hört nicht auf (Endlosschleife). In Jupyter stoppst du sie mit ■ (Interrupt).

c)

```
i = 1
while i != 10:
    i = i + 2
```

Nach 5 Sekunden abgebrochen: Die Schleife hört nicht auf (Endlosschleife). In Jupyter stoppst du sie mit ■ (Interrupt).

d)

```
i = 0
while i < 3:
    print(i)
i = i + 1
```

```
0
0
0
0
0
0
0
0
0
0
0
0
…
```

Hier abgebrochen: Die Ausgabe hört nicht auf (Endlosschleife). In Jupyter stoppst du sie mit ■ (Interrupt).

Deine Antwort:

...

Aufgabe 5: Fehler finden: Wörter zählen ●●○

Das Programm soll so lange Wörter einlesen, bis `stopp` eingegeben wird. Danach soll es ausgeben, wie viele Wörter eingegeben wurden (ohne `stopp`). Es enthält **drei Fehler**.

```
anzahl = 0
wort = input("Wort (stopp zum Beenden): ")
while wort == "stopp":
    anzahl = anzahl + 1
wort = input("Wort (stopp zum Beenden): ")
print("Du hast", anzahl "Wörter eingegeben.")
```

```
  File "<Zelle 7>", line 6
    print("Du hast", anzahl "Wörter eingegeben.")
                            ^^^^^^^^^^^^^^^^^^^^
SyntaxError: invalid syntax
```

1. Finde die drei Fehler. Notiere jeweils die Zeile und was falsch ist.
2. Schreibe das Programm richtig auf. Teste es mit den Eingaben `Hund`, `Katze`, `Maus`, `stopp` und auch mit sofort `stopp`.

Deine Antwort:

…

Aufgabe 6: Ablaufprotokoll: Stellen zählen ●●○

Das Programm zählt, wie viele Stellen eine Zahl hat. Fülle das Ablaufprotokoll aus (– bedeutet: noch kein Wert) und gib an, was das Programm ausgibt.

```
zahl = 4096
stellen = 0
while zahl > 0:
    zahl = zahl // 10
    stellen = stellen + 1
print("Stellen:", stellen)
```

```
Stellen: 4
```

Schritt	Zeile	zahl	stellen	Schritt	Zeile	zahl	stellen
1	1			9	3		
2	2			10	4		
3	3			11	5		
4	4			12	3		
5	5			13	4		
6	3			14	5		
7	4			15	3		
8	5			16	6		

Deine Antwort:

…

Aufgabe 7: Papier falten ●●○

Ein Blatt Papier ist etwa 0,1 mm dick. Faltet man es einmal, ist der Stapel doppelt so dick. Stell dir vor, man könnte das Blatt beliebig oft falten.

1. Schreibe ein Programm, das berechnet, wie oft man falten müsste, bis der Stapel mindestens 1 Meter (1000 mm) dick ist. Gib die Anzahl der Faltungen und die Dicke aus.
2. Begründe, warum hier eine while-Schleife passt.

Deine Antwort:

…

Aufgabe 8: Die 3n+1-Folge ●●●

Mit einer Startzahl wird immer wieder so gerechnet:

- Ist die Zahl gerade, wird sie halbiert.
- Ist sie ungerade, wird sie mit 3 multipliziert und dann 1 addiert.

Das wird wiederholt, bis die Zahl 1 erreicht ist. Aus 6 wird zum Beispiel 3, dann 10, 5, 16, 8, 4, 2 und 1. Das sind 8 Schritte.

1. Schreibe ein Programm, das eine Startzahl einliest, jede neue Zahl ausgibt und am Ende die Anzahl der Schritte. Teste mit 6 und mit 7.
2. Was passiert bei der Startzahl `0`? Erkläre.

Deine Antwort:

…

DS 7 – Übung: for- oder while-Schleife?

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- für eine Aufgabe begründet zwischen for- und while-Schleife wählen,
- ein Programm mit Schleife lesen und seine Ausgabe vorhersagen,
- Summen und Produkte mit einer Schleife berechnen,
- Eingaben mit einer while-Schleife wiederholen lassen,
- meine Programme mit sinnvoll gewählten Testfällen überprüfen.

🗣️ Sprachhilfe

- *„Ich nehme for, weil das besser ist."* → „Ich verwende eine for-Schleife, weil die Anzahl der Durchläufe beim Start feststeht: Es werden genau `n` Zahlen multipliziert."
- *„Ich nehme while, weil man nicht weiß, wie oft."* → „Ich verwende eine while-Schleife, weil das Ende von der Eingabe abhängt: Sie läuft, solange das Passwort falsch ist."
- *„Es hört auf."* → „Die Schleife terminiert, weil `zahl` in jedem Durchlauf größer wird."

Worum es geht

Du kennst zwei Schleifenarten. Heute übst du, beide sicher einzusetzen. Bei jeder Aufgabe überlegst du **zuerst**, welche Schleifenart besser passt, und begründest deine Wahl. Erst dann programmierst du.

1 Das brauchst du

	for-Schleife	while-Schleife
Wann?	Anzahl der Durchläufe steht beim Start fest	Ende hängt von einer Bedingung ab
Kopf	`for i in range(1, 6):`	`while zahl != 0:`
Startwert	steckt in `range`	vor der Schleife selbst setzen
Veränderung	automatisch	im Rumpf selbst programmieren
Gefahr	Stoppwert gehört nicht dazu	Endlosschleife

```python
for i in range(1, 4):
    print("Durchlauf", i)

zahl = 1
while zahl < 10:
    print("zahl ist", zahl)
    zahl = zahl * 2
```

```
Durchlauf 1
Durchlauf 2
Durchlauf 3
zahl ist 1
zahl ist 2
zahl ist 4
zahl ist 8
```

📌 **Merksatz: for-Schleife:** Die Anzahl der Durchläufe steht vorher fest. **while-Schleife:** Das Ende hängt von einer Bedingung ab. Überlege bei jeder Aufgabe **zuerst**, welche Schleifenart passt!

Zwei Muster brauchst du immer wieder:

- **Zusammenzählen:** Summenvariable **vor** der Schleife auf `0`, **in** der Schleife erhöhen, **nach** der Schleife ausgeben.
- **Eingaben wiederholen:** erste Eingabe **vor** der while-Schleife, Bedingung prüft die Eingabe, nächste Eingabe **am Ende** des Rumpfs.

💡 **So testest du ein Programm mit Schleife:** nicht nur eine „normale“ Eingabe, sondern auch die kleinste erlaubte Zahl (oft 0 oder 1), eine Eingabe, bei der die Schleife **gar nicht** läuft, und eine, bei der sie sehr oft läuft. Rechne das erwartete Ergebnis **vorher** von Hand aus.

🔮 Übung 1 – Programm lesen

Lies das Programm, **ohne** es auszuführen.

1. Was gibt es aus?
2. Wie oft wird der Schleifenrumpf ausgeführt? Lege dazu ein Ablaufprotokoll an.
3. Warum wird hier eine while-Schleife verwendet?

```
zahl = 1
while zahl < 50:
    zahl = zahl * 3
print(zahl)
```

```
81
```

Schritt	Zeile	Anweisung	`zahl`
1	1	`zahl = 1`	`1`
2	2	`while zahl < 50:`	
3	3	`zahl = zahl * 3`	
…			

Meine Antworten:

…

📝 Übungen: Schritt für Schritt selbstständig

Das **Musterbeispiel** ist ganz vorgemacht. Bei Übung 2 füllst du Lücken `___`, bei Übung 3 steht nur noch ein Gerüst aus Kommentaren, Übung 4 und die Zusatzaufgabe schreibst du allein.

Schritt	Frage
0	**Schleifenart wählen und begründen:** Steht beim Start fest, wie oft wiederholt wird?
1	**Vor der Schleife:** Welche Variablen brauchen einen Startwert? Welche erste Eingabe?
2	**Schleifenkopf:** `range(...)` oder Bedingung?
3	**Rumpf:** Was passiert pro Durchlauf?
4	**Nach der Schleife:** Was wird ausgegeben? Testfälle mit Grenzfällen.

🧩 Musterbeispiel (vorgemacht) – Notendurchschnitt

Aufgabe: Das Programm fragt, wie viele Noten es sind. Dann liest es die Noten ein und gibt den Durchschnitt aus.

0. **Begründung:** Eine for-Schleife passt. Sobald die Anzahl `n` eingegeben ist, steht fest, dass genau `n` Noten eingelesen werden.
1. **Vor der Schleife:** `n` einlesen, Summenvariable `summe = 0`.
2. **Schleifenkopf:** `for i in range(1, n + 1):`, damit `i` die Nummer der Note ist (1 bis `n`).
3. **Rumpf:** eine Note einlesen und zur Summe addieren.
4. **Nach der Schleife:** Durchschnitt `summe / n` ausgeben, auf zwei Stellen gerundet.

```python
# Schritt 1: vor der Schleife
n = int(input("Wie viele Noten? "))
summe = 0

# Schritt 2 und 3: Schleifenkopf und Rumpf
for i in range(1, n + 1):
    note = int(input("Note " + str(i) + ": "))
    summe += note

# Schritt 4: nach der Schleife
print("Durchschnitt:", round(summe / n, 2))
```

```
Wie viele Noten? 3
Note 1: 2
Note 2: 3
Note 3: 1
Durchschnitt: 2.0
```

Testfälle: Noten 2, 3, 1 → Durchschnitt 2.0. Eine einzige Note 4 → 4.0 (die Schleife läuft genau einmal). Bei `n = 0` bricht das Programm mit `ZeroDivisionError` ab: Wie könnte man das mit einer Verzweigung abfangen?

📝 Übung 2 – Passwortabfrage (Lücken füllen)

Frage so lange nach einem Passwort, bis `informatik10` eingegeben wurde. Gib danach „Zugang gewährt“ aus.

1. **Begründe zuerst:** Welche Schleifenart passt?
2. Ersetze jedes `___`.

Testfälle: sofort richtig; `geheim`, dann `Informatik10`, dann `informatik10` (Python unterscheidet Groß- und Kleinschreibung).

Begründung: …

Lückencode

```python
passwort = ___("Passwort eingeben: ")
while passwort ___ "informatik10":
    passwort = ___("Falsch! Erneut versuchen: ")
print(___)
```

📝 Übung 3 – Fakultät (mit Gerüst)

In der Mathematik schreibt man *n*! („n Fakultät“) für das Produkt aller ganzen Zahlen von 1 bis *n*: 4! = 1 · 2 · 3 · 4 = 24. Für 0! legt man den Wert 1 fest. Die Fakultät gibt zum Beispiel an, auf wie viele Arten man *n* Personen in einer Reihe aufstellen kann.

1. **Begründe zuerst:** Welche Schleifenart passt?
2. Ersetze die Kommentare durch Programmzeilen.

> 💡 **Tipp:** Die Fakultät berechnest du fast wie eine Summe, nur wird multipliziert. Mit welchem Wert muss die Produktvariable **vor** der Schleife starten? Was passiert mit 0?

Kontrolle: 5 → `5! = 120`, 1 → `1! = 1`, 0 → `0! = 1`, 10 → `10! = 3628800`.

Begründung: …

```
# n als Ganzzahl einlesen
# Produktvariable mit passendem Startwert
# Schleife: i von 1 bis einschließlich n
#     Produkt mit i multiplizieren
# Ergebnis ausgeben, z. B. 5! = 120
```

📝 Übung 4 – Primzahl-Test (selbstständig)

Eine **Primzahl** ist eine natürliche Zahl größer als 1, die nur durch 1 und durch sich selbst teilbar ist: 2, 3, 5, 7, 11, 13, … Die 9 ist keine, weil sie durch 3 teilbar ist. Die 1 ist nach Festlegung keine Primzahl.

Prüfe, ob eine eingegebene Zahl eine Primzahl ist. Teste dazu, ob sie durch eine Zahl zwischen 2 und sich selbst minus 1 teilbar ist (`zahl % teiler == 0`).

> 💡 **Tipp:** Arbeite mit einer Variablen, die sich ein Ergebnis merkt. Nimm zuerst an, die Zahl **ist** eine Primzahl: `ist_primzahl = True`. Findest du einen Teiler, war die Annahme falsch. Denke an den Sonderfall 1.

Lege zuerst die Testfälle fest:

Eingabe	erwartete Ausgabe	Warum testen?
`7`		
`9`		
`2`		
`1`		
`97`		

🚀 Zusatzaufgabe – Sparziel (selbstständig)

Auf einem Sparkonto liegen 100 €. Jedes Jahr kommen 5 % Zinsen dazu, das Guthaben wird also mit 1.05 multipliziert. Nach wie vielen Jahren beträgt es zum ersten Mal mindestens 200 €? Gib die Anzahl der Jahre und das Guthaben auf Cent gerundet aus. Begründe deine Wahl der Schleifenart.

Für Schnelle: Beim Primzahl-Test prüft die for-Schleife weiter, auch wenn längst ein Teiler gefunden ist. Schreibe eine Fassung mit while, die beim ersten Teiler aufhört.

📌 Das Wichtigste in Kürze

- Erst die Schleifenart wählen und begründen: Steht die Anzahl beim Start fest → `for`, sonst → `while`.
- Summe startet bei `0`, Produkt startet bei `1`.
- Eingaben wiederholen: erste Eingabe vor der Schleife, nächste am Ende des Rumpfs.
- „Annahme-Variable“: `ist_primzahl = True` annehmen, beim Gegenbeispiel auf `False` setzen.
- Testen mit Grenzfällen: Schleife läuft gar nicht, genau einmal, sehr oft.

Zum Weiterüben: `DS07_Weitere_Uebungen.ipynb`.

DS 7 – Weitere Übungen

Übung: for- oder while-Schleife?

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Welche Schleife passt? ●○○

Entscheide jeweils, ob eine for- oder eine while-Schleife besser passt, und begründe kurz. Du musst die Programme nicht schreiben.

Situation	for oder while?
a) das Taschengeld von 12 Monaten zusammenzählen	
b) Zahlen einlesen, bis eine negative Zahl eingegeben wird	
c) alle geraden Zahlen von 2 bis 50 ausgeben	
d) so lange nachfragen, bis `ja` oder `nein` eingegeben wird	
e) eine eingegebene Zahl so oft halbieren, bis sie kleiner als 1 ist	
f) für jede der 25 Personen einer Klasse eine Note einlesen	

Deine Antwort:

…

Aufgabe 2: Aus for wird while ●○○

Schreibe ein Programm mit einer **while**-Schleife, das genau dasselbe ausgibt wie das folgende Programm. Denke an Startwert, Bedingung und Veränderung der Zählvariablen.

```
for i in range(5, 26, 5):
    print(i)
```

```
5
10
15
20
25
```

Aufgabe 3: Nur die ungeraden ●○○

Lies das Programm, **ohne** es auszuführen.

1. Gib an, was das Programm ausgibt.
2. Gib an, wie oft die Zeile `summe = summe + zahl` ausgeführt wird.

```
summe = 0
for zahl in range(1, 6):
    if zahl % 2 == 1:
        summe = summe + zahl
print(summe)
```

```
9
```

Deine Antwort:

...

Aufgabe 4: Potenzieren ohne ** ●○○

Das Programm soll eine Potenz berechnen, ohne den Operator `**` zu benutzen. Dafür wird die Basis so oft mit sich selbst multipliziert, wie der Exponent angibt: $2^5 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$. Ergänze die drei Lücken `___`. Teste mit Basis `2` und Exponent `10` (Ergebnis 1024) und mit Basis `5` und Exponent `0` (Ergebnis 1).

Lückencode

```
basis = int(input("Basis: "))
exponent = int(input("Exponent: "))
ergebnis = ___
for i in range(___):
    ergebnis = ___
print(basis, "hoch", exponent, "ist", ergebnis)
```

Aufgabe 5: Fehler finden: Vielfache von 7 zählen ●●○

Das Programm soll zählen, wie viele Zahlen von 1 bis 70 durch 7 teilbar sind. Richtig wäre `Anzahl: 10`. Es enthält **drei Fehler**.

```
for zahl in range(1, 70):
    if zahl % 7 = 0:
        anzahl = anzahl + 1
print("Anzahl:", anzahl)
```

```
  File "<Zelle 4>", line 2
    if zahl % 7 = 0:
       ^^^^^^^^
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?
```

1. Finde die drei Fehler. Notiere jeweils die Zeile und was falsch ist.
2. Schreibe das Programm richtig auf.

Deine Antwort:

...

Aufgabe 6: Alle Teiler einer Zahl ●●○

Schreibe ein Programm, das eine ganze Zahl einliest und alle ihre Teiler ausgibt. Am Ende soll es ausgeben, wie viele Teiler es sind. Begründe, welche Schleifenart du verwendest.

So könnte es aussehen:

Ausgabe:

```
Zahl: «12»
1
2
3
4
6
12
Anzahl der Teiler: 6
```

Deine Antwort:

...

Aufgabe 7: Gerade und ungerade zählen ●●○

Schreibe ein Programm, das so lange ganze Zahlen einliest, bis `0` eingegeben wird. Danach soll es ausgeben, wie viele der eingegebenen Zahlen gerade und wie viele ungerade waren. Die `0` zählt nicht mit. Teste mit `4`, `7`, `10`, `3`, `8`, `0` und mit sofort `0`.

Aufgabe 8: Bestweite beim Weitsprung ●●●

Beim Weitsprung werden die Weiten eines Versuchs nach dem anderen eingegeben (in Metern, als Kommazahl). Die Eingabe `0` beendet die Eingabe. Schreibe ein Programm, das danach die **Anzahl der Versuche** und die **beste Weite** ausgibt. Wurde gar keine Weite eingegeben, soll es „Keine Weite eingegeben.“ ausgeben.

So könnte es aussehen:

Ausgabe:

```
Weite in m (0 = Ende): «3.85»
Weite in m (0 = Ende): «4.10»
Weite in m (0 = Ende): «3.92»
Weite in m (0 = Ende): «0»
Versuche: 3
Bestweite: 4.1 m
```

> **Tipp:** Merke dir die bisher beste Weite in einer Variablen `beste`. Weil alle Weiten größer als 0 sind, kann sie bei 0 starten. Bei jeder neuen Weite fragst du: Ist sie größer als die bisher beste? Dann wird sie die neue beste Weite.

DS 8 – Funktionen: Programme aus Bausteinen

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- eine Funktion mit `def` definieren und sie aufrufen,
- Funktionen mit Parametern schreiben und ihnen beim Aufruf Argumente übergeben,
- mit `return` einen Rückgabewert zurückgeben und damit weiterrechnen,
- erklären, warum `print` in einer Funktion etwas anderes ist als `return`,
- beschreiben, was bei einem Funktionsaufruf Schritt für Schritt passiert,
- eine Funktion mit mehreren Aufrufen testen.

🗣️ Sprachhilfe

Beispiel: die Funktion `f` mit dem Parameter `x` und dem Rumpf `return x * x + 1`. Der Aufruf `f(4)` liefert 17.

- *„Zur Eingabe 4 bekomme ich die 17."* → „Wenn ich der Funktion die Zahl 4 gebe, liefert sie mir die 17 als Ergebnis." → „Wenn die Funktion `f` mit dem Wert 4 für den Parameter `x` aufgerufen wird, berechnet sie das Ergebnis 17 und gibt es zurück."
- „Der Funktion wird der Wert 4 übergeben. Die Funktion berechnet 4 · 4 + 1 und gibt 17 zurück."

Worum es geht

Du hast schon viele Funktionen benutzt: `print(...)`, `input(...)`, `int(...)`, `round(...)`. Du schreibst ihren Namen, gibst ihnen in Klammern Werte mit und bekommst ein Ergebnis. Wie sie innen arbeiten, musst du nicht wissen. Heute schreibst du **eigene Funktionen**. So schreibst du eine Rechnung, die du oft brauchst, nur einmal auf. Die Idee kennst du aus Mathe: $f(x) = x^2$ macht aus 4 die 16 und aus 7 die 49.

1 Eine eigene Funktion definieren

Dein Programm soll an mehreren Stellen eine Trennlinie ausgeben. Statt sie jedes Mal abzutippen, gibst du der Anweisung einen Namen.

```
1 def trennlinie():
2     print("--------------------")
3
4 print("Einkaufsliste")
5 trennlinie()
6 print("Milch, Brot, Käse")
7 trennlinie()
```

```
Einkaufsliste
--------------------
Milch, Brot, Käse
--------------------
```

- **Zeile 1:** `def` (englisch *define*, „festlegen") beginnt eine **Funktionsdefinition**: Name, runde Klammern, **Doppelpunkt**. Die Klammern sind leer, weil die Funktion keine Werte braucht.
- **Zeile 2:** Die eingerückten Zeilen unter `def` bilden den **Funktionsrumpf**, wie bei `if` und `for`.
- **Zeile 4:** Hier beginnt das **Hauptprogramm**. Beim Lesen von Zeile 1 und 2 hat Python noch **nichts ausgegeben**, es hat sich nur gemerkt, was `trennlinie` bedeutet.

- **Zeilen 5 und 7:** `trennlinie()` ist ein **Funktionsaufruf**. Erst jetzt wird der Rumpf ausgeführt, und das zweimal, obwohl die Linie nur einmal im Programm steht.

Eine Funktion ist wie ein **Rezept**: Die Definition schreibt das Rezept auf, davon wird noch niemand satt. Erst beim Aufruf wird gekocht, so oft man möchte.

> 📌 **Merksatz:** Eine **Funktion** fasst eine Folge von Anweisungen unter einem Namen zusammen und macht sie wiederverwendbar. **Definition:** `def funktionsname(parameter1, parameter2):` gefolgt vom eingerückten **Funktionsrumpf**.

⚠️ Stolpersteine

Führe beide Zellen aus und lies die letzte Zeile der Fehlermeldung.

```
def trennlinie():
print("--------------------")
```

```
  File "<Zelle 2>", line 2
    print("--------------------")
    ^^^^^
IndentationError: expected an indented block after function definition on line 1
```

```
1 trennlinie2()
2
3 def trennlinie2():
4     print("--------------------")
```

```
Traceback (most recent call last):
  File "<Zelle 3>", line 1, in <module>
NameError: name 'trennlinie2' is not defined
```

1. `IndentationError: expected an indented block after function definition on line 1`: Nach `def` wird ein eingerückter Block erwartet. Fehlt der Doppelpunkt, kommt `SyntaxError: expected ':'`.
2. `NameError: name 'trennlinie2' is not defined`: Python liest von oben nach unten. Eine Funktion muss **definiert sein, bevor** sie aufgerufen wird. Schreibe die Definitionen deshalb an den Anfang. Und vergiss beim Aufruf die Klammern nicht.

2 Parameter: der Funktion Werte mitgeben

Eine Begrüßung soll jedes Mal einen anderen Namen enthalten. Dafür schreibst du in die Klammern einen **Parameter**.

```
1 def gruesse(name):
2     print("Hallo, " + name + "!")
3
4 gruesse("Alex")
5 gruesse("Sam")
```

```
Hallo, Alex!
Hallo, Sam!
```

- **Zeile 1:** `name` ist ein **Parameter**: eine Variable der Funktion, die ihren Wert erst beim Aufruf bekommt, wie eine noch leere Schachtel.
- **Zeile 4:** `"Alex"` ist das **Argument**. Python legt es in den Parameter, als stünde dort `name = "Alex"`, und führt dann den Rumpf aus.

Mehrere Parameter

Mehrere Parameter werden durch Kommas getrennt. Die Argumente werden **der Reihe nach** verteilt. 🔮
Vorhersage: Was gibt die letzte Zeile aus?

```
1 def gruesse(gruss, name):
2     print(gruss + ", " + name + "!")
3 
4 gruesse("Hallo", "Alex")
5 gruesse("Servus", "Sam")
6 gruesse("Sam", "Moin")
```

```
Hallo, Alex!
Servus, Sam!
Sam, Moin!
```

In Zeile 6 sind die Argumente vertauscht. Python bemerkt das nicht: `gruss` bekommt `"Sam"`, `name` bekommt `"Moin"`.

> 📌 **Merksatz: Parameter** sind die „Eingabewerte“ einer Funktion. Beim Aufruf werden ihnen konkrete **Argumente** übergeben, und zwar in derselben Reihenfolge.

⚠️ Stolperstein: falsche Anzahl

Führe die Zelle aus. Was passiert bei `gruesse("Alex", "Sam")`?

```
1 def gruesse(name):
2     print("Hallo, " + name + "!")
3 
4 gruesse()
```

```
Traceback (most recent call last):
  File "<Zelle 6>", line 4, in <module>
TypeError: gruesse() missing 1 required positional argument: 'name'
```

`TypeError: gruesse() missing 1 required positional argument: 'name'` heißt: „Der Funktion fehlt 1 benötigtes Argument.“ Bei zu vielen Argumenten: `takes 1 positional argument but 2 were given`.

🧪 **Probiere es aus:** Rufe `gruesse` mit deinem eigenen Namen auf. Was passiert bei `gruesse(15)`? Erkläre die Fehlermeldung mit dem, was du über das Verketten mit `+` weißt.

3 Rückgabewert: die Funktion liefert ein Ergebnis

`gruesse` gibt etwas auf dem Bildschirm aus, danach ist das Ergebnis für das Programm verloren. Bei `round` ist das anders: `zahl = round(3.14159, 2)` **gibt** `3.14` **zurück**, und es landet in `zahl`. Das geht auch mit eigenen Funktionen, mit `return`.

```
1 def quadrat(zahl):
2     ergebnis = zahl * zahl
3     return ergebnis
4 
5 print(quadrat(4))
6 print(quadrat(7))
7 
8 x = quadrat(10)
9 print("x ist jetzt:", x)
```

```
16
49
x ist jetzt: 100
```

- **Zeile 3:** `return ergebnis` gibt den Wert an die Stelle zurück, an der die Funktion aufgerufen wurde: der **Rückgabewert**.
- **Zeile 5:** Von innen nach außen lesen: `quadrat(4)` liefert `16`, der Aufruf wird durch diesen Wert ersetzt. Es steht gewissermaßen `print(16)` da.
- **Zeile 8:** Diesmal wird der Rückgabewert `100` in `x` gespeichert.

Eine Funktion mit Parametern und Rückgabewert ist wie eine **Maschine**: Oben wirft man das Argument 4 hinein, die Maschine arbeitet, unten kommt 16 heraus.

Weil ein Aufruf wie `quadrat(3)` am Ende für einen Wert steht, darf er überall stehen, wo ein Wert stehen darf: in Rechnungen, Bedingungen oder als Argument. Mit `return` **endet die Funktion sofort**. Ein Rückgabewert muss auch keine Zahl sein.

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

summe = quadrat(3) + quadrat(4)
print("3*3 + 4*4 =", summe)
if quadrat(5) > 20:
    print("5*5 ist größer als 20.")

def begruessung(name, gruss):
    return gruss + ", " + name + "!"

print(begruessung("Alex", "Hallo"))
```

```
3*3 + 4*4 = 25
5*5 ist größer als 20.
Hallo, Alex!
```

📌 **Merksatz:** Mit `return wert` gibt eine Funktion einen **Rückgabewert** an die aufrufende Stelle zurück und beendet die Funktion. Der **Funktionsaufruf** (z. B. `ergebnis = quadrat(4)`) ist zu unterscheiden von der **Funktionsdefinition** (`def quadrat(zahl):` mit Rumpf).

🔮 Übung 1 – Vorhersage

Lies das Programm, **ohne** es auszuführen. Schreibe auf, was es ausgibt. Überprüfe danach.

```
def verdopple(zahl):
    return 2 * zahl

a = verdopple(3)
b = verdopple(a) + 1
print(a, b)
print(verdopple(verdopple(5)))
```

```
6 13
20
```

Meine Vorhersage:

…

4 Was beim Aufruf passiert

Ein Funktionsaufruf unterbricht die Reihenfolge von oben nach unten: Python **springt** in die Funktion, führt den Rumpf aus und **springt** mit dem Rückgabewert **zurück**.

```python
def quadrat(zahl):
    ergebnis = zahl * zahl
    return ergebnis

x = quadrat(10)
y = quadrat(3) + 1
print(x, y)
```

```
100 10
```

1. **Zeile 1:** Python merkt sich die Funktion. Zeilen 2 und 3 werden jetzt **nicht** ausgeführt.
2. **Zeile 5:** Sprung in die Funktion mit `zahl = 10`. Zeile 2 rechnet `ergebnis = 100`, Zeile 3 gibt `100` zurück. `x` bekommt `100`.
3. **Zeile 6:** Sprung mit `zahl = 3`, Rückgabewert `9`. Zurück in Zeile 6: `9 + 1 = 10` wird in `y` gespeichert.
4. **Zeile 7:** Ausgabe von `x` und `y`.

Ablaufprotokoll **innerhalb der Funktion**:

Schritt	Zeile	Anweisung	`zahl`	`ergebnis`
1	2	`ergebnis = zahl * zahl`	`10`	`100`
2	3	`return ergebnis`	`10`	`100`
3	2	`ergebnis = zahl * zahl`	`3`	`9`
4	3	`return ergebnis`	`3`	`9`

Variablen in der Funktion bleiben in der Funktion

`zahl` und `ergebnis` gibt es nur, solange die Funktion läuft. Man nennt sie **lokale Variablen**. Aus der Maschine kommt nur heraus, was mit `return` zurückgegeben wird. Führe die Zelle aus.

```python
def quadrat(zahl):
    ergebnis = zahl * zahl
    return ergebnis

x = quadrat(10)
print(ergebnis)
```

```
Traceback (most recent call last):
  File "<Zelle 11>", line 6, in <module>
NameError: name 'ergebnis' is not defined
```

5 `print` oder `return`?

Der häufigste Fehler mit Funktionen. 🔮 **Vorhersage:** Was steht in `a`, was in `b`?

```python
def quadrat_mit_print(zahl):
    print(zahl * zahl)

def quadrat_mit_return(zahl):
    return zahl * zahl

a = quadrat_mit_print(4)
b = quadrat_mit_return(4)
print("a ist", a)
print("b ist", b)
```

```
16
a ist None
b ist 16
```

- **Zeile 7:** `quadrat_mit_print(4)` zeigt sofort `16` an, hat aber kein `return`. Darum bekommt `a` den besonderen Wert **`None`** (englisch für „nichts“).
- **Zeile 8:** `quadrat_mit_return(4)` zeigt nichts an, liefert aber `16` zurück.

Will man weiterrechnen, bricht die `print`-Variante sogar ab:

```
def quadrat_mit_print(zahl):
    print(zahl * zahl)

summe = quadrat_mit_print(3) + quadrat_mit_print(4)
```

```
9
16
```

```
Traceback (most recent call last):
  File "<Zelle 13>", line 4, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

`TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'`: `None + None` kann Python nicht rechnen. Taucht `NoneType` in einer Fehlermeldung auf, fehlt sehr oft ein `return`.

	`print(...)` in der Funktion	**`return ...`**
Was passiert?	Der Wert wird auf dem Bildschirm angezeigt.	Der Wert wird an die Stelle des Aufrufs zurückgegeben.
Wer bekommt den Wert?	die Person vor dem Bildschirm	das Programm
Was liefert der Aufruf?	`None`	den Wert hinter `return`
Weiterrechnen möglich?	nein	ja

📌 **Merksatz:** Eine Funktion ohne `return` liefert `None` zurück. `print` zeigt einen Wert nur an. Soll das Programm mit dem Ergebnis weiterarbeiten, muss die Funktion es mit `return` zurückgeben.

💡 **Faustregel:** Eine Funktion, die etwas **berechnet**, gibt ihr Ergebnis mit `return` zurück. Das **Ausgeben** mit `print` erledigt das Hauptprogramm.

🔍 Übung 2 – print oder return?

Das Programm soll Minuten in Sekunden umrechnen. Es läuft ohne Fehlermeldung, aber die Ausgabe ist seltsam.

1. Gib an, was das Programm ausgibt.
2. Erkläre, warum die Ausgabe nicht stimmt.
3. Verbessere die Funktion.

```
def minuten_zu_sekunden(minuten):
    print(minuten * 60)

dauer = minuten_zu_sekunden(5)
print("Das sind", dauer, "Sekunden.")
```

```
300
Das sind None Sekunden.
```

Ausgabe und Erklärung:

…

6 Funktionen testen

Eine Funktion kann man beliebig oft mit verschiedenen Argumenten aufrufen. Das ist praktisch zum **Testen**:

1. **Vorher** von Hand ausrechnen, was herauskommen muss.
2. Die Funktion mit diesen Werten aufrufen.
3. Vergleichen. Weicht ein Ergebnis ab, steckt ein Fehler in der Funktion oder in deiner Rechnung.

Teste auch Sonderfälle wie `0`, negative Zahlen oder Kommazahlen. Dort verstecken sich Fehler am häufigsten.

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

print(quadrat(4), "erwartet: 16")
print(quadrat(0), "erwartet: 0")
print(quadrat(-3), "erwartet: 9")
print(quadrat(1.5), "erwartet: 2.25")
print(quadrat(-3) == 9)
```

```
16 erwartet: 16
0 erwartet: 0
9 erwartet: 9
2.25 erwartet: 2.25
True
```

📝 Übungen: Schritt für Schritt selbstständig

Die Übungen werden schrittweise schwieriger. Das **Musterbeispiel** ist ganz vorgemacht. Bei Übung 3 füllst du nur Lücken `___` aus, bei Übung 4 steht nur noch das Gerüst, die Zusatzaufgabe schreibst du allein. Gehe jedes Mal in denselben vier Schritten vor:

Schritt	Frage
1	**Welche Werte** braucht die Funktion? → Parameter
2	**Was soll herauskommen?** → Rückgabewert
3	**Wie rechnet man es aus?** → Rumpf
4	**Stimmt es?** → Testen mit vorher ausgerechneten Werten

🧩 Musterbeispiel (vorgemacht) – Fläche eines Dreiecks

Aufgabe: Schreibe eine Funktion `dreieck_flaeche(grundseite, hoehe)`, die die Fläche eines Dreiecks zurückgibt. Formel: A = g · h / 2.

1. **Welche Werte?** Die Grundseite und die Höhe. Also zwei Parameter: `grundseite` und `hoehe`.
2. **Was soll herauskommen?** Die Fläche, also eine Zahl. Sie wird mit `return` zurückgegeben, nicht mit `print` ausgegeben.
3. **Wie rechnet man?** Die Formel wird fast wörtlich übernommen: `grundseite * hoehe / 2`.
4. **Stimmt es?** Vorher von Hand: g = 4, h = 3 ergibt 4 · 3 / 2 = 6. g = 10, h = 5 ergibt 25. Sonderfall: h = 0 ergibt 0.

Lies den Code Zeile für Zeile und führe ihn aus.

```python
# Schritt 1 und 2: Name, Parameter und Rückgabewert festlegen
def dreieck_flaeche(grundseite, hoehe):
    # Schritt 3: rechnen und das Ergebnis zurückgeben
    flaeche = grundseite * hoehe / 2
    return flaeche

# Schritt 4: testen mit vorher ausgerechneten Werten
print(dreieck_flaeche(4, 3), "erwartet: 6")
print(dreieck_flaeche(10, 5), "erwartet: 25")
print(dreieck_flaeche(7, 0), "erwartet: 0")
```

```
6.0 erwartet: 6
25.0 erwartet: 25
0.0 erwartet: 0
```

Weil `/` immer eine Kommazahl liefert, erscheint `6.0` statt `6`. Das ist richtig.

📝 Übung 3 – Celsius in Fahrenheit (Lücken füllen)

In den USA misst man Temperaturen in Grad Fahrenheit: °F = °C · 9/5 + 32. Die Funktion `celsius_zu_fahrenheit(grad_celsius)` soll den Wert in Fahrenheit **zurückgeben**.

Gehe wie im Musterbeispiel vor:

1. **Welche Werte?** Eine Temperatur in Grad Celsius, der Parameter heißt `grad_celsius`.
2. **Was soll herauskommen?** Die Temperatur in Fahrenheit.
3. **Wie rechnet man?** Übersetze die Formel in Python. Das Malzeichen ist `*`.
4. **Stimmt es?** Die Testaufrufe stehen schon da: Wasser gefriert bei 0 °C = 32 °F und siedet bei 100 °C = 212 °F.

Ersetze jedes `___` durch den richtigen Ausdruck. Solange noch ein `___` dasteht, meldet Python `NameError: name '___' is not defined`.

Lückencode

```python
def celsius_zu_fahrenheit(grad_celsius):
    fahrenheit = ___
    return ___

print(celsius_zu_fahrenheit(0), "erwartet: 32")
print(celsius_zu_fahrenheit(100), "erwartet: 212")
print(celsius_zu_fahrenheit(20), "erwartet: 68")
```

📝 Übung 4 – Maximum von zwei Zahlen (mit Gerüst)

Schreibe eine Funktion `maximum(a, b)`, die die größere von zwei Zahlen zurückgibt. Verwende **nicht** `max()`, sondern eine Verzweigung.

1. **Welche Werte?** Zwei Zahlen, `a` und `b`.
2. **Was soll herauskommen?** Die größere der beiden Zahlen.
3. **Wie rechnet man?** Vergleiche mit `if a > b:` und gib im passenden Zweig `a` oder `b` zurück. Eine Funktion darf mehrere `return` haben.
4. **Stimmt es?** Rechne vorher aus, was bei den Testaufrufen herauskommen muss, und trage es hinter `erwartet:` ein. Teste auch mit zwei gleichen Zahlen.

Das Gerüst gibt die Kopfzeile und die Kommentare vor. Ersetze die Kommentare durch Programmzeilen.

```python
def maximum(a, b):
    # wenn a größer als b ist:
    #     a zurückgeben
    # sonst:
    #     b zurückgeben
    pass

print(maximum(3, 9), "erwartet: ")
print(maximum(17, 5), "erwartet: ")
print(maximum(4, 4), "erwartet: ")
```

```
None erwartet:
None erwartet:
None erwartet:
```

🚀 Zusatzaufgabe – Gerade Zahlen (selbstständig)

1. Schreibe eine Funktion `ist_gerade(zahl)`, die `True` zurückgibt, wenn die Zahl gerade ist, sonst `False`. Der Rückgabewert ist also vom Typ `bool`.
2. Nutze die Funktion in einer `for`-Schleife, um alle geraden Zahlen von 1 bis 20 auszugeben.

Gehe wieder in den vier Schritten vor. Hinweis: Eine Zahl ist gerade, wenn beim Teilen durch 2 der Rest `0` bleibt (`%`).

📌 Das Wichtigste in Kürze

- Eine **Funktion** fasst Anweisungen unter einem Namen zusammen: `def name(parameter):` mit eingerücktem Rumpf. Ausgeführt wird sie erst beim **Aufruf**.
- **Parameter** bekommen beim Aufruf die **Argumente**, der Reihe nach.
- `return wert` gibt einen **Rückgabewert** zurück und beendet die Funktion. Ohne `return` liefert sie `None`.
- `print` zeigt nur an, `return` gibt dem Programm den Wert. Wer berechnet, gibt zurück.
- Variablen in der Funktion sind **lokal**.
- Testen: vorher das erwartete Ergebnis ausrechnen, auch Sonderfälle wie `0` und negative Zahlen prüfen.

Zum Weiterüben: `DS08_Weitere_Uebungen.ipynb`.

DS 8 – Weitere Übungen

Funktionen: Programme aus Bausteinen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Fachbegriffe am Beispiel ●○○

Am Kiosk kostet jede Pfandflasche 25 Cent Pfand zusätzlich zum Preis. Lies das Programm:

```
def preis_mit_pfand(preis, flaschen):
    return preis + flaschen * 0.25

gesamt = preis_mit_pfand(3.5, 6)
print(gesamt)
```

```
5.0
```

Nenne zu diesem Programm:

1. den Namen der Funktion,
2. die Parameter,
3. die Argumente beim Aufruf,
4. die Zeile, in der die Funktion aufgerufen wird,
5. den Rückgabewert für diesen Aufruf.

Deine Antwort:

…

Aufgabe 2: Rückgabewerte vorhersagen ●○○

Gegeben ist die Funktion `rechne`:

```
def rechne(x):
    return 3 * x - 1
```

Bestimme **ohne Computer**, welchen Wert die Aufrufe zurückgeben. Prüfe danach am Computer, z. B. mit `print(rechne(2))`.

Aufruf	Rückgabewert
`rechne(2)`	
`rechne(0)`	
`rechne(-1)`	
`rechne(10)`	
`rechne(rechne(1))`	

Aufgabe 3: Eintritt im Freibad ●○○

Im Freibad kostet der Eintritt 4 Euro. Für jede Stunde, die man bleibt, kommen 2 Euro dazu.

1. Implementiere eine Funktion `eintritt(stunden)`, die den Preis in Euro **zurückgibt**.
2. Teste sie mit den Werten aus der Tabelle. Rechne die erwarteten Werte vorher von Hand aus.

Aufruf	erwartet	tatsächlich
`eintritt(1)`		
`eintritt(3)`		
`eintritt(0)`		

Aufgabe 4: Drei Fehler ●○○

Die Funktion soll die Fläche eines Dreiecks berechnen (Grundseite mal Höhe durch 2). Das Programm enthält **drei Fehler**. Finde sie und schreibe das Programm richtig auf. Teste es: Für die Grundseite 4 und die Höhe 3 muss `6.0` herauskommen.

```
def flaeche_dreieck(g, h)
return g * h / 2

print(flaeche_dreieck(4))
```

```
  File "<Zelle 3>", line 1
    def flaeche_dreieck(g, h)
                              ^
SyntaxError: expected ':'
```

Deine Antwort:

…

Aufgabe 5: Ein Test deckt den Fehler auf ●●○

Die Funktion soll den Mittelwert von zwei Zahlen zurückgeben. Sie läuft ohne Fehlermeldung.

```
def mittelwert(a, b):
    return a + b / 2
```

1. Rechne von Hand aus, was für `mittelwert(4, 6)` herauskommen **muss**.
2. Teste die Funktion mit `mittelwert(4, 6)` und mit `mittelwert(0, 0)`. Notiere die Ergebnisse.
3. Erkläre den Fehler und verbessere die Funktion.
4. Erkläre, warum der Test mit `0` und `0` allein den Fehler nicht gefunden hätte.

Deine Antwort:

…

Aufgabe 6: Kinokarten ●●○

Ein Kino verlangt je nach Alter einen anderen Preis:

- unter 6 Jahren: 0 Euro (frei),
- von 6 bis unter 18 Jahren: 6 Euro,
- ab 18 Jahren: 9 Euro.

1. Implementiere eine Funktion `kinopreis(alter)`, die den Preis **zurückgibt**. Verwende eine mehrseitige Verzweigung.
2. Teste die Funktion an den Grenzen: mit den Altern 5, 6, 17 und 18.

Aufgabe 7: Funktion in der Schleife ●●○

Fülle das Ablaufprotokoll des Hauptprogramms aus. Trage nach jedem Schritt die Werte von `i` und `summe` ein (– bedeutet: noch kein Wert). Gib dann an, was das Programm ausgibt.

```python
def dreifach(zahl):
    return 3 * zahl

summe = 0
for i in range(1, 4):
    summe = summe + dreifach(i)
print("Summe:", summe)
```

```
Summe: 18
```

Schritt	Zeile	Anweisung	i	summe
1	1	`def dreifach(zahl):`	-	-
2	4	`summe = 0`	-	
3	5	`for i in range(1, 4):`		
4	6	`summe = summe + dreifach(i)`		
5	5	`for i in range(1, 4):`		
6	6	`summe = summe + dreifach(i)`		
7	5	`for i in range(1, 4):`		
8	6	`summe = summe + dreifach(i)`		
9	5	`for i in range(1, 4):`		
10	7	`print("Summe:", summe)`		

Deine Antwort:

...

Aufgabe 8: Quersumme als Funktion ●●●

Im Auftrag *Quersumme* (Abschnitt über die while-Schleife) hast du die Quersumme einer Zahl berechnet (Quersumme von 384 ist 3 + 8 + 4 = 15). Zur Erinnerung: `zahl % 10` liefert die letzte Ziffer, `zahl // 10` entfernt sie.

1. Implementiere eine Funktion `quersumme(zahl)`, die die Quersumme **zurückgibt**. Teste sie mit 384 (erwartet: 15), 7 (erwartet: 7) und 1000 (erwartet: 1).
2. Benutze die Funktion in einer `for`-Schleife: Gib alle Zahlen von 1 bis 100 aus, deren Quersumme genau 10 ist.

> **Tipp:** Schreibe die Schleife aus dem Abschnitt über die while-Schleife in den Funktionsrumpf und ersetze das `print` am Ende durch `return`. Im zweiten Teil steht der Aufruf `quersumme(i)` in der Bedingung einer `if`-Anweisung, so wie `ist_gerade(i)` im Auftrag *Gerade Zahlen*.

DS 9 – Übungsstunde: Funktionen lesen, testen und erweitern

Informatik 10 – MINT-Zweig | Textuelle Programmierung und Datenstrukturen

Lernziele

Ich kann

- gegebene Funktionen lesen und ihr Verhalten vorhersagen und erklären,
- einen Fehler in einer Funktion durch geschicktes Testen finden und beheben,
- Funktionen schreiben, die andere Funktionen aufrufen,
- eine vorhandene Funktion zu einer neuen Funktion erweitern,
- mit `math.pi` die Kreiszahl π verwenden.

🗣️ Sprachhilfe

- „Der Aufruf `verarbeite(6, 3)` ruft die Hilfsfunktion `ist_teilbar(6, 3)` auf. Sie liefert `True`, deshalb gibt `verarbeite` den Wert … zurück."
- „Ich teste mit dem Wert …, weil das ein Sonderfall / der Grenzwert der Bedingung ist."
- „Erwartet habe ich …, herausgekommen ist … . Also steckt der Fehler in Zeile …, weil …"

1 Das brauchst du: Funktionen wiederholen

Führe die Zelle aus. Erkläre dann mit den Fachbegriffen **Definition**, **Parameter**, **Argument**, **Aufruf** und **Rückgabewert**, was hier passiert.

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

print(quadrat(3))
print(quadrat(10))
```

```
9
100
```

- `def quadrat(zahl):` **definiert** die Funktion `quadrat` mit dem **Parameter** `zahl`. Der eingerückte Rumpf wird erst beim Aufruf ausgeführt.
- `return zahl * zahl` gibt das Ergebnis als **Rückgabewert** zurück und beendet die Funktion.
- `quadrat(3)` und `quadrat(10)` sind zwei **Aufrufe** derselben Funktion mit verschiedenen **Argumenten**.

Funktionen rufen Funktionen auf

Im Rumpf einer Funktion darf man auch **andere Funktionen** aufrufen. So kann eine größere Funktion kleinere **Hilfsfunktionen** benutzen.

🔮 **Vorhersage:** Was gibt die letzte Zeile aus? Überlege zuerst, dann führe die Zelle aus.

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

def summe_der_quadrate(a, b):
    return quadrat(a) + quadrat(b)

print(summe_der_quadrate(3, 4))
```

```
25
```

- **Zeilen 1–2:** die bekannte Funktion `quadrat`.
- **Zeilen 4–5:** `summe_der_quadrate` rechnet selbst nichts mit `*`. Sie ruft zweimal `quadrat` auf und addiert die beiden Rückgabewerte.
- **Zeile 7:** Der Aufruf springt in `summe_der_quadrate` mit `a = 3` und `b = 4`. Dort springt `quadrat(a)` in die Funktion `quadrat` und kommt mit `9` zurück, `quadrat(b)` liefert `16`. Zum Schluss wird `9 + 16 = 25` zurückgegeben.

Der Vorteil: Jede Funktion erledigt eine kleine, klar umrissene Aufgabe. Ist `quadrat` einmal getestet, kann man sich darauf verlassen, wo immer man sie benutzt.

> 📌 **Merksatz:** Funktionen können **andere Funktionen aufrufen**. Eine größere Funktion nutzt dann kleinere **Hilfsfunktionen**.

2 Code lesen und erklären

Bevor du ein fremdes Programm ausführst, lohnt es sich, es zu **lesen** und vorherzusagen, was passiert.

> 💡 **Tipp:** `a % b` ist der **Rest** der Division (`7 % 3` ergibt `1`), `a // b` die **ganzzahlige Division** (`7 // 3` ergibt `2`). Ist der Rest `0`, geht die Division auf. Lies verschachtelte Aufrufe von innen nach außen und schreibe dir für jeden Aufruf die Werte der Parameter auf.

🔍 Übung 1 – Code lesen

Lies den Code in der nächsten Zelle, **ohne** ihn auszuführen.

1. Beschreibe in eigenen Worten, was die Funktion `verarbeite` macht.
2. Gib an, was `print(verarbeite(6, 3))` und `print(verarbeite(5, 3))` ausgeben.

Schreibe deine Antwort in die Textzelle darunter. Führe den Code erst danach aus und überprüfe deine Vorhersage.

```python
def ist_teilbar(a, b):
    return a % b == 0

def verarbeite(x, y):
    if ist_teilbar(x, y):
        return x // y
    else:
        return x * y

print(verarbeite(6, 3))
print(verarbeite(5, 3))
```

```
2
15
```

Meine Erklärung und Vorhersage:

…

3 Fehler finden durch Testen

Manche Fehler erzeugen keine Fehlermeldung: Das Programm läuft, aber das Ergebnis ist falsch. Solche Fehler findest du nur durch **Testen** mit Werten, deren richtiges Ergebnis du kennst.

🧰 Werkzeug: das Modul `math`

Python bringt viele mathematische Werte und Funktionen im **Modul** `math` mit. Du schreibst einmal am Programmanfang `import math`, so wie `import random` beim Würfeln. Danach steht die Kreiszahl als `math.pi` zur Verfügung.

```
import math
print(math.pi)
```

```
3.141592653589793
```

📝 Übung 2 – Kreisumfang

Die Funktion soll den Umfang eines Kreises berechnen (Formel: $U = 2 \cdot \pi \cdot r$). Sie gibt aber ein falsches Ergebnis zurück. Finde den Fehler und behebe ihn. Teste die verbesserte Funktion, auch mit dem Sonderfall `radius = 0`.

```
import math

def kreisumfang(radius):
    umfang = 2 * radius
    return umfang

# Test: sollte bei radius = 5 etwa 31.4 ergeben
print(kreisumfang(5))
```

```
10
```

Der Fehler:

...

📝 Übung 3 – Grenzfälle testen

Ein Online-Shop gibt **ab** einem Einkaufswert von 50 Euro 10 % Rabatt. Die Funktion `rabatt_preis` soll den Preis nach Abzug des Rabatts zurückgeben.

1. Rechne **zuerst von Hand** aus, was für 20, 50 und 100 Euro herauskommen muss. Trage die Werte hinter `erwartet:` ein.
2. Führe die Tests aus. Welcher Test schlägt fehl?
3. Erkläre den Fehler und verbessere die Funktion.

```
def rabatt_preis(preis):
    if preis > 50:
        return preis * 0.9
    else:
        return preis

print(rabatt_preis(20), "erwartet: ")
print(rabatt_preis(50), "erwartet: ")
print(rabatt_preis(100), "erwartet: ")
```

```
20 erwartet:
50 erwartet:
90.0 erwartet:
```

Welcher Test schlägt fehl, und warum?

…

> 📌 **Merksatz:** Falsche Ergebnisse ohne Fehlermeldung findet man durch **Testen** mit Werten, deren Ergebnis man **vorher** kennt. Besonders wichtig sind **Sonder- und Grenzfälle** wie `0` oder der Grenzwert einer Bedingung.

4 Ein Programm aus mehreren Funktionen

Die Übungen dieser Stunde werden schrittweise selbstständiger: In Übung 1 liest du nur, in Übung 2 und 3 verbesserst du fertigen Code, in Übung 4 füllst du ein Gerüst aus, und die Zusatzaufgabe schreibst du allein. Vorbild für Übung 4 ist das vorgemachte Beispiel `summe_der_quadrate` aus Teil 1.

📝 Übung 4 – Rechteck (mit Gerüst)

Schreibe drei Funktionen:

- `flaeche_rechteck(a, b)` gibt die Fläche eines Rechtecks mit den Seitenlängen `a` und `b` zurück,
- `umfang_rechteck(a, b)` gibt den Umfang dieses Rechtecks zurück,
- `beschreibe_rechteck(a, b)` nutzt die beiden Funktionen oben und liefert einen Satz, den das Hauptprogramm ausgibt, z. B. für die Seiten 4 und 5:

  „Ein Rechteck mit den Seiten 4 und 5 hat die Fläche 20 und den Umfang 18.“

> 💡 **Tipp:** Schreibe zuerst die beiden kleinen Funktionen und teste jede einzeln (Kontrolle: Fläche 20, Umfang 18). Erst wenn sie stimmen, schreibst du `beschreibe_rechteck`. Sie ruft die beiden anderen auf, so wie `summe_der_quadrate` oben `quadrat` aufruft. Den Satz setzt du mit `+` zusammen. Zahlen musst du dafür mit `str(...)` in Text umwandeln.

Gerüst: Ersetze jedes `___`. Führe die Zelle zuerst nur mit Schritt 1 aus und prüfe die Kontrollwerte, bevor du Schritt 2 bearbeitest.

Lückencode

```
# Schritt 1: die beiden Hilfsfunktionen
def flaeche_rechteck(a, b):
    return ___

def umfang_rechteck(a, b):
    return ___

print(flaeche_rechteck(4, 5), umfang_rechteck(4, 5), "erwartet: 20 18")

# Schritt 2: beschreibe_rechteck ruft die beiden Hilfsfunktionen auf
def beschreibe_rechteck(a, b):
    flaeche = ___
    umfang = ___
    return ("Ein Rechteck mit den Seiten " + str(a) + " und " + str(b)
            + " hat die Fläche " + ___ + " und den Umfang " + ___ + ".")

print(beschreibe_rechteck(4, 5))
```

🚀 Zusatzaufgabe – Maximum von drei Zahlen (selbstständig)

Jetzt ohne Gerüst. In DS 8 hast du `maximum(a, b)` geschrieben. Ergänze eine Funktion `maximum3(a, b, c)`, die die größte von **drei** Zahlen zurückgibt. Benutze dafür `maximum`: `maximum3` selbst soll ohne `if` auskommen. Teste mit mindestens drei Aufrufen, bei denen die größte Zahl jeweils an einer anderen Stelle steht.

```python
def maximum(a, b):
    if a > b:
        return a
    else:
        return b

# TODO: maximum3(a, b, c)
```

📌 Das Wichtigste in Kürze

- Programme **lesen**: von innen nach außen auswerten und für jeden Aufruf die Werte der Parameter notieren.
- Funktionen können **andere Funktionen aufrufen**. Große Aufgaben zerlegt man so in kleine **Hilfsfunktionen**, die man einzeln testet.
- Falsche Ergebnisse ohne Fehlermeldung findet man durch **Testen** mit Werten, deren Ergebnis man vorher kennt.
- Besonders wichtig sind Tests mit **Sonder- und Grenzfällen** wie `0` oder dem Grenzwert einer Bedingung.
- Mit `import math` steht die Kreiszahl als `math.pi` zur Verfügung.

Zum Weiterüben: `DS09_Weitere_Uebungen.ipynb`.

DS 9 – Weitere Übungen

Übung: Funktionen lesen, testen und erweitern

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Verschachtelte Aufrufe ●○○

Lies das Programm, **ohne** es auszuführen. Gib an, was jede `print`-Zeile ausgibt. Überprüfe danach am Computer.

```python
def plus_eins(x):
    return x + 1

def doppelt(x):
    return 2 * x

print(doppelt(plus_eins(3)))
print(plus_eins(doppelt(3)))
print(doppelt(doppelt(plus_eins(0))))
```

```
8
7
4
```

Deine Antwort:

…

Aufgabe 2: Kreisfläche ●○○

Die Fläche eines Kreises berechnet man mit der Formel $A = \pi \cdot r \cdot r$.

1. Implementiere eine Funktion `kreisflaeche(radius)`, die die Fläche zurückgibt. Verwende `math.pi`.
2. Teste die Funktion mit den Radien 1, 0 und 2. Zur Kontrolle: Für den Radius 1 muss genau π herauskommen, also etwa 3,14.

Aufgabe 3: Gute Testwerte wählen ●○○

Die Funktion `ist_teenager(alter)` soll `True` zurückgeben, wenn jemand 13 bis einschließlich 19 Jahre alt ist, sonst `False`.

```python
def ist_teenager(alter):
    return alter >= 13 and alter <= 19
```

Nenne vier Testwerte, mit denen du die Funktion besonders gut prüfen kannst, und gib jeweils das erwartete Ergebnis an. Begründe deine Auswahl in einem Satz.

Deine Antwort:

…

Aufgabe 4: Ungerade mit Hilfsfunktion ●○○

Im Abschnitt über Funktionen hast du `ist_gerade(zahl)` geschrieben:

```python
def ist_gerade(zahl):
    return zahl % 2 == 0
```

Implementiere eine Funktion `ist_ungerade(zahl)`, die `True` zurückgibt, wenn die Zahl ungerade ist. Benutze dafür die Funktion `ist_gerade` und den Operator `not`. Teste mit 7 und mit 10.

Aufgabe 5: Wo steckt der Fehler? ●●○

Eine Fahrt dauert 2 Stunden und 15 Minuten. Das Programm soll die Fahrzeit in Minuten berechnen, also 135. Es gibt aber 215 aus.

```python
def stunden_zu_minuten(stunden):
    return stunden * 100

def fahrzeit_minuten(stunden, minuten):
    return stunden_zu_minuten(stunden) + minuten

print(fahrzeit_minuten(2, 15))
```

```
215
```

1. Beschreibe, wie du durch einen **einzelnen Test der Hilfsfunktion** herausfindest, in welcher der beiden Funktionen der Fehler steckt.
2. Behebe den Fehler und teste erneut.

Deine Antwort:

…

Aufgabe 6: Versandkosten ●●○

Ein Online-Shop berechnet 5 Euro Versandkosten. Ab einem Warenwert von 30 Euro ist der Versand kostenlos.

1. Implementiere eine Funktion `versandkosten(warenwert)`, die die Versandkosten zurückgibt (0 oder 5).
2. Implementiere eine Funktion `endpreis(warenwert)`, die den Warenwert plus Versandkosten zurückgibt. Sie soll `versandkosten` aufrufen.
3. Teste `endpreis` mit den Warenwerten 20, 30 und 45. Überlege vorher, was herauskommen muss.

Aufgabe 7: Pizza-Vergleich ●●○

Was ist mehr Pizza: **eine** große Pizza mit 32 cm Durchmesser oder **zwei** kleine mit je 22 cm Durchmesser?

1. Implementiere eine Funktion `pizza_flaeche(durchmesser)`, die die Fläche einer runden Pizza zurückgibt. Sie soll die Funktion `kreisflaeche(radius)` aus dem Auftrag *Kreisfläche* aufrufen. Denk daran: Der Radius ist der halbe Durchmesser.
2. Berechne damit die Fläche der großen Pizza und die Fläche der beiden kleinen Pizzen zusammen. Gib beide Werte gerundet aus und beantworte die Frage.

Aufgabe 8: Primzahlen mit Hilfsfunktionen ●●●

Im Auftrag *Primzahl-Test* (Übung zu for- und while-Schleife) hast du geprüft, ob eine Zahl eine Primzahl ist. Jetzt zerlegst du die Aufgabe in Funktionen.

1. Implementiere eine Funktion `ist_teiler(t, zahl)`, die `True` zurückgibt, wenn `t` ein Teiler von `zahl` ist (die Division geht ohne Rest auf).
2. Implementiere eine Funktion `ist_primzahl(zahl)`, die `True` oder `False` zurückgibt. Sie soll `ist_teiler` benutzen.
3. Gib mit einer Schleife alle Primzahlen von 1 bis 30 aus.

Tipp: Eine Primzahl ist eine Zahl größer als 1, die nur durch 1 und sich selbst teilbar ist. Gehe in `ist_primzahl` wie im Auftrag *Primzahl-Test* vor: Starte mit `ergebnis = zahl > 1` und prüfe in einer Schleife alle möglichen Teiler von 2 bis `zahl - 1`. Findest du einen Teiler, setzt du `ergebnis = False`.

DS 10 – Zeichenketten: Texte Zeichen für Zeichen

Informatik 10 – MINT-Zweig | Datenstrukturen

Lernziele

Ich kann

- eine Zeichenkette als Folge (Sequenz) von Zeichen beschreiben und den Index eines Zeichens angeben,
- mit `text[index]` auf einzelne Zeichen zugreifen und mit `len(text)` die Länge bestimmen,
- Zeichenketten verketten und Ausgaben mit f-Strings bequem zusammensetzen,
- mit einer Schleife alle Zeichen einer Zeichenkette durchlaufen,
- `in`, `.upper()`, `.lower()`, `.isalpha()` und `.isdigit()` anwenden,
- einen `IndexError` erklären und vermeiden.

🗣️ Sprachhilfe

- „Das Zeichen am Index … ist …, weil ab 0 gezählt wird.“
- „Die Schleife durchläuft nacheinander alle Zeichen. In jedem Durchlauf prüft sie, ob …, und erhöht dann …“
- „Der `IndexError` entsteht, weil der Text nur … Zeichen hat. Gültig sind die Indizes 0 bis … .“

Worum es geht

Namen, Passwörter, Nachrichten: Sehr viele Daten in Programmen sind Texte. Bisher hast du Texte nur als Ganzes benutzt. Oft muss ein Programm aber **in einen Text hineinschauen**: Wie lang ist das Passwort? Enthält es eine Ziffer? Mit welchem Buchstaben beginnt der Name? Dafür lernst du heute, wie Python einen Text sieht: als Reihe einzelner Zeichen, jedes an einer festen Position.

1 Eine Zeichenkette ist eine Folge von Zeichen

Für Python besteht ein Text aus einzelnen **Zeichen**, die der Reihe nach hintereinander stehen, wie Perlen auf einer Schnur. Ein Zeichen ist ein Buchstabe, eine Ziffer, ein Satzzeichen oder auch ein Leerzeichen. Die Position eines Zeichens heißt **Index**. Gezählt wird ab **0**:

Index	0	1	2	3	4	5	6	7	8	9
Zeichen	`I`	`n`	`f`	`o`	`r`	`m`	`a`	`t`	`i`	`k`

Die Zeichenkette `"Informatik"` hat 10 Zeichen, aber der größte Index ist 9. Man kann sich den Index als Abstand vom Anfang vorstellen: Das erste Zeichen ist 0 Schritte vom Anfang entfernt.

> 📌 **Merksatz:** Eine **Zeichenkette** (`str`) ist eine **Sequenz von Zeichen**, z. B. `"Informatik"`. Jedes Zeichen hat eine **Position (Index)**, beginnend bei `0`.

Auf einzelne Zeichen zugreifen

Den Index schreibst du in **eckigen Klammern** hinter die Zeichenkette. Vergleiche die Ausgabe mit der Tabelle.

```python
text = "Informatik"
print(text[0])
print(text[2])
print(text[9])
```

```
I
f
k
```

`text[2]` ist das **dritte** Zeichen, `f`. In den eckigen Klammern darf auch eine Variable oder Rechnung stehen, zum Beispiel `text[i]` oder `text[i + 1]`.

Die Länge einer Zeichenkette

Wie viele Zeichen ein Text hat, verrät die Funktion `len` (englisch *length*, „Länge“). Auch das Leerzeichen zählt mit. `""` ist die **leere Zeichenkette** mit der Länge 0.

```python
text = "Informatik"
print(len(text))
print(len("Ada Lovelace"))
print(len(""))

letzter_index = len(text) - 1
print(letzter_index)
print(text[letzter_index])
print(text[-1])
```

```
10
12
0
9
k
k
```

Weil ab 0 gezählt wird, hat das **letzte Zeichen** immer den Index `len(text) - 1`. Kürzer geht es mit einem **negativen Index**, der von hinten zählt: `text[-1]` ist das letzte Zeichen, `text[-2]` das vorletzte.

> 📌 **Merksatz:** Die **Länge** erhält man mit `len(text)`. Das letzte Zeichen hat den Index `len(text) - 1` oder kurz `-1`.

⚠️ Stolperstein: Der Index ist zu groß

Führe die Zelle aus und lies die Fehlermeldung.

```python
text = "Informatik"
print(text[10])
```

```
Traceback (most recent call last):
  File "<Zelle 3>", line 2, in <module>
IndexError: string index out of range
```

`IndexError: string index out of range` heißt: „Der Index liegt außerhalb des gültigen Bereichs.“ Gültig sind bei 10 Zeichen nur die Indizes 0 bis 9 (und −1 bis −10). Prüfe bei diesem Fehler: Wie lang ist der Text, und welcher Index wurde benutzt?

🔮 Übung 1 – Zeichen bestimmen

Gib **ohne** Computer an, was jede Zeile ausgibt. Eine Zeile führt zu einem Fehler: welche, und warum? Schreibe deine Vorhersage in die Textzelle und führe den Code erst danach aus.

```python
wort = "Python"
print(wort[1])
print(len(wort))
print(wort[len(wort) - 1])
print(wort[-2])
print(wort[6])
```

```
y
6
n
o
```

```
Traceback (most recent call last):
  File "<Zelle 4>", line 6, in <module>
IndexError: string index out of range
```

Meine Vorhersage:

…

2 Zeichenketten verketten

Mit `+` hängst du Zeichenketten aneinander. Verkettung und Index-Zugriff lassen sich kombinieren.

```python
vorname = "Ada"
nachname = "Lovelace"
name = vorname + " " + nachname
print(name)
initialen = vorname[0] + nachname[0]
print(initialen)
```

```
Ada Lovelace
AL
```

Ohne das `" "` in der Mitte stünde `AdaLovelace` da. Ada Lovelace gilt übrigens als erste Programmiererin der Welt.

🧰 Werkzeug: f-Strings

Texte mit Zahlen mit `+` zusammenzusetzen ist mühsam: Jede Zahl muss mit `str(...)` umgewandelt werden. Bequemer ist ein **f-String**: ein `f` direkt vor dem ersten Anführungszeichen, und im Text Variablen oder Rechnungen in **geschweiften Klammern** `{...}`. Zeile 3 und Zeile 4 erzeugen dieselbe Ausgabe.

```python
name = "Alex"
alter = 15
print(f"{name} ist {alter} Jahre alt.")
print(name + " ist " + str(alter) + " Jahre alt.")
print(f"{name} hat {len(name)} Buchstaben.")
```

```
Alex ist 15 Jahre alt.
Alex ist 15 Jahre alt.
Alex hat 4 Buchstaben.
```

Vergisst du das `f`, erscheinen die geschweiften Klammern unverändert in der Ausgabe.

> 📌 **Merksatz:** Die **Verkettung** (Konkatenation) zweier Zeichenketten erfolgt mit `+`. Ein **f-String** `f"... {wert} ..."` setzt Werte direkt in einen Text ein.

3 Alle Zeichen durchlaufen

Mit einer `for`-Schleife bearbeitest du nacheinander **jedes Zeichen**. Statt `range(...)` schreibst du hinter `in` einfach die Zeichenkette.

```
for zeichen in "Bit":
    print(zeichen)
```

```
B
i
t
```

Die Schleifenvariable `zeichen` bekommt im ersten Durchlauf `"B"`, dann `"i"`, dann `"t"`. Der Name `zeichen` ist frei gewählt.

Zeichen zählen

🔮 **Vorhersage:** Welche Zahl gibt das Programm aus? Achte auf das große `A`.

```
1 wort = "Anna"
2 anzahl = 0
3 for zeichen in wort:
4     if zeichen == "n":
5         anzahl = anzahl + 1
6 print("Anzahl n:", anzahl)
```

```
Anzahl n: 2
```

- **Zeile 2:** Die Zählvariable `anzahl` startet bei 0.
- **Zeilen 3–5:** Die Schleife läuft über alle vier Zeichen. Nur wenn das aktuelle Zeichen ein `n` ist, wird `anzahl` um 1 erhöht.
- **Zeile 6:** Erst nach der Schleife (nicht eingerückt) wird das Ergebnis ausgegeben.

Das große `A` am Anfang wird **nicht** gezählt, denn `"A"` und `"a"` sind für Python verschiedene Zeichen.

Schritt	Zeile	Anweisung	zeichen	anzahl
1	1	`wort = "Anna"`	-	-
2	2	`anzahl = 0`	-	`0`
3	3	`for zeichen in wort:`	`"A"`	`0`
4	4	`if zeichen == "n":`	`"A"`	`0`
5	3	`for zeichen in wort:`	`"n"`	`0`
6	4	`if zeichen == "n":`	`"n"`	`0`
7	5	`anzahl = anzahl + 1`	`"n"`	`1`
8	3	`for zeichen in wort:`	`"n"`	`1`
9	4	`if zeichen == "n":`	`"n"`	`1`
10	5	`anzahl = anzahl + 1`	`"n"`	`2`
11	3	`for zeichen in wort:`	`"a"`	`2`
12	4	`if zeichen == "n":`	`"a"`	`2`
13	3	`for zeichen in wort:`	`"a"`	`2`
14	6	`print("Anzahl n:", anzahl)`	`"a"`	`2`

Einen neuen Text aufbauen

Man beginnt mit der leeren Zeichenkette `""` und hängt in jedem Durchlauf etwas an. 🔮 **Vorhersage:** Was kommt heraus?

```python
neu = ""
for zeichen in "Bit":
    neu = neu + zeichen + zeichen
print(neu)
```

```
BBiitt
```

Aus `""` wird `"BB"`, dann `"BBii"` und schließlich `"BBiitt"`.

Schleife über die Indizes

Brauchst du auch die **Position**, lässt du die Schleife über alle Indizes laufen. `range(len(text))` liefert genau die gültigen Indizes 0, 1, …, `len(text) - 1`.

```python
text = "Bit"
for i in range(len(text)):
    print(i, text[i])
```

```
0 B
1 i
2 t
```

> 📌 **Merksatz:** `for zeichen in text:` durchläuft nacheinander alle **Zeichen**. `for i in range(len(text)):` durchläuft alle **Indizes**, das Zeichen dazu ist `text[i]`.

4 Zeichen prüfen und umwandeln

Ist ein Zeichen enthalten?

Mit `in` prüfst du, ob ein Zeichen oder ein Textstück vorkommt. Das Ergebnis ist `True` oder `False`. Besonders praktisch ist `in` mit einer Zeichenkette aller „erlaubten“ Zeichen, etwa `"aeiou"` für die Vokale.

```python
print("a" in "Banane")
print("x" in "Banane")
print("nan" in "Banane")
print("B" in "banane")

buchstabe = "e"
if buchstabe in "aeiou":
    print(buchstabe, "ist ein Vokal.")
```

```
True
False
True
False
e ist ein Vokal.
```

Groß- und Kleinschreibung

Für Python sind `"Ja"`, `"ja"` und `"JA"` drei verschiedene Zeichenketten. Ist die Schreibweise egal, wandelt man alles in Klein- oder Großbuchstaben um.

```python
text = "Hallo Welt"
print(text.upper())
print(text.lower())
print(text)
```

```
HALLO WELT
hallo welt
Hallo Welt
```

`text` selbst bleibt **unverändert**: `.upper()` und `.lower()` erzeugen eine neue Zeichenkette. Neu ist die Schreibweise mit dem Punkt **hinter** dem Text. Solche Funktionen heißen **Methoden**.

> 📌 **Definition Methode:** Eine **Methode** ist eine Funktion, die zu einem Wert gehört. Man ruft sie mit einem Punkt hinter dem Wert auf: `text.upper()`. Die Klammern dürfen leer sein, aber nicht fehlen.

⚠️ Stolperstein: `text.lower()` allein ändert nichts

🔮 **Vorhersage:** Was gibt die Zelle aus? Wie musst du Zeile 2 ändern, damit `hallo` herauskommt?

```
1 text = "Hallo"
2 text.lower()
3 print(text)
```

```
Hallo
```

Buchstabe oder Ziffer?

- `text.isalpha()` ist `True`, wenn **alle** Zeichen Buchstaben sind (auch Umlaute und ß),
- `text.isdigit()` ist `True`, wenn **alle** Zeichen Ziffern sind.

```
1 print("Hallo".isalpha())
2 print("Hallo Welt".isalpha())
3 print("2026".isdigit())
4 print("-5".isdigit())
5 print("3.5".isdigit())
```

```
True
False
True
False
False
```

`"Hallo Welt"` enthält ein Leerzeichen, und das ist kein Buchstabe. Bei `"-5"` und `"3.5"` sind Minuszeichen und Punkt keine Ziffern. Mit `.isdigit()` kann ein Programm eine Eingabe prüfen, bevor es sie mit `int(...)` umwandelt, und so den `ValueError` vermeiden.

> 📌 **Merksatz:** `x in text` prüft, ob `x` enthalten ist. `.upper()` und `.lower()` liefern eine **neue** Zeichenkette, darum `text = text.lower()`. `.isalpha()` und `.isdigit()` prüfen auf Buchstaben bzw. Ziffern.

📝 Übungen: Schritt für Schritt selbstständig

Die Übungen werden schrittweise schwieriger. Vorgemacht hast du schon zwei Muster gesehen: **Zeichen zählen** (`"Anna"`) und **einen neuen Text aufbauen** (`"BBiitt"`). Bei Übung 2 füllst du Lücken `___` aus, bei Übung 3 steht nur noch das Gerüst als Kommentar, bei Übung 4 nur die Kopfzeile und die Tests, die Zusatzaufgabe schreibst du allein.

📝 Übung 2 – Zeichen-Steckbrief (Lücken füllen)

Schreibe ein Programm, das ein Wort einliest und ausgibt: die Länge, das erste und das letzte Zeichen und das Wort rückwärts. So könnte es aussehen:

```
Wort eingeben: Python
Länge: 6
Erstes Zeichen: P
Letztes Zeichen: n
Rückwärts: nohtyP
```

💡 **Tipp:** Für das Rückwärts-Wort baust du in einer Schleife einen neuen Text auf wie bei `"BBiitt"`. Der Trick: Hänge jedes Zeichen **vorne** an, also `zeichen + rueckwaerts` statt `rueckwaerts + zeichen`.

Lückencode

```
wort = input("Wort eingeben: ")
print("Länge:", ___)
print("Erstes Zeichen:", wort[___])
print("Letztes Zeichen:", wort[___])

rueckwaerts = ""
for zeichen in wort:
    rueckwaerts = ___
print("Rückwärts:", rueckwaerts)
```

📝 Übung 3 – Vokale zählen (mit Gerüst)

Schreibe ein Programm, das einen Satz einliest und mit einer Schleife zählt, wie viele Vokale (a, e, i, o, u) darin vorkommen. Auch großgeschriebene Vokale sollen mitgezählt werden. Kontrolle: „Informatik macht Spaß“ hat 6 Vokale.

💡 **Tipp:** Kombiniere die Schleife über alle Zeichen, die Zählvariable aus dem Beispiel `"Anna"` und die Prüfung `zeichen in "aeiou"`. Gegen das Problem mit den Großbuchstaben hilft `.lower()`.

```
satz = input("Satz eingeben: ")
# Zählvariable auf 0 setzen
# für jedes Zeichen im Satz (in Kleinbuchstaben):
#     wenn das Zeichen in "aeiou" vorkommt:
#         Zählvariable um 1 erhöhen
# nach der Schleife: Anzahl ausgeben
```

```
Satz eingeben: Informatik macht Spaß
```

📝 Übung 4 – Passwort prüfen (Kopfzeile und Tests vorgegeben)

Ein Passwort gilt hier als sicher, wenn es **mindestens 8 Zeichen** lang ist und **mindestens eine Ziffer** enthält. Schreibe eine Funktion `ist_sicher(passwort)`, die `True` oder `False` zurückgibt. Kopfzeile und Testaufrufe stehen schon da.

💡 **Tipp:** Zähle die Ziffern wie die Vokale in Übung 3, prüfe aber mit `zeichen.isdigit()`. Verknüpfe am Ende beide Bedingungen mit `and`.

```
def ist_sicher(passwort):
    pass   # hier deine Programmzeilen

print(ist_sicher("abc1"), "erwartet: False")          # zu kurz
print(ist_sicher("sommerferien"), "erwartet: False")  # keine Ziffer
print(ist_sicher("sommer2026"), "erwartet: True")
```

```
None erwartet: False
None erwartet: False
None erwartet: True
```

🚀 Zusatzaufgabe – Palindrom (selbstständig)

Ein **Palindrom** ist ein Wort, das vorwärts und rückwärts gelesen gleich ist, z. B. „Anna“ oder „Rentner“. Schreibe eine Funktion `ist_palindrom(wort)`, die `True` oder `False` zurückgibt. Groß- und Kleinschreibung soll keine Rolle

spielen. Teste mit „Anna“, „Rentner“ und „Informatik“.

➕ Zusatzwissen: Ausschnitte (Slicing)

Für die Aufgaben brauchst du das nicht. Mit `text[start:ende]` schneidest du einen **Ausschnitt** heraus. Das Zeichen am Index `start` gehört dazu, das am Index `ende` **nicht** mehr. `text[::-1]` durchläuft den Text rückwärts.

```
text = "Informatik"
print(text[0:4])
print(text[2:6])
print(text[::-1])
```

```
Info
form
kitamrofnI
```

📌 Das Wichtigste in Kürze

- Eine **Zeichenkette** ist eine **Sequenz von Zeichen**. Jedes Zeichen hat einen **Index**, gezählt ab 0.
- `text[i]` liefert das Zeichen am Index `i`; `text[-1]` ist das letzte Zeichen.
- `len(text)` liefert die Länge. Gültige Indizes reichen von 0 bis `len(text) - 1`, sonst gibt es einen **IndexError**.
- Mit `+` verkettet man Zeichenketten. Ein **f-String** `f"... {wert} ..."` setzt Werte bequem in einen Text ein.
- `for zeichen in text:` durchläuft alle Zeichen, `for i in range(len(text)):` alle Indizes.
- `x in text` prüft, ob `x` enthalten ist. `.upper()` und `.lower()` liefern eine neue Zeichenkette; `.isalpha()` und `.isdigit()` prüfen auf Buchstaben bzw. Ziffern.

Zum Weiterüben: `DS10_Weitere_Uebungen.ipynb`.

DS 10 – Weitere Übungen

Zeichenketten: Texte Zeichen für Zeichen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Index und Zeichen ●○○

Gegeben ist `wort = "Freibad"`. Bestimme **ohne Computer** die Werte der Ausdrücke. Prüfe danach mit `print(...)` am Computer.

`wort[0]`	`wort[3]`	`wort[-1]`	`len(wort)`	`wort[len(wort) - 2]`

Gib außerdem an, an welchem Index das Zeichen `b` steht.

Deine Antwort:

…

Aufgabe 2: Ausgabe vorhersagen ●○○

Gib **ohne Computer** an, was das Programm ausgibt. Überprüfe danach am Computer.

```
text = "Klasse 10a"
print(len(text))
print(text[7] + text[8])
print("s" in text)
print("S" in text)
print(text.lower())
```

```
10
10
True
False
klasse 10a
```

Deine Antwort:

…

Aufgabe 3: Initialen ●○○

Implementiere eine Funktion `initialen(vorname, nachname)`, die die Initialen mit Punkten zurückgibt, zum Beispiel für „Mia“ und „Becker“ die Zeichenkette `M. B.`. Teste die Funktion mit zwei verschiedenen Namen.

Aufgabe 4: Buchstaben zählen ●○○

Implementiere eine Funktion `zaehle(text, zeichen)`, die zurückgibt, wie oft `zeichen` in `text` vorkommt. Teste sie mit diesen Aufrufen:

Aufruf	erwartet	tatsächlich
`zaehle("Banane", "a")`	`2`	
`zaehle("Mississippi", "s")`	`4`	
`zaehle("Hallo", "x")`	`0`	

Aufgabe 5: Zwei Fehler finden ●●○

Das Programm soll zählen, wie oft der Buchstabe e im Satz vorkommt, egal ob groß oder klein. Erwartet ist 7. Stattdessen bricht es mit einem Fehler ab.

```
satz = "Eine Elefantenherde"
satz.lower()
anzahl = 0
for i in range(1, len(satz) + 1):
    if satz[i] == "e":
        anzahl = anzahl + 1
print(anzahl)
```

```
Traceback (most recent call last):
  File "<Zelle 2>", line 5, in <module>
IndexError: string index out of range
```

1. Gib an, welche Fehlermeldung erscheint, und erkläre sie.
2. Das Programm enthält noch einen zweiten Fehler. Er würde dafür sorgen, dass die großen E nicht mitgezählt werden. Erkläre ihn.
3. Schreibe das Programm richtig auf.

Deine Antwort:

…

Aufgabe 6: Ohne Leerzeichen ●●○

Implementiere eine Funktion `ohne_leerzeichen(text)`, die eine neue Zeichenkette zurückgibt, in der alle Leerzeichen fehlen. Teste mit `"Hallo Welt"` (erwartet: `HalloWelt`) und mit `"0151 234 567"` (erwartet: `0151234567`).

> **Tipp:** Baue wie im Abschnitt *Einen neuen Text aufbauen* einen neuen Text auf. Beginne mit `""` und hänge nur die Zeichen an, die **kein** Leerzeichen sind. Ein Leerzeichen schreibst du als `" "`.

Aufgabe 7: Vokale tauschen ●●○

Ein bekanntes Sprachspiel: Man ersetzt in einem Satz alle Vokale durch denselben Vokal. Aus „Wir sehen uns morgen“ wird dann „Wir sihin ins mirgin“. Implementiere eine Funktion `nur_i(text)`, die eine neue Zeichenkette zurückgibt, in der jeder Vokal (a, e, i, o, u, auch großgeschrieben) durch ein kleines `i` ersetzt ist. Alle anderen Zeichen bleiben unverändert. Teste mit `"Wir sehen uns morgen"` und mit `"Otto"`.

Aufgabe 8: Handynummer verbergen ●●●

Auf Rechnungen wird eine Handynummer oft nur teilweise gezeigt, zum Beispiel `*********678`. Implementiere eine Funktion `verberge(nummer)`, die eine neue Zeichenkette zurückgibt. Darin ist jedes Zeichen durch `*` ersetzt – außer den **letzten drei** Zeichen. Teste mit `"015112345678"` (erwartet: `*********678`) und `"12345"` (erwartet: `**345`).

> **Tipp:** Hier brauchst du die **Position** jedes Zeichens. Lass die Schleife deshalb über die Indizes laufen: `for i in range(len(nummer)):`. Überlege: Ab welchem Index beginnen die letzten drei Zeichen? Bei 12 Zeichen sind das die Indizes 9, 10 und 11.

DS 11 – Listen: viele Werte unter einem Namen

Informatik 10 – MINT-Zweig | Datenstrukturen

Lernziele

Ich kann

- eine Liste (ein Feld) als Datenstruktur beschreiben, die mehrere Elemente der Reihe nach zu einer Einheit zusammenfasst,
- zwischen dem **Index** und dem **Element** an dieser Stelle unterscheiden,
- über den Index lesend und schreibend auf einzelne Listenelemente zugreifen,
- Elemente mit `append`, `insert`, `pop` und `remove` hinzufügen und entfernen,
- mit einer Schleife alle Elemente einer Liste durchlaufen,
- die Fehlermeldung `IndexError` erklären und den Fehler beheben.

🗣️ Sprachhilfe

- *„Die 5 ist an Stelle 1."* → „Das Listenelement am Index 1 hat den Wert 5."
- *„Ich mach die 3 zur 5."* → „Das Listenelement am Index 1 wird von 3 auf 5 verändert."
- *„Die 6 kommt hinten dran."* → „Mit `append` wird das Element 6 am Ende der Liste hinzugefügt."
- *„Da kommt ein Fehler."* → „Beim Zugriff auf die ungültige Listenposition 5 erhalte ich die Fehlermeldung `IndexError`."

1 Warum Listen?

Speicherst du fünf Noten in `note1` bis `note5`, hat das zwei Schwächen: Kommt eine sechste Note dazu, brauchst du eine neue Variable **und** musst jede Rechnung ändern. Außerdem kann keine Schleife über `note1` bis `note5` laufen, denn für Python sind das fünf unabhängige Namen.

Mit einer **Liste** speicherst du alle Noten in **einer** Variablen:

```
noten = [2, 3, 1, 4, 2]
print(noten)
print(type(noten))

staedte = ["Saarbrücken", "Trier", "Mainz"]
leer = []
print(staedte)
print(leer)
```

```
[2, 3, 1, 4, 2]
<class 'list'>
['Saarbrücken', 'Trier', 'Mainz']
[]
```

- Eine Liste schreibt man in **eckigen Klammern**, die Werte werden durch Kommas getrennt. Der Datentyp heißt `list`.
- Die Werte heißen **Elemente**. Sie stehen in fester Reihenfolge, wie die Waggons eines Zuges: Eine Liste ist eine **lineare** Datenstruktur.
- Zeichenketten in einer Liste zeigt Python mit `'…'` an. Das ist nur die Anzeige.
- Die leere Liste `[]` legt man an und füllt sie später nach und nach.

In anderen Sprachen wie Java heißt diese Datenstruktur **Feld** (*array*) und hat meist eine feste Länge. Eine Python-Liste kann wachsen und schrumpfen.

> 📌 **Merksatz:** Eine **Liste** (in Python: `list`) fasst mehrere **Elemente zu einer Einheit** zusammen, z. B. `noten = [2, 3, 1, 4, 2]`.

2 Index und Element

Denk an die Schließfächer in deiner Schule: Jedes Fach hat eine **Nummer** an der Tür, und darin liegt ein **Inhalt**. Wer „Fach 2“ sagt, meint die Nummer, nicht den Inhalt. Bei einer Liste heißt die Nummer **Index** (ab 0 gezählt wie bei Zeichenketten), der Inhalt **Element**.

Index	0	1	2	3	4
Element	2	3	1	4	2

Lesender Zugriff

Den Index schreibst du in eckigen Klammern hinter den Namen der Liste.

```
noten = [2, 3, 1, 4, 2]
print(noten[0])
print(noten[2])
print(len(noten))
print(noten[len(noten) - 1])
print(noten[-1], noten[-2])
```

```
2
1
5
2
2 4
```

- `noten[2]` liefert die `1`, also das **dritte** Element. Die Zahl in den eckigen Klammern ist der Index, nicht der Wert!
- Das Element `2` kommt zweimal vor: am Index 0 und am Index 4.
- Die Liste hat fünf Elemente, das letzte hat den Index `len(noten) - 1`, also 4. Kürzer: `noten[-1]`.

> 📌 **Merksatz:** Jedes Element besitzt einen **Index**, beginnend bei `0`. Der Index ist **nicht** dasselbe wie der gespeicherte Wert! **Lesender Zugriff:** `liste[index]`. Die Anzahl der Elemente liefert `len(liste)`.

🔮 Übung 1 – Index oder Element?

Gegeben ist `zahlen = [3, 0, 4, 1, 2]`. Bestimme **ohne Computer** die Werte und trage sie in die Tabelle ein. An welchem Index steht das Element `4`? Prüfe danach in der Codezelle.

`zahlen[1]`	`len(zahlen)`	`zahlen[-1]`	`zahlen[zahlen[0]]`	Index von 4

```
zahlen = [3, 0, 4, 1, 2]
# TODO: Prüfe deine Vorhersage mit print(...)
```

3 Schreibender Zugriff: ein Element ändern

Ein Schließfach kann man ausräumen und neu füllen, die Nummer bleibt gleich. Steht der Zugriff mit Index **links** vom `=`, wird das Element **überschrieben**.

```
1 noten = [2, 3, 1, 4, 2]
2 noten[1] = 5
3 print(noten)
4 noten[2] = noten[2] + 1
5 print(noten)
```

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

- **Zeile 2:** Das Element am Index 1 bekommt den Wert `5`. Der alte Wert `3` ist weg.
- **Zeile 4:** Wie bei `x = x + 1` wird zuerst rechts gerechnet: `noten[2]` ist `1`, plus 1 ergibt `2`.

Die Länge der Liste ändert sich dabei nicht.

> 📌 **Merksatz: Schreibender Zugriff:** `liste[index] = neuer_wert` überschreibt das Element an dieser Position.

⚠️ Stolpersteine

Führe die drei Zellen aus und lies jeweils die letzte Zeile der Fehlermeldung.

```
1 # 1. Eine Zeichenkette kann man nicht stellenweise ändern.
2 wort = "Hallo"
3 wort[0] = "B"
```

```
Traceback (most recent call last):
  File "<Zelle 5>", line 3, in <module>
TypeError: 'str' object does not support item assignment
```

```
1 # 2. Den Index 5 gibt es bei fünf Elementen nicht.
2 noten = [2, 3, 1, 4, 2]
3 print(noten[5])
```

```
Traceback (most recent call last):
  File "<Zelle 6>", line 3, in <module>
IndexError: list index out of range
```

```
1 # 3. Auch durch eine Zuweisung wird die Liste nicht länger.
2 noten = [2, 3, 1, 4, 2]
3 noten[5] = 1
```

```
Traceback (most recent call last):
  File "<Zelle 7>", line 3, in <module>
IndexError: list assignment index out of range
```

1. `TypeError: 'str' object does not support item assignment`: Zeichenketten kann man nur lesen. Listen sind dagegen **veränderbar**.
2. `IndexError: list index out of range`: Gültig sind nur die Indizes 0 bis 4 (und −1 bis −5).
3. `IndexError: list assignment index out of range`: Zum Hinzufügen braucht man `append` (Teil 4).

Häufigste Ursache: Man zählt aus Gewohnheit ab 1 und schreibt `liste[len(liste)]`. Prüfe bei einem `IndexError` immer: Wie lang ist die Liste gerade, und welchen Index habe ich benutzt?

> 📌 **Merksatz:** Ein Zugriff auf einen ungültigen Index führt zu einem `IndexError`. Bei einer Liste mit `n` Elementen sind nur die Indizes `0` bis `n - 1` gültig.

4 Listen verändern: Elemente hinzufügen und entfernen

Eine Liste kann wachsen und schrumpfen. Dafür gibt es **Methoden**, die man mit einem Punkt anhängt, wie `text.upper()` bei Zeichenketten.

`append` und `insert`

`liste.append(wert)` hängt ein Element **am Ende** an. `liste.insert(index, wert)` fügt es an einer **beliebigen Stelle** ein: Alle Elemente ab dieser Stelle rücken eine Position nach rechts.

```
noten = [2, 5, 1, 4, 2]
noten.append(6)
print(noten)

einkauf = ["Brot", "Milch", "Eier"]
einkauf.insert(1, "Käse")
print(einkauf)
```

```
[2, 5, 1, 4, 2, 6]
['Brot', 'Käse', 'Milch', 'Eier']
```

Die 6 steht jetzt am Index 5. Der Käse steht am Index 1, die Milch ist von Index 1 auf Index 2 gerückt.

`pop` und `remove`

`liste.pop(index)` entfernt das Element **am Index**, ohne Index das **letzte**. `pop` gibt das entfernte Element zurück, so wie eine Funktion mit `return`. `liste.remove(wert)` sucht ein **Element mit diesem Wert** und entfernt das erste Vorkommen.

```
einkauf = ["Brot", "Käse", "Milch", "Eier"]
letztes = einkauf.pop()
print(letztes)
print(einkauf)
erstes = einkauf.pop(0)
print(erstes)
print(einkauf)

einkauf = ["Brot", "Käse", "Milch", "Käse"]
einkauf.remove("Käse")
print(einkauf)
```

```
Eier
['Brot', 'Käse', 'Milch']
Brot
['Käse', 'Milch']
['Brot', 'Milch', 'Käse']
```

Methode	Wirkung	in den Klammern steht
`liste.append(wert)`	hängt `wert` ans Ende an	ein Element
`liste.insert(i, wert)`	fügt `wert` am Index `i` ein	ein Index und ein Element
`liste.pop()`	entfernt das letzte Element und gibt es zurück	nichts
`liste.pop(i)`	entfernt das Element am Index `i` und gibt es zurück	ein Index
`liste.remove(wert)`	entfernt das erste Element mit dem Wert `wert`	ein Element

⚠️ **Stolperstein:** Ist der Wert nicht in der Liste, meldet `remove` einen `ValueError: list.remove(x): x not in list`. Aus einer leeren Liste kann `pop` nichts entfernen: `IndexError: pop from empty list`.

📌 **Merksatz:** `append` hängt an, `insert` fügt ein, `pop` entfernt am **Index**, `remove` entfernt ein **Element** mit einem bestimmten Wert.

🔮 Übung 2 – Listen-Vorhersage

Lies das Programm, **ohne** es auszuführen. Zeichne die Liste `tiere` nach jeder Zeile als Reihe von Kästchen mit dem Index darüber. Gib dann die Ausgabe an und prüfe am Computer.

```
tiere = ["Hund", "Katze", "Maus"]
tiere.append("Igel")
tiere[0] = "Pferd"
tiere.insert(2, "Esel")
tiere.remove("Katze")
x = tiere.pop()
print(tiere)
print(x)
print(len(tiere))
```

```
['Pferd', 'Esel', 'Maus']
Igel
3
```

Die Liste nach jeder Zeile und meine Vorhersage:

…

5 Listen mit Schleifen durchlaufen

Über die Elemente einer Liste läufst du wie über die Zeichen einer Zeichenkette. Brauchst du auch den **Index**, läufst du mit `range(len(liste))` über alle gültigen Indizes.

```
staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]
for stadt in staedte:
    print("Ich fahre nach", stadt)

for i in range(len(staedte)):
    print(i, staedte[i])
```

```
Ich fahre nach Saarbrücken
Ich fahre nach Trier
Ich fahre nach Mainz
Ich fahre nach Koblenz
0 Saarbrücken
1 Trier
2 Mainz
3 Koblenz
```

🧪 **Probiere es aus:** Ändere die zweite Schleife zu `for i in range(1, len(staedte) + 1):`. Was gibt das Programm aus, bevor es abbricht? Welche Indizes erzeugt `range` jetzt, und welcher davon ist ungültig?

Eine Liste mit einer Schleife füllen

Ein sehr häufiges Muster: mit einer **leeren Liste** beginnen und in der Schleife ein Element nach dem anderen anhängen.

```
quadratzahlen = []
for zahl in range(1, 6):
    quadratzahlen.append(zahl * zahl)
print(quadratzahlen)
```

```
[1, 4, 9, 16, 25]
```

Stünde `print` eingerückt in der Schleife, würde die Liste fünfmal ausgegeben, jedes Mal ein Stück länger.

> 📌 **Merksatz:** `for element in liste:` durchläuft die Elemente, `for i in range(len(liste)):` durchläuft die Indizes.

📝 Übungen: Schritt für Schritt selbstständig

Vorgemacht ist das Muster **Eine Liste mit einer Schleife füllen** (`quadratzahlen`). Bei Übung 3 füllst du Lücken `___` aus, bei Übung 4 steht ein Gerüst aus Kommentaren, die Zusatzaufgabe schreibst du allein.

📝 Übung 3 – Einkaufsliste (Lücken füllen)

Lege eine leere Liste an. Frage nacheinander 5 Artikelnamen ab (mit `input()` in einer Schleife) und füge sie mit `append()` hinzu. Gib die vollständige Liste am Ende aus.

Lückencode

```
einkaufsliste = ___
for i in range(___):
    artikel = input("Artikel eingeben: ")
    einkaufsliste.___(artikel)
print(einkaufsliste)
```

📝 Übung 4 – Index vs. Wert (mit Gerüst)

Gegeben ist `staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]`.

1. Gib den Wert am Index 2 aus.
2. Ändere den Wert am Index 0 zu `"Saarlouis"` und gib die Liste aus.
3. Gib den **Index** der Stadt `"Mainz"` aus.

> 🧰 **Werkzeug:** `liste.index(wert)` liefert den Index des ersten Elements mit diesem Wert. Ist der Wert nicht in der Liste, kommt ein `ValueError`. In DS 12 programmierst du so eine Suche selbst.

```
staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]

# 1. den Wert am Index 2 ausgeben

# 2. den Wert am Index 0 zu "Saarlouis" ändern und die Liste ausgeben

# 3. den Index von "Mainz" ausgeben (erwartet: 2)
```

🚀 Zusatzaufgabe – Liste umkehren (selbstständig)

Erzeuge aus `zahlen = [1, 2, 3, 4, 5]` mit einer Schleife eine **neue** Liste mit den Elementen in umgekehrter Reihenfolge. Verwende keine eingebaute Funktion zum Umkehren. Kontrolle: `[5, 4, 3, 2, 1]`.

> 💡 **Tipp:** `range(4, -1, -1)` erzeugt 4, 3, 2, 1, 0. Die mittlere Zahl ist die Grenze, die **nicht** mehr erreicht wird.

```
zahlen = [1, 2, 3, 4, 5]

# Zusatzaufgabe – Liste per Schleife umkehren
```

📌 Das Wichtigste in Kürze

- Eine **Liste** (ein **Feld**) fasst mehrere **Elemente** in einer festen Reihenfolge zu einer Einheit zusammen: `noten = [2, 3, 1, 4, 2]`.
- Jedes Element hat einen **Index**, beginnend bei 0. Index und Element sind verschiedene Dinge: `noten[2]` ist das Element am Index 2.
- **Lesen:** `liste[i]`. **Schreiben:** `liste[i] = wert`. `liste[-1]` ist das letzte Element.
- `len(liste)` liefert die Anzahl der Elemente. Gültige Indizes sind `0` bis `len(liste) - 1`, sonst kommt ein `IndexError`.
- `append` hängt an, `insert` fügt ein, `pop` entfernt am Index, `remove` entfernt ein Element mit einem bestimmten Wert.
- `for element in liste:` durchläuft die Elemente, `for i in range(len(liste)):` durchläuft die Indizes.

Zum Weiterüben: `DS11_Weitere_Uebungen.ipynb`.

DS 11 – Weitere Übungen

Listen: viele Werte unter einem Namen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Wochentage ●○○

Gegeben ist die Liste `tage = ["Mo", "Di", "Mi", "Do", "Fr"]`. Bestimme **ohne Computer** die Werte der Ausdrücke. Prüfe danach mit `print(...)` am Computer.

`tage[1]`	`tage[-1]`	`len(tage)`	`tage[len(tage) - 3]`

Gib außerdem an, an welchem Index das Element `"Do"` steht.

Deine Antwort:

…

Aufgabe 2: Liste verändern ●○○

Gib **ohne Computer** an, was das Programm ausgibt. Überprüfe danach am Computer.

```
1 punkte = [12, 7, 9]
2 punkte[0] = 10
3 punkte.append(15)
4 punkte[1] = punkte[1] + 3
5 print(punkte)
6 print(len(punkte))
```

```
[10, 10, 9, 15]
4
```

Deine Antwort:

…

Aufgabe 3: Stundenplan ●○○

Am Montag hast du die Fächer Mathe, Deutsch, Informatik und Sport.

1. Lege eine Liste `montag` mit diesen vier Fächern an.
2. Gib das erste und das letzte Fach aus.
3. Sport fällt aus, stattdessen ist Kunst. Ändere das Element mit einem schreibenden Zugriff.
4. Nach der Schule ist noch die Theater-AG. Hänge `"Theater-AG"` an die Liste an.
5. Gib die Liste und die Anzahl der Einträge aus.

Aufgabe 4: IndexError erklären ●○○

Das Programm soll die letzte Farbe der Liste ausgeben. Es bricht aber ab.

```
farben = ["rot", "gelb", "grün"]
print(farben[3])
```

```
Traceback (most recent call last):
  File "<Zelle 2>", line 2, in <module>
IndexError: list index out of range
```

1. Gib an, welche Indizes in dieser Liste gültig sind.
2. Erkläre, warum die Fehlermeldung `IndexError: list index out of range` erscheint.
3. Nenne zwei Möglichkeiten, die letzte Farbe richtig auszugeben.

Deine Antwort:

…

Aufgabe 5: Preiserhöhung am Kiosk ●●○

Am Schulkiosk werden alle Preise um 50 Cent teurer. Die Preise stehen in der Liste `preise = [1.5, 2.0, 0.8, 3.0]` (in Euro).

Schreibe ein Programm, das mit einer Schleife **jedes Element der Liste** um 0.5 erhöht. Gib die Liste danach aus.

> **Tipp:** Du musst die Elemente **in der Liste** verändern. Dafür brauchst du die Schleife über die Indizes: `for i in range(len(preise)):`. Im Rumpf steht dann ein schreibender Zugriff wie im Auftrag *Liste verändern*.

Aufgabe 6: Siegerliste ●●○

Die Liste `sieger = ["Lea", "Tom", "Ada", "Ben"]` enthält die Namen nach der Platzierung beim Sportfest. Schreibe ein Programm, das mit einer Schleife diese Ausgabe erzeugt:

Ausgabe:

```
Platz 1 - Lea
Platz 2 - Tom
Platz 3 - Ada
Platz 4 - Ben
```

Aufgabe 7: Zwei Mannschaften ●●○

Beim Fußball im Sportunterricht stehen alle in einer Reihe: `reihe = ["Ada", "Ben", "Cem", "Dana", "Eli", "Finn"]`. Wer an einem geraden Index steht (0, 2, 4), kommt in Mannschaft A, die anderen in Mannschaft B.

Schreibe ein Programm, das zwei neue Listen `team_a` und `team_b` mit `append` füllt und beide ausgibt.

> **Tipp:** Denk an die Schrittweite von `range` aus dem Abschnitt über die for-Schleife: `range(0, 6, 2)` erzeugt 0, 2, 4. Statt 6 schreibst du besser `len(reihe)`.

Aufgabe 8: Plätze tauschen ●●●

Im Sitzplan `sitzplan = ["Ben", "Ada", "Tom", "Mia", "Lea"]` wollen Tom und Lea die Plätze tauschen.

1. Bestimme mit `.index()` die Indizes von Tom und Lea.
2. Tausche die beiden Elemente mit schreibenden Zugriffen. Gib den Sitzplan vor und nach dem Tausch aus.
3. Erkläre, warum du dafür eine **Hilfsvariable** brauchst.

> **Tipp:** Stell dir zwei Gläser vor: eins mit Saft, eins mit Wasser. Um die Inhalte zu tauschen, brauchst du ein drittes, leeres Glas. Im Programm ist das dritte Glas eine Hilfsvariable, z. B. `merke = sitzplan[i]`.

DS 12 – Algorithmen auf Listen: Summe, Minimum und Suche

Informatik 10 – MINT-Zweig | Datenstrukturen

Lernziele

Ich kann

- mit einer Schleife über die Elemente oder über die Indizes einer Liste laufen und die passende Art auswählen,
- die Summe und den Mittelwert aller Elemente einer Liste berechnen,
- das kleinste und das größte Element einer Liste bestimmen,
- mit der linearen Suche herausfinden, ob und an welchem Index ein Wert in einer Liste vorkommt,
- zählen, wie oft ein Wert in einer Liste vorkommt,
- meine Ergebnisse mit `sum()`, `min()`, `max()` und `sorted()` kontrollieren.

🗣️ Sprachhilfe

- *„Die Funktion findet Tom an Stelle 2."* → „Der Funktion `lineare_suche` werden die Liste `namen` und der Wert `"Tom"` übergeben. Sie durchläuft die Liste Element für Element und gibt den Index 2 zurück."
- *„Bei Lea kommt −1."* → „Ist der gesuchte Wert nicht in der Liste enthalten, gibt die Funktion `−1` zurück."
- *„maximum merkt sich die größte Zahl."* → „Die Hilfsvariable `maximum` wird mit dem ersten Listenelement initialisiert und aktualisiert, sobald ein größeres Element gefunden wird."

Worum es geht

In einer Liste stehen die Noten einer Klassenarbeit. Wie gut war die Klasse im Durchschnitt? Was war die beste Note? Wie viele Einsen gab es? Ein Computer kann nicht „auf einen Blick" sehen. Er muss die Liste **Element für Element** durchgehen. Ein **Algorithmus** ist eine genaue Schritt-für-Schritt-Anleitung, so eindeutig, dass auch ein Computer sie ausführen kann.

1 Zwei Arten, eine Liste zu durchlaufen

Schleife	Schleifenvariable	geeignet, wenn …
`for element in liste:`	jedes **Element**	nur Werte nötig sind
`for i in range(len(liste)):`	jeder **Index**	die Position nötig ist

🔮 **Vorhersage:** Beide Zellen sollen alle Preise verdoppeln. Welche schafft es?

```python
preise = [2.0, 3.5, 1.0]
for i in range(len(preise)):
    preise[i] = preise[i] * 2
print(preise)
```

```
[4.0, 7.0, 2.0]
```

```python
preise = [2.0, 3.5, 1.0]
for preis in preise:
    preis = preis * 2
print(preise)
```

```
[2.0, 3.5, 1.0]
```

In der ersten Zelle steht ein schreibender Zugriff `preise[i] = …`. In der zweiten bekommt `preis` in jedem Durchlauf nur den **Wert** eines Elements. `preis = preis * 2` ändert nur diese Variable, die Liste bekommt davon nichts mit. Es gibt keine Fehlermeldung, die Liste bleibt einfach gleich.

> 📌 **Merksatz:** Über eine Liste kann man **über die Elemente** (`for element in liste:`) oder **über die Indizes** (`for i in range(len(liste)):`) iterieren. Zum Verändern der Liste braucht man die Indizes.

2 Summenbildung

An der Supermarktkasse werden die Preise nacheinander zu einer **Zwischensumme** addiert. Am Anfang steht sie auf 0, bei jedem Artikel kommt sein Preis dazu. Genauso arbeitet der Algorithmus. Die Zwischensumme ist eine **Hilfsvariable**.

```
1 zahlen = [7, 2, 9, 4, 1, 8]
2 summe = 0
3 for zahl in zahlen:
4     summe = summe + zahl
5 print("Summe:", summe)
```

```
Summe: 31
```

- **Zeile 2:** `summe` wird **vor** der Schleife mit 0 **initialisiert**.
- **Zeilen 3–4:** In jedem Durchlauf wird das aktuelle Element addiert.
- **Zeile 5:** Erst **nach** der Schleife steht das Endergebnis in `summe`.

Ablaufprotokoll für die kürzere Liste `[7, 2, 9]`:

Schritt	Zeile	Anweisung	`zahl`	`summe`
1	1	`zahlen = [7, 2, 9]`	-	-
2	2	`summe = 0`	-	`0`
3	3	`for zahl in zahlen:`	`7`	`0`
4	4	`summe = summe + zahl`	`7`	`7`
5	3	`for zahl in zahlen:`	`2`	`7`
6	4	`summe = summe + zahl`	`2`	`9`
7	3	`for zahl in zahlen:`	`9`	`9`
8	4	`summe = summe + zahl`	`9`	`18`
9	3	`for zahl in zahlen:`	`9`	`18`
10	5	`print("Summe:", summe)`	`9`	`18`

> ⚠️ **Stolperstein:** Steht `summe = 0` **in** der Schleife, wird die Zwischensumme bei jedem Durchlauf wieder auf 0 gesetzt. Am Ende steht dann nur das letzte Element in `summe`.

Der Mittelwert

Den **Mittelwert** (Durchschnitt) erhältst du, wenn du die Summe durch die Anzahl `len(liste)` teilst. Das funktioniert für jede Liste, egal wie viele Noten darin stehen.

```python
noten = [2, 3, 1, 4, 2]
summe = 0
for note in noten:
    summe = summe + note
mittelwert = summe / len(noten)
print("Durchschnitt:", mittelwert)
```

```
Durchschnitt: 2.4
```

📌 **Merksatz: Summenbildung:** Eine Hilfsvariable wird vor der Schleife auf `0` gesetzt und in jedem Durchlauf um das aktuelle Element erhöht. **Mittelwert** = Summe / `len(liste)`.

3 Maximum und Minimum

Stell dir vor, du deckst Spielkarten nacheinander auf. Die erste Karte behältst du in der Hand. Bei jeder weiteren vergleichst du: Ist die neue Zahl größer? Dann tauschst du. Am Ende hältst du die größte Zahl in der Hand. Die „Karte in der Hand“ ist wieder eine Hilfsvariable.

```python
zahlen = [7, 2, 9, 4, 1, 8]
maximum = zahlen[0]
for zahl in zahlen:
    if zahl > maximum:
        maximum = zahl
print("Maximum:", maximum)
```

```
Maximum: 9
```

- **Zeile 2:** `maximum` wird mit dem **ersten Element** initialisiert, der ersten Karte.
- **Zeilen 4–5:** Nur wenn das aktuelle Element größer ist, wird `maximum` **aktualisiert**.

Für das **Minimum** dreht man den Vergleich um: `<` statt `>`.

⚠️ Stolperstein: Warum nicht mit 0 beginnen?

🔮 **Vorhersage:** Was gibt das Programm aus? Was wäre richtig?

```python
temperaturen = [-5, -2, -8]
maximum = 0
for t in temperaturen:
    if t > maximum:
        maximum = t
print("Höchste Temperatur:", maximum)
```

```
Höchste Temperatur: 0
```

Keine Temperatur ist größer als 0, also wird `maximum` nie aktualisiert. Ausgegeben wird 0, ein Wert, der gar nicht in der Liste steht. Richtig wäre −2. Mit `maximum = temperaturen[0]` stimmt das Ergebnis.

📌 **Merksatz: Minimum/Maximum:** Eine Hilfsvariable wird mit dem **ersten Element** initialisiert und bei jedem kleineren bzw. größeren Fund aktualisiert.

4 Lineare Suche und Zählen

Du suchst in einem unsortierten Stapel Klassenarbeiten die von Tom. Du nimmst die oberste, schaust auf den Namen, dann die nächste, bis du Toms Arbeit gefunden hast oder der Stapel zu Ende ist. Das ist die **lineare Suche**. Die Funktion gibt den **Index** des Treffers zurück, sonst `-1`.

```python
def lineare_suche(liste, gesuchter_wert):
    for i in range(len(liste)):
        if liste[i] == gesuchter_wert:
            return i
    return -1

namen = ["Ben", "Ada", "Tom", "Mia"]
print(lineare_suche(namen, "Tom"))
print(lineare_suche(namen, "Lea"))
```

```
2
-1
```

- **Zeile 2:** Weil der Index zurückgegeben werden soll, läuft die Schleife über die **Indizes**.
- **Zeile 4:** Bei einem Treffer beendet `return i` die Funktion **sofort**, auch mitten in der Schleife. Mia wird gar nicht mehr angeschaut.
- **Zeile 5:** wird nur erreicht, wenn die Schleife ohne Treffer zu Ende gelaufen ist.

Warum `-1`? Von vorne gezählt ist ein Index nie negativ. Aber Vorsicht: `namen[-1]` wäre `"Mia"`! Prüfe das Ergebnis also mit `if position == -1:`, bevor du es als Index benutzt.

⚠️ Stolperstein: `return -1` im `else`

🔮 **Vorhersage:** Tom steht in der Liste. Was gibt diese Fassung aus, und warum?

```python
def lineare_suche(liste, gesuchter_wert):
    for i in range(len(liste)):
        if liste[i] == gesuchter_wert:
            return i
        else:
            return -1

namen = ["Ben", "Ada", "Tom", "Mia"]
print(lineare_suche(namen, "Tom"))
```

```
-1
```

Beim ersten Element `"Ben"` ist der Vergleich falsch, der `else`-Teil wird ausgeführt, und `return` beendet die Funktion sofort. „Nicht gefunden“ steht erst fest, wenn **alle** Elemente verglichen wurden, also **nach** der Schleife.

> 📌 **Merksatz: Lineare Suche:** Man durchläuft die Liste Element für Element und vergleicht mit dem Suchwert, bis er gefunden wurde oder die Liste zu Ende ist.

Zählen

Beim Zählen führt man eine **Strichliste**: Die Hilfsvariable beginnt bei 0 und wird um **1** erhöht, wenn ein Element die Bedingung erfüllt.

```python
noten = [2, 3, 1, 4, 2, 5, 1]
anzahl = 0
for note in noten:
    if note <= 2:
        anzahl = anzahl + 1
print("Anzahl der Noten 1 oder 2:", anzahl)
```

```
Anzahl der Noten 1 oder 2: 4
```

> 📌 **Merksatz: Zählen:** Hilfsvariable mit 0 beginnen und um 1 erhöhen, wenn ein Element die Bedingung erfüllt.

5 Übungen

🧰 Werkzeug: fertige Funktionen zur Kontrolle

Python bringt für diese Aufgaben fertige Funktionen mit. Benutze sie, um deine eigenen Funktionen zu prüfen, zum Beispiel `print(minimum(zahlen) == min(zahlen))`.

Funktion	Ergebnis
`sum(liste)`	die Summe aller Elemente
`min(liste)`, `max(liste)`	das kleinste bzw. größte Element
`sorted(liste)`	eine **neue**, aufsteigend sortierte Liste
`wert in liste`	`True`, wenn `wert` vorkommt

```
zahlen = [7, 2, 9, 4, 1, 8]
print(sum(zahlen))
print(min(zahlen), max(zahlen))
print(sorted(zahlen))
print(zahlen)
print(9 in zahlen)
```

```
31
1 9
[1, 2, 4, 7, 8, 9]
[7, 2, 9, 4, 1, 8]
True
```

Schritt für Schritt selbstständig

Vorgemacht sind die Muster **Summe** und **Maximum** oben. Bei Übung 1 füllst du Lücken `___` aus, bei Übung 2 steht ein Gerüst aus Kommentaren, bei Übung 3 nur die Kopfzeile und der Test. Übung 4 ist eine Leseaufgabe, die Zusatzaufgabe schreibst du allein.

📝 Übung 1 – Durchschnitt berechnen (Lücken füllen)

Schreibe eine Funktion `durchschnitt(liste)`, die den Durchschnitt aller Zahlen einer Liste zurückgibt. Kontrolle: `durchschnitt([7, 2, 9, 4, 1, 8])` ergibt etwa 5,17 (Summe 31, 6 Elemente).

Lückencode

```
def durchschnitt(liste):
    summe = ___
    for wert in liste:
        summe = ___
    return ___ / len(liste)

print(durchschnitt([7, 2, 9, 4, 1, 8]), "erwartet: etwa 5.17")
```

📝 Übung 2 – Minimum finden (mit Gerüst)

Schreibe eine Funktion `minimum(liste)`, die das kleinste Element zurückgibt, ohne `min()` zu verwenden. Teste mit `[7, 2, 9, 4, 1, 8]` und mit einer Liste aus negativen Zahlen. Vergleiche mit `min()`.

```python
def minimum(liste):
    # Hilfsvariable mit dem ersten Element initialisieren
    # für jeden Wert in der Liste:
    #     wenn der Wert kleiner ist als die Hilfsvariable:
    #         Hilfsvariable aktualisieren
    # nach der Schleife: Hilfsvariable zurückgeben
    pass

print(minimum([7, 2, 9, 4, 1, 8]), "erwartet: 1")
print(minimum([-5, -2, -8]), "erwartet: -8")
```

```
None erwartet: 1
None erwartet: -8
```

📝 Übung 3 – Häufigkeit zählen (Kopfzeile und Test vorgegeben)

Schreibe eine Funktion `zaehle_vorkommen(liste, wert)`, die zählt, wie oft `wert` in `liste` vorkommt. Anders als die lineare Suche soll sie **alle** Vorkommen zählen. Kontrolle: `zaehle_vorkommen([1, 2, 2, 3, 2, 4], 2)` ergibt 3.

```python
def zaehle_vorkommen(liste, wert):
    pass   # hier deine Programmzeilen (Vorbild: Zählen mit Strichliste)

print(zaehle_vorkommen([1, 2, 2, 3, 2, 4], 2), "erwartet: 3")
```

```
None erwartet: 3
```

🔍 Übung 4 – Code lesen

Lies die Funktion, **ohne** sie auszuführen.

1. Gib an, was das Programm ausgibt.
2. Beschreibe in einem Satz, was `geheim` allgemein berechnet.
3. Erkläre, warum das Ergebnis nicht `3` ist, obwohl am Index 3 auch eine 8 steht.

```python
def geheim(liste):
    position = 0
    for i in range(len(liste)):
        if liste[i] > liste[position]:
            position = i
    return position

print(geheim([3, 8, 5, 8, 1]))
```

```
1
```

Meine Antworten:

…

🚀 Zusatzaufgabe – Zwei größte Werte (selbstständig)

Finde mit einer Schleife die **zwei größten** Werte der Liste und gib sie aus, ohne `sorted()`. Kontrolle: 22 und 17.

> 💡 **Tipp:** Merke dir in **zwei** Hilfsvariablen den größten und den zweitgrößten Wert. Initialisiere sie mit den ersten beiden Elementen, in der richtigen Reihenfolge. Ist ein weiteres Element größer als der größte Wert, oder nur größer als der zweitgrößte?

```python
zahlen = [4, 17, 9, 22, 3, 15]

# Zusatzaufgabe – zwei größte Werte finden
```

📌 Das Wichtigste in Kürze

- Über eine Liste iteriert man über die Elemente (`for element in liste:`) oder über die Indizes (`for i in range(len(liste)):`). Zum Verändern braucht man die Indizes.
- **Summe:** Hilfsvariable vor der Schleife auf 0 setzen, in jedem Durchlauf das Element addieren. **Mittelwert** = Summe / `len(liste)`.
- **Maximum/Minimum:** Hilfsvariable mit dem ersten Element initialisieren und bei jedem größeren bzw. kleineren Element aktualisieren.
- **Lineare Suche:** Element für Element vergleichen, bis der Wert gefunden oder die Liste zu Ende ist. Ergebnis: der Index (oder `-1`) bzw. `True`/`False`.
- **Zählen:** Hilfsvariable mit 0 beginnen und um 1 erhöhen, wenn ein Element die Bedingung erfüllt.
- `sum`, `min`, `max`, `sorted` und `in` erledigen das fertig, gut zur Kontrolle eigener Funktionen.

Zum Weiterüben: `DS12_Weitere_Uebungen.ipynb`.

DS 12 – Weitere Übungen

Algorithmen auf Listen: Summe, Minimum und Suche

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Maximum von Hand ●○○

Das Programm sucht das Maximum der Liste `[4, 9, 2, 9, 6]`:

```
zahlen = [4, 9, 2, 9, 6]
maximum = zahlen[0]
for zahl in zahlen:
    if zahl > maximum:
        maximum = zahl
print("Maximum:", maximum)
```

```
Maximum: 9
```

Fülle die Tabelle aus: Ist das Element größer als das bisherige Maximum? Welchen Wert hat `maximum` nach diesem Durchlauf? Gib dann an, wie oft `maximum` insgesamt geändert wird.

Durchlauf	`zahl`	`zahl > maximum?`	`maximum` danach
vor der Schleife	–	–	`4`
1	`4`		
2	`9`		
3	`2`		
4	`9`		
5	`6`		

Deine Antwort:

…

Aufgabe 2: Summe und Zählen ●○○

In der Liste stehen die Punkte, die ein Team in sechs Spielen geholt hat. Gib **ohne Computer** an, was das Programm ausgibt, und beschreibe in einem Satz, was `anzahl` zählt.

```
punkte = [3, 0, 1, 3, 0, 3]
summe = 0
anzahl = 0
for p in punkte:
    summe = summe + p
    if p == 0:
        anzahl = anzahl + 1
print(summe, anzahl)
```

```
10 2
```

Deine Antwort:

…

Aufgabe 3: Playlist ●○○

In einer Liste steht die Länge jedes Songs einer Playlist in Minuten: `songs = [3, 4, 2, 5, 3]`.

1. Implementiere eine Funktion `gesamtdauer(liste)`, die die Summe aller Elemente zurückgibt.
2. Implementiere eine Funktion `mittlere_dauer(liste)`, die die durchschnittliche Länge zurückgibt. Sie soll `gesamtdauer` aufrufen.
3. Teste beide Funktionen mit der Playlist. Erwartet sind 17 Minuten und 3,4 Minuten.

Aufgabe 4: Kälteste Nacht ●○○

In einer Woche wurden nachts diese Temperaturen gemessen (in °C): `nachts = [-3, 2, -7, 0, 4, -1, 1]`.

Implementiere eine Funktion `kaelteste(temperaturen)`, die die niedrigste Temperatur zurückgibt. Verwende nicht `min()`. Teste mit der Liste. Erwartet ist −7.

Aufgabe 5: Freies Schließfach ●●○

In der Liste `faecher` steht, wem welches Schließfach gehört. Ein leeres Fach ist mit der leeren Zeichenkette `""` gekennzeichnet:

```
faecher = ["Ada", "Ben", "", "Mia", ""]
```

1. Implementiere eine Funktion `erstes_freies(faecher)`, die mit einer linearen Suche den **Index** des ersten freien Fachs zurückgibt. Gibt es kein freies Fach, soll sie `-1` zurückgeben.
2. Gib im Hauptprogramm aus, welches Fach frei ist. Die Fächer sind für Menschen ab 1 nummeriert, also ist der Index 2 das Fach Nummer 3. Ist kein Fach frei, soll „Alle Fächer sind belegt.“ erscheinen.
3. Teste auch mit einer Liste ganz ohne freies Fach.

Aufgabe 6: Zwei Fehler im Maximum ●●○

Die Funktion soll die beste Tordifferenz einer Saison zurückgeben. Sie liefert falsche Ergebnisse.

```
def beste_differenz(differenzen):
    beste = 0
    for d in differenzen:
        if d > beste:
            beste = d
        return beste
```

1. Teste die Funktion mit `[3, -1, 5]` (erwartet: 5) und mit `[-2, -1, -5]` (erwartet: −1). Notiere die Ergebnisse.
2. Erkläre die beiden Fehler und verbessere die Funktion.

Deine Antwort:

…

Aufgabe 7: Frosttage ●●○

Ein Tag mit einer Temperatur **unter** 0 °C heißt Frosttag.

1. Implementiere eine Funktion `frosttage(temperaturen)`, die zählt, wie viele Werte der Liste unter 0 liegen.
2. Teste mit der Liste aus dem Auftrag *Kälteste Nacht* (erwartet: 3) und mit `[0, 1, 2]` (erwartet: 0).

Aufgabe 8: Besser als der Durchschnitt ●●●

Implementiere eine Funktion `ueber_durchschnitt(liste)`, die zählt, wie viele Elemente **größer** als der Durchschnitt der Liste sind. Teste mit:

Aufruf	erwartet	tatsächlich
`ueber_durchschnitt([2, 4, 6, 8])`	`2`	
`ueber_durchschnitt([10, 10, 10])`	`0`	
`ueber_durchschnitt([1, 1, 1, 9])`	`1`	

> **Tipp:** Du brauchst **zwei** Schritte nacheinander. Erst berechnest du mit einer Schleife den Durchschnitt. Erst danach kannst du mit einer zweiten Schleife zählen, denn vorher kennst du den Vergleichswert noch nicht.

DS 13 – Teamprojekt: Verschlüsselung mit der Caesar-Chiffre (Teil 1)

Informatik 10 – MINT-Zweig | Projektarbeit

Lernziele

Ich kann

- das Verfahren der Caesar-Chiffre beschreiben und von Hand anwenden,
- die Position eines Buchstabens im Alphabet bestimmen und mit `% 26` am Ende des Alphabets vorne weiterzählen,
- Funktionen zum Verschlüsseln eines Buchstabens und eines ganzen Textes implementieren und systematisch testen,
- ein Programmierprojekt im Team planen, Teilaufgaben verteilen und die Ergebnisse zusammenführen.

🗣️ Sprachhilfe

- „Der Buchstabe … hat im Alphabet die Position … . Mit dem Schlüssel … ergibt sich die Position …, also der Buchstabe … .“
- „Am Ende des Alphabets rechnen wir `% 26`, damit nach Z wieder A kommt: … % 26 ergibt … .“
- „Die Funktion … bekommt die Parameter … und gibt … zurück. Das ist ihre Schnittstelle.“
- „Wir haben mit dem Randfall … getestet, weil dort am ehesten ein Fehler steckt.“

1 Wie die Caesar-Chiffre funktioniert

Wie verschickt man eine Nachricht so, dass nur Eingeweihte sie lesen können? Julius Caesar soll ein einfaches Verfahren benutzt haben: Jeder Buchstabe wird im Alphabet um eine feste Anzahl von Stellen nach hinten verschoben. Diese Anzahl ist der **Schlüssel**. Am Ende des Alphabets geht es vorne weiter.

Schlüssel 3:

Klartext	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Geheimtext	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	**A**	**B**	**C**

So wird aus dem **Klartext** `HALLO` der **Geheimtext** `KDOOR`: H → K, A → D, L → O, L → O, O → R. Wer den Schlüssel kennt, entschlüsselt, indem er jeden Buchstaben um 3 Stellen **zurück** verschiebt.

> 📌 **Definition:** Beim **Verschlüsseln** wird ein lesbarer **Klartext** mithilfe eines **Schlüssels** in einen **Geheimtext** umgewandelt. Beim **Entschlüsseln** wird aus dem Geheimtext mit dem Schlüssel wieder der Klartext.

✏️ Übung 1 – Von Hand verschlüsseln

Benutze die Tabelle oben.

1. Verschlüssele das Wort `INFORMATIK`.
2. Entschlüssele den Geheimtext `ZDVVHU`.
3. Beschreibe in einem Satz, wie du beim Entschlüsseln vorgegangen bist.

I	N	F	O	R	M	A	T	I	K

Meine Antworten zu 2 und 3:

...

2 Das Projekt im Team planen

Professionelle Softwareteams tippen nicht alle gleichzeitig drauflos. Sie gehen in Schritten vor:

1. **Verstehen und planen:** Was soll das Programm können? Wie löst man das Problem erst einmal ohne Computer?
2. **Zerlegen und verteilen:** Für jede Teilaufgabe wird eine Funktion geplant, jede Person übernimmt eine.
3. **Umsetzen und testen:** Jede Funktion wird mit vorher festgelegten Beispielen getestet.
4. **Zusammenführen:** Die Funktionen werden zu einem Programm zusammengesetzt und gemeinsam getestet.
5. **Dokumentieren und vorstellen.**

Damit ihr gleichzeitig arbeiten könnt, legt ihr **vorher** fest, wie jede Funktion heißt, welche Parameter sie bekommt und was sie zurückgibt.

> 📌 **Definition:** Die **Schnittstelle** einer Funktion legt fest, wie sie heißt, welche Parameter sie erwartet und welchen Rückgabewert sie liefert. Wer die Funktion benutzt, muss nur die Schnittstelle kennen.

Teil	Funktion mit Parametern	Rückgabewert
A	`verschluessle_buchstabe(buchstabe, schluessel)`	der verschobene Buchstabe
B	`verschluessle_text(text, schluessel)`	der Geheimtext
C	Hauptprogramm (keine Funktion)	– (fragt ab und gibt aus)

B benutzt A. Trotzdem kann man B schon programmieren, bevor A fertig ist: Man kennt ja die Schnittstelle von A.

✏️ Übung 2 – Projektplan

Plant gemeinsam auf Papier, **bevor** ihr programmiert:

1. Wie ermittelt man zu einem Buchstaben seine Position im Alphabet?
2. Wie kommt man von der Position zurück zum (verschobenen) Buchstaben?
3. Was passiert am Ende des Alphabets, z. B. bei `Z` mit dem Schlüssel 3?

Verteilt dann die Teilaufgaben. Niemand testet die eigene Funktion allein.

Teilaufgabe	programmiert von	getestet von	fertig?
A: einen Buchstaben verschlüsseln			
B: einen Text verschlüsseln			
C: interaktives Programm			

Unsere Antworten zu den Fragen 1 bis 3:

...

3 Werkzeuge für das Projekt

Das Alphabet als Zeichenkette

Den Index eines Zeichens findet ihr mit `.index()`, das kennt ihr von den Listen.

```python
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print(alphabet)
print("Position von C:", alphabet.index("C"))
print("Buchstabe an Position 5:", alphabet[5])
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Position von C: 2
Buchstabe an Position 5: F
```

`alphabet.index("C")` geht vom **Zeichen** zur **Position** (2), `alphabet[5]` von der **Position** zum **Zeichen** (F).

⚠️ Stolperstein: Zeichen, die nicht im Alphabet stehen

Führe die Zelle aus.

```python
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print(alphabet.index("a"))
```

```
Traceback (most recent call last):
  File "<Zelle 2>", line 2, in <module>
ValueError: substring not found
```

`ValueError: substring not found` heißt „Teilzeichenkette nicht gefunden“. Prüft deshalb vorher mit `zeichen in alphabet`, ob das Zeichen ein Großbuchstabe ist.

Am Ende des Alphabets: der Rest-Operator

X hat die Position 23, um 3 verschoben ergibt das 26. Die gibt es nicht. Nach Z soll es wieder bei A weitergehen, wie auf einer Uhr nach 12 wieder die 1 kommt. Das leistet der **Rest-Operator** `%`: Für Zahlen von 0 bis 25 ändert sich nichts, aus 26 wird 0, aus 27 wird 1.

🔮 **Vorhersage:** Welche vier Zeilen gibt die Schleife aus?

```python
for position in range(22, 26):
    neue_position = (position + 3) % 26
    print(position, "->", neue_position)
```

```
22 -> 25
23 -> 0
24 -> 1
25 -> 2
```

W (22) wird zu Z (25), X (23) zu A (0), Y zu B, Z zu C. Die Klammern sind wichtig: erst addieren, dann den Rest bilden.

> 📌 **Merksatz:** Mit `(position + schluessel) % 26` erhält man immer eine gültige Position von 0 bis 25. Am Ende des Alphabets geht es dadurch vorne weiter.

> 🧰 **Zweiter Weg (freiwillig):** Der Computer speichert jedes Zeichen als Zahl. `ord("A")` ist 65, `chr(68)` ist `"D"`. Die Position im Alphabet ist `ord(buchstabe) - ord("A")`, zurück geht es mit `chr(position + ord("A"))`. Entscheidet euch im Team für **einen** Weg.

4 Teilaufgabe A – einen Buchstaben verschlüsseln (Lücken füllen)

Die Teilaufgaben werden schrittweise selbstständiger: A ist ein Lückencode, B hat ein Gerüst aus Kommentaren, C setzt ihr allein zusammen.

Schreibt eine Funktion `verschluessle_buchstabe(buchstabe, schluessel)`, die **einen** Großbuchstaben um `schluessel` Stellen verschiebt und den neuen Buchstaben zurückgibt. Die drei Schritte kennt ihr aus Übung 2 (Projektplan): Position finden, verschieben und mit `% 26` umlaufen lassen, Buchstaben ablesen. Ersetzt jedes `___`.

> 💡 **Tipp:** Gute Tests prüfen nicht nur den **Normalfall** (A → D), sondern auch die **Randfälle**: das Ende des Alphabets (X, Y, Z) und besondere Schlüssel wie 0 oder 25.

Aufruf	erwartet	tatsächlich	✓ oder ✗
`verschluessle_buchstabe("A", 3)`	`D`		
`verschluessle_buchstabe("X", 3)`	`A`		
`verschluessle_buchstabe("Z", 1)`	`A`		
`verschluessle_buchstabe("M", 0)`	`M`		
`verschluessle_buchstabe("B", 25)`	`A`		

Lückencode

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def verschluessle_buchstabe(buchstabe, schluessel):
    position = alphabet.index(___)
    neue_position = (___ + ___) % 26
    return alphabet[___]

print(verschluessle_buchstabe("A", 3))    # erwartet: D
print(verschluessle_buchstabe("X", 3))    # erwartet: A
print(verschluessle_buchstabe("Z", 1))    # erwartet: A
print(verschluessle_buchstabe("M", 0))    # erwartet: M
print(verschluessle_buchstabe("B", 25))   # erwartet: A
```

5 Teilaufgabe B – einen ganzen Text verschlüsseln (mit Gerüst)

Ihr braucht das Muster *Einen neuen Text aufbauen* aus DS 10. Es funktioniert wie die Summenbildung, nur mit Text: Man beginnt mit `""` und hängt in der Schleife mit `+` Zeichen an.

```
text = "ABC"
ergebnis = ""
for zeichen in text:
    ergebnis = ergebnis + zeichen + zeichen
print(ergebnis)
```

```
AABBCC
```

1. Schreibt eine Funktion `verschluessle_text(text, schluessel)`, die in einer Schleife jeden Buchstaben mit `verschluessle_buchstabe` verschlüsselt und den Geheimtext zurückgibt. Kontrolle: `verschluessle_text("HALLO", 3)` ergibt `KDOOR`.
2. **Zusatz-Herausforderung:** Was passiert bei `"Hallo Welt!"`? Überlegt euch im Team eine Lösung, z. B. Leerzeichen und Satzzeichen unverändert lassen und den Text vorher mit `.upper()` umwandeln. Kontrolle: `KDOOR ZHOW!`.

Die Zelle benutzt eure Funktion aus Teilaufgabe A. Führt die Zelle oben deshalb zuerst aus.

```python
def verschluessle_text(text, schluessel):
    # ergebnis beginnt als leere Zeichenkette
    # für jedes Zeichen im Text:
    #     das Zeichen mit verschluessle_buchstabe verschlüsseln
    #     und an ergebnis anhängen
    # ergebnis zurückgeben
    pass

print(verschluessle_text("HALLO", 3), "erwartet: KDOOR")
# Zusatz-Herausforderung:
# print(verschluessle_text("Hallo Welt!", 3), "erwartet: KDOOR ZHOW!")
```

```
None erwartet: KDOOR
```

6 Teilaufgabe C – das interaktive Programm (selbstständig)

Setzt beide Funktionen zu einem vollständigen Programm zusammen, in dieser Reihenfolge: Alphabet, Funktion A, Funktion B, Hauptprogramm. Das Hauptprogramm fragt Klartext und Schlüssel mit `input()` ab und gibt den Geheimtext aus. Testet mit mindestens drei Eingaben. Kontrolle: „Treffen um acht“ mit Schlüssel 5 ergibt `YWJKKJS ZR FHMY`.

Schreibt an den Anfang einen **Kommentar** mit dem Programmnamen, eurem Team und einem Satz dazu, was das Programm tut.

```python
# Caesar-Chiffre: Texte verschlüsseln
# Team:
# Was das Programm tut:

# TODO: Teilaufgabe C – Alphabet, Funktion A, Funktion B, Hauptprogramm
```

📒 Projekttagebuch

Tragt am Ende der Arbeitsphase ein, was ihr geschafft habt, welche Probleme aufgetreten sind und was als Nächstes kommt. Ihr braucht es in DS 14.

Datum	erledigt	Probleme und wie wir sie gelöst haben	nächster Schritt

📌 Das Wichtigste in Kürze

- Die **Caesar-Chiffre** verschiebt jeden Buchstaben um eine feste Anzahl von Stellen (den **Schlüssel**) im Alphabet. Aus dem **Klartext** wird der **Geheimtext**.
- Die Position eines Buchstabens ist sein Index in `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"` (oder `ord(buchstabe) - ord("A")`).
- `(position + schluessel) % 26` sorgt dafür, dass es nach Z wieder bei A weitergeht.
- Einen neuen Text baut man auf, indem man mit `""` beginnt und in einer Schleife Zeichen anhängt.
- Im Team zerlegt man das Problem in Teilaufgaben, legt vorher die **Schnittstellen** fest, testet jede Funktion mit Normal- und Randfällen und führt die Teile am Ende zusammen.

> **In DS 14** entwickelt ihr die passende **Entschlüsselung** und recherchiert Berufsfelder rund um die Softwareentwicklung. Zum Weiterüben: `DS13_Weitere_Uebungen.ipynb`.

DS 13 – Weitere Übungen

Projekt Verschlüsselung (Teil 1): die Caesar-Chiffre

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Caesar mit Schlüssel 5 ●○○

Benutze das Alphabet mit Positionen. Ein Tipp: Schreibe dir zuerst die Positionen auf, addiere 5 und rechne bei Ergebnissen ab 26 noch 26 ab.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

1. Verschlüssele das Wort `PIZZA` mit dem Schlüssel 5.
2. Entschlüssele den Geheimtext `KJWNJS`. Er wurde mit dem Schlüssel 5 verschlüsselt.

Deine Antwort:

…

Aufgabe 2: Rechnen mit % 26 ●○○

Berechne zu jeder Zeile die neue Position `(position + schluessel) % 26` und den Buchstaben, der dort steht. Benutze die Positionstabelle aus dem vorigen Auftrag.

position	schluessel	(position + schluessel) % 26	Buchstabe
24	4		
10	13		
3	30		
25	26		

Was fällt dir in der letzten Zeile auf? Erkläre.

Deine Antwort:

…

Aufgabe 3: Zweimal verschlüsseln ●●○

Die Funktionen `verschluessle_buchstabe` und `verschluessle_text` (verbesserte Fassung aus Teilaufgabe B) stehen vor diesen beiden Zeilen:

```
print(verschluessle_text("HALLO", 13))
print(verschluessle_text(verschluessle_text("HALLO", 13), 13))
```

```
Traceback (most recent call last):
  File "<Zelle 1>", line 1, in <module>
NameError: name 'verschluessle_text' is not defined
```

1. Gib an, was die beiden Zeilen ausgeben.

2. Erkläre, warum in der zweiten Zeile wieder der Klartext herauskommt.
3. Gibt es einen weiteren Schlüssel zwischen 1 und 25 mit dieser Eigenschaft? Begründe.

Deine Antwort:

…

Aufgabe 4: Geheimtext in Fünfergruppen ●●●

Früher hat man Geheimtexte ohne Leerzeichen und in Gruppen zu je fünf Buchstaben aufgeschrieben. So verrät die Länge der Wörter nichts mehr über den Klartext.

1. Implementiere eine Funktion `fuenfergruppen(text)`. Sie entfernt zuerst alle Leerzeichen und fügt dann nach je fünf Zeichen ein Leerzeichen ein. Aus `"YWJKKJS ZR FHMY"` wird so `"YWJKK JSZRF HMY"`.
2. Teste die Funktion zusammen mit `verschluessle_text`: Verschlüssele „Treffen um acht“ mit dem Schlüssel 5 und gib das Ergebnis in Fünfergruppen aus.

> **Tipp:** Zerlege die Aufgabe in zwei Schritte. Schritt 1: Baue einen neuen Text ohne Leerzeichen auf (Zeichen nur anhängen, wenn es kein `" "` ist). Schritt 2: Laufe über die **Indizes** dieses Textes. Vor jedem Zeichen, dessen Index durch 5 teilbar ist (außer beim Index 0), hängst du zuerst ein Leerzeichen an. „Durch 5 teilbar“ prüfst du mit `i % 5 == 0`.

DS 14 – Teamprojekt: Verschlüsselung (Teil 2) und Berufe in der Softwareentwicklung

Informatik 10 – MINT-Zweig | Projektarbeit

Lernziele

Ich kann

- erklären, wie Entschlüsselung und Verschlüsselung bei der Caesar-Chiffre zusammenhängen, und eine Entschlüsselungsfunktion implementieren,
- mit dem Ausprobieren aller Schlüssel (Brute-Force) einen Geheimtext knacken und die Sicherheit der Caesar-Chiffre beurteilen,
- ein Programmierprojekt testen, dokumentieren und kurz vorstellen,
- Berufe, Berufsfelder und Berufsbiografien mit Bezug zur Softwareentwicklung mit seriösen Quellen recherchieren und die Ergebnisse geordnet festhalten.

🗣️ Sprachhilfe

- „Beim Entschlüsseln wird jeder Buchstabe um den Schlüssel **zurück** verschoben. Das ist dasselbe wie Verschlüsseln mit `26 - s`, weil …“
- „Der Hin-und-zurück-Test zeigt, dass …, denn nach dem Entschlüsseln kommt wieder … heraus.“
- „Die Caesar-Chiffre ist unsicher, weil es nur … sinnvolle Schlüssel gibt.“
- „Laut … (Quelle, Datum) gehört zu diesem Beruf …“

1 Rückblick: der Stand aus Teil 1

Eine geheime Nachricht nützt nur etwas, wenn man sie auch wieder **entschlüsseln** kann. Kopiert eure fertigen Funktionen aus DS 13 in die nächste Zelle. Sie sollten ungefähr so aussehen wie hier. Führt die Zelle aus: Es muss `KDOOR ZHOW` herauskommen.

```python
# Eure Funktionen aus DS 13 (oder diese Fassung)
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def verschluessle_buchstabe(buchstabe, schluessel):
    position = alphabet.index(buchstabe)
    neue_position = (position + schluessel) % 26
    return alphabet[neue_position]

def verschluessle_text(text, schluessel):
    ergebnis = ""
    for zeichen in text.upper():
        if zeichen in alphabet:
            neu = verschluessle_buchstabe(zeichen, schluessel)
            ergebnis = ergebnis + neu
        else:
            ergebnis = ergebnis + zeichen
    return ergebnis

print(verschluessle_text("HALLO WELT", 3))
```

```
KDOOR ZHOW
```

> 📌 **Merksatz:** Verschlüsseln mit der Caesar-Chiffre: Position des Buchstabens bestimmen, um den Schlüssel verschieben, mit `% 26` umlaufen lassen und den Buchstaben an der neuen Position ablesen. Zeichen, die keine Buchstaben sind, bleiben unverändert.

2 Entschlüsseln: zurück verschieben

Beim Entschlüsseln rechnet man `- schluessel` statt `+ schluessel`. Was passiert am **Anfang** des Alphabets? A hat die Position 0, und 0 − 3 = −3.

🔮 **Vorhersage:** Was geben die beiden Zeilen aus?

```
print((0 - 3) % 26)
print((1 - 3) % 26)
```

```
23
24
```

In Python liefert `% 26` immer ein Ergebnis von 0 bis 25, auch bei negativen Zahlen. Aus A (0) wird X (23), aus B (1) wird Y (24).

Zweiter Weg: Stell dir das Alphabet als Uhr mit 26 Stunden vor. 3 Schritte zurück landen an derselben Stelle wie 23 Schritte vorwärts. Zurückschieben um `s` ist dasselbe wie Vorwärtsschieben um `26 - s`.

> 📌 **Merksatz:** Entschlüsseln mit dem Schlüssel `s` entspricht dem Verschlüsseln mit `-s` oder `26 - s`.

📝 Teilaufgabe D – Entschlüsseln

Schreibt eine Funktion `entschluessle_text(geheimtext, schluessel)`. Ihr könnt `verschluessle_text` wiederverwenden: Die Funktion braucht nur eine Zeile. Ersetzt die beiden `___`. Testet mit einem **Hin-und-zurück-Test**: verschlüsseln, dann wieder entschlüsseln. Es muss der ursprüngliche Text herauskommen (in Großbuchstaben).

Klartext	Schlüssel	verschlüsselt	wieder entschlüsselt	✓ oder ✗
`HALLO WELT`	3			
`Treffen um acht`	5			

Lückencode

```
def entschluessle_text(geheimtext, schluessel):
    return verschluessle_text(___, ___)

geheim = verschluessle_text("HALLO WELT", 3)
print(geheim)
print(entschluessle_text(geheim, 3), "erwartet: HALLO WELT")
print(entschluessle_text("YWJKKJS ZR FHMY", 5), "erwartet: TREFFEN UM ACHT")
```

3 Wie sicher ist die Caesar-Chiffre?

🚀 Erweiterung 1 – Brute-Force-Entschlüsselung (mit Gerüst)

Eine Funktion `brute_force(geheimtext)` probiert automatisch alle 26 Schlüssel durch und gibt alle Ergebnisse aus. So sucht man den richtigen Klartext von Hand heraus. Testet mit `KDOOR ZHOW`.

```
def brute_force(geheimtext):
    # für jeden schluessel von 0 bis 25:
    #     schluessel und entschluesselten Text ausgeben
    pass

brute_force("KDOOR ZHOW")
```

🚀 Erweiterung 2 – Menü (selbstständig)

Erweitert euer Programm um ein Text-Menü: 1 = verschlüsseln, 2 = entschlüsseln, 3 = beenden. Das Menü erscheint so lange, bis „beenden“ gewählt wurde.

> 💡 **Tipp:** Vorher steht nicht fest, wie oft das Menü erscheint. Welche Schleife passt also?

⚖️ Beurteilung: Wie sicher ist die Caesar-Chiffre?

1. Wie viele verschiedene Schlüssel gibt es, die den Text tatsächlich verändern?
2. Wie lange braucht ein Programm, das **alle Schlüssel nacheinander ausprobiert** (**Brute-Force**, „rohe Gewalt“), um einen Geheimtext zu knacken?
3. Was müsste ein sicheres Verfahren anders machen?

Unsere Beurteilung:

…

4 Das Projekt abschließen

Zum Abschluss gehört mehr als ein laufendes Programm. Schreibt an den Anfang der Datei einen **Kommentarkopf**, zum Beispiel:

```
# Caesar-Chiffre: Texte ver- und entschlüsseln
# Team: Alex, Sam, Lea
# Funktionen: verschluessle_buchstabe, verschluessle_text,
#             entschluessle_text
# Bekannte Grenzen: Umlaute und ß werden nicht verschlüsselt.
```

📝 Projektabschluss

1. Ergänzt euer Programm um einen Kommentarkopf.
2. Führt mit dem fertigen Programm noch einmal **alle** Tests aus den Teilaufgaben A bis D durch.
3. Bereitet eine Vorstellung von etwa einer Minute vor: Was kann euer Programm? Aus welchen Funktionen besteht es? Welches Problem war am schwierigsten, und wie habt ihr es gelöst? Was würdet ihr noch verbessern?
4. Schaut im Team zurück:

Rückblick im Team	Stichpunkte
Das lief bei der Zusammenarbeit gut:	
Das würden wir beim nächsten Projekt anders machen:	
Das habe ich in diesem Projekt gelernt:	

---

5 🧭 Berufliche Orientierung: Berufe rund um die Softwareentwicklung

Euer Programm ist klein. Nach demselben Prinzip (planen, zerlegen, programmieren, testen, zusammenführen) entsteht auch große Software: in Autos und Smartphones, in Krankenhäusern, Banken und Behörden, in Spielen und Lern-Apps. Daran arbeiten Menschen mit ganz unterschiedlichen Ausbildungswegen.

Recherchiert in Partner- oder Kleingruppenarbeit zu **einem** Beruf bzw. Berufsfeld, zum Beispiel:

- Fachinformatikerin oder Fachinformatiker für Anwendungsentwicklung
- Softwareentwicklerin oder Softwareentwickler
- IT-Sicherheitsexpertin oder IT-Sicherheitsexperte (hier spielt Verschlüsselung eine große Rolle)
- Data Scientist oder Data Engineer
- UX/UI-Designerin oder UX/UI-Designer (Gestaltung von Bedienoberflächen)

📌 **Definition:** Eine **Berufsbiografie** beschreibt den beruflichen Werdegang einer realen Person: Schulabschluss, Ausbildung oder Studium, Berufseinstieg und die weiteren Stationen bis zur heutigen Tätigkeit.

Seriöse Quellen finden

- **BERUFENET** der Bundesagentur für Arbeit (`berufenet.arbeitsagentur.de`): Steckbriefe mit Aufgaben, Zugangswegen und Ausbildung.
- **planet-beruf.de** und **abi.de**: Berufs- und Studienwahl für Schülerinnen und Schüler, auch mit Erfahrungsberichten.
- **Hochschulkompass** (`hochschulkompass.de`): Studiengänge an deutschen Hochschulen.
- Die **Berufsberatung** der Agentur für Arbeit und das **Berufsinformationszentrum (BiZ)**.
- Für Berufsbiografien: Interviews und Porträts, z. B. auf Karriereseiten von Unternehmen. Bedenke: Unternehmen werben dort auch für sich.

💡 **Tipp:** Prüfe jede Information mit drei Fragen: **Wer** hat sie veröffentlicht? **Wann**? Wird etwas **verkauft oder beworben**? Bestätige wichtige Angaben mit einer zweiten Quelle. Vorsicht bei KI-Chatbots: Sie können überzeugend klingende, aber falsche Angaben erfinden. Schreibe nur auf, was du in einer Quelle gefunden hast, und notiere sie.

Leitfragen

1. Welche Aufgaben und Tätigkeiten gehören zu diesem Berufsbild?
2. Welche Ausbildung bzw. welches Studium wird benötigt?
3. Welche Programmiersprachen oder Werkzeuge werden häufig eingesetzt?
4. Wie hat sich das Berufsbild in den letzten Jahren verändert, z. B. durch KI-Werkzeuge?
5. Was interessiert euch persönlich an diesem Berufsfeld, oder was eher nicht?
6. Welche Person aus diesem Berufsfeld könnt ihr finden? Recherchiert ihren Werdegang.

📝 Berufsfeld-Steckbrief

(Doppelklick zum Bearbeiten dieser Zelle)

	Stichpunkte	Quelle
Beruf/Berufsfeld		
Aufgaben und Tätigkeiten		
Ausbildung oder Studium		
Werkzeuge und Programmiersprachen		
Veränderung des Berufsbilds		
Meine Einschätzung		–

📝 Berufsbiografie

Sucht den Werdegang einer **realen** Person, z. B. in einem Interview oder Porträt. Erfindet nichts: Was ihr nicht herausfinden könnt, kennzeichnet ihr mit „unbekannt“.

Station	Zeitraum	Was genau?
Schulabschluss		
Ausbildung oder Studium		
Berufseinstieg		
heutige Tätigkeit		
Quelle (mit Datum)		

🎤 Kurzbeitrag (eine Minute)

Was macht man in diesem Beruf? Wie kommt man dorthin? Was hat euch bei der Recherche überrascht?

Stichpunkte:

…

📌 Das Wichtigste in Kürze

- Entschlüsseln heißt: jeden Buchstaben um den Schlüssel **zurück** verschieben. Das entspricht dem Verschlüsseln mit `-s` oder `26 - s`.
- `% 26` liefert in Python auch bei negativen Zahlen ein Ergebnis von 0 bis 25.
- Mit **Brute-Force** probiert man alle Schlüssel aus. Weil es nur 25 sinnvolle Schlüssel gibt, ist die Caesar-Chiffre nicht sicher.
- Zum Projektabschluss gehören: alle Tests wiederholen, das Programm dokumentieren (Kommentarkopf), es vorstellen und im Team zurückblicken.
- Berufe und Berufsbiografien recherchiert man mit seriösen Quellen wie BERUFENET, gibt jede Angabe mit Quelle an und erfindet nichts dazu.

Zum Weiterüben: `DS14_Weitere_Uebungen.ipynb`.

DS 14 – Weitere Übungen

Projekt Verschlüsselung (Teil 2) und Berufe in der Softwareentwicklung

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Die Aufgaben stehen auch im Skript im Abschnitt „Weitere Übungen“; die Lösungen im Lösungsheft.

Aufgabe 1: Entschlüsseln und Gegenschlüssel ●○○

Der Geheimtext `WGLYPI` wurde mit der Caesar-Chiffre und dem Schlüssel 4 verschlüsselt.

1. Entschlüssele ihn von Hand. Benutze die Positionstabelle aus dem Auftrag *Caesar mit Schlüssel 5* im ersten Projektteil.
2. Gib den Gegenschlüssel an: Mit welchem Schlüssel muss man **verschlüsseln**, um dasselbe Ergebnis zu bekommen wie beim Entschlüsseln mit dem Schlüssel 4?

Deine Antwort:

…

Aufgabe 2: Wer macht was? ●○○

In eurem Projekt habt ihr geplant, Schnittstellen festgelegt, programmiert, getestet, dokumentiert und die Sicherheit beurteilt. In einer Softwarefirma übernehmen oft verschiedene Personen diese Aufgaben.

Ordne drei dieser Projektschritte jeweils einem Beruf oder Berufsfeld aus dem Abschnitt *Berufliche Orientierung* zu. Begründe jede Zuordnung in einem Satz.

Deine Antwort:

…

Aufgabe 3: Der häufigste Buchstabe verrät den Schlüssel ●●○

Du hast den Geheimtext `NOB OVOPKXD QORD JEW COO` abgefangen. Den Schlüssel kennst du nicht. In deutschen Texten ist **E** meistens der häufigste Buchstabe.

1. Bestimme den häufigsten Buchstaben im Geheimtext. Du kannst von Hand zählen oder die Funktion `zaehle(text, zeichen)` aus dem Auftrag *Buchstaben zählen* im Abschnitt über Zeichenketten benutzen.
2. Nimm an, dass dieser Buchstabe für E steht. Berechne daraus den Schlüssel.
3. Entschlüssele den Text mit `entschluessle_text` und prüfe, ob ein sinnvoller Satz herauskommt.

Deine Antwort:

…

Aufgabe 4: Knacken mit Suchwort ●●●

Die Funktion `brute_force` aus der Erweiterung gibt alle 26 Entschlüsselungen aus. Bei langen Texten ist es mühsam, darin den Klartext zu suchen.

Implementiere eine Funktion `knacke(geheimtext, wort)`, die alle Schlüssel von 0 bis 25 ausprobiert, aber nur die Ergebnisse ausgibt, in denen das Wort `wort` vorkommt (zusammen mit dem Schlüssel). Teste mit dem Geheimtext `KDOOR ZHOW` und dem Suchwort `"WELT"` sowie mit dem Geheimtext aus dem vorigen Auftrag und dem Suchwort `"DER"`.

Tipp: Mit `wort in text` prüfst du, ob ein Textstück in einer Zeichenkette vorkommt (Abschnitt über Zeichenketten). Diese Bedingung setzt du in die Schleife von `brute_force` vor das `print`.

DS 15 – Grundgatter AND, OR, XOR, NOT und Belegungstabellen

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich kann die logischen Grundgatter benennen und skizzieren und die zugehörigen Belegungstabellen angeben.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Vermutungen sammeln, wie ein Computer mit den Zuständen 0 und 1 Entscheidungen realisiert.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	AND, OR, XOR und NOT aus Alltagssituationen und Schüleraussagen entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Belegungstabellen gemeinsam mit Handzeichen oder Signalkarten aufbauen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Grundgatter in CircuitVerse bauen und systematisch testen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	IEC- und ANSI-Symbole zuordnen und Tabellen mit Fachsprache erläutern.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Majority-Schaltung oder Python-Kontrolle als Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

⚠️ Wichtiger Hinweis zur Gatter-Symbolik in CircuitVerse

In deutschen Lehrbüchern werden Logikgatter meist nach **IEC 60617** dargestellt: rechteckige Symbole mit Kennbuchstaben, z. B. `&` für AND, `≥1` für OR, `=1` für XOR und ein kleiner Kreis am Ausgang für die Negation.

CircuitVerse verwendet stattdessen die in den USA gebräuchliche Symbolik (ANSI/IEEE Std 91) – also die klassischen, unterschiedlich geformten Gattersymbole (AND = "D-Form", OR = spitz zulaufende Kurvenform, XOR = OR-Form mit zusätzlicher Linie, NOT = Dreieck mit kleinem Kreis am Ausgang).

Funktional sind beide Notationen identisch – nur die Zeichnung unterscheidet sich. Achtet trotzdem darauf, in Klassenarbeiten/Heftaufschrieben weiterhin die IEC-Symbole zu verwenden, wie sie im Schulbuch stehen. In CircuitVerse geht es uns nur um das **Verhalten** der Schaltungen, nicht um die Prüfungsschreibweise.

📌 Merksätze

Gatter	IEC-Symbol	Verhalten
AND (Konjunktion)	`&`	Ausgang ist `1`, wenn **alle** Eingänge `1` sind
OR (Disjunktion)	`≥1`	Ausgang ist `1`, wenn **mindestens ein** Eingang `1` ist
XOR (Exklusiv-Oder)	`=1`	Ausgang ist `1`, wenn die Eingänge **unterschiedlich** sind
NOT (Negation)	kleiner Kreis	Ausgang ist das **Gegenteil** des Eingangs

- Eine **Belegungstabelle** (Wahrheitstabelle) listet für jede mögliche Kombination von Eingangswerten den zugehörigen Ausgangswert auf.
- Bei n Eingängen gibt es 2^n mögliche Belegungen.

✏️ Zusatzübung – Grundgatter zusätzlich von Hand skizzieren

*Hinweis: „Skizzieren" bedeutet laut Lehrplan-Operatorendefinition, einen Sachverhalt aufs Wesentliche reduziert **grafisch oder sprachlich** darzustellen – das Bauen der Gatter in CircuitVerse gleich im Praxisteil unten erfüllt diese Kompetenz bereits. Die folgende Papierübung ist daher freiwillig, aber eine gute zusätzliche Übung, z. B. als Vorbereitung auf eine Klassenarbeit ohne Rechner.*

Zeichnet **freiwillig** (auf Papier oder in dieser Zelle als Beschreibung) die vier Grundgatter AND, OR, XOR und NOT **mit ihren IEC-Symbolen** (siehe Tabelle oben) und beschriftet jeweils Ein- und Ausgänge mit `A`, `B` bzw. `Q`.

🖥️ Praxisteil: CircuitVerse

1. Öffnet circuitverse.org/simulator (Login optional, aber empfohlen zum Speichern).
2. Im linken Bereich unter **Input** findet ihr **Input**-Bausteine (Schalter) – zieht zwei davon ins Arbeitsfeld.
3. Unter **Output** findet ihr die **Output**-LED – zieht eine davon ins Arbeitsfeld.
4. Unter **Gates** findet ihr AND, OR, XOR, NOT – zieht ein Gatter eurer Wahl dazwischen.
5. Verbindet die Bausteine per Drag&Drop von Anschluss zu Anschluss mit Leitungen.
6. Klickt auf die Input-Schalter, um zwischen `0` und `1` umzuschalten, und beobachtet die Output-LED (leuchtet = `1`).

📝 Übung 1 – Alle vier Grundgatter testen

Baut in CircuitVerse **nacheinander** je eine Schaltung mit AND, OR, XOR und NOT (bei NOT reicht ein Eingang). Testet **alle** Eingangskombinationen und übertragt eure Beobachtungen in die Tabellen unten (Doppelklick zum Bearbeiten dieser Zelle).

AND

A	B	Ausgang
0	0	
0	1	
1	0	
1	1	

OR

A	B	Ausgang
0	0	
0	1	
1	0	
1	1	

XOR

A	B	Ausgang
0	0	
0	1	
1	0	
1	1	

🐍 Kontrolle mit Python

Nutzt die Zelle unten, um eure handschriftlich ermittelten Belegungstabellen zu überprüfen. Python bringt die logischen Operatoren `and`, `or` bereits mit; XOR entspricht `!=` (ungleich) bei Wahrheitswerten.

```python
from itertools import product

def zeige_belegungstabelle(gattername, funktion, anzahl_eingaenge=2):
    print(f"--- {gattername} ---")
    for eingaben in product([0, 1], repeat=anzahl_eingaenge):
        ausgang = funktion(*eingaben)
        print(eingaben, "->", int(ausgang))

zeige_belegungstabelle("AND", lambda a, b: a and b)
zeige_belegungstabelle("OR", lambda a, b: a or b)
zeige_belegungstabelle("XOR", lambda a, b: a != b)
zeige_belegungstabelle("NOT", lambda a: not a, anzahl_eingaenge=1)
```

```
--- AND ---
(0, 0) -> 0
(0, 1) -> 0
(1, 0) -> 0
(1, 1) -> 1
--- OR ---
(0, 0) -> 0
(0, 1) -> 1
(1, 0) -> 1
(1, 1) -> 1
--- XOR ---
(0, 0) -> 0
(0, 1) -> 1
(1, 0) -> 1
(1, 1) -> 0
--- NOT ---
(0,) -> 1
(1,) -> 0
```

📝 Übung 2 – Kombinierte Schaltung

Baut in CircuitVerse eine Schaltung mit **drei Eingängen** A, B, C, die genau dann `1` ausgibt, wenn **mindestens zwei** der drei Eingänge `1` sind (Mehrheitsentscheidung/"Majority-Gatter"). Nutzt dafür mehrere AND- und OR-Gatter.

Tipp: Die Schaltung lässt sich als `(A AND B) OR (B AND C) OR (A AND C)` beschreiben.

✅ Kontrolle Übung 2 mit Python

```python
def majority(a, b, c):
    return (a and b) or (b and c) or (a and c)

zeige_belegungstabelle("Majority", majority, anzahl_eingaenge=3)
```

```
--- Majority ---
(0, 0, 0) -> 0
(0, 0, 1) -> 0
(0, 1, 0) -> 0
(0, 1, 1) -> 1
(1, 0, 0) -> 0
(1, 0, 1) -> 1
(1, 1, 0) -> 1
(1, 1, 1) -> 1
```

DS 16 – Entwurf logischer Schaltungen

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich entwerfe logische Schaltungen in gegebenen Sachzusammenhängen, auch unter Verwendung geeigneter Software (CircuitVerse).

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Eine einfache technische Anforderung in Wenn-dann-Sätzen formulieren.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Vorgehen vom Sachtext über Variablen und Belegungstabelle zur Schaltung entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Die Alarmanlage gemeinsam modellieren und in CircuitVerse prüfen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Die Fahrstuhltür in Tandems bearbeiten.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Lösungswege vergleichen und jede Verbindung aus der Anforderung begründen lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Abstimmung mit Vetorecht als Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

📌 Merksätze

- Beim Schaltungsentwurf übersetzt man eine **Anforderung in Textform** zunächst in eine **Belegungstabelle**, danach in eine passende **Gatterkombination**.
- Vorgehen:
    1. Eingangs- und Ausgangsgrößen benennen
    2. Belegungstabelle aufstellen (alle Fälle durchgehen, in denen der Ausgang `1` sein soll)
    3. Passende Gatter auswählen und verschalten
    4. Schaltung testen (alle Belegungen durchprobieren)

💡 Beispiel: Alarmanlage

Eine Alarmanlage soll auslösen (Ausgang = 1), wenn die **Tür geöffnet** ist **und** die Anlage **scharfgeschaltet** ist, **oder** wenn der **Rauchmelder** anschlägt (unabhängig vom Scharfschaltungsstatus).

Eingänge: `Tuer`, `Scharf`, `Rauch` → Ausgang: `Alarm = (Tuer AND Scharf) OR Rauch`

Baut diese Schaltung zur Kontrolle selbst in CircuitVerse nach, bevor ihr mit den Übungen startet.

```python
from itertools import product

def alarm(tuer, scharf, rauch):
    return (tuer and scharf) or rauch

for t, s, r in product([0, 1], repeat=3):
    print(f"Tuer={t}, Scharf={s}, Rauch={r} -> Alarm={int(alarm(t, s, r))}")
```

```
Tuer=0, Scharf=0, Rauch=0 -> Alarm=0
Tuer=0, Scharf=0, Rauch=1 -> Alarm=1
Tuer=0, Scharf=1, Rauch=0 -> Alarm=0
Tuer=0, Scharf=1, Rauch=1 -> Alarm=1
Tuer=1, Scharf=0, Rauch=0 -> Alarm=0
Tuer=1, Scharf=0, Rauch=1 -> Alarm=1
Tuer=1, Scharf=1, Rauch=0 -> Alarm=1
Tuer=1, Scharf=1, Rauch=1 -> Alarm=1
```

📝 Übung 1 – Fahrstuhltür

Eine Fahrstuhltür darf sich nur schließen (Ausgang `Schliessen = 1`), wenn **beide** Lichtschranken (`LS1`, `LS2`) frei sind (`0` = blockiert, `1` = frei) **und** der Zeitgeber (`Zeit`) abgelaufen ist.

1. Stellt die Belegungstabelle auf.
2. Baut die Schaltung in CircuitVerse.
3. Kontrolliert euer Ergebnis mit Python.

```python
# TODO: Übung 1 - Funktion schliessen(ls1, ls2, zeit) definieren
def schliessen(ls1, ls2, zeit):
    pass

for ls1, ls2, zeit in product([0, 1], repeat=3):
    print(f"LS1={ls1}, LS2={ls2}, Zeit={zeit} -> Schliessen={schliessen(ls1, ls2, zeit)}")
```

```
LS1=0, LS2=0, Zeit=0 -> Schliessen=None
LS1=0, LS2=0, Zeit=1 -> Schliessen=None
LS1=0, LS2=1, Zeit=0 -> Schliessen=None
LS1=0, LS2=1, Zeit=1 -> Schliessen=None
LS1=1, LS2=0, Zeit=0 -> Schliessen=None
LS1=1, LS2=0, Zeit=1 -> Schliessen=None
LS1=1, LS2=1, Zeit=0 -> Schliessen=None
LS1=1, LS2=1, Zeit=1 -> Schliessen=None
```

📝 Übung 2 – Abstimmungsschaltung mit Vetorecht

Drei Personen A, B, C stimmen ab (0 = dagegen, 1 = dafür). Ein Antrag ist angenommen, wenn die **Mehrheit** dafür ist – **außer** Person A stimmt dagegen, dann ist der Antrag **in jedem Fall** abgelehnt (Vetorecht).

1. Überlegt euch zunächst in Worten, wann der Ausgang `1` sein muss.
2. Baut die Schaltung in CircuitVerse.
3. Vergleicht mit der Python-Kontrolle.

```python
# TODO: Übung 2 - Funktion antrag_angenommen(a, b, c) definieren
def antrag_angenommen(a, b, c):
    pass

for a, b, c in product([0, 1], repeat=3):
    print(f"A={a}, B={b}, C={c} -> Angenommen={antrag_angenommen(a, b, c)}")
```

```
A=0, B=0, C=0 -> Angenommen=None
A=0, B=0, C=1 -> Angenommen=None
A=0, B=1, C=0 -> Angenommen=None
A=0, B=1, C=1 -> Angenommen=None
A=1, B=0, C=0 -> Angenommen=None
A=1, B=0, C=1 -> Angenommen=None
A=1, B=1, C=0 -> Angenommen=None
A=1, B=1, C=1 -> Angenommen=None
```

DS 17 – Addition im Binärsystem und Halbaddierer

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich kann die Addition zweier natürlicher Zahlen im Binärsystem durchführen und den Aufbau und die Funktionsweise eines Halbaddierers erläutern.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Eine bekannte Dezimaladdition mit Übertrag auf Binärzahlen übertragen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Die vier Fälle der Addition zweier Bits gemeinsam entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Aus Summenbit und Übertrag die Belegungstabelle und den Halbaddierer herleiten.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Halbaddierer in CircuitVerse bauen und alle vier Belegungen testen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Funktion von XOR und AND im Halbaddierer erläutern lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Weitere Handrechnungen oder optionale Python-Kontrolle.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

📌 Merksätze

- Die **binäre Addition** funktioniert wie die schriftliche Addition im Dezimalsystem, nur mit den Ziffern `0` und `1`: `0+0=0`, `0+1=1`, `1+0=1`, `1+1=10` (also `0` schreiben, `1` **Übertrag**)
- Ein **Halbaddierer** addiert **zwei einzelne Bits** (`A`, `B`) und liefert zwei Ausgänge:
  - **Summe (S):** `A XOR B`
  - **Übertrag (C, Carry):** `A AND B`
- "Halb"-Addierer deshalb, weil er **keinen eingehenden Übertrag** aus einer vorherigen Stelle berücksichtigen kann – das kann erst der Volladdierer (DS 18).

💡 Beispiel: Binäre Addition von Hand

```
  0101   (= 5)
+ 0011   (= 3)
-------
  1000   (= 8)
```

Rechts beginnend: 1+1=10 (0, Übertrag 1) → 0+1+1(Übertrag)=10 (0, Übertrag 1) → 1+0+1(Übertrag)=10 (0, Übertrag 1) → 0+0+1(Übertrag)=1

```python
# Kontrolle mit Python (nur zur Überprüfung, nicht als Ersatz fürs Kopfrechnen!)
a = int("0101", 2)
b = int("0011", 2)
summe = a + b
print(bin(summe))
```

```
0b1000
```

📝 Übung 1 – Von Hand rechnen

Berechnet **von Hand** (auf Papier, Schritt für Schritt mit Übertrag) folgende binäre Additionen und kontrolliert anschließend mit Python:

1. `0110 + 0101`
2. `1011 + 0111`
3. `1111 + 0001`

```python
# TODO: Übung 1 - Kontrolle eurer Handrechnungen
aufgaben = [("0110", "0101"), ("1011", "0111"), ("1111", "0001")]
for a_str, b_str in aufgaben:
    a = int(a_str, 2)
    b = int(b_str, 2)
    print(a_str, "+", b_str, "=", bin(a + b))
```

```
0110 + 0101 = 0b1011
1011 + 0111 = 0b10010
1111 + 0001 = 0b10000
```

🖥️ Praxisteil: Halbaddierer in CircuitVerse bauen

1. Legt zwei Input-Schalter (`A`, `B`) an.
2. Fügt ein **XOR-Gatter** hinzu (Eingänge: `A`, `B`) → Ausgang ist die **Summe S**.
3. Fügt ein **AND-Gatter** hinzu (Eingänge: `A`, `B`) → Ausgang ist der **Übertrag C**.
4. Verbindet beide Gatterausgänge mit je einer Output-LED.
5. Testet alle vier Belegungen und vergleicht mit der Merksatz-Formel oben.

📝 Übung 2 – Halbaddierer in Python nachbilden

Schreibt eine Funktion `halbaddierer(a, b)`, die ein Tupel `(summe, uebertrag)` zurückgibt, und testet sie für alle vier Belegungen.

DS 18 – Volladdierer und Carry-Ripple-Addierer

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich erläutere den Aufbau und die Funktionsweise eines Volladdierers und skizziere und erläutere einen Carry-Ripple-Addierer.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Das Problem eines eingehenden Übertrags bei mehrstelliger Addition untersuchen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Ein- und Ausgänge des Volladdierers aus konkreten Additionsfällen entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Den Volladdierer aus zwei Halbaddierern und einem OR-Gatter gemeinsam zusammensetzen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Ausgewählte Belegungen testen und einen Carry-Ripple-Addierer skizzieren.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Weitergabe des Übertrags schrittweise erläutern lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	CircuitVerse-Ausbau oder optionale Python-Kontrolle.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

📌 Merksätze

- Ein **Volladdierer** addiert **drei Bits**: die beiden zu addierenden Bits `A` und `B` sowie einen **eingehenden Übertrag** `Cin` aus der vorherigen Stelle. Er liefert die **Summe S** und einen **ausgehenden Übertrag** `Cout`.
- Ein Volladdierer lässt sich aus **zwei Halbaddierern und einem OR-Gatter** aufbauen:
  1. Halbaddierer 1: `A` und `B` → Zwischensumme `S1` und Übertrag `C1`
  2. Halbaddierer 2: `S1` und `Cin` → Summe `S` und Übertrag `C2`
  3. `Cout = C1 OR C2`
- Ein **Carry-Ripple-Addierer** verkettet mehrere Volladdierer: Der `Cout` einer Stelle wird zum `Cin` der nächsthöheren Stelle. So werden mehrstellige Binärzahlen addiert – der Übertrag "rieselt" (englisch: *ripple*) von Stelle zu Stelle durch.

🖥️ Praxisteil: Volladdierer in CircuitVerse bauen

1. Baut **zwei Halbaddierer** wie in DS 17 (je ein XOR- und ein AND-Gatter).
2. Verbindet den Summenausgang des ersten Halbaddierers mit einem Eingang des zweiten Halbaddierers, der zweite Eingang ist `Cin`.
3. Verknüpft die beiden Übertrags-Ausgänge (`C1`, `C2`) mit einem **OR-Gatter** → das ist `Cout`.
4. Testet alle **8** möglichen Belegungen von `A`, `B`, `Cin` und vergleicht mit der Tabelle unten.

Tipp: In CircuitVerse gibt es unter *Decoders/Adders* auch fertige "Full Adder"-Bausteine – baut den Volladdierer aber **zuerst selbst aus Gattern**, um die Funktionsweise zu verstehen, bevor ihr den fertigen Baustein zum Testen nutzt.

🐍 Kontrolle mit Python

```python
from itertools import product

def volladdierer(a, b, cin):
    s1 = a != b
    c1 = a and b
    s = s1 != cin
    c2 = s1 and cin
    cout = c1 or c2
    return int(s), int(cout)

for a, b, cin in product([0, 1], repeat=3):
    s, cout = volladdierer(a, b, cin)
    print(f"A={a} B={b} Cin={cin} -> S={s} Cout={cout}")
```

```
A=0 B=0 Cin=0 -> S=0 Cout=0
A=0 B=0 Cin=1 -> S=1 Cout=0
A=0 B=1 Cin=0 -> S=1 Cout=0
A=0 B=1 Cin=1 -> S=0 Cout=1
A=1 B=0 Cin=0 -> S=1 Cout=0
A=1 B=0 Cin=1 -> S=0 Cout=1
A=1 B=1 Cin=0 -> S=0 Cout=1
A=1 B=1 Cin=1 -> S=1 Cout=1
```

📝 Übung 1 – 4-Bit Carry-Ripple-Addierer

Schreibt eine Funktion `carry_ripple_addierer(a_bits, b_bits)`, die zwei 4-Bit-Zahlen (jeweils als Liste von vier Bits, z. B. `[0,1,0,1]` für `0101`) mithilfe eures `volladdierer` **stellenweise von rechts nach links** addiert und das Ergebnis als 5-Bit-Liste (inkl. abschließendem Übertrag) zurückgibt.

Tipp: Der `Cin` der niedrigsten Stelle ist `0`. Verarbeitet die Listen von hinten (rechts, niedrigstwertiges Bit) nach vorne.

```python
# TODO: Übung 1 - carry_ripple_addierer(a_bits, b_bits)

# Test:
# print(carry_ripple_addierer([0,1,0,1], [0,0,1,1]))  # 5 + 3 = 8 -> [0,1,0,0,0]
```

✏️ Zusatzübung – Carry-Ripple-Addierer zusätzlich skizzieren

Hinweis: Der CircuitVerse-Aufbau und eure Python-Implementierung oben erfüllen die Kompetenz „skizzieren und erläutern" bereits (grafische bzw. sprachliche, aufs Wesentliche reduzierte Darstellung). Diese Papier-Übung ist freiwillig und dient der zusätzlichen Festigung, z. B. für eine Klassenarbeit ohne Rechner.

Skizziert **freiwillig** (auf Papier oder als Beschreibung in dieser Zelle) das **Blockschaltbild** eines 4-Bit-Carry-Ripple-Addierers: vier Volladdierer-Blöcke nebeneinander, mit den Eingängen `A0..A3`, `B0..B3`, den Summenausgängen

`S0..S3` sowie den Pfeilen, die zeigen, wie der `Cout` jeder Stelle zum `Cin` der nächsten Stelle wird. Erläutert in 1–2 Sätzen, warum die Bezeichnung "Ripple" (rieseln) passend ist.

🚀 Zusatzaufgabe

Baut in CircuitVerse einen **2-Bit Carry-Ripple-Addierer** aus zwei Volladdierern (bzw. einem Halb- und einem Volladdierer für die niedrigste Stelle) und testet ihn mit mehreren Zahlenpaaren.

DS 19 – Rückkopplung als Speichertechnik: Das Flipflop

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich skizziere und erläutere eine logische Schaltung, die das Prinzip der Rückkopplung als Speichertechnik verwendet.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Frage entwickeln, wie eine Schaltung einen früheren Zustand behalten kann.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Rückkopplung und Speicherzustand anhand zweier gekoppelter Gatter erschließen.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	SR-Flipflop gemeinsam skizzieren und Zustandsänderungen vor dem Test vorhersagen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Schaltung in CircuitVerse bauen und die Testfolge protokollieren.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Speicherwirkung mit den Begriffen Rückkopplung, Setzen und Rücksetzen erklären lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Weitere Flipflops oder optionale Simulation nur als Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

📌 Merksätze

- Bisherige Schaltungen (Addierer etc.) sind **kombinatorisch**: Der Ausgang hängt nur vom **aktuellen** Eingang ab.
- Ein **Flipflop** kann sich hingegen ein Bit **merken** – der Ausgang hängt auch von der **Vorgeschichte** ab. Flipflops sind die Grundlage für schnelle Speicher im Computer, z. B. Register (der Arbeitsspeicher eines PCs speichert Bits dagegen meist als elektrische Ladung).
- Möglich wird das durch **Rückkopplung**: Der Ausgang einer Schaltung wird auf einen ihrer eigenen Eingänge zurückgeführt.
- Das einfachste Beispiel ist das **SR-Flipflop** (Set-Reset), gebaut aus zwei über Kreuz rückgekoppelten **NOR-Gattern**:
  - `S = 1` (Set) setzt den gespeicherten Wert auf `1`
  - `R = 1` (Reset) setzt den gespeicherten Wert auf `0`
  - `S = 0` und `R = 0`: Der zuletzt gespeicherte Wert **bleibt erhalten** – das ist der "Speicher-Effekt"!

- `S = 1` und `R = 1` gleichzeitig ist ein **verbotener Zustand** und sollte vermieden werden.

✏️ Zusatzübung – SR-Flipflop zusätzlich von Hand skizzieren

Hinweis: Der CircuitVerse-Aufbau im nächsten Schritt zusammen mit eurer schriftlichen Erklärung in Übung 2 erfüllt die Kompetenz „skizzieren und erläutern" bereits (grafische bzw. sprachliche, aufs Wesentliche reduzierte Darstellung). Diese Papier-Skizze vorab ist freiwillig, kann euch aber als Bauanleitung für den nächsten Schritt dienen.

Skizziert **freiwillig**, bevor ihr baut (auf Papier oder als Beschreibung in dieser Zelle): zwei NOR-Gatter, deren Ausgang jeweils auf einen Eingang des **anderen** Gatters zurückgeführt wird ("über Kreuz"). Beschriftet die freien Eingänge mit `S` und `R` sowie die beiden Ausgänge mit `Q` und `Q'`.

🖥️ Praxisteil: SR-Flipflop in CircuitVerse bauen

1. Legt zwei Input-Schalter `S` und `R` sowie zwei Output-LEDs (`Q` und `Q'`, das negierte Q) an.
2. Fügt **zwei NOR-Gatter** hinzu (falls kein NOR verfügbar ist: OR-Gatter + nachgeschaltetem NOT-Gatter).
3. Verschaltet **über Kreuz**:
    - Eingang des ersten NOR-Gatters: `R` und der Ausgang des **zweiten** NOR-Gatters
    - Eingang des zweiten NOR-Gatters: `S` und der Ausgang des **ersten** NOR-Gatters
4. Der Ausgang des ersten Gatters ist `Q`, der des zweiten ist `Q'`.
5. **Testet die Speicherwirkung:** Setzt kurz `S=1, R=0` → `Q` wird `1`. Schaltet danach `S` wieder auf `0` (also `S=0, R=0`) → beobachtet, dass `Q` weiterhin `1` bleibt, obwohl kein Eingang mehr aktiv ist!

📝 Übung 1 – Zustände protokollieren

Führt die Testreihenfolge unten in eurer CircuitVerse-Schaltung durch und tragt jeweils den beobachteten Wert von `Q` ein (Doppelklick zum Bearbeiten):

Schritt	S	R	Q (beobachtet)	Erklärung
1	1	0		Setzen
2	0	0		Speichern (Q bleibt wie zuvor)
3	0	1		Rücksetzen
4	0	0		Speichern (Q bleibt wie zuvor)

📝 Übung 2 – Erklärung in eigenen Worten

Beantwortet in dieser Zelle in 2–3 Sätzen: Warum kann eine **rein kombinatorische** Schaltung (ohne Rückkopplung) kein Bit speichern? Was genau ermöglicht die Rückkopplung, das eine gewöhnliche Gatterkette nicht kann?

(Eure Antwort hier)

🚀 Zusatzaufgabe (Vertiefung)

Recherchiert (z. B. über eine Websuche) den Unterschied zwischen dem hier gebauten **SR-Flipflop** und einem **D-Flipflop** oder **JK-Flipflop**. Was ist deren Vorteil gegenüber dem einfachen SR-Flipflop?

🐍 Optionale Simulation in Python

Rein kombinatorisch (ohne Speicher) lässt sich ein Flipflop nicht direkt als einfache Funktion simulieren – genau das ist ja der Punkt aus Übung 2! Mit einer **Zustandsvariablen**, die zwischen Aufrufen erhalten bleibt, lässt sich der Speicher-Effekt aber nachbilden. Optional zur Vertiefung:

```python
# Optionale Vertiefung: SR-Flipflop-Verhalten mit einer Zustandsvariable simulieren
q = 0  # Anfangszustand (nicht Teil einer "reinen" Schaltung, sondern unser Gedächtnis)

def sr_flipflop(s, r, q):
    if s == 1 and r == 0:
        return 1
    elif s == 0 and r == 1:
        return 0
    elif s == 0 and r == 0:
        return q  # Speichern: alter Zustand bleibt erhalten
    else:
        raise ValueError("S=1 und R=1 gleichzeitig ist der verbotene Zustand!")

# Testreihenfolge wie in Übung 1
for s, r in [(1, 0), (0, 0), (0, 1), (0, 0)]:
    q = sr_flipflop(s, r, q)
    print(f"S={s} R={r} -> Q={q}")
```

```
S=1 R=0 -> Q=1
S=0 R=0 -> Q=1
S=0 R=1 -> Q=0
S=0 R=0 -> Q=0
```

DS 20 – Klassische Aussagenlogik: Syntax und Termbäume

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich beschreibe logische Schaltungen durch aussagenlogische Terme, definiere den Begriff des aussagenlogischen Terms induktiv und stelle Terme als Termbäume dar.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Eine bekannte Schaltung zunächst in Worten beschreiben.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Konjunktion, Disjunktion, Negation und induktiven Termaufbau aus Schaltungsteilen entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Einen Termbaum gemeinsam von innen nach außen konstruieren.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Einen gegebenen Term als Termbaum oder Schaltung darstellen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Zusammenhang zwischen Teilterm, Knoten und Teilschaltung erläutern lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Term für „genau eine von drei“ oder Python-Kontrolle.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

📌 Merksätze

- Die **Aussagenlogik** beschreibt dieselben Zusammenhänge wie logische Schaltungen – nur symbolisch statt schaltungstechnisch.
- **Induktive Definition eines aussagenlogischen Terms:**
  1. Jede **Aussagenvariable** (z. B. `A`, `B`, `C`) ist ein Term.
  2. Ist `t` ein Term, so ist auch `¬t` (**Negation**) ein Term.
  3. Sind `t1` und `t2` Terme, so sind auch `(t1 ∧ t2)` (**Konjunktion**, entspricht AND) und `(t1 ∨ t2)` (**Disjunktion**, entspricht OR) Terme.
  4. Nichts anderes ist ein Term.
- Ein **Termbaum** stellt einen Term als Baumstruktur dar: Die Blätter sind die Aussagenvariablen, die inneren Knoten die Operatoren (`¬`, `∧`, `∨`).

- Entsprechung zu Schaltungen: `∧` ↔ AND-Gatter, `∨` ↔ OR-Gatter, `¬` ↔ NOT-Gatter.

💡 Beispiel: Term zu Termbaum

Term: `(A ∧ B) ∨ ¬C`

Termbaum:



Dieser Term entspricht der Schaltung: `A` und `B` gehen in ein AND-Gatter, `C` geht in ein NOT-Gatter, deren Ausgänge gehen in ein OR-Gatter.

📝 Übung 1 – Term zu Termbaum

Zeichnet (auf Papier oder als ASCII-Skizze in dieser Zelle) den Termbaum zu folgendem Term:

`¬(A ∨ B) ∧ C`

(Euer Termbaum hier)

📝 Übung 2 – Schaltung zu Term

Ihr habt in DS 15/16 folgende Schaltung gebaut: `Alarm = (Tuer ∧ Scharf) ∨ Rauch`. Zeichnet dazu den Termbaum.

(Euer Termbaum hier)

🐍 Kontrolle: Terme als Python-Funktionen

Terme lassen sich direkt als Python-Ausdrücke mit `and`, `or`, `not` schreiben. Nutzt das, um eure Termbäume auf Plausibilität zu prüfen.

```python
from itertools import product

def term1(a, b, c):
    # ¬(A ∨ B) ∧ C
    return (not (a or b)) and c

def term2(tuer, scharf, rauch):
    # (Tuer ∧ Scharf) ∨ Rauch
    return (tuer and scharf) or rauch

print("--- Term 1: ¬(A ∨ B) ∧ C ---")
for a, b, c in product([0, 1], repeat=3):
    print(f"A={a} B={b} C={c} -> {int(term1(a,b,c))}")
```

```
--- Term 1: ¬(A ∨ B) ∧ C ---
A=0 B=0 C=0 -> 0
A=0 B=0 C=1 -> 1
A=0 B=1 C=0 -> 0
A=0 B=1 C=1 -> 0
A=1 B=0 C=0 -> 0
A=1 B=0 C=1 -> 0
A=1 B=1 C=0 -> 0
A=1 B=1 C=1 -> 0
```

📝 Übung 3 – Vom Term zur Schaltung

Baut nun die **umgekehrte Richtung**: Nehmt den Term `¬(A ∨ B) ∧ C` aus Übung 1 (bzw. `term1` aus der Python-Kontrolle oben) und **baut die zugehörige Schaltung in CircuitVerse** (ein OR-Gatter für `A ∨ B`, ein NOT-Gatter dahinter, ein weiteres AND-Gatter mit `C`). Testet alle 8 Belegungen und vergleicht mit eurem Termbaum und der Python-Ausgabe von oben.

📝 Übung 4 – Eigenen Term formulieren

Formuliert einen aussagenlogischen Term für folgende Situation: „Der Ausgang ist `1`, wenn **genau eine** der drei Variablen `A`, `B`, `C` den Wert `1` hat." Notiert den Term symbolisch **und** als Python-Funktion.

```
# TODO: Übung 4 - Term und Python-Funktion für "genau eine von drei"
def term3(a, b, c):
    pass
```

DS 21 – Belegungstabellen, Äquivalenz und Disjunktive Normalform (DNF)

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich erstelle Belegungstabellen zu gegebenen aussagenlogischen Termen, erläutere den Begriff der Äquivalenz und erstelle die DNF zu gegebenen Termen bzw. Belegungstabellen.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Die Ausgabe eines Terms für ausgewählte Belegungen vorhersagen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Aus den wahren Tabellenzeilen gemeinsam die Grundidee der DNF entdecken.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Eine vollständige DNF aufbauen und Äquivalenz über gleiche Belegungstabellen sichern.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Zu einer kleinen Tabelle selbst eine DNF formulieren.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Jeden DNF-Baustein einer Tabellenzeile zuordnen und Äquivalenz erläutern lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	De-Morgan-Vergleich oder Python-Kontrolle als Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

📌 Merksätze

- Zwei aussagenlogische Terme heißen **äquivalent**, wenn sie für **jede** mögliche Belegung denselben Wahrheitswert liefern – also **dieselbe Belegungstabelle** besitzen. (Äquivalenz ist ein Begriff **über** Terme, kein eigener Operator!)
- Die **Disjunktive Normalform (DNF)** ist eine standardisierte Darstellung eines Terms:
  - Für **jede** Zeile der Belegungstabelle, in der der Ausgang `1` ist, bildet man ein **Minterm**: die Konjunktion aller Variablen, wobei Variablen mit Wert `0` negiert werden.
  - Alle Minterme werden anschließend **disjunktiv** (mit `∨`) verknüpft.
- Beispiel: Ist der Ausgang bei `A=1,B=0` und bei `A=0,B=1` gleich `1` (sonst `0`), lautet die DNF: `(A ∧ ¬B) ∨ (¬A ∧ B)` — das ist übrigens genau XOR!

💡 Beispiel: DNF aus Belegungstabelle ablesen

A	B	Ausgang
0	0	0
0	1	1
1	0	1
1	1	0

Zeilen mit Ausgang `1`: `(A=0,B=1)` und `(A=1,B=0)` → DNF: `(¬A ∧ B) ∨ (A ∧ ¬B)`

```python
from itertools import product

def dnf_beispiel(a, b):
    return (not a and b) or (a and not b)

for a, b in product([0, 1], repeat=2):
    print(f"A={a} B={b} -> {int(dnf_beispiel(a, b))}")
```

```
A=0 B=0 -> 0
A=0 B=1 -> 1
A=1 B=0 -> 1
A=1 B=1 -> 0
```

📝 Vorübung – Von einem Term zur Belegungstabelle

Bevor ihr DNFs bestimmt: Erstellt **von Hand** die vollständige Belegungstabelle (3 Variablen, 8 Zeilen) zu folgendem Term:

`(A ∨ B) ∧ ¬C`

Überprüft euer Ergebnis anschließend mit der Python-Zelle unten.

```python
from itertools import product

def vorbuebung_term(a, b, c):
    return (a or b) and (not c)

for a, b, c in product([0, 1], repeat=3):
    print(f"A={a} B={b} C={c} -> {int(vorbuebung_term(a,b,c))}")
```

```
A=0 B=0 C=0 -> 0
A=0 B=0 C=1 -> 0
A=0 B=1 C=0 -> 1
A=0 B=1 C=1 -> 0
A=1 B=0 C=0 -> 1
A=1 B=0 C=1 -> 0
A=1 B=1 C=0 -> 1
A=1 B=1 C=1 -> 0
```

📝 Übung 1 – DNF ablesen

Gegeben ist folgende Belegungstabelle:

A	B	C	Ausgang
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Bestimmt die DNF (auf Papier) und überprüft sie anschließend mit Python.

```
# TODO: Übung 1 - DNF als Python-Funktion eingeben und gegen die Tabelle prüfen
def meine_dnf(a, b, c):
    pass

for a, b, c in product([0, 1], repeat=3):
    print(f"A={a} B={b} C={c} -> {int(meine_dnf(a,b,c)) if meine_dnf(a,b,c) is not None else '?'}")
```

```
A=0 B=0 C=0 -> ?
A=0 B=0 C=1 -> ?
A=0 B=1 C=0 -> ?
A=0 B=1 C=1 -> ?
A=1 B=0 C=0 -> ?
A=1 B=0 C=1 -> ?
A=1 B=1 C=0 -> ?
A=1 B=1 C=1 -> ?
```

📝 Übung 2 – Äquivalenz prüfen

Prüft **automatisiert mit Python**, ob die beiden folgenden Terme äquivalent sind, indem ihr für **alle** Belegungen vergleicht, ob beide Funktionen stets denselben Wert liefern:

`Term A: ¬(A ∧ B)` und `Term B: ¬A ∨ ¬B` (De Morgansches Gesetz)

```
def term_a(a, b):
    return not (a and b)

def term_b(a, b):
    return (not a) or (not b)

# TODO: Übung 2 - prüft für alle Belegungen, ob term_a und term_b immer gleich sind
aequivalent = True
for a, b in product([0, 1], repeat=2):
    # TODO: Vergleich einbauen und ggf. aequivalent = False setzen
    pass

print("Äquivalent:", aequivalent)
```

```
Äquivalent: True
```

DS 22 – Karnaugh-Veitch-Diagramme und Minimierung

Informatik 10 – MINT-Zweig | Informatiksysteme

Lernziel

Ich vereinfache die DNF (mit bis zu vier Variablen) mithilfe von Karnaugh-Veitch-Diagrammen.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Eine lange DNF und die zugehörige aufwendige Schaltung als Optimierungsproblem betrachten.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Nachbarschaft und Blockbildung im KV-Diagramm durch Mustervergleich entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Ein Drei-Variablen-Beispiel gemeinsam eintragen, gruppieren und vereinfachen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Ein weiteres Drei-Variablen-Diagramm bearbeiten.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Gruppenbildung und resultierenden vereinfachten Term begründen lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Vier Variablen oder Alarmanlagen-Beispiel als Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Für die Eingangswerte … liefert die Schaltung den Ausgangswert …“
- „Der Term/die Tabelle repräsentiert die Schaltung, weil …“
- „Die Rückkopplung bzw. Gruppierung bewirkt …“

📌 Merksätze

- Ein **Karnaugh-Veitch-Diagramm (KV-Diagramm)** ordnet alle Belegungen so in einem Raster an, dass sich **benachbarte Felder nur in genau einer Variablen** unterscheiden.
- Ziel: Benachbarte `1`-Felder werden zu **Blöcken** der Größe 1, 2, 4, 8 … zusammengefasst. Je größer der Block, desto **weniger Variablen** braucht der zugehörige Term.
- Innerhalb eines Blocks fällt jede Variable weg, die sowohl **negiert als auch nicht-negiert** vorkommt – übrig bleiben nur die Variablen, die im gesamten Block **konstant** sind.
- Das KV-Diagramm liefert so eine **minimierte DNF**, die zur ursprünglichen DNF **äquivalent**, aber kürzer ist – wichtig für **kostengünstigere Schaltungen** (weniger Gatter).

💡 Beispiel: Minimierung von Hand

Belegungstabelle mit den `1`-Zeilen `A=0,B=0` und `A=0,B=1`:

DNF: `(¬A ∧ ¬B) ∨ (¬A ∧ B)`

Im KV-Diagramm liegen beide Felder nebeneinander (beide mit `A=0`) → sie lassen sich zu einem Block zusammenfassen, in dem `B` wegfällt:

Minimierte Form: `¬A`

Das lässt sich leicht mit Python nachprüfen – beide Terme müssen für alle Belegungen denselben Wert liefern:

```python
from itertools import product

def unminimiert(a, b):
    return (not a and not b) or (not a and b)

def minimiert(a, b):
    return not a

for a, b in product([0, 1], repeat=2):
    print(a, b, unminimiert(a, b) == minimiert(a, b))
```

```
0 0 True
0 1 True
1 0 True
1 1 True
```

📝 Übung 1 – KV-Diagramm zeichnen und minimieren

Gegeben ist folgende Belegungstabelle für drei Variablen:

A	B	C	Ausgang
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

1. Zeichnet auf Papier ein KV-Diagramm und findet die größtmöglichen Blöcke.
2. Notiert die minimierte Form.
3. Überprüft eure minimierte Form mit Python gegen die unminimierte DNF.

```python
def unminimiert_a(a, b, c):
    return ((not a and not b and not c) or (not a and not b and c) or
            (a and not b and not c) or (a and not b and c))

# TODO: Übung 1 - eure minimierte Form hier eintragen
def minimiert_a(a, b, c):
    pass

for a, b, c in product([0, 1], repeat=3):
    print(a, b, c, "->", unminimiert_a(a, b, c) == minimiert_a(a, b, c))
```

```
0 0 0 -> False
0 0 1 -> False
0 1 0 -> False
0 1 1 -> False
1 0 0 -> False
1 0 1 -> False
1 1 0 -> False
1 1 1 -> False
```

📝 Übung 2 – Von der Alarmanlage zurück zur Minimierung

Erinnert euch an die Alarmanlage aus DS 16: `Alarm = (Tuer ∧ Scharf) ∨ Rauch`. Ist dieser Term bereits minimal, oder lässt er sich mit einem KV-Diagramm weiter vereinfachen? Begründet eure Antwort in Worten (kein Python nötig).

(Eure Begründung hier)

🚀 Zusatzaufgabe – Vier Variablen

Erstellt selbst eine Belegungstabelle mit **vier** Variablen (`A, B, C, D`) und mindestens vier `1`-Zeilen, die sich zu einem einzigen Block zusammenfassen lassen. Prüft eure Minimierung mit Python.

DS 23 – Problem, Probleminstanz und abstrahierte Darstellung

Informatik 10 – MINT-Zweig | Informatisches Problemlösen

Lernziel

Ich unterscheide zwischen einem Problem und einer Probleminstanz und untersuche kleine, abstrahiert dargestellte Probleminstanzen, um strukturelle Eigenschaften zu erkennen.

> **Hinweis:** In diesem Themenfeld steht laut Lehrplan die **Auseinandersetzung mit Problemstrukturen und Lösungsstrategien** im Vordergrund – die Umsetzung in einer Programmiersprache ist **nicht** vorgesehen. Python-Zellen in diesem und den folgenden Notebooks sind daher **optional** und dienen nur zur Veranschaulichung.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Ein allgemeines Routenproblem und eine konkrete Karte gegenüberstellen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Problem, Probleminstanz und abstrahierte Darstellung aus den Unterschieden entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Die TSP-Instanz gemeinsam als Tabelle und gedanklich als Graph untersuchen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Beispiele als Problem oder Probleminstanz klassifizieren und begründen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Symmetrie der Entfernungstabelle und ihre Bedeutung erläutern lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Eigene Instanz formulieren; Python-Liste nur als optionale Demonstration.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Es handelt sich um ein Problem/eine Probleminstanz, weil …“
- „Der Ansatz prüft/wählt … und benötigt dabei …“
- „Die Heuristik ist hier geeignet/nicht geeignet, weil …“

📌 Merksätze

- Ein **Problem** ist eine allgemeine Fragestellung mit "Leerstellen" (z. B. "Finde die kürzeste Route zwischen beliebigen Städten").
- Eine **Probleminstanz** ist eine konkrete Ausprägung dieses Problems mit festen Werten (z. B. "Finde die kürzeste Route zwischen genau diesen 5 gegebenen Städten mit diesen Entfernungen").
- Probleminstanzen werden häufig **abstrahiert dargestellt**, um sie handhabbar zu machen, z. B. als:
  - **Tabelle** (z. B. Entfernungstabelle zwischen Orten)
  - **Graph** (Knoten = Objekte, Kanten = Beziehungen/Entfernungen)

    - **Liste** (z. B. Liste von Gegenständen mit Gewicht und Wert)
- Das Untersuchen mehrerer **Varianten** einer Probleminstanz hilft, **Symmetrien** und Muster zu erkennen, die spätere Lösungsstrategien vereinfachen können.

💡 Beispiel: Problem des Handlungsreisenden (TSP) als Graph

Problem: Finde für eine beliebige Menge von Städten mit gegebenen paarweisen Entfernungen die kürzeste Rundreise, die jede Stadt genau einmal besucht und zum Ausgangspunkt zurückkehrt.

Probleminstanz: 4 Städte A, B, C, D mit folgender Entfernungstabelle:

	A	B	C	D
A	–	10	15	20
B	10	–	35	25
C	15	35	–	30
D	20	25	30	–

Diese Tabelle ist eine **abstrahierte Darstellung** – sie enthält nur die für das Problem relevanten Informationen (Entfernungen), keine Straßennamen, keine Geographie etc.

📝 Übung 1 – Problem oder Probleminstanz?

Entscheidet für jede Aussage, ob es sich um ein **Problem** (allgemein) oder eine **Probleminstanz** (konkret) handelt, und begründet kurz:

1. "Packe möglichst wertvolle Gegenstände in einen Rucksack mit maximal 10 kg Tragkraft, wobei diese fünf Gegenstände zur Verfügung stehen: ..."
2. "Packe möglichst wertvolle Gegenstände in einen Rucksack mit begrenzter Tragkraft."
3. "Finde den kürzesten Weg von Saarbrücken nach Trier im gegebenen Straßennetz."
4. "Finde in einem beliebigen Graphen den kürzesten Weg zwischen zwei beliebigen Knoten."

(Eure Antworten hier)

📝 Übung 2 – Symmetrien erkennen

Betrachtet die Entfernungstabelle aus dem Beispiel oben.

1. Welche strukturelle Eigenschaft erkennt ihr an der Tabelle bezüglich `Entfernung(A,B)` und `Entfernung(B,A)`? Was bedeutet das für die Anzahl der Rundreisen, die ihr tatsächlich unterscheiden müsst?
2. Wie viele **verschiedene** Rundreisen (unter Berücksichtigung dieser Symmetrie) gibt es bei 4 Städten? Zählt systematisch.

(Eure Antworten hier)

🐍 Optionale Veranschaulichung mit Python

Die folgende Zelle listet alle möglichen Rundreisen für die obige Probleminstanz auf – nur zur Veranschaulichung, keine Pflichtaufgabe.

```python
from itertools import permutations

staedte = ["A", "B", "C", "D"]
start = staedte[0]
restliche = staedte[1:]

rundreisen = list(permutations(restliche))
print(f"Anzahl möglicher Rundreisen (ohne Symmetriebetrachtung): {len(rundreisen)}")
for reise in rundreisen:
    print(start, "->", " -> ".join(reise), "->", start)
```

```
Anzahl möglicher Rundreisen (ohne Symmetriebetrachtung): 6
A -> B -> C -> D -> A
A -> B -> D -> C -> A
A -> C -> B -> D -> A
A -> C -> D -> B -> A
A -> D -> B -> C -> A
A -> D -> C -> B -> A
```

🚀 Zusatzaufgabe

Formuliert selbst ein Problem und eine dazu passende Probleminstanz aus einem Kontext eurer Wahl (z. B. Stundenplanung, Einkaufstour, Gruppeneinteilung).

DS 24 – Optimale Lösung und Brute-Force-Ansatz

Informatik 10 – MINT-Zweig | Informatisches Problemlösen

Lernziel

Ich definiere eine optimale Lösung als beste zulässige Lösung bezüglich eines Kriteriums und beschreibe einen Brute-Force-Ansatz zu ihrer Ermittlung.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Frage stellen, wie sich die garantiert kürzeste Route finden lässt.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Zulässige und optimale Lösung sowie systematisches Ausprobieren aus der TSP-Instanz entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Rundreisen geordnet auflisten, Längen berechnen und die optimale Route bestimmen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Anzahl der Rundreisen für kleine Städtezahlen ermitteln.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Brute Force mit Garantie und schnell wachsendem Aufwand erklären lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Python-Auswertung oder eigene Fünf-Städte-Instanz ausschließlich optional.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Es handelt sich um ein Problem/eine Probleminstanz, weil …“
- „Der Ansatz prüft/wählt … und benötigt dabei …“
- „Die Heuristik ist hier geeignet/nicht geeignet, weil …“

📌 Merksätze

- Eine **zulässige Lösung** erfüllt alle Bedingungen des Problems (z. B. "besucht jede Stadt genau einmal").
- Eine **optimale Lösung** ist unter allen zulässigen Lösungen diejenige, die bezüglich eines festgelegten Kriteriums (z. B. Gesamtlänge) **am besten** ist.
- Ein **Brute-Force-Ansatz** ("rohe Gewalt") ermittelt die optimale Lösung, indem er **alle** zulässigen Lösungen systematisch erzeugt, bewertet und die beste auswählt.
- Vorteil: Brute-Force liefert garantiert die **optimale** Lösung.
- Nachteil: Die Anzahl der zu prüfenden Lösungen wächst bei vielen Problemen sehr schnell (z. B. **faktoriell** beim TSP) – Brute-Force wird für größere Instanzen schnell unpraktikabel.

💡 Beispiel: TSP von Hand mit Brute-Force

Wir verwenden die Entfernungstabelle aus DS 23 (4 Städte A, B, C, D). Berechnet **von Hand** die Gesamtlänge für die Rundreise `A → B → C → D → A`:

`10 (A-B) + 35 (B-C) + 30 (C-D) + 20 (D-A) = 95`

Wiederholt das für mindestens zwei weitere Rundreisen und vergleicht die Längen.

(Eure Berechnungen hier)

🐍 Kontrolle: Brute-Force in Python

Die folgende Zelle probiert **alle** Rundreisen aus und findet die optimale Lösung – vergleicht das Ergebnis mit euren Handrechnungen.

```python
from itertools import permutations

entfernungen = {
    ("A", "B"): 10, ("A", "C"): 15, ("A", "D"): 20,
    ("B", "C"): 35, ("B", "D"): 25, ("C", "D"): 30,
}

def entfernung(ort1, ort2):
    if (ort1, ort2) in entfernungen:
        return entfernungen[(ort1, ort2)]
    return entfernungen[(ort2, ort1)]

def rundreise_laenge(reise):
    laenge = 0
    for i in range(len(reise) - 1):
        laenge += entfernung(reise[i], reise[i + 1])
    laenge += entfernung(reise[-1], reise[0])
    return laenge

staedte = ["A", "B", "C", "D"]
beste_reise = None
kuerzeste_laenge = None

for permutation in permutations(staedte[1:]):
    reise = [staedte[0]] + list(permutation)
    laenge = rundreise_laenge(reise)
    print(reise, "->", laenge)
    if kuerzeste_laenge is None or laenge < kuerzeste_laenge:
        kuerzeste_laenge = laenge
        beste_reise = reise

print("\nOptimale Rundreise:", beste_reise, "mit Länge", kuerzeste_laenge)
```

```
['A', 'B', 'C', 'D'] -> 95
['A', 'B', 'D', 'C'] -> 80
['A', 'C', 'B', 'D'] -> 95
['A', 'C', 'D', 'B'] -> 80
['A', 'D', 'B', 'C'] -> 95
['A', 'D', 'C', 'B'] -> 95

Optimale Rundreise: ['A', 'B', 'D', 'C'] mit Länge 80
```

🚀 Optionale Vertiefung – Neue Probleminstanz mit Python

Stellt eine eigene Entfernungstabelle für **5 Städte** auf (frei erfunden, plausible Werte) und passt den Python-Code oben entsprechend an, um die optimale Rundreise zu finden.

📝 Übung 2 – Wachstum beobachten

Zählt die Anzahl der geprüften Rundreisen bei 4, 5 und 6 Städten (Tipp: Es sind `(n-1)!` Reisen bei `n` Städten, wenn man die Startstadt festhält). Was fällt euch am Wachstum auf?

(Eure Beobachtung hier)

✅ Hinweis zu Übung 1

Kopiert dazu einfach den `entfernungen`-Dictionary-Block von oben, fügt die fünfte Stadt mit allen nötigen Paar-Entfernungen hinzu und erweitert die `staedte`-Liste.

DS 25 – Heuristiken: Der Greedy-Ansatz

Informatik 10 – MINT-Zweig | Informatisches Problemlösen

Lernziel

Ich definiere den Begriff der Heuristik, beschreibe eine Heuristik zur Problemlösung und wende sie auf Probleminstanzen an.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Das Problem des schnell wachsenden Brute-Force-Aufwands erneut aufgreifen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Nearest Neighbor als schrittweise lokale Entscheidung aus Schülerideen entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Die Route ab A gemeinsam bestimmen und mit dem Optimum vergleichen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Die Heuristik von einem anderen Startpunkt aus von Hand anwenden.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Vorteil, fehlende Optimalitätsgarantie und Einfluss der Instanz beurteilen lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Gegenbeispiel suchen; Python nur als optionale Kontrolle.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Es handelt sich um ein Problem/eine Probleminstanz, weil …“
- „Der Ansatz prüft/wählt … und benötigt dabei …“
- „Die Heuristik ist hier geeignet/nicht geeignet, weil …“

📌 Merksätze

- Eine **Heuristik** ist eine Lösungsstrategie, die **schnell** eine (meist gute, aber nicht garantiert optimale) Lösung liefert, ohne alle Möglichkeiten zu prüfen.
- Eine **Greedy-Heuristik** ("gierig") trifft in jedem Schritt die Entscheidung, die im **Moment** am besten erscheint, ohne spätere Auswirkungen zu berücksichtigen.
- Beim TSP: Die Greedy-Heuristik **Nearest Neighbor** wählt von der aktuellen Stadt aus immer die **nächstgelegene, noch nicht besuchte** Stadt.
- Heuristiken sind **schneller** als Brute-Force, liefern aber **nicht immer** die optimale Lösung – manchmal führt eine "im Moment beste" Entscheidung in eine Sackgasse.

💡 Beispiel: Nearest Neighbor von Hand

Mit der Entfernungstabelle aus DS 23/24, Start in `A`:

1. Von `A` aus: nächstgelegene unbesuchte Stadt ist `B` (Entfernung 10).
2. Von `B` aus: nächstgelegene unbesuchte Stadt ist `D` (Entfernung 25, da `C` mit 35 weiter ist).
3. Von `D` aus: einzige verbleibende Stadt ist `C` (Entfernung 30).
4. Zurück nach `A` (Entfernung 15).

Route: `A → B → D → C → A`, Länge: `10 + 25 + 30 + 15 = 80`

Die Route `A → B → D → C → A` hat die Länge **80** und ist für diese Instanz zugleich optimal. Das Beispiel zeigt also: Eine Heuristik kann das Optimum treffen, garantiert dies aber nicht.

🐍 Kontrolle: Nearest-Neighbor-Heuristik in Python

```python
entfernungen = {
    ("A", "B"): 10, ("A", "C"): 15, ("A", "D"): 20,
    ("B", "C"): 35, ("B", "D"): 25, ("C", "D"): 30,
}

def entfernung(ort1, ort2):
    if (ort1, ort2) in entfernungen:
        return entfernungen[(ort1, ort2)]
    return entfernungen[(ort2, ort1)]

def nearest_neighbor(staedte, start):
    unbesucht = set(staedte)
    unbesucht.remove(start)
    route = [start]
    aktuell = start

    while unbesucht:
        naechste = min(unbesucht, key=lambda stadt: entfernung(aktuell, stadt))
        route.append(naechste)
        unbesucht.remove(naechste)
        aktuell = naechste

    return route

def rundreise_laenge(reise):
    laenge = 0
    for i in range(len(reise) - 1):
        laenge += entfernung(reise[i], reise[i + 1])
    laenge += entfernung(reise[-1], reise[0])
    return laenge

staedte = ["A", "B", "C", "D"]
route = nearest_neighbor(staedte, "A")
print("Nearest-Neighbor-Route:", route)
print("Länge:", rundreise_laenge(route))
```

```
Nearest-Neighbor-Route: ['A', 'B', 'D', 'C']
Länge: 80
```

📝 Übung 1 – Anderer Startpunkt

Wendet die Nearest-Neighbor-Heuristik **von Hand** an, aber startet diesmal in Stadt `C`. Vergleicht die Route und Länge mit dem Ergebnis von Start `A`. Was fällt euch auf?

🚀 Optionale Vertiefung – Heuristik vs. Optimum

Vergleicht für eure **eigene 5-Städte-Instanz aus DS 24 Übung 1**: Wie lang ist die Route der Nearest-Neighbor-Heuristik im Vergleich zur optimalen (Brute-Force-)Route? Berechnet die **prozentuale Abweichung**.

DS 26/27 – Vertiefung am Beispiel: Das Rucksackproblem

Informatik 10 – MINT-Zweig | Informatisches Problemlösen

Lernziel

Ich wende Brute-Force- und Greedy-Ansätze auf ein neues Beispielproblem (Rucksackproblem) an und vergleiche beide Strategien.

Hinweis: Dieses Notebook deckt zwei Doppelstunden (DS 26 und DS 27) ab. Die beiden Verlaufspläne unten zeigen, wie sich die Inhalte auf zwei separate 90-Minuten-Einheiten aufteilen lassen — DS 26 endet nach der Brute-Force-Lösung (Übung 1), DS 27 beginnt mit der Greedy-Heuristik (Übung 2).

🕘 Stundenverlaufsplan – DS 26 (90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg	10'	Ein konkretes Packproblem mit Gewichtsgrenze vorstellen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Modellierung	25'	Gegenstände, Gewicht, Wert, Tragkraft, zulässige und optimale Auswahl herausarbeiten.	Unterrichtsgespräch; Tabelle am Board
Systematisches Ausprobieren	20'	Kleine Teilmengen gemeinsam bilden, bewerten und einen Brute-Force-Plan formulieren.	Plenum; gemeinsames Arbeiten an der Tabelle
Kurzer Transfer	15'	Weitere Teilmengen prüfen und die Anzahl `2^n` für kleine `n` bestimmen.	Einzelarbeit, anschließend 4 Tandems + 1 Trio
Auswertung	10'	Optimale Auswahl und exponentiell wachsenden Aufwand erklären lassen.	Plenum
Reserve / Vertiefung	10'	Python-Code nur demonstrieren oder freiwillig nachvollziehen.	Flexibel

🕘 Stundenverlaufsplan – DS 27 (90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Wiedereinstieg	10'	Brute-Force-Ergebnis und zentrale Begriffe aus DS 26 aktivieren.	Plenum
Fragend-entwickelnde Erarbeitung	25'	Wert-Gewicht-Verhältnis als mögliche schnelle Auswahlregel entdecken und ordnen.	Unterrichtsgespräch; gemeinsame Tabelle
Heuristik anwenden	20'	Greedy-Auswahl von Hand durchführen und jeden Schritt begründen.	Plenum mit kurzen Denkpausen
Kurzer Transfer	15'	Greedy-Ergebnis mit der optimalen Auswahl vergleichen und ein Gegenbeispiel untersuchen.	Einzelarbeit, anschließend 4 Tandems + 1 Trio
Bewertung	10'	Vorteile, Aufwand und fehlende Optimalitätsgarantie beurteilen.	Plenum
Reserve / Vertiefung	10'	Eigenes Gegenbeispiel oder freiwillige Python-Implementierung.	Flexibel

🗣️ Sprachhilfe

- „Es handelt sich um ein Problem/eine Probleminstanz, weil …“
- „Der Ansatz prüft/wählt … und benötigt dabei …“
- „Die Heuristik ist hier geeignet/nicht geeignet, weil …“

📌 Merksätze

- Beim **Rucksackproblem (Knapsack Problem)** stehen mehrere Gegenstände mit jeweils einem **Gewicht** und einem **Wert** zur Verfügung. Gesucht ist die Auswahl von Gegenständen mit **maximalem Gesamtwert**, deren **Gesamtgewicht** eine vorgegebene Tragkraft nicht übersteigt.
- **Brute-Force-Ansatz:** Alle möglichen **Teilmengen** der Gegenstände werden gebildet und bewertet – bei `n` Gegenständen gibt es `2^n` Teilmengen (**exponentieller Aufwand**).
- **Greedy-Heuristik:** Die Gegenstände werden nach ihrem **Wert-Gewicht-Verhältnis** (Wert ÷ Gewicht) sortiert und **so lange** eingepackt, bis kein weiterer Gegenstand mehr hineinpasst.

💡 Probleminstanz

Rucksack-Tragkraft: **10 kg**

Gegenstand	Gewicht (kg)	Wert (€)
Zelt	5	60
Schlafsack	4	40
Kocher	2	30
Proviant	3	25
Erste-Hilfe-Set	1	20

Aufgabe zum Einstieg (von Hand): Findet durch Ausprobieren mindestens zwei zulässige Kombinationen (Gesamtgewicht ≤ 10 kg) und berechnet jeweils den Gesamtwert.

(Eure Kombinationen und Werte hier)

🐍 Brute-Force-Lösung

```python
from itertools import combinations

gegenstaende = [
    ("Zelt", 5, 60),
    ("Schlafsack", 4, 40),
    ("Kocher", 2, 30),
    ("Proviant", 3, 25),
    ("Erste-Hilfe-Set", 1, 20),
]
tragkraft = 10

beste_kombination = None
bester_wert = 0

for r in range(len(gegenstaende) + 1):
    for kombination in combinations(gegenstaende, r):
        gesamtgewicht = sum(g[1] for g in kombination)
        gesamtwert = sum(g[2] for g in kombination)
        if gesamtgewicht <= tragkraft and gesamtwert > bester_wert:
            bester_wert = gesamtwert
            beste_kombination = kombination

print("Optimale Kombination:", [g[0] for g in beste_kombination])
print("Gesamtwert:", bester_wert)
print("Gesamtgewicht:", sum(g[1] for g in beste_kombination))
```

```
Optimale Kombination: ['Zelt', 'Schlafsack', 'Erste-Hilfe-Set']
Gesamtwert: 120
Gesamtgewicht: 10
```

📝 Übung 1 – Anzahl der Teilmengen

Wie viele Teilmengen müssen bei `n` Gegenständen geprüft werden? Berechnet dies für die vorliegende Instanz mit `2^n` und begründet das Ergebnis. Die folgende Code-Zelle kann optional zur Kontrolle ergänzt werden.

📝 Übung 2 – Greedy-Heuristik von Hand anwenden

Berechnet für jeden Gegenstand das **Wert-Gewicht-Verhältnis**, sortiert die Gegenstände danach absteigend und packt sie in dieser Reihenfolge, sofern sie noch passen. Vergleicht das Ergebnis mit der optimalen Brute-Force-Lösung. Haltet jeden Auswahlschritt in einer Tabelle fest.

```python
# OPTIONAL: Greedy-Heuristik für das Rucksackproblem programmieren

def greedy_rucksack(gegenstaende, tragkraft):
    # Tipp: sortiert(gegenstaende, key=lambda g: g[2]/g[1], reverse=True)
    pass
```

📝 Übung 3 – Grenzen der Heuristik

Überlegt euch (oder konstruiert) eine Probleminstanz, bei der die Greedy-Heuristik **nicht** die optimale Lösung findet. Testet eure Instanz mit beiden Ansätzen.

DS 28 – Aufwand verschiedener Lösungsansätze vergleichen

Informatik 10 – MINT-Zweig | Informatisches Problemlösen

Lernziel

Ich vergleiche den Aufwand verschiedener Lösungsansätze miteinander.

> Der Begriff **Aufwand** wird hier in einem **didaktisch reduzierten, anschaulichen** Sinn verwendet (Anzahl der geprüften Möglichkeiten bzw. Laufzeit) – eine formale Komplexitätstheorie folgt erst in der Oberstufe.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	An bekannte Anzahlen aus TSP und Rucksackproblem anknüpfen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Faktorielles und exponentielles Wachstum anschaulich über Wertetabellen entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Eine kleine Messreihe gemeinsam interpretieren; Code dient nur der Demonstration.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Für ausgewählte Werte von `n` die Zahl `2^n` berechnen und vergleichen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Unterschied zwischen geprüften Möglichkeiten und gemessener Zeit erklären lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Diagramm oder Hochrechnung als Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Es handelt sich um ein Problem/eine Probleminstanz, weil …“
- „Der Ansatz prüft/wählt … und benötigt dabei …“
- „Die Heuristik ist hier geeignet/nicht geeignet, weil …“

📌 Merksätze

- Der **Aufwand** eines Lösungsansatzes lässt sich anschaulich als **Anzahl der zu prüfenden Möglichkeiten** oder als **tatsächlich gemessene Rechenzeit** beschreiben.
- Beim TSP (Brute-Force) wächst der Aufwand **faktoriell** mit der Anzahl der Städte: `(n-1)!`
- Beim Rucksackproblem (Brute-Force) wächst der Aufwand **exponentiell** mit der Anzahl der Gegenstände: `2^n`
- Greedy-Heuristiken prüfen dagegen **wesentlich weniger Möglichkeiten**; der Aufwand entsteht hier vor allem durch das Sortieren und das einmalige Durchlaufen der Gegenstände.

- Schon bei mäßig größeren Probleminstanzen wird der Unterschied zwischen Brute-Force und Heuristik **drastisch** spürbar.

🐍 Messung: Wachstum des Brute-Force-Aufwands (TSP)

Die folgende Zelle misst, wie stark die Anzahl der zu prüfenden Rundreisen und die tatsächliche Rechenzeit mit der Anzahl der Städte wächst.

```python
import time
from itertools import permutations

def anzahl_und_zeit_bruteforce(anzahl_staedte):
    staedte = list(range(anzahl_staedte))
    start = time.time()
    anzahl = 0
    for _ in permutations(staedte[1:]):
        anzahl += 1
    dauer = time.time() - start
    return anzahl, dauer

print(f"{'Städte':>7} | {'Anzahl Routen':>15} | {'Zeit (s)':>10}")
for n in range(2, 9):
    anzahl, dauer = anzahl_und_zeit_bruteforce(n)
    print(f"{n:>7} | {anzahl:>15} | {dauer:>10.4f}")
```

```
 Städte |   Anzahl Routen |   Zeit (s)
      2 |               1 |     0.0000
      3 |               2 |     0.0000
      4 |               6 |     0.0000
      5 |              24 |     0.0000
      6 |             120 |     0.0000
      7 |             720 |     0.0000
      8 |            5040 |     0.0002
```

📝 Übung 1 – Tabelle interpretieren

Betrachtet die Ausgabe oben:

1. Um welchen Faktor wächst die Anzahl der Routen jeweils von `n` zu `n+1` Städten?
2. Schätzt anhand des Musters (ohne es auszuführen!), wie lange die Berechnung bei **12 Städten** ungefähr dauern würde.

(Eure Antworten hier)

📝 Übung 2 – Rucksackproblem: Brute-Force vs. Greedy im Vergleich

Erstellt für eine wachsende Anzahl an Gegenständen (z. B. 5, 10, 15, 18) eine Tabelle mit der Anzahl der von Brute Force geprüften Teilmengen (`2^n`). Vergleicht dieses Wachstum mit der Greedy-Heuristik, die die Gegenstände sortiert und anschließend einmal durchläuft. Die Code-Zelle kann optional zur Kontrolle ergänzt werden.

```python
# OPTIONAL: Tabelle mit n und 2**n zur Kontrolle ausgeben
for n in [5, 10, 15, 18]:
    pass  # TODO: 2**n berechnen und ausgeben
```

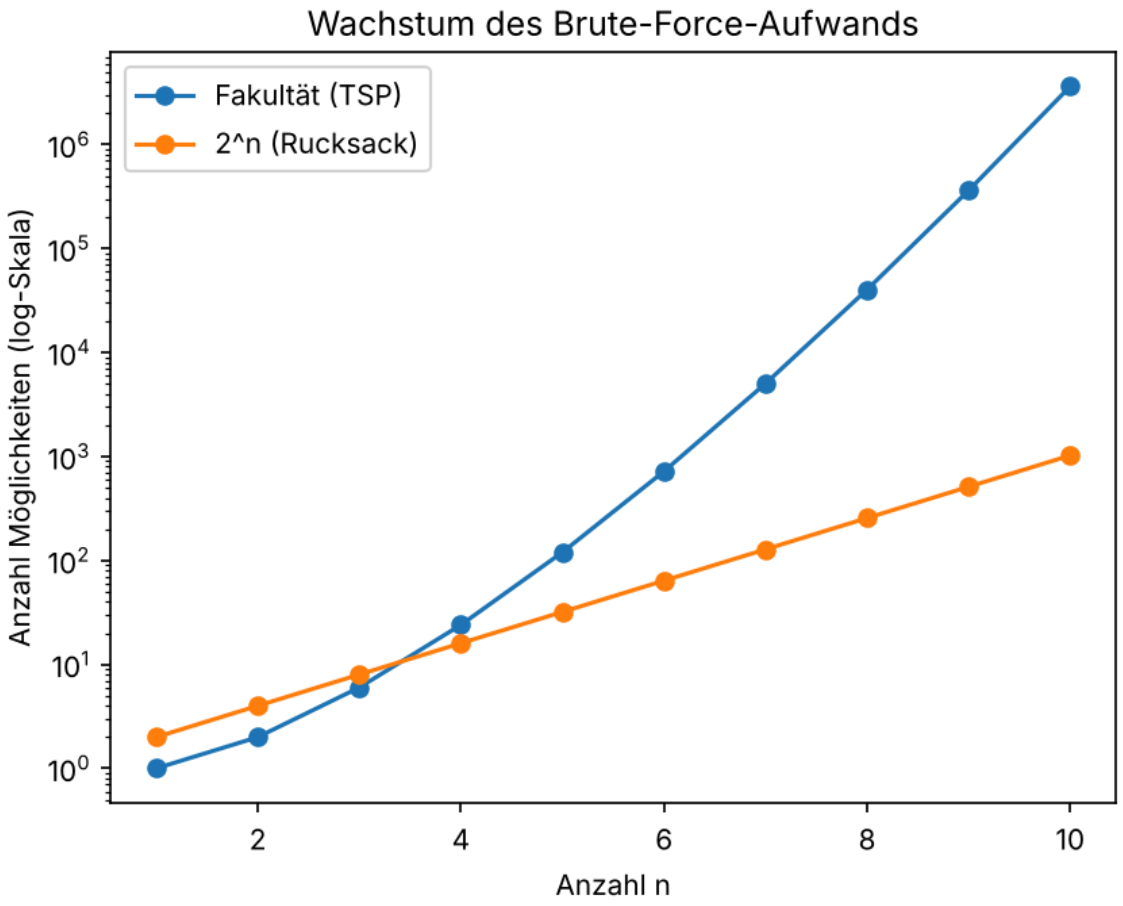
🚀 Zusatzaufgabe – Diagramm

Falls ihr mit `matplotlib` vertraut seid: Stellt das faktorielle Wachstum (TSP) und das exponentielle Wachstum (Rucksack) grafisch gegenüber.

```python
import matplotlib.pyplot as plt
import math

n_werte = list(range(1, 11))
faktoriell = [math.factorial(n) for n in n_werte]
exponentiell = [2**n for n in n_werte]

plt.plot(n_werte, faktoriell, marker="o", label="Fakultät (TSP)")
plt.plot(n_werte, exponentiell, marker="o", label="2^n (Rucksack)")
plt.yscale("log")
plt.xlabel("Anzahl n")
plt.ylabel("Anzahl Möglichkeiten (log-Skala)")
plt.legend()
plt.title("Wachstum des Brute-Force-Aufwands")
plt.show()
```



DS 29 – Bewertung von Heuristiken & Abschluss

Informatik 10 – MINT-Zweig | Informatisches Problemlösen

Lernziel

Ich beurteile, wie gut eine Heuristik bei verschiedenen Probleminstanzen funktioniert, und festige die Inhalte des Themenfelds.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Kriterien sammeln, nach denen eine schnelle, nicht garantiert optimale Lösung beurteilt werden kann.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Prozentuale Abweichung und die Notwendigkeit mehrerer Instanzen aus Beispielen entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Das vorbereitete Experiment gemeinsam auswerten; Python bleibt Demonstrationswerkzeug.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Eine begründete Bewertung der Nearest-Neighbor-Heuristik formulieren.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Urteile anhand von Optimum, Abweichung, Aufwand und Instanzvielfalt vergleichen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Abschlussfragen als Lernstandscheck oder Vorbereitung auf einen Kurztest.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Es handelt sich um ein Problem/eine Probleminstanz, weil …“
- „Der Ansatz prüft/wählt … und benötigt dabei …“
- „Die Heuristik ist hier geeignet/nicht geeignet, weil …“

📌 Merksätze

- Um eine Heuristik zu **bewerten**, vergleicht man ihre Lösung mit der **optimalen** Lösung (sofern diese bekannt bzw. berechenbar ist) – üblich ist die **prozentuale Abweichung**.
- Eine Heuristik kann bei **manchen** Probleminstanzen die optimale Lösung treffen und bei **anderen** deutlich schlechter abschneiden – deshalb reicht ein einzelner Test nicht aus.
- Zur verlässlichen Bewertung testet man eine Heuristik über **mehrere, möglichst unterschiedliche** Probleminstanzen und betrachtet z. B. die **durchschnittliche** Abweichung.

🐍 Experiment: Nearest-Neighbor über mehrere zufällige Instanzen

Die folgende Zelle erzeugt mehrere zufällige TSP-Instanzen, löst sie sowohl mit Brute-Force (optimal) als auch mit Nearest-Neighbor (Heuristik) und berechnet die durchschnittliche Abweichung.

```python
import random
from itertools import permutations

def zufaellige_instanz(anzahl_staedte, seed):
    random.seed(seed)
    staedte = list(range(anzahl_staedte))
    entfernungen = {}
    for i in range(anzahl_staedte):
        for j in range(i + 1, anzahl_staedte):
            entfernungen[(i, j)] = random.randint(5, 50)
    return staedte, entfernungen

def entfernung(entfernungen, a, b):
    return entfernungen[(a, b)] if (a, b) in entfernungen else entfernungen[(b, a)]

def rundreise_laenge(entfernungen, reise):
    laenge = 0
    for i in range(len(reise) - 1):
        laenge += entfernung(entfernungen, reise[i], reise[i + 1])
    laenge += entfernung(entfernungen, reise[-1], reise[0])
    return laenge

def optimale_route(staedte, entfernungen):
    beste, beste_laenge = None, None
    for perm in permutations(staedte[1:]):
        reise = [staedte[0]] + list(perm)
        laenge = rundreise_laenge(entfernungen, reise)
        if beste_laenge is None or laenge < beste_laenge:
            beste, beste_laenge = reise, laenge
    return beste, beste_laenge

def nearest_neighbor(staedte, entfernungen, start):
    unbesucht = set(staedte)
    unbesucht.remove(start)
    route = [start]
    aktuell = start
    while unbesucht:
        naechste = min(unbesucht, key=lambda s: entfernung(entfernungen, aktuell, s))
        route.append(naechste)
        unbesucht.remove(naechste)
        aktuell = naechste
    return route

abweichungen = []
for seed in range(10):
    staedte, entfernungen = zufaellige_instanz(6, seed)
    _, opt_laenge = optimale_route(staedte, entfernungen)
    nn_route = nearest_neighbor(staedte, entfernungen, staedte[0])
    nn_laenge = rundreise_laenge(entfernungen, nn_route)
    abweichung = (nn_laenge - opt_laenge) / opt_laenge * 100
    abweichungen.append(abweichung)
    print(f"Instanz {seed}: Optimal={opt_laenge}, Nearest-Neighbor={nn_laenge}, Abweichung=
{abweichung:.1f}%")

print(f"\nDurchschnittliche Abweichung: {sum(abweichungen)/len(abweichungen):.1f}%")
```

```
Instanz 0: Optimal=119, Nearest-Neighbor=119, Abweichung=0.0%
Instanz 1: Optimal=92, Nearest-Neighbor=92, Abweichung=0.0%
Instanz 2: Optimal=79, Nearest-Neighbor=79, Abweichung=0.0%
Instanz 3: Optimal=122, Nearest-Neighbor=124, Abweichung=1.6%
Instanz 4: Optimal=81, Nearest-Neighbor=81, Abweichung=0.0%
Instanz 5: Optimal=113, Nearest-Neighbor=113, Abweichung=0.0%
Instanz 6: Optimal=78, Nearest-Neighbor=109, Abweichung=39.7%
```

```
Instanz 7: Optimal=67, Nearest-Neighbor=75, Abweichung=11.9%
Instanz 8: Optimal=83, Nearest-Neighbor=87, Abweichung=4.8%
Instanz 9: Optimal=114, Nearest-Neighbor=135, Abweichung=18.4%

Durchschnittliche Abweichung: 7.7%
```

📝 Übung 1 – Auswertung

Beantwortet anhand der Ausgabe oben:

1. Gab es eine Instanz, bei der die Heuristik die optimale Lösung getroffen hat (Abweichung 0 %)?
2. Wie groß war die größte beobachtete Abweichung?
3. Ist die Nearest-Neighbor-Heuristik eurer Einschätzung nach für den praktischen Einsatz geeignet? Begründet.

(Eure Antworten hier)

📝 Übung 2 (Abschlussaufgabe) – Zusammenfassung

Beantwortet in eigenen Worten (als Wiederholung des gesamten Themenfelds):

1. Was ist der Unterschied zwischen einem Problem und einer Probleminstanz?
2. Wann liefert ein Brute-Force-Ansatz garantiert die optimale Lösung, und wo liegt sein Nachteil?
3. Was zeichnet eine Greedy-Heuristik aus, und welches Risiko birgt sie?
4. Nennt je ein Beispiel für ein Problem mit faktoriellem und eines mit exponentiellem Brute-Force-Aufwand.

(Eure Antworten hier – als Vorbereitung für einen möglichen Kurztest)

DS 30 – Sprachmodelle: Regelbasierte Systeme vs. n-Gramm-Modelle

Informatik 10 – MINT-Zweig | Künstliche Intelligenz

Lernziel

Ich erläutere den Begriff des Sprachmodells und unterscheide zwischen regelbasierten Systemen und n-Gramm-Modellen.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Antworten eines einfachen Chatprogramms zeigen und Funktionshypothesen sammeln.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Regelbasiertes System und statistisches Sprachmodell aus Beispielreaktionen unterscheiden.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Mini-ELIZA gemeinsam testen und eine Regel im Code nachvollziehen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Eigene Testsätze formulieren und Grenzen des Regelsystems dokumentieren.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Regeln und Statistik mit passenden Fachbegriffen vergleichen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Eine zusätzliche Regel implementieren.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Auf den Kontext … folgt mit der Häufigkeit/Wahrscheinlichkeit … das Wort …“
- „Der erzeugte Text verändert sich, weil …“
- „Im Vergleich zum n-Gramm-Modell kann das LLM …, jedoch …“

📌 Merksätze

- Ein **Sprachmodell** ist ein System, das sprachliche Strukturen bzw. Wahrscheinlichkeiten erfasst, um Texte zu analysieren oder neu zu erzeugen.
- **Regelbasierte Systeme** (z. B. das historische Programm **ELIZA**, 1966) reagieren auf Basis **fest programmierter Regeln** und Mustererkennung (z. B. "Wenn der Satz 'Ich fühle mich …' enthält, antworte mit 'Warum fühlst du dich …?'"). Sie "verstehen" die Sprache nicht, sondern wenden feste Muster an.
- **n-Gramm-Modelle** basieren auf der **statistischen Analyse großer Textmengen**: Sie erfassen, wie häufig bestimmte Wortfolgen (n-Gramme) in einem Textkorpus vorkommen, und nutzen diese Häufigkeiten zur Vorhersage des nächsten Wortes.

- Moderne **Large Language Models (LLMs)** nutzen komplexe neuronale Netze, beruhen aber im Kern auf einem ähnlichen Prinzip: Wahrscheinlichkeiten für die Fortsetzung eines Textes zu schätzen.

💡 Beispiel: Ein Mini-ELIZA (regelbasiert)

Das folgende, stark vereinfachte Programm reagiert nach festen Mustern – es "versteht" nichts, sondern erkennt nur Schlüsselwörter.

```python
def mini_eliza(satz):
    satz = satz.lower()
    if "ich fühle mich" in satz:
        gefuehl = satz.split("ich fühle mich")[-1].strip()
        return f"Warum fühlst du dich {gefuehl}?"
    elif "ich brauche" in satz:
        bedarf = satz.split("ich brauche")[-1].strip()
        return f"Warum brauchst du {bedarf}?"
    elif "?" in satz:
        return "Interessante Frage - was denkst du selbst darüber?"
    else:
        return "Erzähl mir mehr davon."

print(mini_eliza("Ich fühle mich müde"))
print(mini_eliza("Ich brauche mehr Zeit"))
print(mini_eliza("Wie geht es dir?"))
print(mini_eliza("Die Sonne scheint"))
```

```
Warum fühlst du dich müde?
Warum brauchst du mehr zeit?
Interessante Frage - was denkst du selbst darüber?
Erzähl mir mehr davon.
```

📝 Übung 1 – Grenzen des regelbasierten Ansatzes

Testet `mini_eliza` mit mindestens drei eigenen Sätzen, die **nicht** zu den vorgesehenen Mustern passen. Beschreibt, warum das System hier keine sinnvolle Antwort liefern kann.

📝 Übung 2 – Regel erweitern

Erweitert `mini_eliza` um mindestens eine weitere Regel eurer Wahl (z. B. Reaktion auf "Ich mag …" oder "Ich hasse …").

```python
# TODO: Übung 2 - mini_eliza um eine neue Regel erweitern
def mini_eliza_erweitert(satz):
    pass
```

📝 Übung 3 – Vergleich in Worten

Beantwortet: Warum stößt ein rein regelbasiertes System bei der Vielfalt natürlicher Sprache schnell an Grenzen? Welchen grundsätzlich anderen Ansatz verfolgen n-Gramm-Modelle (Stichwort: Regeln vs. Statistik)? Diese Frage wird in den nächsten Doppelstunden praktisch vertieft.

(Eure Antwort hier)

DS 31 – n-Gramm-Analyse von Textkorpora (von Hand)

Informatik 10 – MINT-Zweig | Künstliche Intelligenz

Lernziel

Ich simuliere die Textkörperanalyse durch das Ermitteln von n-Gramm-Wahrscheinlichkeiten in gegebenen Textkörpern.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Einen kurzen Satz in aufeinanderfolgende Wortgruppen zerlegen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Uni-, Bi- und Trigramme sowie Häufigkeiten aus den Schülerbeispielen entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Die Bigramme des Beispielkorpus gemeinsam vollständig auszählen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Für einen kurzen Korpus Häufigkeiten und Folgewortwahrscheinlichkeiten bestimmen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Die Berechnung einer bedingten Folgewortwahrscheinlichkeit sprachlich erklären lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Python-Kontrolle oder Trigramme als Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Auf den Kontext … folgt mit der Häufigkeit/Wahrscheinlichkeit … das Wort …“
- „Der erzeugte Text verändert sich, weil …“
- „Im Vergleich zum n-Gramm-Modell kann das LLM …, jedoch …“

📌 Merksätze

- Ein **n-Gramm** ist eine Folge von `n` aufeinanderfolgenden Wörtern (oder Zeichen) aus einem Text.
  - **Unigramm** (n=1): einzelne Wörter
  - **Bigramm** (n=2): Wortpaare
  - **Trigramm** (n=3): Wortdreiergruppen
- Bei der **Textkörperanalyse** zählt man, wie oft jedes n-Gramm in einem **Textkorpus** (Trainingstext) vorkommt.
- Aus diesen Häufigkeiten lässt sich die **Wahrscheinlichkeit** ableiten, mit der auf ein bestimmtes Wort (bzw. eine Wortfolge) ein bestimmtes Folgewort folgt – das ist die Grundlage für die spätere Textgenerierung (DS 33).

💡 Beispiel: Bigramme von Hand zählen

Textkorpus: **"der hund läuft der hund bellt der hund schläft"**

Bigramme (Wortpaare): `(der, hund)`, `(hund, läuft)`, `(läuft, der)`, `(der, hund)`, `(hund, bellt)`, `(bellt, der)`, `(der, hund)`, `(hund, schläft)`

Zählt man die Häufigkeiten, ergibt sich für das Wort **"der"**, gefolgt von **"hund"**: 3-mal – das ist das mit Abstand häufigste Bigramm mit "der" als erstem Wort.

📝 Übung 1 – Von Hand zählen

Gegeben ist der Textkorpus:

"die katze schläft die katze spielt die katze frisst die katze schläft"

Zählt **von Hand** (auf Papier oder in dieser Zelle) alle Bigramme und deren Häufigkeit. Welches Wort folgt im Korpus am häufigsten direkt auf "die"?

(Eure Auszählung hier, z. B. als Tabelle)

🐍 Kontrolle mit Python

Überprüft eure Handauszählung mit dem folgenden Code.

```python
def erstelle_bigramme(text):
    woerter = text.split()
    bigramme = []
    for i in range(len(woerter) - 1):
        bigramme.append((woerter[i], woerter[i + 1]))
    return bigramme

def zaehle_haeufigkeiten(bigramme):
    haeufigkeiten = {}
    for bigramm in bigramme:
        haeufigkeiten[bigramm] = haeufigkeiten.get(bigramm, 0) + 1
    return haeufigkeiten

korpus = "die katze schläft die katze spielt die katze frisst die katze schläft"
bigramme = erstelle_bigramme(korpus)
haeufigkeiten = zaehle_haeufigkeiten(bigramme)

for bigramm, anzahl in haeufigkeiten.items():
    print(bigramm, ":", anzahl)
```

```
('die', 'katze') : 4
('katze', 'schläft') : 2
('schläft', 'die') : 1
('katze', 'spielt') : 1
('spielt', 'die') : 1
('katze', 'frisst') : 1
('frisst', 'die') : 1
```

📝 Übung 2 – Wahrscheinlichkeiten berechnen

Berechnet aus den Häufigkeiten oben die **Wahrscheinlichkeit** für jedes Wort, das auf "die" folgt (Anzahl des jeweiligen Bigramms geteilt durch die Gesamtanzahl aller Bigramme, die mit "die" beginnen).

🚀 Zusatzaufgabe – Trigramme

Erweitert den Code so, dass **Trigramme** (Dreiergruppen) statt Bigramme gezählt werden. Verwendet euren eigenen, etwas längeren Beispielsatz.

DS 32 – Rechnergestützte Textkörperanalyse

Informatik 10 – MINT-Zweig | Künstliche Intelligenz

Lernziel

Ich verwalte n-Gramm-Wahrscheinlichkeiten tabellarisch und analysiere größere Textkörper rechnergestützt.

> Der Lehrplan schlägt als unterrichtsbegleitende Software z. B. **SoekiaGPT** vor – ein kostenloses, ohne Login nutzbares Webtool, in dem man eigene Textsammlungen hochladen oder vorgegebene Sammlungen (z. B. Märchen) nutzen kann; es zeigt n-Gramme, deren Häufigkeiten und die daraus resultierenden Wortvorschläge farblich aufbereitet an. In diesem Notebook bauen wir die Textkörperanalyse zusätzlich **selbst in Python** nach, um die Funktionsweise "unter der Haube" zu verstehen – beide Zugänge ergänzen sich gut: SoekiaGPT für die anschauliche Demonstration an größeren, echten Textsammlungen, Python für das Verständnis des zugrunde liegenden Algorithmus.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Grenzen der Handauszählung bei einem längeren Text sichtbar machen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Tabellarische Verwaltung von Kontexten, Folgewörtern und Häufigkeiten entwickeln.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	SoekiaGPT oder den vorbereiteten Python-Code gemeinsam erkunden und Ausgaben deuten.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Einen kleinen eigenen Korpus analysieren und auffällige Folgewörter notieren.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Ergebnisse verschiedener Korpora vergleichen und Datenabhängigkeit anbahnen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Wahrscheinlichkeitsfunktion in Python nur bei passendem Lerntempo vertiefen.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Auf den Kontext … folgt mit der Häufigkeit/Wahrscheinlichkeit … das Wort …“
- „Der erzeugte Text verändert sich, weil …“
- „Im Vergleich zum n-Gramm-Modell kann das LLM …, jedoch …“

📌 Merksätze

- Bei größeren Textkorpora ist eine **Auszählung von Hand** nicht mehr praktikabel – man nutzt Software, die die Häufigkeiten **automatisiert** ermittelt.
- Die Ergebnisse werden meist **tabellarisch** verwaltet: Zu jedem "Kontext" (z. B. einem Wort oder Wortpaar) wird eine Liste möglicher Folgewörter mit ihrer jeweiligen Häufigkeit/Wahrscheinlichkeit gespeichert.

- Eine gängige Datenstruktur dafür ist ein **Dictionary von Dictionaries**: `{kontext: {folgewort: anzahl}}`.

💡 Beispiel: Größerer Textkorpus automatisiert analysieren

```
korpus = """
der hund läuft im park der hund spielt mit dem ball
die katze sitzt auf dem baum die katze beobachtet den hund
der hund bellt laut der hund läuft zur katze
die katze springt vom baum die katze rennt weg
der hund läuft ihr hinterher der hund ist müde
"""

def erstelle_bigramm_tabelle(text):
    woerter = text.split()
    tabelle = {}
    for i in range(len(woerter) - 1):
        aktuelles_wort = woerter[i]
        naechstes_wort = woerter[i + 1]
        if aktuelles_wort not in tabelle:
            tabelle[aktuelles_wort] = {}
        tabelle[aktuelles_wort][naechstes_wort] =
tabelle[aktuelles_wort].get(naechstes_wort, 0) + 1
    return tabelle

tabelle = erstelle_bigramm_tabelle(korpus)

# Ausgabe für ein Beispielwort
print("Mögliche Folgewörter nach 'der':")
for wort, anzahl in tabelle["der"].items():
    print(f"  {wort}: {anzahl}x")
```

```
Mögliche Folgewörter nach 'der':
  hund: 6x
```

📝 Übung 1 – Eigene Analyse

Gebt für die Wörter `"die"` und `"hund"` jeweils aus, welche Folgewörter mit welcher Häufigkeit in der Tabelle vorkommen.

📝 Übung 2 – Wahrscheinlichkeiten berechnen

Schreibt eine Funktion `wahrscheinlichkeiten(tabelle, wort)`, die für ein gegebenes Wort ein Dictionary `{folgewort: wahrscheinlichkeit}` zurückgibt (Wahrscheinlichkeit = Anzahl ÷ Gesamtanzahl aller Folgewörter zu diesem Wort).

📝 Übung 3 – Eigenen Textkorpus analysieren

Sucht euch einen eigenen, etwas längeren Text (z. B. ein paar Sätze aus einem Buch, Songtext-Zusammenfassung in eigenen Worten, oder frei erfunden) und lasst ihn mit `erstelle_bigramm_tabelle` analysieren. Welches Wort hat die meisten unterschiedlichen Folgewörter?

DS 33 – Textgenerierung und Sampling-Strategien

Informatik 10 – MINT-Zweig | Künstliche Intelligenz

Lernziel

Ich benenne unterschiedliche Sampling-Strategien und wende sie zur Auswahl möglicher Folgewörter an; ich beschreibe die Auswirkungen unterschiedlicher n-Gramm-Längen auf den generierten Text.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Aus einer bekannten Tabelle das nächste Wort zunächst von Hand auswählen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Greedy, Random und Top-k mit Karten, Würfel oder Zufallszahlen fragend-entwickelnd vergleichen.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Einen kurzen Text gemeinsam erzeugen und anschließend die vorbereitete Simulation ausführen.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Den Korpus wechseln und Veränderungen des erzeugten Textes untersuchen.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Auswirkungen von Sampling, Trainingsdaten und n-Gramm-Länge erklären lassen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Top-k- oder Trigramm-Programmierung ausschließlich als optionale Vertiefung.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Auf den Kontext … folgt mit der Häufigkeit/Wahrscheinlichkeit … das Wort …“
- „Der erzeugte Text verändert sich, weil …“
- „Im Vergleich zum n-Gramm-Modell kann das LLM …, jedoch …“

📌 Merksätze

- **Textgenerierung (next word prediction):** Ausgehend von einem Startwort/-kontext wird mithilfe der n-Gramm-Tabelle immer wieder das **nächste Wort** bestimmt und an den Text angehängt.
- **Sampling-Strategien** legen fest, **wie** aus mehreren möglichen Folgewörtern eines ausgewählt wird:
  - **Greedy:** Es wird **immer** das **wahrscheinlichste** Folgewort gewählt → deterministisch, aber oft eintönig/sich wiederholend.
  - **Random:** Es wird **zufällig** unter allen möglichen Folgewörtern gewählt, **gewichtet nach ihrer Wahrscheinlichkeit** → abwechslungsreicher, aber manchmal unsinnig.
  - **Top-k:** Es werden nur die `k` wahrscheinlichsten Folgewörter betrachtet, unter denen dann zufällig (gewichtet) ausgewählt wird → Kompromiss zwischen Vorhersagbarkeit und Abwechslung.

- Die **Länge des n-Gramms** beeinflusst die Textqualität: Größere n (z. B. Trigramme statt Bigramme) erzeugen meist **kohärentere**, aber dafür **weniger variantenreiche** Texte, da der Kontext spezifischer und damit seltener im Korpus vorkommt.

✍️ Von Hand: Die ersten Schritte der Textgenerierung

Bevor ihr die Textgenerierung per Code laufen lasst: Nehmt eure Bigramm-Tabelle aus DS 31/32 (oder die Beispieltabelle mit "der hund läuft..." aus DS 32). Beginnt beim Wort **"der"** und wählt **von Hand** jeweils das **wahrscheinlichste** Folgewort (Greedy-Prinzip), bis ihr 5 Wörter notiert habt. Vergleicht euer von Hand erzeugtes Ergebnis anschließend mit der Greedy-Ausgabe der Code-Zelle unten – sie sollten übereinstimmen!

💡 Beispiel: Textgenerierung mit Greedy- und Random-Sampling

```python
import random

korpus = """
der hund läuft im park der hund spielt mit dem ball
die katze sitzt auf dem baum die katze beobachtet den hund
der hund bellt laut der hund läuft zur katze
die katze springt vom baum die katze rennt weg
der hund läuft ihr hinterher der hund ist müde
"""

def erstelle_bigramm_tabelle(text):
    woerter = text.split()
    tabelle = {}
    for i in range(len(woerter) - 1):
        a, b = woerter[i], woerter[i + 1]
        tabelle.setdefault(a, {})
        tabelle[a][b] = tabelle[a].get(b, 0) + 1
    return tabelle

tabelle = erstelle_bigramm_tabelle(korpus)

def naechstes_wort_greedy(tabelle, wort):
    folgewoerter = tabelle.get(wort)
    if not folgewoerter:
        return None
    return max(folgewoerter, key=folgewoerter.get)

def naechstes_wort_random(tabelle, wort):
    folgewoerter = tabelle.get(wort)
    if not folgewoerter:
        return None
    woerter = list(folgewoerter.keys())
    gewichte = list(folgewoerter.values())
    return random.choices(woerter, weights=gewichte, k=1)[0]

def generiere_text(tabelle, startwort, laenge, strategie):
    text = [startwort]
    aktuelles_wort = startwort
    for _ in range(laenge - 1):
        naechstes = strategie(tabelle, aktuelles_wort)
        if naechstes is None:
            break
        text.append(naechstes)
        aktuelles_wort = naechstes
    return " ".join(text)

print("Greedy:", generiere_text(tabelle, "der", 10, naechstes_wort_greedy))
print("Random:", generiere_text(tabelle, "der", 10, naechstes_wort_random))
print("Random:", generiere_text(tabelle, "der", 10, naechstes_wort_random))
```

```
Greedy: der hund läuft im park der hund läuft im park
Random: der hund spielt mit dem ball die katze die katze
Random: der hund bellt laut der hund läuft zur katze rennt
```

📝 Übung 1 – Beobachtung Greedy vs. Random

Führt die Zelle oben **mehrfach** aus. Was fällt euch beim Vergleich der Greedy- und der Random-Ausgabe auf? Warum liefert Greedy immer denselben Text, Random aber unterschiedliche?

(Eure Beobachtung hier)

🚀 Optionale Vertiefung A – Top-k-Sampling programmieren

Implementiert eine Funktion `naechstes_wort_top_k(tabelle, wort, k=2)`, die nur unter den `k` häufigsten Folgewörtern (gewichtet nach Häufigkeit) zufällig auswählt.

```
# OPTIONAL: naechstes_wort_top_k implementieren

def naechstes_wort_top_k(tabelle, wort, k=2):
    pass

# Test:
# print(generiere_text(tabelle, "der", 10, lambda t, w: naechstes_wort_top_k(t, w, k=2)))
```

🚀 Optionale Vertiefung B – Einfluss der n-Gramm-Länge

Baut analog zu DS 31/32 eine **Trigramm-Tabelle** auf (Kontext = Wortpaar statt Einzelwort) und passt die Textgenerierung entsprechend an. Vergleicht die generierten Texte: Wirken sie mit Trigrammen "sinnvoller" als mit Bigrammen? Warum bricht die Trigramm-Generierung bei kleinem Korpus schneller ab?

📝 Übung 2 – Abhängigkeit von den Trainingsdaten

Ersetzt im Beispielcode oben den `korpus` durch einen **inhaltlich ganz anderen** Text, z. B.:

```
korpus = """
heute scheint die sonne heute regnet es stark
morgen scheint die sonne morgen ist es bewölkt
heute ist es bewölkt heute regnet es leicht
"""
```

Führt die Textgenerierung mit `generiere_text(tabelle, "heute", 8, naechstes_wort_random)` erneut aus.

1. Wie stark unterscheidet sich der erzeugte Text vom Hund/Katze-Beispiel?
2. Was bedeutet das für die Aussage "Ein Sprachmodell erzeugt objektiven, neutralen Text"?
3. **Optional:** Probiert dieselbe Beobachtung mit SoekiaGPT aus, indem ihr dort zwischen zwei unterschiedlichen Dokumentensammlungen wechselt – das Prinzip ist identisch.

DS 34 – Halluzination, n-Gramm-Modelle vs. moderne LLMs & Berufsorientierung

Informatik 10 – MINT-Zweig | Künstliche Intelligenz

Lernziel

Ich erkläre das Phänomen der Halluzination anhand von n-Gramm-Modellen, beschreibe Unterschiede zwischen n-Gramm-Modellen und modernen LLMs und recherchiere Auswirkungen generativer KI-Systeme auf Berufe.

🕘 Stundenverlaufsplan (Doppelstunde, 90 Minuten)

Orientierungsrahmen für 11 SuS: Die ersten 80 Minuten bilden den Kern. Die letzten 10 Minuten sind bewusst flexibel und werden nicht mit neuem Pflichtstoff belegt.

Phase	Zeit	Inhalt/Aktion	Sozialform
Einstieg und Aktivierung	10'	Einen plausibel klingenden, aber falschen oder unsinnigen Modellsatz beurteilen.	Plenum; kurze Einzeldenkzeit
Fragend-entwickelnde Erarbeitung	25'	Halluzination am n-Gramm-Modell erklären und die Grenzen der Analogie zu modernen LLMs klären.	Unterrichtsgespräch; Beiträge der SuS werden sichtbar festgehalten
Gemeinsame Umsetzung und Sicherung	20'	Beobachtbare Unterschiede zwischen n-Gramm-Modell und LLM gemeinsam tabellarisch sichern.	Plenum; Live-Coding, Tafelbild oder gemeinsame Simulation
Kurzer Transfer	15'	Auswirkungen generativer KI auf ein Berufsfeld anhand der Leitfragen recherchieren.	Einzelarbeit mit anschließendem Tandemabgleich (4 Tandems + 1 Trio)
Auswertung	10'	Ergebnisse kurz vorstellen und Aussagen mit konkreten Beispielen begründen.	Plenum; Lösung erklären und Fachsprache sichern
Reserve / Vertiefung	10'	Berufsbiografie, Chancen-Risiken-Abwägung oder Abschlussreflexion vertiefen.	Flexibel; bei Bedarf für Fragen, Fehlersuche oder Abschluss nutzen

🗣️ Sprachhilfe

- „Auf den Kontext … folgt mit der Häufigkeit/Wahrscheinlichkeit … das Wort …“
- „Der erzeugte Text verändert sich, weil …“
- „Im Vergleich zum n-Gramm-Modell kann das LLM …, jedoch …“

📌 Merksätze

- **Halluzination** bezeichnet das Phänomen, dass ein Sprachmodell Text erzeugt, der **grammatikalisch bzw. statistisch plausibel**, aber **inhaltlich falsch oder unsinnig** ist.
- Bei n-Gramm-Modellen entsteht Halluzination, weil das Modell **keinerlei "Verständnis"** der Bedeutung hat – es reiht Wörter nur nach **statistischer Wahrscheinlichkeit** aneinander, unabhängig davon, ob der entstehende Satz sachlich korrekt ist.
- Moderne **LLMs** unterscheiden sich von einfachen n-Gramm-Modellen u. a. in:
  - **Sprachqualität:** LLMs erzeugen deutlich flüssigeren, grammatikalisch korrekteren Text.
  - **Kontextberücksichtigung:** LLMs können sehr viel längere Kontexte einbeziehen (nicht nur 1–2 vorherige Wörter).

- **Korrektheit:** LLMs erzeugen häufig zuverlässigere Antworten, können aber weiterhin halluzinieren. Das n-Gramm-Modell dient hierfür als **vereinfachte Analogie**; die Ursachen bei modernen LLMs sind wesentlich komplexer und nicht mit dem einfachen Modell gleichzusetzen.
- **Vielseitigkeit:** LLMs können unterschiedlichste Aufgaben (Übersetzen, Zusammenfassen, Programmieren …) bewältigen, während ein einfaches n-Gramm-Modell nur Text im Stil seines Trainingskorpus fortsetzt.

💡 Beispiel: Halluzination provozieren

Mit einem sehr kleinen, unpassenden Trainingskorpus erzeugt unser n-Gramm-Modell aus DS 33 leicht unsinnige, aber grammatikalisch klingende Sätze – das ist eine vereinfachte Analogie zur Halluzination bei LLMs.

```python
import random

korpus = """
der hund läuft im park der hund spielt mit dem ball
die katze sitzt auf dem baum die katze beobachtet den hund
der hund bellt laut der hund läuft zur katze
die katze springt vom baum die katze rennt weg
der hund läuft ihr hinterher der hund ist müde
"""

def erstelle_bigramm_tabelle(text):
    woerter = text.split()
    tabelle = {}
    for i in range(len(woerter) - 1):
        a, b = woerter[i], woerter[i + 1]
        tabelle.setdefault(a, {})
        tabelle[a][b] = tabelle[a].get(b, 0) + 1
    return tabelle

def naechstes_wort_random(tabelle, wort):
    folgewoerter = tabelle.get(wort)
    if not folgewoerter:
        return None
    woerter = list(folgewoerter.keys())
    gewichte = list(folgewoerter.values())
    return random.choices(woerter, weights=gewichte, k=1)[0]

def generiere_text(tabelle, startwort, laenge, strategie):
    text = [startwort]
    aktuelles_wort = startwort
    for _ in range(laenge - 1):
        naechstes = strategie(tabelle, aktuelles_wort)
        if naechstes is None:
            break
        text.append(naechstes)
        aktuelles_wort = naechstes
    return " ".join(text)

tabelle = erstelle_bigramm_tabelle(korpus)
for _ in range(5):
    print(generiere_text(tabelle, "die", 8, naechstes_wort_random))
```

```
die katze die katze rennt weg der hund
die katze springt vom baum die katze die
die katze springt vom baum die katze springt
die katze beobachtet den hund ist müde
die katze die katze beobachtet den hund spielt
```

📝 Übung 1 – Halluzinationen einordnen

Führt die Zelle oben mehrfach aus und sucht einen erzeugten Satz heraus, der **grammatikalisch plausibel, aber inhaltlich seltsam oder unsinnig** ist. Erklärt in eigenen Worten, **warum** genau dieser Fehler entsteht (Bezug zur Funktionsweise des n-Gramm-Modells).

(Euer Beispielsatz und eure Erklärung hier)

📝 Übung 2 – Vergleichstabelle

Erstellt eine Tabelle, die n-Gramm-Modelle und moderne LLMs bezüglich Sprachqualität, Korrektheit, Kontextberücksichtigung und Vielseitigkeit gegenüberstellt (Doppelklick zum Bearbeiten):

Kriterium	n-Gramm-Modell	Modernes LLM
Sprachqualität		
Korrektheit		
Kontextberücksichtigung		
Vielseitigkeit		

🧭 Berufliche Orientierung: Auswirkungen generativer KI auf Berufe

Recherchiert in Partner- oder Kleingruppenarbeit (ca. 20–25 Minuten) die Auswirkungen generativer KI-Systeme auf **ein** Berufsfeld eurer Wahl, z. B.:

- Journalismus / Content-Erstellung
- Softwareentwicklung
- Grafikdesign / Illustration
- Kundenservice / Support
- Übersetzung

Leitfragen für eure Recherche

1. Welche Tätigkeiten dieses Berufsfelds werden bereits heute durch generative KI unterstützt oder verändert?
2. Entstehen durch generative KI **neue** Aufgaben oder sogar neue Berufsbilder in diesem Bereich (z. B. "Prompt Engineering")?
3. Welche Tätigkeiten lassen sich (aus heutiger Sicht) **nicht** oder nur schwer durch KI ersetzen – und warum?
4. Wie bewertet ihr diese Entwicklung – welche Chancen und welche Risiken seht ihr?

Haltet eure Ergebnisse in Stichpunkten fest und bereitet einen kurzen mündlichen Beitrag (ca. 1 Minute) für das Plenum vor. 5. Findet (falls möglich) eine **reale Person** mit Bezug zu diesem Berufsfeld und recherchiert ihre Berufsbiografie (Werdegang: Ausbildung, erste Stelle, heutige Position) – z. B. über eine Unternehmens-Website, ein Interview oder ein Karriereportal.

(Platz für eure Rechercheergebnisse)

Berufsfeld:

Bereits veränderte/unterstützte Tätigkeiten:

Neue Aufgaben/Berufsbilder:

(Noch) nicht ersetzbare Tätigkeiten:

Chancen und Risiken – unsere Einschätzung: