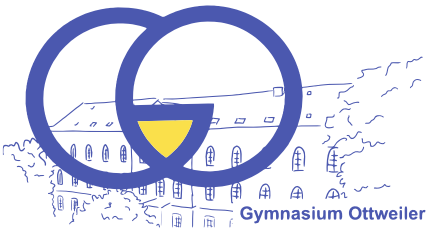
Kapitel 1

Programmieren mit Python



Skript Informatik · Klasse 10 · MINT

In diesem Kapitel lernst du, Programme in der Programmiersprache **Python** zu schreiben. Aus der Blockprogrammierung kennst du schon Bausteine wie Wiederholungen und Bedingungen. Jetzt schreibst du dieselben Ideen als Text. Das ist am Anfang ungewohnt: Ein fehlendes Komma oder eine falsche Einrückung genügt, und das Programm läuft nicht. Dieses Skript zeigt dir deshalb jeden Schritt einzeln, erklärt jedes Beispiel Zeile für Zeile und zeigt dir die typischen Fehler, bevor sie dir passieren.

So arbeitest du mit diesem Skript

Das Skript ist zum **selbstständigen Lernen** gedacht. Du kannst damit jede Unterrichtsstunde vorbereiten, nacharbeiten oder nachholen. Am besten arbeitest du so:

1. **Lies einen Abschnitt in Ruhe.** Die Abschnitte bauen aufeinander auf. Überspringe nichts, auch wenn es leicht aussieht.
2. **Tippe die Beispiele selbst ab** und starte sie. Kopieren ist bequemer, aber beim Abtippen lernst du die Schreibweise. Vergleiche dein Ergebnis mit der abgedruckten Ausgabe.
3. **Bearbeite die Arbeitsaufträge.** Programme schreibst und testest du direkt am Computer. Wo du etwas überlegen, vorhersagen oder begründen sollst, ist unter dem Auftrag Platz zum Schreiben.
4. **Kontrolliere mit dem Lösungsheft** – aber erst, wenn du es selbst versucht hast. Oft gibt es mehrere richtige Lösungen. Wichtig ist, dass dein Programm das Richtige tut.
5. **Hake am Ende den Selbstcheck ab.** Wenn du einen Punkt nicht abhaken kannst, lies den passenden Teil noch einmal.

Die Kästen im Skript haben feste Farben. So weißt du sofort, was dich erwartet:

Kasten	Farbe	Was darin steht
Merke	rot	das Wichtigste, das du dir merken musst
Definition	grün	was ein Fachbegriff genau bedeutet
Leitidee, Lernziele, Beispiel	gelb	Einstieg, Ziele und ausführliche Beispiele
Stolperstein, Werkzeug, Tipp	gelb	typische Fehler, kleine Hilfsmittel, Hinweise
So kannst du es sagen	gelb	Mustersätze, um Programme in Fachsprache zu beschreiben
Versuch, Arbeitsauftrag	blau	hier bist du dran: ausprobieren oder eine Aufgabe lösen
Auswertung, Beurteilung	lila	Ergebnisse zusammentragen und bewerten

Programme stehen in grauen Kästen. Die Farben im Code helfen beim Lesen, genau wie in vielen Programmierumgebungen. Was ein Programm ausgibt, steht in einem weißen Kasten mit der Aufschrift **AUSGABE**. Was du selbst eintippst, ist darin **blau und fett** gedruckt. Fehlermeldungen stehen in einem ebensolchen Kasten mit der Aufschrift **FEHLERMELDUNG**.

So kannst du es sagen

Mit diesen Satzanfängen kannst du in jedem Abschnitt beschreiben und erklären, was ein Programm tut:

- *„Für die Eingabe … liefert das Programm …, weil …“*
- *„Die Bedingung/Schleife/Funktion wird ausgeführt, wenn …“*
- *„Ich ändere …, damit …“*

Tipp

Es ist egal, ob du im Jupyter-Notebook aus dem Unterricht programmierst oder in einem Editor wie Thonny. Die Programme in diesem Skript laufen überall, wo Python 3 installiert ist. Wie du Python zu Hause einrichtest, steht im Anhang.

1.1 Variablen, Eingabe und Ausgabe

Leitidee

Ein Programm ist eine Liste von Anweisungen, die der Computer der Reihe nach ausführt – von oben nach unten. In der Blockprogrammierung hast du die Anweisungen als Bausteine zusammengeschoben. In Python schreibst du sie als Text. In diesem Abschnitt lernst du die drei wichtigsten Dinge, die fast jedes Programm braucht: Es **gibt etwas aus**, es **merkt sich Werte** und es **liest Eingaben** der Nutzerin oder des Nutzers ein. Am Ende schreibst du ein Programm, das dich nach deinem Namen fragt und dir einen Steckbrief ausgibt.

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.1.1 Das erste Programm: Ausgaben mit `print`

Das kürzeste sinnvolle Programm besteht aus einer einzigen Zeile:

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

AUSGABE

```
Hallo Welt!
```

Das Wort `print` ist englisch und heißt „drucken“. Gemeint ist: Zeige etwas auf dem Bildschirm an. Was angezeigt werden soll, steht in den **runden Klammern** dahinter. Der Text steht in **Anführungszeichen**. Die Anführungszeichen selbst erscheinen nicht in der Ausgabe. Sie zeigen Python nur, wo der Text anfängt und wo er aufhört.

Ein Programm hat meistens mehrere Zeilen. Python führt sie **nacheinander von oben nach unten** aus. Jede `print`-Anweisung beginnt eine neue Zeile in der Ausgabe:

```
1 print("Guten Morgen!")
2 print("Heute lernen wir Python.")
3 print("Los geht's.")
```

AUSGABE

```
Guten Morgen!
Heute lernen wir Python.
Los geht's.
```

Text oder Zahl?

Mit `print` kannst du auch Zahlen ausgeben und sogar rechnen. Vergleiche die beiden Zeilen genau:

```
print(3 + 4)
print("3 + 4")
```

AUSGABE

```
7
3 + 4
```

- **Zeile 1:** Hier stehen keine Anführungszeichen. Python liest `3 + 4` als Rechnung, rechnet sie aus und gibt das Ergebnis `7` aus.
- **Zeile 2:** Jetzt steht `3 + 4` in Anführungszeichen. Für Python ist das ein Text wie jeder andere. Er wird genau so ausgegeben, wie er dasteht.

Du kannst in eine `print`-Anweisung auch mehrere Dinge schreiben. Trenne sie durch **Kommas**. Python setzt in der Ausgabe automatisch je ein Leerzeichen dazwischen:

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

AUSGABE

```
Ich bin 15 Jahre alt.
In fünf Jahren bin ich 20 Jahre alt.
```

Merke

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

Versuch: Probiere es aus

Tippe das Programm mit „Guten Morgen!“ ab und starte es. Ändere dann die Texte. Was passiert, wenn du eine der drei Zeilen an den Anfang verschiebst?

Wenn etwas schiefgeht

Beim Abtippen passieren Fehler – auch Profis jeden Tag. Python bricht dann ab und zeigt eine **Fehlermeldung**. Das ist kein Grund zur Sorge: Die Meldung ist ein Hinweis, wo das Problem liegt. Hier fehlt das schließende Anführungszeichen:

```
print("Hallo)
```

FEHLERMELDUNG

```
SyntaxError: unterminated string literal (detected at line 1)
```

Die Fehlermeldung ist englisch. **SyntaxError** bedeutet: Die Schreibweise stimmt nicht – Python versteht den Satz nicht. *Unterminated string literal* heißt „nicht abgeschlossener Text“. Python sagt dir sogar die Zeile: *detected at line 1*. Die Lösung: das fehlende `"` ergänzen.

Und was passiert, wenn du die Anführungszeichen ganz vergisst?

```
print(Hallo)
```

FEHLERMELDUNG

```
NameError: name 'Hallo' is not defined
```

Ohne Anführungszeichen hält Python `Hallo` nicht für einen Text, sondern für einen **Namen** – so wie die Variablen, die du gleich kennenlernst. Einen Namen `Hallo` gibt es aber nicht. Deshalb meldet Python einen **NameError**: „Der Name ‘Hallo’ ist nicht definiert.“

Stolperstein

Die häufigsten Tippfehler am Anfang: ein vergessenes Anführungszeichen, eine vergessene Klammer, `Print` statt `print` (Python unterscheidet Groß- und Kleinschreibung!) und deutsche Anführungszeichen „…“ statt `"…"`. Wenn eine Fehlermeldung kommt, schau dir zuerst die genannte Zeile an – und die Zeile davor.

1.1.2 Variablen: Werte speichern

Programme müssen sich Dinge merken: einen Namen, einen Punktestand, einen Preis. Dafür gibt es **Variablen**.

Stell dir eine Variable wie eine **beschriftete Schachtel** vor. Außen auf der Schachtel steht ein Name, zum Beispiel `alter`. In der Schachtel liegt ein Wert, zum Beispiel `15`. Immer wenn das Programm den Namen `alter` liest, schaut es in die Schachtel und nimmt den Wert, der gerade darin liegt.



Drei Variablen: Außen steht der Name, innen liegt der Wert. Darunter steht, welche Art von Wert es ist (der Datentyp)

Definition: Variable

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

In Python legst du eine Variable an, indem du ihr einen Wert gibst:

```
1 alter = 15
2 print(alter)
3 print("Ich bin", alter, "Jahre alt.")
```

AUSGABE

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

- **Zeile 1:** Die Variable `alter` wird angelegt und bekommt den Wert `15`. Auf dem Bildschirm passiert dabei nichts.
- **Zeile 2:** `alter` steht ohne Anführungszeichen. Python schaut deshalb in die Schachtel `alter` und gibt ihren Inhalt aus: `15`.
- **Zeile 3:** Text und Variable werden durch Kommas getrennt ausgegeben.

Das Gleichheitszeichen ist ein Zuweisungszeichen

Das Zeichen = bedeutet in Python **nicht** „ist gleich“ wie in Mathe. Es bedeutet: **„bekommt den Wert“**. Lies `alter = 15` also so: „Die Variable `alter` bekommt den Wert 15.“ Man nennt das eine **Zuweisung**. Dabei wird immer zuerst die rechte Seite ausgerechnet, dann wird das Ergebnis in die Variable links gelegt.

Deshalb ist die folgende Zeile in Python sinnvoll, obwohl sie in Mathe falsch wäre:

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

AUSGABE

```
15
16
```

- **Zeile 1:** `alter` bekommt den Wert `15`.
- **Zeile 3:** Zuerst wird die rechte Seite ausgerechnet: Der aktuelle Wert von `alter` ist `15`, plus `1` ergibt `16`. Dann bekommt `alter` diesen neuen Wert. Der alte Wert `15` ist weg – er wurde **überschrieben**.

Wie sich die Werte Schritt für Schritt ändern, zeigt ein **Ablaufprotokoll**. Darin steht nach jeder ausgeführten Zeile, welchen Wert die Variable gerade hat:

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

Gymnasium Ottweiler

Merke

- Eine **Variable** ist ein benannter Speicherplatz im Arbeitsspeicher, in dem ein Wert abgelegt wird.
- 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

Den Namen einer Variablen darfst du selbst wählen. Dabei gelten ein paar Regeln:

- Erlaubt sind Buchstaben, Ziffern und der Unterstrich _. Ein Name darf **nicht mit einer Ziffer beginnen** und **keine Leerzeichen** enthalten.
- Groß- und Kleinschreibung zählt: `alter` und `Alter` sind zwei verschiedene Namen.
- Verwende **sprechende Namen**, an denen man erkennt, was gespeichert ist: `preis` statt `p`, `anzahl_artikel` statt `aa`. Mehrere Wörter verbindest du mit _.
- Verzichte auf Umlaute und ß in Namen: `groesse` statt `größe`. Das funktioniert zwar meist, kann aber Probleme machen.

Name	erlaubt?	Grund
`punkte`	✓	
`punkte_runde_2`	✓	
`2_runde`	✗	beginnt mit einer Ziffer
`meine punkte`	✗	enthält ein Leerzeichen

Stolperstein

Wenn du eine Variable benutzt, bevor sie einen Wert bekommen hat, oder dich beim Namen vertippst, meldet Python einen **NameError**:

```
alter = 15
print(Alter)
```

FEHLERMELDUNG

```
NameError: name 'Alter' is not defined. Did you mean: 'alter'?
```

Neuere Python-Versionen geben dir sogar einen Tipp: *Did you mean: 'alter'?* – „Meintest du 'alter'?"

Arbeitsauftrag 1: Vorhersage

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

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

Meine Vorhersage:

1.1.3 Datentypen: Zahlen, Texte und Wahrheitswerte

In einer Variablen können ganz unterschiedliche Arten von Werten liegen: ganze Zahlen, Kommazahlen, Texte. Die Art eines Wertes heißt **Datentyp**. Python merkt sich zu jedem Wert seinen Datentyp, denn davon hängt ab, was man mit dem Wert machen kann. Mit Zahlen kann man rechnen, mit Texten nicht.

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

Definition: Datentyp

Der **Datentyp** eines Wertes legt fest, welche Art von Daten er darstellt (z. B. ganze Zahl, Kommazahl, Text oder Wahrheitswert) und welche Operationen damit möglich sind.

Drei Dinge sind besonders wichtig:

- Kommazahlen schreibt man in Python mit einem **Punkt**, nicht mit einem Komma: `1.68`, nicht `1,68`.
- Alles in Anführungszeichen ist ein Text, auch wenn Ziffern darin stehen: `"15"` ist eine Zeichenkette, `15` eine Zahl.
- `True` und `False` schreibt man mit großem Anfangsbuchstaben und ohne Anführungszeichen. Wahrheitswerte brauchst du später für Bedingungen.

Mit `type(...)` kannst du Python fragen, welchen Datentyp ein Wert hat:

```
1 name = "Alex"
2 alter = 15
3 groesse = 1.68
4 volljaehrig = False
5 print(type(name))
6 print(type(alter))
7 print(type(groesse))
8 print(type(volljaehrig))
```

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

Die Ausgabe `<class 'str'>` bedeutet einfach: „Dieser Wert ist vom Typ `str`."

Merke

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

Rechnen mit Zahlen

Mit `int`- und `float`-Werten kannst du rechnen. Die Grundrechenarten schreibt man mit `+`, `-`, `*` (mal) und `/` (geteilt). Es gilt wie in Mathe **Punkt vor Strich**, und Klammern werden zuerst ausgerechnet:

```
1 a = 7
2 b = 2
3 print(a + b)
4 print(a - b)
5 print(a * b)
6 print(a / b)
7 print(2 + 3 * 4)
8 print((2 + 3) * 4)
```

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

Beachte Zeile 6: Das Ergebnis einer Division mit `/` ist **immer** ein `float`, auch wenn es aufgeht. `6 / 2` ergibt `3.0`, nicht `3`. Weitere Rechenzeichen lernst du im nächsten Abschnitt kennen.

Stolperstein

Schreibst du eine Kommazahl versehentlich mit Komma, gibt es **keine** Fehlermeldung – Python versteht aber etwas völlig anderes:

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

AUSGABE
```
(2, 50)
```

Python hat zwei Zahlen gespeichert, `2` und `50`. Mit `preis` kann man jetzt nicht mehr rechnen. Richtig ist `preis = 2.50`.

Arbeitsauftrag 2: Datentypen bestimmen

Gib zu jedem Wert den Datentyp an (`int`, `float`, `str` oder `bool`). Überprüfe deine Antworten anschließend mit `type(...)`.

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

1.1.4 Eingaben mit `input`

Bisher stehen alle Werte fest im Programm. Spannender wird es, wenn die Nutzerin oder der Nutzer etwas eingeben kann. Dafür gibt es `input(...)`:

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

AUSGABE

```
Wie heißt du? Lena
Hallo Lena
```

- **Zeile 1:** `input` zeigt zuerst den Text in den Klammern an, die sogenannte **Eingabeaufforderung**. Dann **wartet das Programm**, bis jemand etwas eintippt und die Eingabetaste drückt. Der eingetippte Text (hier blau gedruckt) wird in der Variablen `name` gespeichert.
- **Zeile 2:** Der gespeicherte Name wird ausgegeben.

Tipp

Setze ans Ende der Eingabeaufforderung ein Leerzeichen: `"Wie heißt du? "`. Sonst klebt die Eingabe direkt am Fragezeichen.

Die Eingabe ist immer ein Text

Probiere einmal dieses Programm mit der Eingabe `7`. Was erwartest du?

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

AUSGABE

```
Gib eine Zahl ein: 7
77
<class 'str'>
```

Überraschung: Python rechnet nicht `7 + 7 = 14`, sondern gibt `77` aus. Der Grund steht in der letzten Zeile: `zahl` hat den Typ `str`. Das, was jemand eintippt, ist für Python **immer eine Zeichenkette**, auch wenn nur Ziffern darin stehen. Und das Pluszeichen hängt zwei Zeichenketten einfach aneinander: `"7" + "7"` ergibt `"77"`.

Merke

- Mit `input(...)` liest man eine Eingabe von der Nutzerin oder dem Nutzer ein. **Die Eingabe ist immer vom Typ `str`!**

1.1.5 Typumwandlung: aus Text wird eine Zahl

Damit du mit einer Eingabe rechnen kannst, musst du sie in eine Zahl **umwandeln**. Dafür gibt es drei Befehle, die genauso heißen wie die Datentypen:

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

Definition: Typumwandlung

Bei einer **Typumwandlung** wird ein Wert in einen anderen Datentyp umgewandelt, zum Beispiel die Zeichenkette `"7"` in die ganze Zahl `7`.

Das folgende Programm zeigt alles zusammen. Es fragt nach einer Lieblingszahl, zeigt, dass die Eingabe ein Text ist, und rechnet nach der Umwandlung mit ihr:

```
1 lieblingszahl = input("Gib deine Lieblingszahl ein: ")
2 print("Deine Eingabe war:", lieblingszahl, "- Typ:", type(lieblingszahl))
3
4 lieblingszahl_int = int(lieblingszahl)
5 print("Deine Zahl mal 2 ist:", lieblingszahl_int * 2)
```

AUSGABE

```
Gib deine Lieblingszahl ein: 7
Deine Eingabe war: 7 - Typ: <class 'str'>
Deine Zahl mal 2 ist: 14
```

- **Zeile 1:** Die Eingabe wird als Zeichenkette in `lieblingszahl` gespeichert.
- **Zeile 2:** Die Ausgabe bestätigt: Der Typ ist `str`.
- **Zeile 3:** Eine leere Zeile. Sie hat keine Wirkung, macht das Programm aber übersichtlicher.
- **Zeile 4:** `int(lieblingszahl)` macht aus dem Text `"7"` die Zahl `7`. Das Ergebnis wird in einer neuen Variablen gespeichert.
- **Zeile 5:** Jetzt wird wirklich gerechnet: `7 * 2 = 14`.

Eingabe und Umwandlung in einer Zeile

Meistens schreibt man Eingabe und Umwandlung in **eine** Zeile. Das ist kürzer, und man braucht keine zweite Variable:

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

AUSGABE

```
Wie alt bist du? 15
Nächstes Jahr bist du 16
```

Lies Zeile 1 **von innen nach außen**: Zuerst wird `input(...)` ausgeführt und liefert den Text `"15"`. Dieser Text wird an `int(...)` übergeben und in die Zahl `15` umgewandelt. Erst diese Zahl wird in `alter` gespeichert. Achte darauf, dass am Ende **zwei** schließende Klammern stehen – eine für `input`, eine für `int`.

Für Kommazahlen nimmst du `float` statt `int`:

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

AUSGABE

```
Wie groß bist du in Metern? 1.68
In Zentimetern sind das 168.0
```

Merke

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

Stolperstein

Die Umwandlung klappt nur, wenn der Text wirklich wie eine Zahl aussieht. Sonst meldet Python einen **ValueError** („falscher Wert“):

```
alter = int(input("Wie alt bist du? "))
```

FEHLERMELDUNG

```
ValueError: invalid literal for int() with base 10: 'sieben'
```

Dasselbe passiert bei `int("2.5")` (eine Kommazahl ist keine ganze Zahl) und bei `float("2,5")` (Komma statt Punkt). Bei einem ValueError lohnt sich also ein Blick darauf, **was eingegeben wurde**.

1.1.6 Texte zusammensetzen

Mit Kommas in `print` setzt Python immer ein Leerzeichen zwischen die Teile. Manchmal willst du das nicht, zum Beispiel wenn direkt nach einem Namen ein Komma oder ein Punkt folgen soll. Dann kannst du Texte mit `+` **aneinanderhängen** (verketten). Dabei kommt kein Leerzeichen dazu – die brauchst du selbst in den Anführungszeichen:

```
1 name = "Alex"
2 print("Hallo, " + name + "!")
3 print("Hallo,", name, "!")
```

AUSGABE

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

In Zeile 3 stört das Leerzeichen vor dem Ausrufezeichen. Mit `+` (Zeile 2) sitzt alles genau richtig.

Stolperstein

Mit `+` lassen sich nur **Texte mit Texten** verbinden. Eine Zahl muss vorher mit `str(...)` in einen Text umgewandelt werden. Sonst bricht das Programm ab:

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

FEHLERMELDUNG

```
Traceback (most recent call last):
  File "steckbrief.py", line 2, in <module>
    print("Du bist " + alter + " Jahre alt.")
          ~~~~~~~~~~^~~~~~~
TypeError: can only concatenate str (not "int") to str
```

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

Eine Fehlermeldung lesen

Die Meldung oben ist eine vollständige Fehlermeldung, wie du sie auf dem Bildschirm siehst. Lies sie **von unten nach oben**:

1. **Letzte Zeile:** die Art des Fehlers und eine kurze Beschreibung. Hier: `TypeError` – ein Fehler mit den Datentypen. *can only concatenate str (not "int") to str* heißt: „Kann nur `str` (nicht `int`) an `str` anhängen."
2. **Die Zeilen darüber** zeigen, wo der Fehler passiert ist: in Zeile 2 (*line 2*). Darunter steht diese Zeile noch einmal, und die Zeichen ~~~^~~~ zeigen auf die Stelle, an der der Fehler auftritt.
3. Die erste Zeile *Traceback (most recent call last)* kannst du überlesen.

Die vier Fehlerarten, die dir am Anfang am häufigsten begegnen:

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

So kannst du es sagen

So beschreibst du ein Programm in Fachsprache – vom einfachen zum genauen Satz:

- *„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."*

1.1.7 Übungen

Arbeitsauftrag 3: Fehler finden

Das folgende Programm soll nach zwei Zahlen fragen und ihre Summe ausgeben. Es enthält aber **drei Fehler**. Finde sie, erkläre jeweils, welche Fehlermeldung oder welches falsche Verhalten sie auslösen, und schreibe das Programm richtig auf.

```
1 zahl1 = input("Erste Zahl: )
2 zahl2 = input("Zweite Zahl: ")
3 print("Die Summe ist", Zahl1 + zahl2)
```

Die Fehler:

Arbeitsauftrag 4: Steckbrief

Schreibe ein Programm, das folgende Angaben mit `input()` abfragt und anschließend als vollständigen Satz ausgibt:

- Name
- Alter (als Zahl!)
- Lieblingsfach

Gib zum Schluss in einer zweiten Zeile das **doppelte Alter** aus. Daran siehst du, ob das Alter wirklich als Zahl gespeichert ist.

So könnte die Ausgabe aussehen:

AUSGABE

```
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
```

Arbeitsauftrag 5: Einfacher Rechner

Schreibe ein Programm, das zwei Zahlen einliest (denk an die Typumwandlung!) und **Summe, Differenz und Produkt** der beiden Zahlen ausgibt.

Werkzeug

Mit `round(wert, stellen)` rundest du eine Zahl auf eine bestimmte Anzahl von Nachkommastellen. `round(3.14159, 2)` ergibt `3.14`. Lässt du die Stellen weg, wird auf eine ganze Zahl gerundet: `round(2.7)` ergibt `3`.

Arbeitsauftrag 6 (Zusatz): Gesamtpreis

Frage einen Preis in Euro (als Kommazahl, `float`) sowie eine Anzahl an Artikeln (als `int`) ab und berechne den Gesamtpreis. Runde die Ausgabe sinnvoll (Tipp: `round(wert, 2)`).

1.1.8 Weitere Übungen

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.

Arbeitsauftrag 7: Mit oder ohne Anführungszeichen? ●○○

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

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

Meine Vorhersage:

Gymnasium Ottweiler

Arbeitsauftrag 8: 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(...))`.

```
1 a = input("Gib eine Zahl ein: ")
2 b = int(input("Gib eine Zahl ein: "))
3 c = float(input("Gib eine Zahl ein: "))
4 d = "3.5"
5 e = 3.5
6 f = False
```

Variable	a	b	c	d	e	f
Datentyp						

Arbeitsauftrag 9: 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.

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

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

Ausgabe:

Arbeitsauftrag 10: 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`.

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

Arbeitsauftrag 11: 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**.

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

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).

Die Fehler:

Arbeitsauftrag 12: 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.
```

Arbeitsauftrag 13: 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.

Begründung:

Arbeitsauftrag 14: 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.

Testergebnisse und Erklärung:

Gymnasium Ottweiler

1.1.9 Das Wichtigste in Kürze

- Ein Programm ist eine Folge von Anweisungen, die von oben nach unten ausgeführt werden.
- `print(...)` gibt Werte aus. Mehrere Werte werden durch Kommas getrennt und mit Leerzeichen ausgegeben.
- Eine **Variable** ist ein benannter Speicherplatz. Mit = bekommt sie einen Wert (**Zuweisung**); ein neuer Wert überschreibt den alten.
- Wichtige **Datentypen**: `int` (ganze Zahl), `float` (Kommazahl mit Punkt), `str` (Zeichenkette) und `bool` (`True`/`False`).
- `input(...)` liest eine Eingabe ein. Die Eingabe ist **immer** vom Typ `str`.
- Mit `int(...)`, `float(...)` und `str(...)` wandelt man Werte in einen anderen Datentyp um (**Typumwandlung**).
- Fehlermeldungen liest man von unten nach oben: Fehlerart, Beschreibung, dann die Zeilennummer.

Selbstcheck

- ☐ Ich kann Texte und Zahlen mit `print` ausgeben.
- ☐ Ich kann erklären, was eine Variable ist und was bei `x = x + 1` passiert.
- ☐ Ich kenne die Datentypen `int`, `float`, `str` und `bool` und kann zu einem Wert den Typ angeben.
- ☐ Ich kann mit `input` eine Eingabe einlesen und weiß, dass sie immer eine Zeichenkette ist.
- ☐ Ich kann eine Eingabe mit `int` oder `float` in eine Zahl umwandeln.
- ☐ Ich kann eine Fehlermeldung lesen und weiß, was `SyntaxError`, `NameError`, `TypeError` und `ValueError` bedeuten.

1.2 Operatoren: rechnen, vergleichen, verknüpfen

Leitidee

Ein Film dauert 135 Minuten. Wie viele Stunden und Minuten sind das? Ist eine Zahl gerade oder ungerade? Darf eine 15-Jährige ohne Ausweis ins Kino? Für solche Fragen muss ein Programm **rechnen**, **vergleichen** und mehrere Aussagen **verknüpfen**. Die Zeichen, mit denen man das in Python aufschreibt, heißen **Operatoren**. Einige kennst du schon: `+`, `-`, `*` und `/`. In diesem Abschnitt kommen neue Rechenzeichen dazu und vor allem Operatoren, deren Ergebnis `True` oder `False` ist. Mit diesen Wahrheitswerten kann ein Programm im nächsten Abschnitt Entscheidungen treffen.

Lernziele

Nach diesem Abschnitt kannst du

- 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 den Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=` und `>=` vergleichen,
- Wahrheitswerte mit `and`, `or` und `not` verknüpfen und das Ergebnis vorhersagen,
- die Zuweisung `=` und den Vergleich `==` sicher unterscheiden.

1.2.1 Was ist ein Operator?

In der Rechnung `7 + 3` steht das Pluszeichen zwischen zwei Zahlen. Es sagt Python, **was** mit den beiden Zahlen geschehen soll: Sie werden addiert. Ein solches Zeichen heißt **Operator**. Die Werte, mit denen gerechnet wird, heißen **Operanden**. Das Ergebnis ist wieder ein Wert, hier `10`. Eine solche Verbindung aus Werten, Variablen und Operatoren, die Python zu einem Wert ausrechnet, nennt man einen **Ausdruck**.

Definition: Operator

Ein **Operator** ist ein Zeichen oder ein Wort, das eine Operation mit einem oder mehreren Werten (den **Operanden**) ausführt und daraus einen neuen Wert berechnet. Beispiel: In `7 + 3` ist `+` der Operator, `7` und `3` sind die Operanden.

In diesem Abschnitt lernst du drei Gruppen von Operatoren kennen: **arithmetische Operatoren** zum Rechnen (Ergebnis ist eine Zahl), **Vergleichsoperatoren** und **logische Operatoren** (Ergebnis ist jeweils ein Wahrheitswert, also `True` oder `False`).

1.2.2 Arithmetische Operatoren: mehr als die Grundrechenarten

Die vier Grundrechenarten `+`, `-`, `*` und `/` kennst du schon aus dem Abschnitt über Variablen. Python hat drei weitere Rechenoperatoren. Hier sind alle sieben auf einen Blick, jeweils mit den Zahlen 7 und 3:

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`

Die drei neuen Operatoren schauen wir uns jetzt genauer an.

Ganzzahldivision `//` und Rest `%`

Erinnere dich an die Grundschule: Dort hast du „mit Rest“ geteilt. 17 Kekse werden gerecht an 5 Kinder verteilt. Jedes Kind bekommt 3 Kekse, und 2 Kekse bleiben übrig. Man schreibt: 17 : 5 = 3 Rest 2.

Genau diese beiden Zahlen liefern dir die Operatoren `//` und `%`:

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

AUSGABE

```
3
2
3.4
```

- **Zeile 1:** `//` ist die **Ganzzahldivision**. Sie gibt an, wie oft die 5 *ganz* in die 17 passt: 3-mal. Bei positiven Zahlen fallen die Nachkommastellen einfach weg. Wie viele Kekse bekommt jedes Kind? 3.
- **Zeile 2:** `%` heißt **Modulo** und liefert den **Rest** der Division: $17 - 3 \cdot 5 = 2$. Wie viele Kekse bleiben übrig? 2.
- **Zeile 3:** Zum Vergleich die normale Division mit `/`. Sie rechnet mit Nachkommastellen weiter und liefert einen `float`.

Du kannst das Ergebnis immer überprüfen: $3 \cdot 5 + 2 = 17$. Ganzzahl-Ergebnis mal Teiler plus Rest ergibt wieder die Ausgangszahl. Übrigens: Das Zeichen `%` hat in Python **nichts mit Prozent** zu tun. `17 % 5` bedeutet „Rest von 17 geteilt durch 5“.

Anwendung: Minuten in Stunden und Minuten umrechnen

Wozu braucht man die Division mit Rest? Ein typisches Beispiel ist das Umrechnen von Zeiten. Eine Stunde hat 60 Minuten. Wie viele *volle* Stunden in einer Anzahl von Minuten stecken, liefert `// 60`. Die Minuten, die danach übrig bleiben, liefert `% 60`:

```
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.")
```

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

- **Zeile 1:** Die Eingabe wird gleich in eine ganze Zahl umgewandelt und in `minuten_gesamt` gespeichert.
- **Zeile 2:** `135 // 60` ergibt `2`, denn die 60 passt zweimal ganz in die 135.
- **Zeile 3:** `135 % 60` ergibt `15`, denn $135 - 2 \cdot 60 = 15$ Minuten bleiben übrig.
- **Zeilen 4–5:** Die Ausgabe ist lang und deshalb auf zwei Zeilen verteilt. Innerhalb der Klammern von `print` darf man in der nächsten Zeile weiterschreiben.

Versuch: Probiere es aus

Starte das Programm mit den Eingaben `60`, `59` und `600`. Stimmen die Ergebnisse? Was passiert bei `0`? Ändere das Programm dann so, dass es eine Anzahl von **Sekunden** in volle Minuten und restliche Sekunden umrechnet. Zum Beispiel sind 200 Sekunden 3 Minuten und 20 Sekunden.

Anwendung: Ist eine Zahl gerade?

Eine Zahl ist **gerade**, wenn sie sich ohne Rest durch 2 teilen lässt. Der Rest beim Teilen durch 2 ist also `0`. Bei ungeraden Zahlen bleibt immer der Rest `1`:

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

AUSGABE
```
0 1 0 2
```

14 ist gerade (Rest `0`), 15 ist ungerade (Rest `1`). Dasselbe funktioniert mit jedem Teiler: 2024 ist durch 4 teilbar (Rest `0`), 2026 nicht (Rest `2`).

Merke

Eine Zahl `a` ist genau dann durch `b` teilbar, wenn `a % b` den Wert `0` ergibt. Insbesondere ist eine Zahl gerade, wenn `zahl % 2` gleich `0` ist.

Potenzieren mit **

Mit zwei Sternchen berechnest du eine **Potenz**. `2 ** 3` bedeutet $2^3 = 2 \cdot 2 \cdot 2 = 8$:

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

AUSGABE
```
8 1000000 25
```

Die erste Zahl ist die Basis, die zweite der Exponent. `5 ** 2` ist also „5 hoch 2“, das Quadrat von 5.

Stolperstein

In manchen Programmen (z. B. in Tabellenkalkulationen) schreibt man „hoch“ mit dem Zeichen `^`. In Python bedeutet `^` etwas ganz anderes. Es gibt **keine** Fehlermeldung, aber ein falsches Ergebnis:

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

AUSGABE
```
1
```

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

Merke

Arithmetische Operatoren: `+`, `-`, `*`, `/` (Division, Ergebnis ist immer ein `float`), `//` (Ganzzahldivision), `%` (Modulo, also der Rest der Division) und `**` (Potenz).

Was wird zuerst gerechnet?

Wie in Mathe gilt eine feste Reihenfolge: zuerst Klammern, dann Potenzen `**`, dann Punktrechnung, zuletzt Strichrechnung `+` und `-`. `//` und `%` zählen zur Punktrechnung, genau wie `*` und `/`.

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

AUSGABE

```
11
25
7
```

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

Wenn du unsicher bist, setze Klammern. Sie schaden nie und machen das Programm leichter lesbar.

Arbeitsauftrag 15: Rechenausdrücke

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

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

1.2.3 Vergleichsoperatoren: wahr oder falsch?

Viele Fragen kann man nur mit Ja oder Nein beantworten: Ist 7 größer als 3? Ist die Eingabe gleich „ja“? Python beantwortet solche Fragen mit einem **Wahrheitswert**: `True` (wahr, ja) oder `False` (falsch, nein). Den passenden Datentyp `bool` kennst du schon aus dem Abschnitt über Datentypen.

Eine solche Frage stellst du mit einem **Vergleichsoperator**:

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`

Das Zeichen `!=` liest man als „ungleich“. Das Ausrufezeichen steht für „nicht“. Bei `<=` und `>=` steht das Gleichheitszeichen immer **rechts**.

```
1 a = 7
2 b = 3
3 print(a > b)
4 print(a == b)
5 print(a != b)
6 print(a <= 7)
```

AUSGABE

```
True
False
True
True
```

- **Zeilen 1–2:** Die Variablen `a` und `b` bekommen die Werte `7` und `3`.
- **Zeilen 3–5:** Ist 7 größer als 3? Ja (`True`). Ist 7 gleich 3? Nein (`False`). Ist 7 ungleich 3? Ja (`True`).
- **Zeile 6:** Ist 7 kleiner oder gleich 7? Ja, denn „gleich“ reicht schon. Also `True`.

Ergebnisse von Vergleichen speichern

Ein Wahrheitswert ist ein ganz normaler Wert. Du kannst ihn deshalb in einer Variablen speichern:

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

AUSGABE

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

- **Zeile 2:** Wie bei jeder Zuweisung wird zuerst die rechte Seite ausgewertet: Ist 15 größer oder gleich 18? Nein, das Ergebnis ist `False`. Erst dann wird dieser Wert in `ist_volljaehrig` gespeichert.
- **Zeile 4:** Der Typ ist `bool`, also ein Wahrheitswert.

Für Variablen mit Wahrheitswerten wählt man gern Namen, die wie eine Ja-Nein-Frage klingen: `ist_volljaehrig`, `hat_ausweis`, `ist_gerade`.

Texte vergleichen

Auch Texte kann man mit `==` und `!=` vergleichen. Das brauchst du, wenn eine Nutzerin zum Beispiel „ja“ oder „nein“ eingibt. Dabei kommt es auf jedes Zeichen an, auch auf Groß- und Kleinschreibung:

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

AUSGABE

```
True False False
```

Im zweiten Vergleich unterscheiden sich die Texte im ersten Buchstaben, im dritten durch ein Leerzeichen am Ende. Für Python sind das verschiedene Texte.

Stolperstein

Vergleichst du eine Eingabe mit einer Zahl und hast die Typumwandlung vergessen, gibt es **keine** Fehlermeldung. Das Ergebnis ist aber immer `False`:

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

AUSGABE

```
Gib eine Zahl ein: 7
False
```

Der Text `"7"` ist nicht dasselbe wie die Zahl `7`. Abhilfe: `zahl = int(input(...))`.

Zuweisung = oder Vergleich ==?

Diese beiden Zeichen sehen sich ähnlich, bedeuten aber etwas völlig anderes:

- `x = 5` ist eine **Zuweisung**: „`x` bekommt den Wert 5.“ Dabei wird nichts geprüft.
- `x == 5` ist ein **Vergleich**: „Ist `x` gleich 5?“ Das Ergebnis ist `True` oder `False`. Dabei wird nichts verändert.

Verwechselst du die beiden, meldet Python oft einen Fehler. Python ahnt sogar, was du gemeint hast:

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

FEHLERMELDUNG

```
  File "programm.py", line 1
    print(7 = 7)
          ^^^
SyntaxError: expression cannot contain assignment, perhaps you meant "=="?
```

Expression cannot contain assignment heißt: „Ein Ausdruck darf keine Zuweisung enthalten.“ Und *perhaps you meant* `"=="?` bedeutet „Meintest du vielleicht ==?“

Merke

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

So kannst du es sagen

So beschreibst du einen Vergleich in Fachsprache – vom einfachen zum genauen Satz:

- *„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 teilt das Ergebnis mit.“*

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

Oft reicht eine einzige Frage nicht aus. Ein Film ist ab 12 Jahren freigegeben, und an der Kasse wird ein Ausweis verlangt. Du darfst also nur hinein, wenn **beide** Bedingungen erfüllt sind: Du bist mindestens 12 **und** du hast deinen Ausweis dabei. Solche Verknüpfungen schreibt man mit **logischen Operatoren**. Sie verknüpfen Wahrheitswerte und liefern wieder einen Wahrheitswert.

Es gibt drei logische Operatoren:

- `and` heißt „und“. Das Ergebnis ist nur dann `True`, wenn **beide** Seiten `True` sind. Sobald eine Seite `False` ist, ist das Ganze `False`.
- `or` heißt „oder“. Das Ergebnis ist `True`, wenn **mindestens eine** Seite `True` ist – also **auch, wenn beide** wahr sind. Das ist anders als bei „entweder … oder“ im Alltag („Tee oder Kaffee?“). Nur wenn beide Seiten `False` sind, ist das Ergebnis `False`.
- `not` heißt „nicht“. Es steht **vor** einem einzelnen Wahrheitswert und kehrt ihn um: Aus `True` wird `False` und aus `False` wird `True`.

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

AUSGABE

```
False
True
False
False
```

- **Zeile 1:** Die rechte Seite ist `False`. Für `and` müssten beide Seiten wahr sein, also `False`.
- **Zeile 2:** Für `or` genügt eine wahre Seite, also `True`.
- **Zeile 3:** `not` kehrt `True` um.
- **Zeile 4:** Zuerst wird der Vergleich ausgewertet: `7 > 3` ist `True`. Dann kehrt `not` das Ergebnis um.

Die drei Operatoren auf einen Blick. Dabei stehen `A` und `B` für zwei beliebige Wahrheitswerte:

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

Merke

Logische Operatoren verknüpfen Wahrheitswerte:

- `and` – beide Bedingungen müssen wahr sein,
- `or` – mindestens eine Bedingung muss wahr sein,
- `not` – kehrt einen Wahrheitswert um.

Auch Wahrheitswerte kann man mit `==` und `!=` vergleichen. `A == B` ist `True`, wenn beide **denselben** Wahrheitswert haben. `A != B` ist `True`, wenn sie **verschieden** sind – also genau dann, wenn **genau einer** von beiden wahr ist:

A	B	A == B	A != B
False	False	True	False
False	True	False	True
True	False	False	True
True	True	True	False

Die Spalte `A != B` merkst du dir am besten als „entweder … oder“: wahr, wenn genau eine Seite wahr ist. Diese Verknüpfung begegnet dir in Kapitel 2 als XOR wieder.

Vergleiche verknüpfen

Meistens verknüpft man mit `and` und `or` keine festen Werte, sondern Vergleiche. Hier wird geprüft, ob eine Person jugendlich ist, also mindestens 13 und jünger als 18 Jahre alt:

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

AUSGABE

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

Python wertet Zeile 2 in dieser Reihenfolge aus:

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

Rechnungen und Vergleiche werden also **vor** den logischen Operatoren ausgewertet. Unter den logischen Operatoren gilt: zuerst `not`, dann `and`, zuletzt `or`. Wenn du `and` und `or` in einem Ausdruck mischst, setze Klammern. Dann sieht jeder sofort, was zusammengehört.

Stolperstein

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

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

FEHLERMELDUNG

```
SyntaxError: invalid syntax
```

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

Klammern entscheiden: Darf ich zocken?

Bei drei Bedingungen kommt es darauf an, was zusammengehört. Ein Beispiel aus dem Alltag:

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

Zwei Eltern formulieren ihre Regel fürs Zocken unterschiedlich:

- **Regel 1:** `(A and B) or C` – „Du darfst zocken, wenn die Hausaufgaben fertig **und** das Zimmer aufgeräumt ist – **oder** wenn Wochenende ist." Am Wochenende reicht also `C = True` allein.
- **Regel 2:** `A and (B or C)` – „Du darfst zocken, wenn die Hausaufgaben fertig sind **und** zusätzlich das Zimmer aufgeräumt **oder** Wochenende ist." Hier müssen die Hausaufgaben **immer** fertig sein.

Probiere es aus: Die Hausaufgaben sind nicht fertig, das Zimmer ist nicht aufgeräumt, aber es ist Wochenende.

```
1 A = False   # Hausaufgaben sind fertig
2 B = False   # Zimmer ist aufgeräumt
3 C = True    # Es ist Wochenende
4
5 ausdruck_1 = (A and B) or C
6 ausdruck_2 = A and (B or C)
7
8 print("(A and B) or C =", ausdruck_1)
9 print("A and (B or C) =", ausdruck_2)
```

AUSGABE

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

- **Zeilen 1–3:** Die drei Bedingungen werden als Wahrheitswerte gespeichert. Die Kommentare hinter `#` erklären, wofür jede Variable steht.
- **Zeile 5:** Zuerst wird die Klammer berechnet: `False and False` ergibt `False`. Dann folgt `False or True`, das ist `True`.
- **Zeile 6:** Zuerst die Klammer: `False or True` ergibt `True`. Dann `False and True`, das ist `False`.

Nach Regel 1 darfst du also zocken, nach Regel 2 nicht. Die **Klammerung** entscheidet darüber, welche Bedingung **zwingend** erfüllt sein muss. 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 `(A and B) or C` und `A and (B or C)` bedeuten also **nicht** dasselbe – man sagt: Sie sind **nicht äquivalent**.

Merke

Klammern bestimmen, welcher Teilausdruck zuerst ausgewertet wird. `(A and B) or C` und `A and (B or C)` sind **nicht äquivalent**. Setze deshalb immer Klammern, wenn du `and` und `or` in einem Ausdruck mischst.

Bedingungen verneinen: die Regeln von De Morgan

Manchmal will man das **Gegenteil** einer verknüpften Bedingung prüfen, zum Beispiel: Ist eine Person **nicht** jugendlich? Am einfachsten setzt man `not` vor die ganze Bedingung in Klammern:

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

AUSGABE

```
True
True
```

- **Zeile 2:** Die Klammer wird zuerst ausgewertet: `20 >= 13` ist `True`, `20 < 18` ist `False`, zusammen mit `and` also `False`. `not` macht daraus `True`. Die Person ist nicht jugendlich.
- **Zeile 3:** Dieselbe Frage ohne `not`: Die Person ist nicht jugendlich, wenn sie **jünger als 13 oder mindestens 18** ist. Auch das ergibt `True`.

Beide Zeilen liefern für **jedes** Alter dasselbe Ergebnis. Probiere es mit anderen Werten aus, zum Beispiel `alter = 15` oder `alter = 9`. Beim Umformen ist zweierlei passiert: Jeder einzelne Vergleich wurde verneint (aus >= wurde <, aus < wurde >=), und aus `and` wurde `or`. Das ist kein Zufall, sondern eine allgemeine Regel. Sie heißt nach dem englischen Mathematiker **Augustus De Morgan**:

A	B	not (A and B)	not A or not B	not (A or B)	not A and not B
False	False	True	True	True	True
False	True	True	True	False	False
True	False	True	True	False	False
True	True	False	False	False	False

Die dritte und vierte Spalte stimmen in jeder Zeile überein, ebenso die fünfte und sechste. Im Alltag kennst du das auch: „Es ist **nicht** so, dass es regnet **oder** windet“ bedeutet „Es regnet **nicht und** es windet **nicht**“.

Merke

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).

In der Mathematik und in Kapitel 2 schreibt man $\wedge$ für `and`, $\vee$ für `or` und $\neg$ für `not`. Die beiden Regeln lauten dann $\neg(A \wedge B) \equiv \neg A \vee \neg B$ und $\neg(A \vee B) \equiv \neg A \wedge \neg B$. Das Zeichen $\equiv$ heißt: Beide Seiten haben für alle Werte von A und B dasselbe Ergebnis.

Um einen einzelnen Vergleich zu verneinen, brauchst du kein `not`. Es genügt, den Vergleichsoperator umzudrehen:

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

Stolperstein

Der häufigste Fehler: Man verneint die Teilbedingungen, vergisst aber, `and` in `or` umzuwandeln.

```
alter = 20
print(alter < 13 and alter >= 18)
```

AUSGABE

```
False
```

Kein Mensch ist gleichzeitig jünger als 13 **und** mindestens 18 Jahre alt. Diese Bedingung ist also **immer** `False`, egal welches Alter eingegeben wird. Richtig ist `alter < 13 or alter >= 18`.

Arbeitsauftrag 16: Bedingungen verneinen

Schreibe zu jeder Bedingung eine Bedingung **ohne** `not`, die genau das Gegenteil prüft. Teste deine Antworten am Computer mit ein paar Werten.

1. `zahl >= 1 and zahl <= 6` (eine Würfelzahl)
2. `note == 1 or note == 2`
3. `not (temperatur > 30 or regen)` *(Tipp: `regen` ist eine Variable mit einem Wahrheitswert; „nicht `regen`“ darfst du hier als `not regen` stehen lassen.)*

Meine Bedingungen:

Alles zusammen

Das folgende Programm zeigt noch einmal alle drei Gruppen von Operatoren:

```
1 a = 7
2 b = 3
3
4 print("Summe:", a + b)
5 print("Ganzzahldivision:", a // b)
6 print("Rest (Modulo):", a % b)
7 print("Potenz:", a ** 2)
8
9 print("a größer b?", a > b)
10 print("a gleich b?", a == b)
11
12 alter = 15
13 hat_ausweis = True
14 print("Darf ins Kino (ab 12) und hat Ausweis dabei:",
15       alter >= 12 and hat_ausweis)
```

AUSGABE

```
Summe: 10
Ganzzahldivision: 2
Rest (Modulo): 1
Potenz: 49
a größer b? True
a gleich b? False
Darf ins Kino (ab 12) und hat Ausweis dabei: True
```

- **Zeilen 4–7:** arithmetische Operatoren. Die 3 passt 2-mal ganz in die 7, Rest 1. `7 ** 2` ist $7^2 = 49$.
- **Zeilen 9–10:** Vergleiche. Ihr Ergebnis ist ein Wahrheitswert.
- **Zeilen 12–13:** Die Variable `hat_ausweis` speichert direkt den Wahrheitswert `True`.
- **Zeilen 14–15:** Zuerst wird `alter >= 12` ausgewertet, das ergibt `True`. Dann wird `True and True` berechnet. Das Ergebnis ist `True`. Probiere aus, was sich ändert, wenn du `hat_ausweis = False` oder `alter = 11` setzt.

Arbeitsauftrag 17: Wahr oder falsch?

Es gilt `x = 5` und `y = 10`. Bestimme für jeden Ausdruck, ob er `True` oder `False` ergibt. Überprüfe deine Antworten danach am Computer.

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`	

1.2.5 Übungen

Arbeitsauftrag 18: Gerade oder ungerade?

Schreibe ein Programm, das eine Ganzzahl einliest und mit Hilfe des Modulo-Operators `%` einen Wahrheitswert ausgibt, der angibt, ob die Zahl gerade ist.

So könnte es aussehen:

AUSGABE
```
Ganzzahl eingeben: 14
Ist gerade: True
```

Arbeitsauftrag 19: Schaltjahr-Check

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

Schreibe ein Programm, das eine Jahreszahl einliest und einen Wahrheitswert ausgibt, der angibt, ob es ein Schaltjahr ist. Verwende dafür `and`, `or` und `not` bzw. Vergleichsoperatoren. Du brauchst dafür nur Operatoren, sonst nichts.

Tipp

Zerlege die Regel in Teile und übersetze jeden Teil einzeln:

- „durch 4 teilbar“ → `jahr % 4 == 0`
- „nicht durch 100 teilbar“ → `jahr % 100 != 0`
- „durch 400 teilbar“ → `jahr % 400 == 0`

Ein Jahr ist also Schaltjahr, wenn *die ersten beiden Teile zusammen* wahr sind **oder** wenn der dritte Teil wahr ist. Setze Klammern um das, was zusammengehört. Teste dein Programm mit 2024 (Schaltjahr), 2023 (kein Schaltjahr), 1900 (kein Schaltjahr) und 2000 (Schaltjahr).

1.2.6 Weitere Übungen

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.

Arbeitsauftrag 20: 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`	

Arbeitsauftrag 21: 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`	

Arbeitsauftrag 22: 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			

Arbeitsauftrag 23: 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`.

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

Arbeitsauftrag 24: 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.

Meine Ausdrücke:

Arbeitsauftrag 25: 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**.

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

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.

Die Fehler:

Gymnasium Ottweiler

Arbeitsauftrag 26: 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:

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

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	

Zu 2.:

Gymnasium Ottweiler

Arbeitsauftrag 27: 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`?

1.2.7 Das Wichtigste in Kürze

- Ein **Operator** verarbeitet einen oder mehrere Werte (Operanden) zu einem neuen Wert.
- **Arithmetische Operatoren:** `+ - * /` sowie `//` (Ganzzahldivision), `%` (Rest) und `**` (Potenz). Potenz vor Punkt vor Strich, Klammern zuerst.
- `zahl // 60` und `zahl % 60` zerlegen z. B. Minuten in Stunden und restliche Minuten. `zahl % 2 == 0` prüft, ob eine Zahl gerade ist.
- **Vergleichsoperatoren** `== != < > <= >=` liefern immer `True` oder `False`.
- **Logische Operatoren:** `and` (beide wahr), `or` (mindestens eine wahr), `not` (umkehren).
- **Klammern** legen fest, was zuerst ausgewertet wird: `(A and B) or C` und `A and (B or C)` sind nicht äquivalent.
- **Regeln von De Morgan:** `not (A and B)` ist dasselbe wie `not A or not B`, `not (A or B)` dasselbe wie `not A and not B`. So verneint man verknüpfte Bedingungen.
- `=` weist einen Wert zu, `==` vergleicht zwei Werte.

Selbstcheck

- ☐ Ich kann erklären, was `//`, `%` und `**` berechnen, und Beispiele im Kopf ausrechnen.
- ☐ Ich kann eine Anzahl Minuten mit `//` und `%` in Stunden und Minuten umrechnen.
- ☐ Ich kann mit `%` prüfen, ob eine Zahl gerade oder durch eine andere Zahl teilbar ist.
- ☐ Ich kenne die sechs Vergleichsoperatoren und weiß, dass ihr Ergebnis ein Wahrheitswert ist.
- ☐ Ich kann vorhersagen, was `and`, `or` und `not` ergeben.
- ☐ Ich kann erklären, warum `(A and B) or C` etwas anderes bedeutet als `A and (B or C)`.
- ☐ Ich kann eine verknüpfte Bedingung mit den Regeln von De Morgan verneinen.
- ☐ Ich verwechsle `=` und `==` nicht mehr.

Gymnasium Ottweiler

1.3 Verzweigungen: Programme treffen Entscheidungen

Leitidee

Jeden Tag triffst du Entscheidungen nach einfachen Regeln: *Wenn* es regnet, *dann* nehme ich einen Schirm mit. *Wenn* der Bus kommt, steige ich ein, *sonst* warte ich. Auch Programme müssen entscheiden: Ist das Passwort richtig? Hat die Spielerin genug Punkte? Ist die Zahl positiv? Bisher hat Python jede Zeile deines Programms ausgeführt, immer von oben nach unten. Jetzt lernst du, wie ein Programm abhängig von einer **Bedingung** nur bestimmte Anweisungen ausführt. Aus der Blockprogrammierung kennst du dafür die Bausteine „falls … dann“ und „falls … dann … sonst“.

Lernziele

Nach diesem Abschnitt kannst du

- erklären, woraus eine Verzweigung besteht, und die Fachbegriffe *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 Programmablaufplan und einem Ablaufprotokoll darstellen,
- Bedingungen mit `and` und `or` formulieren und Verzweigungen verschachteln.

1.3.1 Die einseitige Verzweigung: `if`

Stell dir eine Weiche bei der Eisenbahn vor. Je nachdem, wie sie gestellt ist, fährt der Zug geradeaus oder biegt ab. In einem Programm übernimmt eine **Bedingung** die Rolle der Weiche. Eine Bedingung ist eine Frage, deren Antwort `True` oder `False` ist – genau wie die Vergleiche aus dem Abschnitt über Operatoren.

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

AUSGABE

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

- **Zeile 1:** Die Variable `temperatur` bekommt den Wert `30`.
- **Zeile 2:** `if` ist englisch und heißt „wenn“ oder „falls“. Dahinter steht die Bedingung `temperatur > 25`. Python prüft sie: 30 ist größer als 25, die Bedingung ist also `True`. Am Ende der Zeile steht ein **Doppelpunkt**.
- **Zeile 3:** Diese Zeile ist **eingerückt**, sie beginnt mit vier Leerzeichen. Dadurch gehört sie zur `if`-Anweisung. Sie wird nur ausgeführt, weil die Bedingung wahr ist.
- **Zeile 4:** Diese Zeile ist nicht eingerückt. Sie gehört nicht mehr zur `if`-Anweisung und wird in jedem Fall ausgeführt.

Was passiert, wenn die Bedingung falsch ist? Überlege zuerst, dann lies weiter.

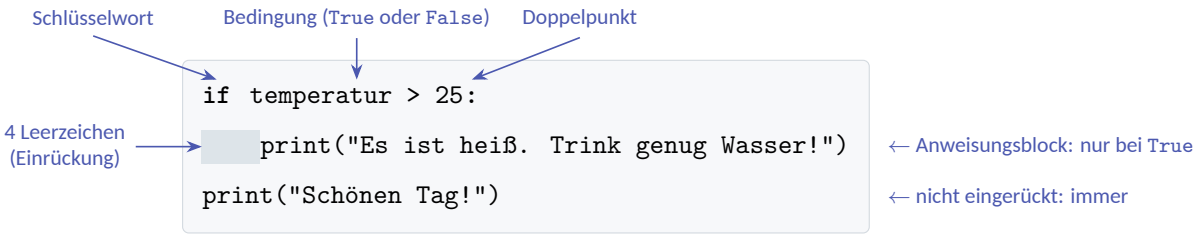
```
1 temperatur = 18
2 if temperatur > 25:
3     print("Es ist heiß. Trink genug Wasser!")
4 print("Schönen Tag!")
```

AUSGABE

```
Schönen Tag!
```

Jetzt ist `temperatur > 25` falsch (`False`). Python **überspringt** die eingerückte Zeile 3 und macht direkt mit Zeile 4 weiter.

Die `if`-Anweisung steuert also, welche Zeilen ausgeführt werden. Anweisungen, die so den Ablauf eines Programms steuern, nennt man **Kontrollstrukturen**.



Aufbau einer einseitigen Verzweigung in Python. Nur der eingerückte Anweisungsblock hängt von der Bedingung ab

Definition: Bedingung

Eine **Bedingung** ist ein Ausdruck, dessen Wert entweder wahr (`True`) oder falsch (`False`) ist, zum Beispiel `temperatur > 25`.

Definition: Verzweigung

Eine **Verzweigung** ist eine Kontrollstruktur, die abhängig von einer Bedingung festlegt, welche Anweisungen ausgeführt werden. Man nennt sie auch *bedingte Anweisung*.

Gymnasium Ottweiler

Merke

- Eine **Verzweigung** (bedingte Anweisung) besteht aus einer Bedingung und einem oder mehreren Anweisungsblöcken.
- **Einseitige Verzweigung:** `if Bedingung:` – der eingerückte Block wird nur ausgeführt, wenn die Bedingung `True` ist.

1.3.2 Einrückung: Was gehört zum Block?

Ein Block kann aus mehreren Zeilen bestehen. Woher weiß Python, wo der Block aufhört? An der **Einrückung**. Alle Zeilen direkt unter dem `if`, die gleich weit eingerückt sind, gehören zum Block. Die erste Zeile, die wieder ganz links beginnt, gehört nicht mehr dazu.

Vergleiche die beiden Programme. Sie unterscheiden sich nur in der Einrückung von Zeile 4:

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

AUSGABE

```
Spiel beendet.
```

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

AUSGABE

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

In beiden Programmen ist die Bedingung `3 >= 10` falsch. Im ersten Programm gehören die Zeilen 3 und 4 zum Block und werden beide übersprungen. Im zweiten Programm gehört nur Zeile 3 zum Block. Zeile 4 wird immer ausgeführt – und das Programm verteilt einen Preis, obwohl niemand gewonnen hat.

Merke

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

Stolperstein

Vergisst du den Doppelpunkt, meldet Python `expected ':'` („Doppelpunkt erwartet“). Das Zeichen `^` zeigt genau auf die Stelle, an der er fehlt:

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

FEHLERMELDUNG

```
  File "programm.py", line 2
    if zahl > 0
               ^
SyntaxError: expected ':'
```

Tipp

Die meisten Programmierumgebungen rücken nach einem Doppelpunkt automatisch ein, wenn du die Eingabetaste drückst. Mit der Tabulatortaste (Tab) rückst du selbst ein. Achte darauf, dass alle Zeilen eines Blocks **genau gleich weit** eingerückt sind.

Stolperstein

Fehlt die Einrückung nach dem Doppelpunkt ganz, bricht Python mit einem **IndentationError** ab (*indentation* heißt „Einrückung“):

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

FEHLERMELDUNG

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

Übersetzt: „Nach der `if`-Anweisung in Zeile 2 wurde ein eingerückter Block erwartet.“ Python zeigt auf Zeile 3, denn dort hätte der Block beginnen müssen. Abhilfe: Zeile 3 um vier Leerzeichen einrücken.

Auch zu viel Einrückung ist ein Fehler. Eine Zeile darf nur eingerückt sein, wenn sie zu einem Block gehört, der mit einer Zeile mit Doppelpunkt beginnt:

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

FEHLERMELDUNG

```
IndentationError: unexpected indent
```

Unexpected indent heißt „unerwartete Einrückung“. Prüfe die genannte Zeile und rücke sie richtig ein.

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

Oft soll ein Programm in **beiden** Fällen etwas tun: wenn die Bedingung wahr ist, das eine, und sonst das andere. Dafür gibt es `else` (englisch „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.")
```

AUSGABE

```
Gib eine Zahl ein: 7
Die Zahl ist größer als 4.
Fertig.
```

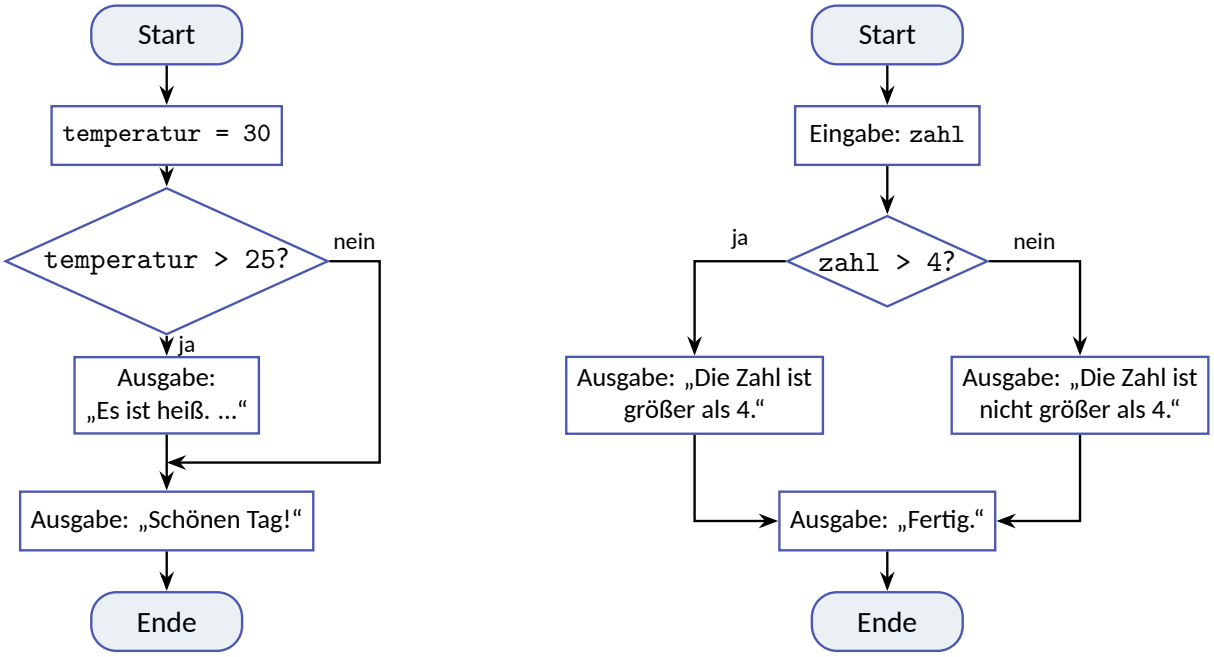
- **Zeile 1:** Die Eingabe wird in eine ganze Zahl umgewandelt und in `zahl` gespeichert.
- **Zeile 2:** Die Bedingung `zahl > 4` wird geprüft. Für 7 ist sie wahr.
- **Zeile 3:** Dieser eingerückte Block ist der **dann-Teil**. Er wird ausgeführt, wenn die Bedingung wahr ist.
- **Zeile 4:** `else:` steht genau unter dem `if` (gleiche Einrückung) und hat **keine** eigene Bedingung, aber einen Doppelpunkt. Schreibst du trotzdem eine Bedingung dahinter, etwa `else zahl < 4:`, meldet Python `SyntaxError: expected ':'`.
- **Zeile 5:** Der **sonst-Teil**. Er wird nur ausgeführt, wenn die Bedingung falsch ist. Bei der Eingabe 7 wird er übersprungen.
- **Zeile 6:** Nicht eingerückt, wird also in jedem Fall ausgeführt.

Es wird also **immer genau einer** der beiden Teile ausgeführt, nie beide und nie keiner. Weil das Programm zwischen zwei Möglichkeiten (Alternativen) wählt, nennt man die zweiseitige Verzweigung auch **Alternativanweisung**.

Den Ablauf darstellen

Wie ein Programm mit Verzweigung abläuft, kann man in einem **Programmablaufplan** zeichnen. Rauten stehen für Bedingungen, Rechtecke für Anweisungen. Die Pfeile zeigen, in welcher Reihenfolge Python die Anweisungen ausführt. An einer Raute teilt sich der Weg in „ja“ (Bedingung wahr) und „nein“ (Bedingung falsch).

Gymnasium Ottweiler



Programmablaufpläne: links die einseitige Verzweigung aus dem Temperatur-Beispiel, rechts die zweiseitige Verzweigung aus dem Programm oben

Ein **Ablaufprotokoll** zeigt, welche Zeilen bei einer bestimmten Eingabe wirklich ausgeführt werden. Hier für das Programm oben, links mit der Eingabe 7, rechts mit der Eingabe 2:

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

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

Bei der Eingabe 7 ist die Bedingung in Zeile 2 wahr, also geht es mit Zeile 3 weiter. Bei der Eingabe 2 ist sie falsch, also springt Python zu Zeile 5. Die Zeile `else:` selbst tut nichts. Sie zeigt Python nur, wo der sonst-Teil beginnt. Deshalb taucht Zeile 4 im Protokoll nicht auf.

Merke

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

So kannst du es sagen

So beschreibst du eine Verzweigung in Fachsprache – vom einfachen zum genauen Satz:

- *„Ich bekomme raus, ob die Zahl größer als 4 ist.“*
- *„Die Anweisung gibt ‚Die Zahl ist größer als 4.‘ aus, wenn eine Zahl größer als 4 ist, und ‚Die Zahl ist nicht größer als 4.‘, wenn sie es nicht ist.“*
- *„Die Alternativanweisung prüft, ob eine Zahl größer als 4 ist. Ist dies der Fall, wird ‚Die Zahl ist größer als 4.‘ ausgegeben und sonst ‚Die Zahl ist nicht größer als 4.‘ “*

Nützliche Wendungen: *Die Verzweigung besteht aus … · Es wird geprüft, ob … · Ist die Bedingung erfüllt, wird der dann-Teil ausgeführt, sonst der sonst-Teil.*

Arbeitsauftrag 28: Glatteis-Warnung

Lies das Programm. Beschreibe in zwei bis drei Sätzen mit Fachbegriffen, was es tut. Gib dann an, was es bei den Eingaben `-3`, `0` und `5` ausgibt.

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

Gymnasium Ottweiler

Beschreibung und Ausgaben:

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

Manchmal gibt es mehr als zwei Möglichkeiten. Ein Programm soll aus einer Punktzahl von 0 bis 100 eine Note machen. Dafür gibt es `elif`. Das ist eine Abkürzung für *else if*, also „sonst, wenn“:

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

AUSGABE

```
Punktzahl (0-100) eingeben: 80
Note: gut
```

- **Zeile 1:** Die Punktzahl wird eingelesen und in `note` gespeichert.
- **Zeile 3:** Die erste Bedingung wird geprüft: Ist 80 größer oder gleich 90? Nein.
- **Zeile 5:** Weil die erste Bedingung falsch war, wird die nächste geprüft: Ist 80 größer oder gleich 75? Ja.
- **Zeile 6:** Der zugehörige Block wird ausgeführt: „Note: gut“.
- **Zeilen 7–10:** Werden übersprungen. Sobald eine Bedingung zutrifft, prüft Python die weiteren **gar nicht mehr**, obwohl auch `80 >= 50` wahr wäre.

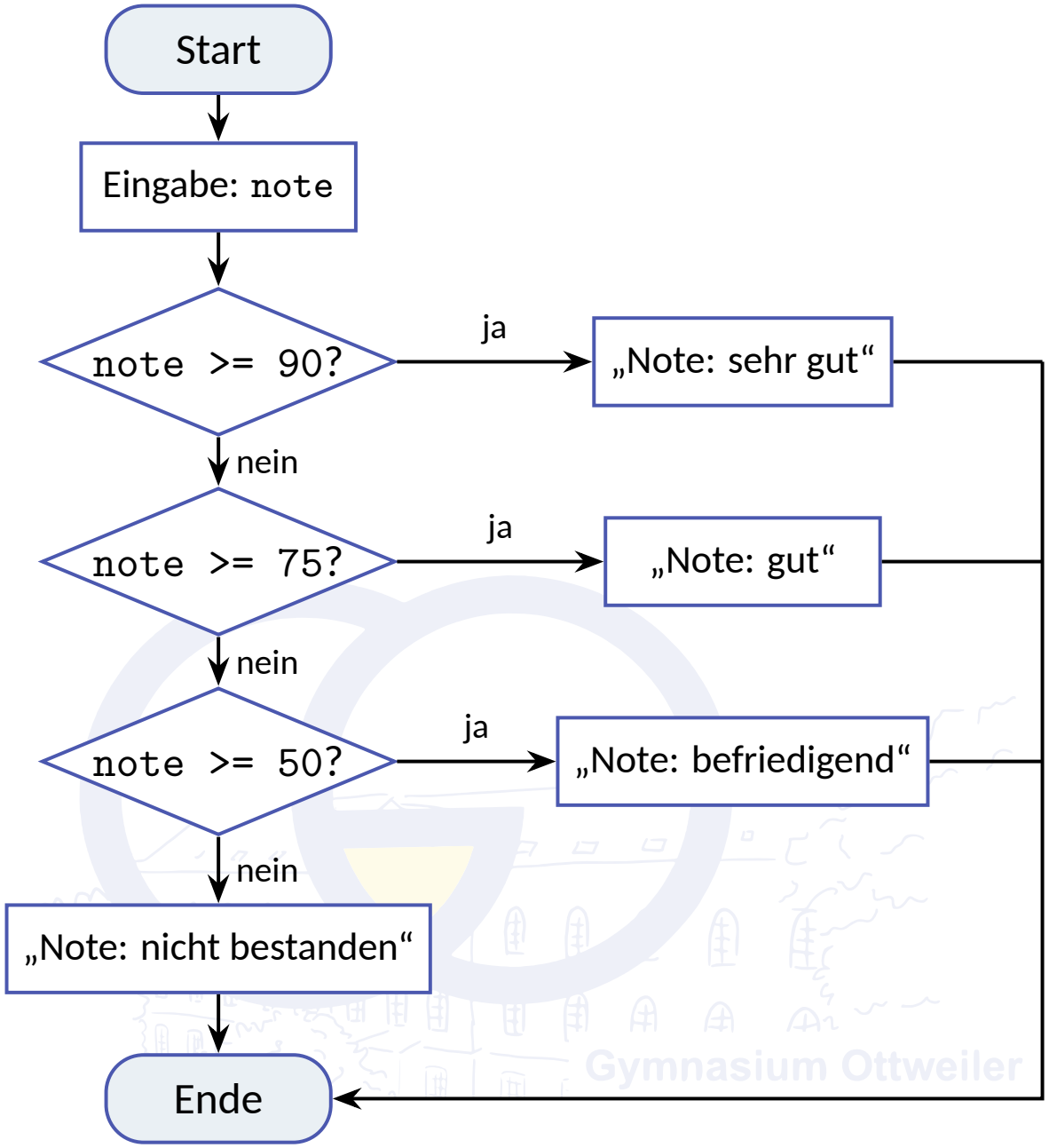
Der `else`-Teil am Ende fängt alle Fälle auf, in denen **keine** der Bedingungen wahr war. Er darf auch fehlen. Dann passiert in diesem Fall einfach nichts.

Das Ablaufprotokoll zeigt, welche Zeilen ausgeführt werden. Links für die Eingabe 80, rechts für die Eingabe 30:

Schritt	Zeile	note
1	1	80
2	3	80
3	5	80
4	6	80

Schritt	Zeile	note
1	1	30
2	3	30
3	5	30
4	7	30
5	10	30

Bei der Eingabe 30 sind alle drei Bedingungen falsch (Zeilen 3, 5 und 7). Deshalb wird der `else`-Teil in Zeile 10 ausgeführt.



Programmablaufplan des Notenrechners: Die Bedingungen werden von oben nach unten geprüft. Die erste, die zutrifft, entscheidet

Merke

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

Stolperstein

Bei `elif` kommt es auf die **Reihenfolge** an. Hier stehen die Bedingungen in der falschen Reihenfolge:

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

AUSGABE

```
Punktzahl (0-100) eingeben: 95
Note: befriedigend
```

95 Punkte, aber nur „befriedigend“! Die erste Bedingung `95 >= 50` ist wahr, also wird ihr Block ausgeführt und der Rest übersprungen. Die Fälle „sehr gut“ und „gut“ werden nie erreicht. Regel: Bei `>=` die **größte** Grenze zuerst prüfen, bei `<` die **kleinste**.

Arbeitsauftrag 29: Größte von zwei Zahlen

Schreibe ein Programm, das zwei Zahlen einliest und ausgibt, welche die größere ist – oder ob beide gleich sind. Teste dein Programm mit mindestens drei Eingabepaaren: einmal ist die erste Zahl größer, einmal die zweite, einmal sind beide gleich.

Arbeitsauftrag 30: Altersklassifikation

Schreibe ein Programm, das ein Alter einliest und es so einordnet:

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

Teste dein Programm besonders mit den **Grenzwerten** 5, 6, 12, 13, 17 und 18.

Gymnasium Ottweiler

1.3.5 Bedingungen mit `and` und `or`

Eine Bedingung darf alles sein, was `True` oder `False` ergibt – also auch Verknüpfungen mit `and`, `or` und `not` aus dem Abschnitt über Operatoren:

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

AUSGABE

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

- **Zeile 1:** Der Tag wird als Text eingelesen. Hier wird nicht umgewandelt, denn es soll ein Text verglichen werden.

- **Zeile 2:** Die Bedingung ist wahr, wenn **mindestens einer** der beiden Vergleiche wahr ist. `tag == "Samstag"` ist falsch, `tag == "Sonntag"` ist wahr. Also ist die ganze Bedingung wahr.

Stolperstein

Ein häufiger Fehler ist, den zweiten Vergleich abzukürzen: `if tag == "Samstag" or "Sonntag":`. Das sieht aus wie Deutsch, bedeutet in Python aber etwas anderes. Python liest `"Sonntag"` als eigene Bedingung, und ein Text, der nicht leer ist, zählt als wahr. Die Bedingung ist deshalb **immer** erfüllt:

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

AUSGABE

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

Schreibe auf **beiden** Seiten von `or` und `and` einen vollständigen Vergleich.

Stolperstein

Schreibst du in einer Bedingung `=` statt `==`, meldet Python einen Syntaxfehler und macht gleich einen Vorschlag:

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

FEHLERMELDUNG

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

„Meintest du vielleicht `==` …?“ – genau. Den zweiten Vorschlag `:=` brauchst du nicht.

1.3.6 Verschachtelte Verzweigungen

Im Block einer Verzweigung dürfen beliebige Anweisungen stehen – auch eine weitere Verzweigung. Man spricht dann von einer **verschachtelten** Verzweigung. Ein Beispiel ist eine Anmeldung: Nur wenn der Benutzername bekannt ist, wird überhaupt nach dem Passwort gefragt.

```
1 benutzer = input("Benutzername: ")
2 if benutzer == "alex":
3     passwort = input("Passwort: ")
4     if passwort == "geheim123":
5         print("Willkommen, Alex!")
6     else:
7         print("Falsches Passwort.")
8 else:
9     print("Unbekannter Benutzer.")
```

AUSGABE

```
Benutzername: alex
Passwort: geheim123
Willkommen, Alex!
```

- **Zeile 2:** Die äußere Verzweigung prüft den Benutzernamen.
- **Zeilen 3–7:** Das ist der dann-Teil der äußeren Verzweigung. Alle diese Zeilen sind mindestens vier Leerzeichen eingerückt.
- **Zeilen 4–7:** Darin steht eine zweite, **innere** Verzweigung. Ihre Blöcke (Zeilen 5 und 7) sind noch einmal vier Leerzeichen weiter eingerückt, also acht Leerzeichen.
- **Zeilen 8–9:** Das `else` steht genau unter dem äußeren `if`. Es gehört deshalb zur äußeren Verzweigung.

An der Einrückung erkennst du, welches `else` zu welchem `if` gehört: Es steht immer genau darunter. Das Ablaufprotokoll für die Eingaben `alex` und `1234` zeigt, dass Python die innere Verzweigung erreicht, dort aber den sonst-Teil ausführt:

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

Der Strich – bedeutet: Die Variable `passwort` gibt es zu diesem Zeitpunkt noch nicht.

Merke

Verzweigungen können **verschachtelt** werden: Eine `if`-Anweisung steht im Block einer anderen. Die Einrückung zeigt, welcher Block zu welcher Bedingung gehört.

Arbeitsauftrag 31 (Zusatz): Achterbahn-Check

Ein Freizeitpark hat folgende Regeln für eine Achterbahn:

- Mindestgröße: 140 cm.
- Ist man kleiner als 140 cm, aber mindestens 120 cm, darf man nur in Begleitung eines Erwachsenen mitfahren.
- Ist man kleiner als 120 cm, darf man nicht mitfahren.

Schreibe ein Programm, das die Körpergröße (in cm) abfragt und ob eine erwachsene Begleitung dabei ist (`ja`/`nein`). Es soll die passende Meldung ausgeben. Nutze eine **verschachtelte** Verzweigung.

1.3.7 Weitere Übungen

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.

Arbeitsauftrag 32: Bestanden? ●○○

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

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

Meine Vorhersagen:

Arbeitsauftrag 33: 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:

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

Programm B:

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

Ausgaben und Erklärung:

Arbeitsauftrag 34: 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`.

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

Arbeitsauftrag 35: 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")
```

Eingabe	`36.8`	`37.4`	`37.5`	`38.4`	`38.5`	`39.2`
Ausgabe						

Arbeitsauftrag 36: 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.

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

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.

Die Fehler:

Arbeitsauftrag 37: 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.

Arbeitsauftrag 38: 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.

Arbeitsauftrag 39: 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.

1.3.8 Das Wichtigste in Kürze

- Eine **Verzweigung** besteht aus einer **Bedingung** (Wert `True` oder `False`) und einem oder mehreren Anweisungsblöcken.
- `if Bedingung:` – der Block wird nur ausgeführt, wenn die Bedingung wahr ist (**einseitig**).
- `if … else:` – entweder der dann-Teil oder der sonst-Teil wird ausgeführt (**zweiseitig**).
- `if … elif … else:` – nur der Block der ersten zutreffenden Bedingung wird ausgeführt (**mehrseitig**). Die Reihenfolge ist wichtig.
- Hinter jeder Bedingung und hinter `else` steht ein **Doppelpunkt**. Die **Einrückung** (4 Leerzeichen) legt fest, was zum Block gehört.
- Bedingungen können mit `and`, `or` und `not` verknüpft werden – auf beiden Seiten mit vollständigem Vergleich.
- Verzweigungen können **verschachtelt** werden.

Selbstcheck

- ☐ Ich kann erklären, was eine Bedingung und was eine Verzweigung ist.
- ☐ Ich kann ein-, zwei- und mehrseitige Verzweigungen mit `if`, `else` und `elif` schreiben.
- ☐ Ich weiß, warum bei `elif` die Reihenfolge der Bedingungen wichtig ist.
- ☐ Ich kann an der Einrückung erkennen, welche Zeilen zu einem Block gehören, und kann einen `IndentationError` beheben.
- ☐ Ich kann zu einem Programm mit Verzweigung einen Programmablaufplan zeichnen und ein Ablaufprotokoll erstellen.
- ☐ Ich kann eine Verzweigung mit den Fachbegriffen dann-Teil und sonst-Teil beschreiben.

1.4 Übung: Verzweigungen lesen, testen und erweitern

Leitidee

Programmiererinnen und Programmierer schreiben nicht nur neue Programme. Mindestens genauso oft **lesen** sie fremden Code, **suchen Fehler** darin und **erweitern** ein Programm, das schon da ist. Die Fehlersuche heißt auf Englisch *Debugging* – wörtlich „Entwanzen“, denn einen Programmfehler nennt man einen *Bug* („Käfer“). In diesem Abschnitt übst du genau diese drei Tätigkeiten mit Verzweigungen. Dazu lernst du, wie man ein Programm mit gut ausgewählten Eingaben **testet**, und du lernst eine neue Form der Verzweigung kennen: die **Mehrfachauswahl**.

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.4.1 Das brauchst du

Alles, was du für diesen Abschnitt brauchst, kennst du aus den Abschnitten über Operatoren und Verzweigungen. Hier ist es noch einmal kurz zusammengefasst. Der Aufbau einer mehrseitigen Verzweigung sieht so aus (`elif` und `else` sind dabei freiwillig):

```
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
```

Merke

- `if` / `elif` / `else` – nur der Block der **ersten zutreffenden** Bedingung wird ausgeführt.
- Achte auf **Doppelpunkt** `:` und korrekte **Einrückung**.
- `==` vergleicht, `=` weist zu.

Die häufigsten Fehler bei Verzweigungen und was Python dazu meldet:

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

1.4.2 Programme testen

Ein Programm, das ohne Fehlermeldung läuft, ist noch lange nicht richtig. Es kann trotzdem eine falsche Ausgabe liefern. Deshalb **testet** man es: Man führt es mit verschiedenen Eingaben aus und vergleicht die Ausgabe mit dem, was herauskommen **soll**.

Eine Eingabe zusammen mit der erwarteten Ausgabe nennt man einen **Testfall**. Wichtig ist, dass du die erwartete Ausgabe **vorher** überlegst – nicht erst, wenn du das Ergebnis siehst. Bei Programmen mit Verzweigungen wählst du die Testfälle so:

1. **Jeder Zweig mindestens einmal.** Für jeden Block der Verzweigung brauchst du eine Eingabe, bei der genau dieser Block ausgeführt wird.
2. **Die Grenzfälle.** Das sind die Werte direkt an einer Grenze. Bei der Bedingung `alter < 18` sind das 17 und 18. Hier passieren die meisten Fehler, zum Beispiel wenn `<` statt `<=` geschrieben wurde.

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

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 zwischen positiv und negativ
`1` und `-1`	positiv bzw. negativ	direkt neben der Grenze

Genau dieses Programm reparierst du in der nächsten Aufgabe.

1.4.3 Fehler finden

Arbeitsauftrag 40: Fehler finden (Debugging)

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

```
1 zahl = int(input("Zahl eingeben: "))
2 if zahl = 0
3     print("Die Zahl ist null")
4 elif zahl > 0:
5 print("Die Zahl ist positiv")
6 else:
7     print("Die Zahl ist negativ")
```

1. Finde die drei Fehler, **bevor** du das Programm ausführst. Notiere jeweils die Zeile und was falsch ist.
2. Schreibe die korrigierte Version auf.
3. Teste dein korrigiertes Programm mit den Testfällen aus dem Abschnitt „Programme testen“.

Tipp

Wenn du das fehlerhafte Programm ausführst, meldet Python immer nur den **ersten** Fehler, den es findet. Behebe ihn und starte das Programm erneut. Dann erscheint die Meldung zum nächsten Fehler. Die Tabelle im Abschnitt „Das brauchst du“ hilft dir, die Meldungen zu verstehen.

Die Fehler:

Gymnasium Ottweiler

1.4.4 Code lesen

Arbeitsauftrag 41: Code lesen und erklären

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

```
1 zahl = 17
2 if zahl % 15 == 0:
3     print("FizzBuzz")
4 elif zahl % 3 == 0:
5     print("Fizz")
6 elif zahl % 5 == 0:
7     print("Buzz")
8 else:
9     print(zahl)
```

1. Notiere, was das Programm für `zahl = 17` ausgibt. Begründe mit den Bedingungen.
2. Sage ebenso die Ausgabe für `zahl = 9`, `zahl = 10` und `zahl = 30` voraus.
3. Erkläre, warum die Bedingung `zahl % 15 == 0` **zuerst** geprüft wird.

Führe das Programm danach aus (ändere dazu den Wert in Zeile 1) und prüfe deine Vorhersagen.

Meine Vorhersagen und Begründungen:

Gymnasium Ottweiler

Arbeitsauftrag 42: Programmablaufplan zeichnen

Zeichne einen Programmablaufplan zum Programm aus dem Auftrag *Code lesen und erklären*. Verwende Rauten für die Bedingungen und Rechtecke für die Anweisungen. Beschrifte die Pfeile an den Rauten mit „ja“ und „nein“.

Tipp

Orientiere dich am Programmablaufplan des Notenrechners im Abschnitt über Verzweigungen. Jedes `if` und jedes `elif` wird zu einer Raute. Der „nein“-Pfeil führt jeweils zur nächsten Raute.

Mein Programmablaufplan:

1.4.5 Mehrfachauswahl: `CASE OF`

Oft wird **ein und derselbe Wert** mit mehreren festen Möglichkeiten verglichen. Ein Programm soll zum Beispiel eine Schulnote in Worten ausgeben. Mit `if` und `elif` sieht das so aus:

```
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.")
```

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

Das funktioniert, ist aber umständlich: In jeder Zeile steht wieder `note == ...`. Für solche Fälle gibt es eine eigene Form der Verzweigung, die **Mehrfachauswahl**. Man beschreibt sie unabhängig von einer Programmiersprache oft so:

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

Lies das so: „Im Fall, dass `note` den Wert 1 hat, gib ‚sehr gut' aus. Im Fall 2 …" `OTHERWISE` („andernfalls") ist der **Auffangfall**. Er greift, wenn keiner der genannten Werte passt.

Definition: Mehrfachauswahl

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.

Die Mehrfachauswahl in Python: `match` und `case`

In Python schreibt man die Mehrfachauswahl mit den Schlüsselwörtern `match` („vergleiche") und `case` („Fall"):

```
1  note = int(input("Note (1-6): "))
2  match note:
3      case 1:
4          print("sehr gut")
5      case 2:
6          print("gut")
7      case 3:
8          print("befriedigend")
9      case 4:
10         print("ausreichend")
11     case 5:
12         print("mangelhaft")
13     case 6:
14         print("ungenügend")
15     case _:
16         print("Das ist keine gültige Note.")
```

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

- **Zeile 2:** `match note:` legt fest, **welcher Wert** verglichen wird. Wie bei `if` steht am Ende ein Doppelpunkt, und alles Weitere ist eingerückt.
- **Zeilen 3–14:** Jedes `case` nennt einen festen Wert, dahinter einen Doppelpunkt. Python geht die Fälle von oben nach unten durch. Beim ersten passenden Fall wird der eingerückte Block darunter ausgeführt, die übrigen Fälle werden übersprungen. Hier passt `case 2`.
- **Zeilen 15–16:** `case _` ist der **Auffangfall**. Der Unterstrich `_` passt zu jedem Wert. Dieser Fall entspricht dem `else` bzw. dem `OTHERWISE`. Er muss als **letzter** Fall stehen.

Die Mehrfachauswahl funktioniert genauso mit Texten. Das nutzt du gleich für einen Taschenrechner:

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

AUSGABE

```
Subtraktion
```

Merke

- `CASE OF` eignet sich, wenn **ein Ausdruck mit mehreren festen Werten** verglichen wird. In Python heißt diese Verzweigung `match`/`case`.
- `case _` ist der Auffangfall und entspricht hier `else`. Für Bereiche und frei formulierte Bedingungen ist `if` meist geeigneter.

Wann nimmst du was? Die Mehrfachauswahl passt nur, wenn ein Wert mit **festen** Werten verglichen wird: Noten 1 bis 6, Operatoren `+ - * /`, Menüpunkte `"a"`, `"b"`, `"c"`. Für **Bereiche** wie „unter 6 Jahren“ oder „zwischen 6 und 17“ und für Bedingungen mit `and` und `or` bleibt `if`/`elif`/`else` die richtige Wahl.

Stolperstein

`match` und `case` gibt es erst seit **Python 3.10** (erschienen 2021). Mit einer älteren Version meldet Python bei `match note:` einen `SyntaxError`. Thonny und die meisten aktuellen Umgebungen bringen eine neuere Version mit. Welche Version du hast, zeigt dir das Programm `import sys` und `print(sys.version)` – oder du startest einfach das Beispiel oben.

Ein zweiter häufiger Fehler: `case _` steht nicht am Ende. Dann passt er zu allem, und die Fälle darunter werden nie erreicht. Python meldet in diesem Fall einen `SyntaxError`.

1.4.6 Ein Programm erweitern

Im Abschnitt über Variablen hast du im Auftrag *Einfacher Rechner* ein Programm geschrieben, das zwei Zahlen einliest und mit ihnen rechnet. Eine kurze Version davon sieht so aus:

```
a = float(input("Erste Zahl: "))
b = float(input("Zweite Zahl: "))
print("Summe:", a + b)
```

AUSGABE

```
Erste Zahl: 7
Zweite Zahl: 2
Summe: 9.0
```

Dieses Programm rechnet immer plus. Mit einer Mehrfachauswahl kann die Nutzerin selbst wählen, welche Rechnung ausgeführt wird.

Arbeitsauftrag 43: Taschenrechner mit CASE OF

Schreibe einen Taschenrechner, der zwei Zahlen sowie einen Operator (`+`, `-`, `*`, `/`) abfragt. Setze die Mehrfachauswahl in Python mit `match`/`case` um. Fange den Fall ab, dass durch 0 geteilt werden soll, und gib bei einem unbekannten Operator eine verständliche Meldung aus. Teste dein Programm mit allen vier Operatoren, mit einer Division durch 0 und mit einem unbekannten Operator.

So könnte es aussehen:

AUSGABE

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

Tipp

Gehe in drei Schritten vor:

1. Lies die erste Zahl, den Operator und die zweite Zahl ein. Der Operator ist ein Text und wird **nicht** umgewandelt.
2. Unterscheide die Operatoren mit `match op:` und je einem `case "+":`, `case "-":` und so weiter. `case _` übernimmt die Rolle des Auffangfalls für alle unbekannten Operatoren.
3. Im Fall `"/"` brauchst du zusätzlich eine **verschachtelte** Verzweigung mit `if`: Nur wenn die zweite Zahl ungleich 0 ist, darf geteilt werden.

Ohne diese Prüfung bricht das Programm bei einer Division durch 0 mit einem `ZeroDivisionError` (Division durch null) ab.

Arbeitsauftrag 44 (Zusatz): Eintrittspreis

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

Schreibe ein Programm, das zuerst das Alter abfragt. **Nur** bei Personen ab 18 Jahren soll es zusätzlich fragen, ob ein Ausweis vorliegt (`ja`/`nein`). Dann gibt es den Eintrittspreis aus. Lege vorher eine Tabelle mit Testfällen an, die alle Zweige und alle Grenzfälle abdeckt, und teste dein Programm damit.

Meine Testfälle (Eingabe, erwartete Ausgabe):

1.4.7 Weitere Übungen

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.

Arbeitsauftrag 45: 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")
```

Meine Vorhersagen:

Arbeitsauftrag 46: 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.")
```

Arbeitsauftrag 47: 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`.

```
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.")
```

Arbeitsauftrag 48: 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")
```

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?

Arbeitsauftrag 49: 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.")
```

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`.

Die Fehler:

Arbeitsauftrag 50: 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	

Begründungen zu a) bis e):

Arbeitsauftrag 51: 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.
```

Arbeitsauftrag 52: 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.

Meine Testfälle (Eingaben, erwartete Ausgabe):

1.4.8 Das Wichtigste in Kürze

- Bei `if` / `elif` / `else` wird nur der Block der ersten zutreffenden Bedingung ausgeführt. Deshalb kommt es auf die Reihenfolge an: der speziellste Fall zuerst.
- Python meldet immer nur den **ersten** Fehler. Behebe Fehler einzeln, von oben nach unten, und starte das Programm danach jeweils neu.
- Ein **Testfall** besteht aus einer Eingabe und der erwarteten Ausgabe. Teste jeden Zweig mindestens einmal und alle **Grenzfälle**.
- Beim Code-Lesen hilft ein Ablaufprotokoll: Welche Bedingung ist wahr, welche Zeile wird als Nächstes ausgeführt?
- Wird **ein Wert** mit mehreren **festen** Werten verglichen, eignet sich die **Mehrfachauswahl** (`CASE OF`). In Python: `match wert:` mit `case …:` für jeden Fall und `case _:` als Auffangfall.
- Mit einer verschachtelten Verzweigung fängt man Sonderfälle ab, zum Beispiel die Division durch 0.

Selbstcheck

- ☐ Ich kann die Fehlermeldungen `expected ':'`, `IndentationError` und `invalid syntax … '=='` erklären und den Fehler beheben.
- ☐ Ich kann die Ausgabe eines Programms mit `if` / `elif` / `else` vorhersagen, ohne es auszuführen.
- ☐ Ich kann erklären, warum bei `elif` die Reihenfolge der Bedingungen wichtig ist.
- ☐ Ich kann zu einem Programm Testfälle für alle Zweige und alle Grenzfälle angeben.
- ☐ Ich kann eine Mehrfachauswahl als `CASE OF` beschreiben und in Python mit `match`/`case` umsetzen.
- ☐ Ich kann einen Taschenrechner mit `match`/`case` schreiben, der eine Division durch 0 abfängt.

1.5 Die for-Schleife: Anweisungen wiederholen

Leitidee

Stell dir vor, dein Programm soll die Zahlen von 1 bis 100 ausgeben. Mit dem, was du bisher kannst, bräuchtest du dafür 100 `print`-Zeilen. Das ist mühsam, und bei einem Tippfehler in Zeile 73 suchst du lange. Viel besser wäre eine Anweisung wie: „Wiederhole das 100-mal." Aus der Blockprogrammierung kennst du dafür den Baustein „wiederhole 10-mal". In Python heißt diese Wiederholung **for-Schleife**. Mit ihr gibst du Zahlenreihen aus, berechnest Summen und zeichnest einfache Muster – mit nur zwei oder drei Zeilen Code.

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.5.1 Die Idee: Wiederholen statt abschreiben

Statt dreimal `print("Hallo!")` untereinanderzuschreiben, kannst du Python sagen: „Mache das dreimal." Das sieht so aus:

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

AUSGABE

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

- **Zeile 1:** Diese Zeile ist der **Schleifenkopf**. Er sagt: „Mache das Folgende dreimal." Was `i` und `range(3)` genau bedeuten, lernst du gleich. Der Kopf endet mit einem **Doppelpunkt**.
- **Zeile 2:** Diese Zeile ist **eingerückt** (4 Leerzeichen). Alles, was eingerückt unter dem Kopf steht, wird wiederholt. Diesen Teil nennt man den **Schleifenrumpf**.

Der Vorteil wird deutlich, wenn du statt `3` die Zahl `100` schreibst: Das Programm bleibt zwei Zeilen lang, gibt aber 100-mal „Hallo!" aus.

Definition: Schleife

Eine **Schleife** ist eine Kontrollstruktur, die einen Block von Anweisungen (den **Schleifenrumpf**) mehrfach hintereinander ausführt. Jede einzelne Ausführung des Schleifenrumpfs heißt **Durchlauf**.

Definition: for-Schleife

Eine **for-Schleife** (auch **Zählschleife**) ist eine Schleife, bei der die Anzahl der Durchläufe schon vor dem Start der Schleife feststeht.

Merke

- Eine **Schleife** wiederholt einen Anweisungsblock mehrfach.
- Die **for-Schleife** wird verwendet, wenn die **Anzahl der Wiederholungen vorher bekannt** ist (z. B. „10-mal“, „für jede Zahl von 1 bis 100“).

1.5.2 Aufbau einer for-Schleife

Die Zählvariable

Im Schleifenkopf steht eine Variable, hier `i`. Sie heißt **Zählvariable** (oder Schleifenvariable). Bei jedem Durchlauf bekommt sie automatisch den nächsten Wert. Du kannst sie im Schleifenrumpf benutzen, zum Beispiel ausgeben:

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

AUSGABE

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

- **Zeile 1:** `range(1, 4)` liefert der Reihe nach die Zahlen `1`, `2` und `3`. Die Zählvariable `i` bekommt im ersten Durchlauf den Wert `1`, im zweiten den Wert `2`, im dritten den Wert `3`.
- **Zeile 2:** Der Schleifenrumpf. Er gibt in jedem Durchlauf „Runde“ und den aktuellen Wert von `i` aus.
- **Zeile 3:** Diese Zeile ist **nicht** eingerückt. Sie gehört also nicht mehr zur Schleife. Sie wird erst ausgeführt, wenn die Schleife ganz fertig ist – und zwar nur einmal.

Du kannst dir die for-Schleife wie eine Läuferin auf der Sportbahn vorstellen: Sie läuft ihre Runden und zählt dabei mit. Die Zahl, die sie gerade zählt, ist der Wert der Zählvariablen. Nach der letzten Runde hört sie auf und geht zur Bank (Zeile 3).

Der Ablauf Schritt für Schritt

Was genau passiert, zeigt ein **Ablaufprotokoll**. Es listet jede ausgeführte Zeile auf und den Wert von `i` **nach** dieser Zeile:

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

Beachte drei Dinge:

- Python springt nach dem Rumpf (Zeile 2) immer wieder **zurück zum Kopf** (Zeile 1). Dort bekommt `i` den nächsten Wert.
- Im Schritt 7 ist Python noch einmal im Kopf. Jetzt liefert `range(1, 4)` aber keine Zahl mehr. Die Schleife ist zu Ende, `i` behält den letzten Wert `3`, und es geht mit Zeile 3 weiter.
- Die Zeile 1 wird viermal ausgeführt, der Rumpf dreimal.

Merke

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

Einrückung: Was gehört zur Schleife?

Wie bei `if` entscheidet in Python die **Einrückung**, welche Anweisungen zusammengehören. Der Schleifenrumpf darf mehrere Zeilen haben. Alle müssen gleich weit eingerückt sein. Vergleiche die beiden Programme. Sie unterscheiden sich nur in der Einrückung der letzten Zeile:

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

AUSGABE

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

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

AUSGABE

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

Im ersten Programm gehören beide `print`-Anweisungen zum Rumpf und werden dreimal ausgeführt. Im zweiten Programm gehört nur die erste dazu. Die letzte Zeile wird erst nach der Schleife ausgeführt, also nur einmal.

Stolperstein

Vergisst du die Einrückung, meldet Python einen Fehler, bevor das Programm überhaupt startet:

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

FEHLERMELDUNG

```
  File "programm.py", line 2
    print(i)
    ^^^^^
IndentationError: expected an indented block after 'for' statement on line 1
```

IndentationError heißt „Einrückungsfehler“. Die Meldung sagt: „Nach der for-Anweisung in Zeile 1 wurde ein eingerückter Block erwartet.“ Die Zeichen ^^^^^ zeigen auf die Zeile 2, die eingerückt sein müsste. Rücke den Rumpf mit 4 Leerzeichen (oder der Tab-Taste) ein.

Fehlt der Doppelpunkt am Ende des Kopfes, kommt diese Meldung:

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

FEHLERMELDUNG

```
SyntaxError: expected ':'
```

Versuch: Probiere es aus

Tippe das Programm mit „Runde“ ab. Ändere die Zahlen in `range(1, 4)` und beobachte, wie sich die Ausgabe ändert. Rücke dann `print("Fertig")` ein. Was passiert jetzt?

1.5.3 Zahlenfolgen mit `range`

`range` heißt auf Deutsch „Bereich“. Es erzeugt eine Folge von ganzen Zahlen, die die Zählvariable nacheinander annimmt. Es gibt drei Formen:

- **`range(stop)`** beginnt bei `0` und endet **vor** `stop`. `range(5)` liefert `0, 1, 2, 3, 4` – genau 5 Zahlen. Diese Form nimmst du, wenn es nur auf die Anzahl der Durchläufe ankommt.
- **`range(start, stop)`** beginnt bei `start` und endet wieder **vor** `stop`. Die Zahl `stop` selbst ist nie dabei.
- **`range(start, stop, schritt)`** zählt mit der **Schrittweite** `schritt` weiter. Mit einer negativen Schrittweite zählt man 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

Merke

- `range(n)` erzeugt die Zahlen `0, 1, ..., n-1`.
- `range(start, stop)` erzeugt die Zahlen von `start` bis `stop-1`.
- `range(start, stop, schritt)` erlaubt eine eigene Schrittweite.

Tipp

Eine Eselsbrücke: Die Zahl `stop` ist wie die Ziellinie beim Laufen. Man zählt bis kurz davor, die Linie selbst gehört nicht mehr dazu. Bei `range(start, stop)` gibt es genau `stop - start` Zahlen.

Beispiel: Zahlen ausgeben und das Einmaleins

Das folgende Programm zeigt zwei Schleifen hintereinander. Überlege zuerst, was es ausgibt. Lies dann weiter.

```
1 for i in range(1, 6):
2     print(i)
3
4 print("---")
5
6 for i in range(1, 11):
7     print("7 *", i, "=", 7 * i)
```

AUSGABE

```
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:** Die erste Schleife gibt die Zahlen von 1 bis 5 aus. Für „1 bis 5“ braucht man `range(1, 6)`, weil der Stoppwert nicht dazugehört.
- **Zeile 4:** Ein Trennstrich. Er steht nicht eingerückt, gehört also zu keiner Schleife und wird einmal ausgegeben.
- **Zeilen 6–7:** Die zweite Schleife läuft mit `i` von 1 bis 10. In jedem Durchlauf wird eine Zeile des Siebener-Einmaleins ausgegeben. `"7 *"` und `"="` sind Texte, `i` und `7 * i` sind Zahlen. Der Wert `7 * i` wird in jedem Durchlauf neu ausgerechnet.
- Beide Schleifen benutzen den Namen `i`. Das ist erlaubt: Die zweite Schleife gibt `i` einfach wieder neue Werte.

Rückwärts zählen

Mit der Schrittweite `-1` zählt die Schleife rückwärts. `for i in range(5, 0, -1):` lässt `i` die Werte `5, 4, 3, 2, 1` annehmen – ein Countdown. Auch hier gehört der Stoppwert `0` nicht mehr dazu.

Stolperstein

Der häufigste Fehler bei `range`: Man vergisst, dass der Stoppwert **nicht** dazugehört. `range(1, 10)` erzeugt die Zahlen von 1 bis **9**, nicht bis 10. Python meldet dabei keinen Fehler – das Programm tut einfach nicht ganz das, was du wolltest. Prüfe deshalb immer die erste und die letzte Zahl der Ausgabe. Für „von 1 bis 10“ schreibst du `range(1, 11)`.

Arbeitsauftrag 53: range-Detektiv

1. Gib an, welche Zahlen die folgenden `range`-Aufrufe erzeugen.
2. Gib zu den Zahlenfolgen in der unteren Tabelle einen passenden `range`-Aufruf an.

Überprüfe deine Antworten am Computer mit einer Schleife, die `i` ausgibt.

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	

Arbeitsauftrag 54: Zahlenreihe

Gib mit einer for-Schleife alle geraden Zahlen von 2 bis 20 aus (Tipp: Schrittweite nutzen!).

1.5.4 Summen bilden

Schleifen sind besonders nützlich, wenn viele Zahlen zusammengezählt werden sollen. Denk an eine Kasse im Supermarkt: Am Anfang steht die Anzeige auf 0. Bei jedem Artikel, der über das Band läuft, kommt sein Preis dazu. Am Ende steht der Gesamtbetrag auf der Anzeige.

Genauso geht man im Programm vor. Man braucht eine Variable, die die Zwischensumme speichert. Sie wird **vor** der Schleife auf `0` gesetzt. **In** der Schleife kommt bei jedem Durchlauf eine Zahl dazu. **Nach** der Schleife steht in ihr das Ergebnis. Eine solche Variable heißt **Summenvariable** (Fachwort: **Akkumulator**, von lateinisch *accumulare* = anhäufen).

Das Programm berechnet die Summe der Zahlen von 1 bis 5:

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

AUSGABE

```
Summe: 15
```

- **Zeile 1:** Die Summenvariable bekommt den Startwert `0` – die Kasse ist leer.
- **Zeile 2:** `i` läuft von 1 bis 5.

- **Zeile 3:** Der aktuelle Wert von `i` wird zur bisherigen Summe addiert. Das Ergebnis wird wieder in `summe` gespeichert. Diese Zeile kennst du schon aus `alter = alter + 1`: Erst wird rechts gerechnet, dann links gespeichert.
- **Zeile 4:** Nicht eingerückt – die Summe wird einmal ausgegeben, wenn die Schleife fertig ist.

Wie die Summe wächst, zeigt das Ablaufprotokoll. Lies vor allem 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

Im Schritt 1 hat `i` noch keinen Wert (–), denn die Schleife hat noch nicht begonnen. Im Schritt 12 liefert `range` keine Zahl mehr, die Schleife endet.

Merke

Eine Summe berechnet man mit einer Summenvariablen in drei Schritten:

1. **vor** der Schleife: `summe = 0`
2. **in** der Schleife: `summe = summe + zahl`
3. **nach** der Schleife: Ergebnis ausgeben

Die Kurzschreibweise +=

Zeilen wie `summe = summe + i` kommen so oft vor, dass es dafür eine Abkürzung gibt: `summe += i`. Lies das als „`summe` wird um `i` erhöht“. Beide Schreibweisen bewirken genau dasselbe.

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`

Im Summenprogramm kannst du also statt Zeile 3 auch `summe += i` schreiben. Du darfst beide Schreibweisen benutzen. Am Anfang ist die lange Form oft leichter zu lesen.

Stolperstein

Die Summenvariable steht in der Schleife. Dann wird sie in jedem Durchlauf wieder auf 0 gesetzt, und alles bisher Gezählte geht verloren:

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

AUSGABE

```
Summe: 5
```

Am Ende steht in `summe` nur noch die letzte Zahl `5`. Python meldet keinen Fehler, das Ergebnis ist einfach falsch. Der Startwert gehört **vor** die Schleife.

Die Ausgabe ist eingerückt. Dann wird nach jedem Durchlauf eine Zwischensumme ausgegeben:

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

AUSGABE

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

Das kann zum Nachprüfen nützlich sein. Wenn nur das Ergebnis gefragt ist, darf `print` aber nicht eingerückt sein.

Arbeitsauftrag 55: Summe berechnen

Berechne mit einer for-Schleife die Summe aller Zahlen von 1 bis 100 (ohne eine fertige Summenfunktion zu verwenden) und gib das Ergebnis aus.

Die Obergrenze eingeben lassen

Die Schleife wird noch nützlicher, wenn die Nutzerin oder der Nutzer die Obergrenze selbst eingibt. Weil `range` nur mit ganzen Zahlen arbeitet, muss die Eingabe mit `int` umgewandelt werden:

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

AUSGABE

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

- **Zeile 1:** Die Obergrenze wird eingelesen und in eine ganze Zahl umgewandelt.
- **Zeile 3:** Damit `n` selbst noch mitgezählt wird, muss der Stoppwert `n + 1` sein.
- **Zeilen 4–5:** wie im Beispiel oben, mit der Kurzschreibweise `+=`.

Stolperstein

Vergisst du die Umwandlung mit `int`, ist `n` eine Zeichenkette. Damit kann `range` nichts anfangen:

```
1 n = input("Bis zu welcher Zahl soll ich addieren? ")
2 for i in range(1, n):
3     print(i)
```

FEHLERMELDUNG

```
TypeError: 'str' object cannot be interpreted as an integer
```

Die Meldung heißt: „Ein Wert vom Typ `str` kann nicht als ganze Zahl verstanden werden.“ Dieselbe Meldung mit `'float'` bekommst du, wenn du `float` statt `int` benutzt. `range` braucht immer **ganze Zahlen**.

Arbeitsauftrag 56: Ablaufprotokoll

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

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

Schritt	Zeile	Anweisung	Wert von `i`	Wert von `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:

1.5.5 Muster ausgeben

Mit Schleifen kann man auch Bilder aus Zeichen zeichnen. Dafür brauchst du ein kleines Hilfsmittel.

Werkzeug: Texte vervielfachen mit *

Multiplizierst du eine Zeichenkette mit einer ganzen Zahl, wird der Text so oft hintereinandergehängt:

```
print("ab" * 3)
print("-" * 20)
```

AUSGABE

```
ababab
--------------------
```

Die Zahl muss eine ganze Zahl (`int`) sein. Sie darf auch eine Variable sein, zum Beispiel die Zählvariable einer Schleife.

Arbeitsauftrag 57 (Zusatz): Sternendreieck

Gib mit einer for-Schleife folgendes Muster aus:

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

So kannst du es sagen

So beschreibst du eine for-Schleife in Fachsprache – vom einfachen zum genauen Satz:

- *„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.“*

1.5.6 Weitere Übungen

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.

Arbeitsauftrag 58: 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):`			

Arbeitsauftrag 59: Runde für Runde ●○○

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

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

Meine Vorhersage:

Arbeitsauftrag 60: 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 `___`.

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

Arbeitsauftrag 61: 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
```

Arbeitsauftrag 62: 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**.

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

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

Die Fehler:

Gymnasium Ottweiler

Arbeitsauftrag 63: 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.

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

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)`		

Ausgabe:

Arbeitsauftrag 64: 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.
```

Arbeitsauftrag 65: 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.

1.5.7 Das Wichtigste in Kürze

- Eine **Schleife** wiederholt einen Anweisungsblock (den **Schleifenrumpf**) mehrfach.
- Die **for-Schleife** nimmt man, wenn die Anzahl der Wiederholungen vorher feststeht.
- Aufbau: `for i in range(...):` – Kopf mit Doppelpunkt, darunter der eingerückte Rumpf. Was nicht eingerückt ist, kommt nach der Schleife.
- Die **Zählvariable** bekommt bei jedem Durchlauf automatisch den nächsten Wert aus `range`.
- `range(n)` liefert `0` bis `n-1`, `range(start, stop)` liefert `start` bis `stop-1`, `range(start, stop, schritt)` zählt mit eigener Schrittweite. Der Stoppwert gehört **nie** dazu.
- Summen: Summenvariable **vor** der Schleife auf 0 setzen, **in** der Schleife erhöhen, **nach** der Schleife ausgeben. `summe += i` ist die Kurzform von `summe = summe + i`.

Selbstcheck

- ☐ Ich kann erklären, was eine Schleife ist und wann man eine for-Schleife verwendet.
- ☐ Ich kann eine for-Schleife mit Doppelpunkt und richtig eingerücktem Rumpf schreiben.
- ☐ Ich kann zu einem `range`-Aufruf angeben, welche Zahlen er erzeugt – und umgekehrt.
- ☐ Ich kann mit einer Schleife eine Summe berechnen und weiß, wo `summe = 0` stehen muss.
- ☐ Ich kann `+=` benutzen und weiß, was es bedeutet.
- ☐ Ich kann zu einer Schleife ein Ablaufprotokoll aufstellen.

1.6 Die while-Schleife: Wiederholen, solange eine Bedingung gilt

Leitidee

Mit der for-Schleife wiederholst du Anweisungen eine feste Anzahl von Malen. Oft weiß man aber vorher nicht, wie viele Wiederholungen nötig sind. Beim „Mensch ärgere dich nicht“ würfelst du, **bis** eine Sechs kommt – das kann einmal dauern oder zehnmal. Ein Geldautomat fragt die PIN ab, **solange** sie falsch ist. Für solche Fälle gibt es die **while-Schleife**. Sie prüft vor jedem Durchlauf eine Bedingung und hört auf, sobald die Bedingung nicht mehr erfüllt ist. In diesem Abschnitt lernst du, wie man sie schreibt, wie man gefährliche Endlosschleifen vermeidet und wann man besser for und wann besser while verwendet.

Lernziele

Nach diesem Abschnitt kannst du

- eine while-Schleife mit Bedingung und eingerücktem Schleifenrumpf schreiben,
- den Ablauf einer while-Schleife in einem Ablaufprotokoll und einem Programmablaufplan 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.

1.6.1 Wiederholen, solange eine Bedingung wahr ist

Das Wort *while* ist englisch und heißt „solange“. Eine while-Schleife sagt: „**Solange** die Bedingung wahr ist, führe den Rumpf aus.“ Die Bedingung kennst du schon von `if`: Es ist ein Ausdruck, der `True` oder `False` ergibt, zum Beispiel `i <= 3` oder `eingabe != "stopp"`.

Das erste Beispiel zählt von 1 bis 3 – das könnte die for-Schleife auch, aber so siehst du gut, wie die while-Schleife arbeitet:

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

AUSGABE

```
1
2
3
Ende
```

- **Zeile 1:** Die Variable `i` bekommt den Startwert `1`. Das musst du bei while selbst erledigen.
- **Zeile 2:** Der Schleifenkopf: `while`, dann die Bedingung, dann ein Doppelpunkt. Vor **jedem** Durchlauf prüft Python, ob `i <= 3` wahr ist. Nur dann wird der Rumpf ausgeführt.
- **Zeilen 3–4:** Der eingerückte Schleifenrumpf. Er gibt `i` aus und erhöht `i` danach um 1. Diese Erhöhung ist wichtig: Ohne sie bliebe `i` immer `1`, und die Bedingung wäre immer wahr.
- **Zeile 5:** Nicht eingerückt – wird ausgeführt, sobald die Bedingung falsch ist.

Das Ablaufprotokoll zeigt, wie Python nach dem Rumpf immer wieder zur Bedingung in Zeile 2 zurückspringt:

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

Im Schritt 11 hat `i` den Wert `4`. Die Bedingung `4 <= 3` ist falsch. Die Schleife **endet**, und Python macht mit Zeile 5 weiter. Am Ende hat `i` also den Wert `4`, obwohl zuletzt `3` ausgegeben wurde.

Definition: while-Schleife

Eine **while-Schleife** (bedingte Schleife) wiederholt ihren Schleifenrumpf, **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.

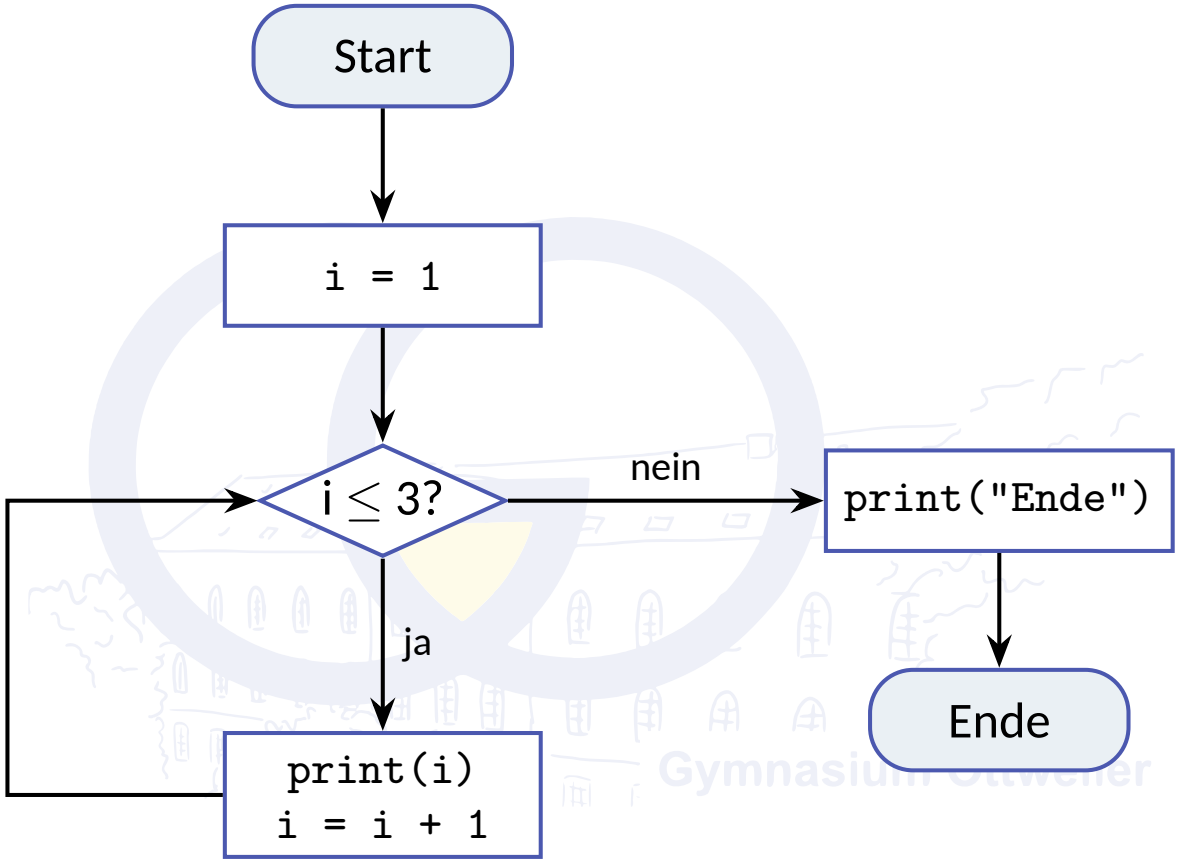
Gymnasium Ottweiler

Merke

- Die **while-Schleife** wird verwendet, wenn die **Anzahl der Wiederholungen vorher nicht bekannt** ist – sie läuft, **solange** eine Bedingung wahr ist.
- Syntax: `while Bedingung:` gefolgt vom eingerückten Anweisungsblock.

Der Ablauf als Bild

Den Ablauf einer Schleife kann man auch zeichnen. In einem **Programmablaufplan** stehen Bedingungen in Rauten und Anweisungen in Rechtecken. Pfeile zeigen, in welcher Reihenfolge es weitergeht. Bei einer Schleife führt ein Pfeil vom Rumpf **zurück zur Bedingung** – daher der Name „Schleife“.



Programmablaufplan der while-Schleife aus dem Beispiel: Solange die Bedingung in der Raute wahr ist (ja), wird der Rumpf ausgeführt und danach die Bedingung erneut geprüft

1.6.2 Beispiel: Zahlen addieren, bis 0 eingegeben wird

Jetzt ein Beispiel, bei dem die for-Schleife nicht weiterhilft. Das Programm soll beliebig viele Zahlen addieren. Wie viele es sind, entscheidet die Nutzerin oder der Nutzer beim Eintippen: Die Eingabe `0` beendet die Schleife.

```
1 summe = 0
2 zahl = int(input("Zahl eingeben (0 zum Beenden): "))
3
4 while zahl != 0:
5     summe = summe + zahl
6     zahl = int(input("Zahl eingeben (0 zum Beenden): "))
7
8 print("Summe aller Eingaben:", summe)
```

AUSGABE

```
Zahl eingeben (0 zum Beenden): 4
Zahl eingeben (0 zum Beenden): 7
Zahl eingeben (0 zum Beenden): 2
Zahl eingeben (0 zum Beenden): 0
Summe aller Eingaben: 13
```

- **Zeile 1:** Die Summenvariable startet bei 0 – genau wie bei der for-Schleife.
- **Zeile 2:** Die **erste** Zahl wird schon **vor** der Schleife eingelesen. Sonst könnte Python die Bedingung in Zeile 4 gar nicht prüfen, denn `zahl` hätte noch keinen Wert.
- **Zeile 4:** Solange die eingegebene Zahl nicht 0 ist, wird der Rumpf ausgeführt.
- **Zeile 5:** Die Zahl wird zur Summe addiert.
- **Zeile 6:** Die **nächste** Zahl wird eingelesen. Danach springt Python zurück zu Zeile 4 und prüft die neue Zahl. Diese Zeile sorgt dafür, dass sich die Bedingung ändern kann.
- **Zeile 8:** Nach der Schleife wird die Summe ausgegeben.

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

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

Im Schritt 12 ist `zahl` gleich `0`. Die Bedingung `0 != 0` ist falsch, die Schleife endet. Die `0` selbst wird nicht mehr addiert (das würde aber auch nichts ändern).

Tipp

Das Muster aus diesem Beispiel brauchst du immer wieder, wenn eine Schleife von Eingaben abhängt:

1. **vor** der Schleife die erste Eingabe einlesen,
2. im Kopf die Eingabe prüfen,
3. **am Ende** des Rumpfs die nächste Eingabe einlesen.

Versuch: Probiere es aus

Starte das Programm und gib gleich als Erstes `0` ein. Was passiert? Erkläre mit der Definition der while-Schleife, warum der Rumpf kein einziges Mal ausgeführt wird.

1.6.3 Endlosschleifen

In der while-Schleife bist **du** dafür verantwortlich, dass die Bedingung irgendwann falsch wird. Vergisst du das, hört die Schleife nie auf. Hier soll ein Countdown von 3 herunterzählen. Findest du den Fehler?

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

AUSGABE

```
3
4
5
6
7
...
```

In Zeile 4 wird `zahl` erhöht statt verringert. Die Zahl wird also immer größer, und `zahl > 0` bleibt für immer wahr. Das Programm gibt endlos Zahlen aus und erreicht `print("Start!")` nie. Richtig wäre `zahl = zahl - 1`.

Definition: Endlosschleife

Eine **Endlosschleife** ist eine Schleife, deren Bedingung nie falsch wird. Sie **terminiert nicht**, das heißt: Sie endet nie von selbst.

Woran erkennst du eine Endlosschleife? Das Programm wird nicht fertig. Entweder läuft die Ausgabe immer weiter, oder es passiert scheinbar gar nichts mehr (die Schleife läuft dann ohne Ausgabe). In Jupyter steht links neben deinem Programm lange ein Sternchen `[*]`.

Wie brichst du sie ab? In Jupyter klickst du oben auf die Stopp-Schaltfläche (ein kleines Quadrat). In Thonny klickst du auf den roten Stopp-Knopf. In einem Terminal drückst du `Strg` + `C`. Python bricht dann mit dieser Meldung ab:

FEHLERMELDUNG

```
KeyboardInterrupt
```

KeyboardInterrupt heißt „Unterbrechung über die Tastatur“. Das ist kein Fehler in deinem Code, sondern nur die Bestätigung, dass du das Programm angehalten hast. Den eigentlichen Fehler musst du danach selbst suchen.

Merke

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

Stolperstein

So findest du die Ursache einer Endlosschleife:

1. Welche Variablen stehen in der Bedingung?
2. Wird mindestens eine davon **im Rumpf** verändert? Typische vergessene Zeilen sind `i = i + 1` oder ein neues `input(...)` am Ende des Rumpfs.
3. Wird sie in die **richtige Richtung** verändert, sodass die Bedingung irgendwann falsch wird?
4. Steht die Änderung wirklich **eingerückt** im Rumpf und nicht erst nach der Schleife?

1.6.4 Eingaben prüfen: Wiederholen, bis die Eingabe stimmt

Eine typische Aufgabe für die while-Schleife: Ein Programm soll so lange nachfragen, bis eine **gültige** Eingabe kommt. Wie oft sich jemand vertippt, weiß man vorher nicht. Das folgende Programm akzeptiert nur Schulnoten von 1 bis 6:

```
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)
```

AUSGABE

```
Gib deine Note ein (1 bis 6): 8
Ungültig! Erlaubt sind nur 1 bis 6.
Gib deine Note ein (1 bis 6): 0
Ungültig! Erlaubt sind nur 1 bis 6.
Gib deine Note ein (1 bis 6): 2
Gespeichert: Note 2
```

- **Zeile 1:** Die erste Eingabe wird vor der Schleife eingelesen.
- **Zeile 2:** Die Bedingung beschreibt eine **ungültige** Eingabe: kleiner als 1 **oder** größer als 6. Solange das zutrifft, wird wiederholt.
- **Zeilen 3–4:** Hinweis ausgeben und neu fragen. Die neue Eingabe überschreibt die alte.
- **Zeile 5:** Hier kommt das Programm nur an, wenn die Note gültig ist.

Tipp

Beim Formulieren der Bedingung hilft die Frage: „**Wann muss ich noch einmal fragen?**“ Die Antwort ist die Bedingung der while-Schleife. Hier: „wenn die Note kleiner als 1 oder größer als 6 ist“.

1.6.5 Zufallszahlen

Bei Spielen braucht man oft den Zufall: einen Würfelwurf oder eine geheime Zahl, die sich das Programm selbst ausdenkt. Dafür gibt es in Python ein Zusatzmodul.

Werkzeug: Zufallszahlen mit random.randint

Das **Modul** `random` ist eine Sammlung von Befehlen rund um den Zufall. Damit du es benutzen kannst, schreibst du **einmal am Anfang** des Programms `import random`. Danach liefert `random.randint(a, b)` eine zufällige ganze Zahl von `a` bis `b`. Anders als bei `range` gehören hier **beide** Grenzen dazu: `random.randint(1, 6)` liefert 1, 2, 3, 4, 5 oder 6 – wie ein Würfel.

Das folgende Programm würfelt so lange, bis eine Sechs fällt, und zählt die Würfe. Wie oft gewürfelt wird, entscheidet der Zufall – ein klarer Fall für while:

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

So könnte es aussehen (bei jedem Start kommt etwas anderes heraus):

AUSGABE

```
Gewürfelt: 4
Gewürfelt: 1
Gewürfelt: 6
Endlich eine Sechs! Anzahl der Würfe: 3
```

- **Zeile 1:** Das Modul `random` wird geladen.
- **Zeilen 3–5:** Der erste Wurf wird vor der Schleife ausgeführt, gezählt und ausgegeben.
- **Zeile 6:** Solange keine Sechs gewürfelt wurde, geht es weiter.
- **Zeilen 7–9:** Neu würfeln, den Zähler um 1 erhöhen, den Wurf ausgeben. Durch das neue Würfeln kann sich die Bedingung ändern.
- **Zeile 10:** Nach der Schleife steht fest: Der letzte Wurf war eine Sechs.

So kannst du es sagen

So beschreibst du eine while-Schleife in Fachsprache – vom einfachen zum genauen Satz:

- *„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.“*

1.6.6 for oder while?

Viele Aufgaben kann man mit beiden Schleifenarten lösen. Das Zählen von 1 bis 3 sieht mit for und while so aus:

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

AUSGABE

```
1
2
3
```

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

AUSGABE

```
1
2
3
```

Beide Programme geben dasselbe aus. Die for-Schleife ist aber kürzer und sicherer: Startwert, Bedingung und Erhöhung stecken alle in `range(1, 4)`. Bei der while-Schleife musst du an alle drei selbst denken – vergisst du die Erhöhung, entsteht eine Endlosschleife. Umgekehrt eignet sich die for-Schleife nicht für Aufgaben wie „würfle bis zur Sechs“, weil `range` eine feste Anzahl braucht.

Bei der Wahl hilft eine einzige Frage: **Steht beim Start der Schleife fest, wie oft wiederholt wird?** Das ist auch dann der Fall, wenn die Anzahl vorher eingegeben wurde, zum Beispiel bei „Summe von 1 bis n“.

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
Ziffern einer Zahl einzeln abarbeiten	`while`	so lange, bis keine Ziffer mehr übrig ist

Merke

Faustregel: Ist die Anzahl der Durchläufe vorher bekannt → `for`. Hängt das Ende von einer Bedingung oder Eingabe ab → `while`.

Arbeitsauftrag 66: for oder while?

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) 20-mal „Hallo“ ausgeben	
b) Wörter einlesen, bis „stopp“ eingegeben wird	
c) das Einmaleins einer eingegebenen Zahl ausgeben	
d) ein Guthaben von 100 € wächst jedes Jahr um 5 %: Nach wie vielen Jahren sind es mindestens 200 €?	

Begründungen zu a) bis d):

Gymnasium Ottweiler

1.6.7 Übungen

Arbeitsauftrag 67: Zahl erraten

Lege eine geheime Zahl im Code fest (z. B. `geheim = 42`). Die Nutzerin oder der Nutzer soll so lange raten dürfen, bis die richtige Zahl eingegeben wurde. Gib nach jedem Versuch „zu hoch“ oder „zu niedrig“ aus.

Tipp

Nutze das Muster aus dem Abschnitt über das Addieren von Eingaben: erste Eingabe vor der Schleife, Bedingung „Zahl ist noch nicht richtig“, neue Eingabe am Ende des Rumpfs. Für „zu hoch“ oder „zu niedrig“ brauchst du im Rumpf eine Verzweigung mit `if` und `else`.

Arbeitsauftrag 68: Ratespiel mit Zufallszahl

Erweitere dein Programm aus dem Auftrag *Zahl erraten*:

1. Die geheime Zahl soll mit `random.randint` zufällig zwischen 1 und 100 gewählt werden.
2. Das Programm soll die Versuche zählen und am Ende ausgeben, wie viele Versuche gebraucht wurden.

Schreibe nur die Zeilen auf, die du ergänzt oder änderst, und gib jeweils an, wo sie stehen.

Arbeitsauftrag 69: Quersumme

Berechne mit einer while-Schleife die Quersumme einer eingegebenen Zahl (z. B. ist die Quersumme von 384 gleich 3 + 8 + 4 = 15).

Tipp: `zahl % 10` liefert die letzte Ziffer, `zahl // 10` entfernt sie.

Tipp

Probiere die beiden Rechenzeichen zuerst einzeln aus: `384 % 10` ergibt `4` (der Rest beim Teilen durch 10), `384 // 10` ergibt `38`. Wiederholst du das mit `38`, bekommst du `8` und `3`. Und mit `3` bekommst du `3` und `0`. Bei `0` ist keine Ziffer mehr übrig – das ist deine Abbruchbedingung.

Arbeitsauftrag 70 (Zusatz): Endlosschleife erkennen

Der folgende Code enthält eine Endlosschleife. Finde den Fehler und korrigiere ihn. Ändere **nicht** die Bedingung `i < 10`, sondern behebe den eigentlichen Fehler im Schleifenrumpf.

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

Die Fehler:

1.6.8 Weitere Übungen

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.

Arbeitsauftrag 71: 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.

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

Gymnasium Ottweiler

Meine Antworten:

Arbeitsauftrag 72: 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`.

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

Arbeitsauftrag 73: 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).

Arbeitsauftrag 74: 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
```

c)

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

d)

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

Meine Entscheidungen und Begründungen:

Arbeitsauftrag 75: 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**.

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

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`.

Die Fehler:

Arbeitsauftrag 76: 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.

```
1 zahl = 4096
2 stellen = 0
3 while zahl > 0:
4     zahl = zahl // 10
5     stellen = stellen + 1
6 print("Stellen:", stellen)
```

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		

Ausgabe:

Arbeitsauftrag 77: 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.

Begründung:

Arbeitsauftrag 78: 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.

Zu 2.:

1.6.9 Das Wichtigste in Kürze

- Die **while-Schleife** wiederholt ihren Rumpf, **solange** die Bedingung wahr ist. Die Bedingung wird vor jedem Durchlauf geprüft.
- Aufbau: `while Bedingung:` – darunter der eingerückte Rumpf.
- Startwert und Veränderung der Variablen in der Bedingung musst du selbst programmieren. Ändert sich die Bedingung nie, entsteht eine **Endlosschleife**. Abbrechen mit der Stopp-Schaltfläche oder `Strg` + `C`.
- Bei Eingaben gilt das Muster: erste Eingabe vor der Schleife, nächste Eingabe am Ende des Rumpfs.
- `import random` und `random.randint(a, b)` liefern eine zufällige ganze Zahl von `a` bis einschließlich `b`.
- Steht die Anzahl der Durchläufe vorher fest → `for`. Hängt das Ende von einer Bedingung oder Eingabe ab → `while`.

Selbstcheck

- ☐ Ich kann eine while-Schleife schreiben und erklären, wann sie terminiert.
- ☐ Ich kann zu einer while-Schleife ein Ablaufprotokoll und einen Programmablaufplan erstellen.
- ☐ Ich kann eine Endlosschleife erkennen, abbrechen und den Fehler finden.
- ☐ Ich kann eine Eingabe so lange wiederholen lassen, bis sie gültig ist.
- ☐ Ich kann mit `random.randint` Zufallszahlen erzeugen.
- ☐ Ich kann begründen, ob für eine Aufgabe eine for- oder eine while-Schleife besser passt.

Gymnasium Ottweiler

1.7 Übung: for- oder while-Schleife?

Leitidee

Du kennst jetzt zwei Arten von Schleifen: die for-Schleife und die while-Schleife. In diesem Abschnitt übst du, beide sicher einzusetzen. Dabei geht es nicht nur um das Programmieren selbst. Bei jeder Aufgabe überlegst du **zuerst**, welche Schleifenart besser passt, und begründest deine Wahl. Erst dann schreibst du das Programm. Diese Entscheidung zu treffen, ist eine wichtige Fähigkeit beim Programmieren – sie macht Programme kürzer und schützt vor Endlosschleifen.

Lernziele

Nach diesem Abschnitt kannst du

- 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,
- deine Programme mit sinnvoll gewählten Testfällen überprüfen.

1.7.1 Das brauchst du

Die beiden Schleifenarten im Vergleich:

	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

So sehen die beiden Grundformen aus:

```
1 for i in range(1, 4):
2     print("Durchlauf", i)
3
4 zahl = 1
5 while zahl < 10:
6     print("zahl ist", zahl)
7     zahl = zahl * 2
```

AUSGABE

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

Merke

- **for-Schleife:** Die Anzahl der Durchläufe steht vorher fest.
- **while-Schleife:** Das Ende hängt von einer Bedingung ab, die Anzahl ist vorher unbekannt.

Bei jeder Aufgabe unten gilt: Überlege zuerst, welche Schleifenart passt, bevor du programmierst!

Zwei Muster brauchst du immer wieder:

- **Zusammenzählen:** Summenvariable **vor** der Schleife auf `0` setzen, **in** der Schleife erhöhen (`summe = summe + zahl` oder `summe += zahl`), **nach** der Schleife ausgeben.
- **Eingaben wiederholen:** erste Eingabe **vor** der while-Schleife, Bedingung prüft die Eingabe, nächste Eingabe **am Ende** des Rumpfs.

Tipp

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

1.7.2 Aufträge

Arbeitsauftrag 79: Programm lesen

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

1. Gib an, was das Programm ausgibt.
2. Gib an, wie oft der Schleifenrumpf ausgeführt wird.
3. Begründe, warum hier eine while-Schleife verwendet wird.

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

Deine Antworten:

Fakultät

In der Mathematik schreibt man $n!$ (gesprochen „n Fakultät“) für das Produkt aller ganzen Zahlen von 1 bis n. Zum Beispiel ist $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$ und $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. 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.

Arbeitsauftrag 80: Fakultät

Berechne die Fakultät einer eingegebenen Zahl n ($n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$). Welche Schleifenart passt hier besser? Begründe deine Wahl, bevor du programmierst.

Tipp

Die Fakultät berechnest du fast wie eine Summe – nur wird multipliziert statt addiert. Überlege genau, mit welchem Wert die Produktvariable **vor** der Schleife starten muss. Was passiert, wenn du sie wie eine Summe mit 0 beginnen lässt?

Begründung:

Gymnasium Ottweiler

Arbeitsauftrag 81: Passwortabfrage

Frage so lange nach einem Passwort, bis das korrekte Passwort `informatik10` eingegeben wurde. Gib danach „Zugang gewährt“ aus. Welche Schleifenart passt hier? Begründe.

Tipp

Das Passwort ist ein Text. Die Eingabe musst du deshalb **nicht** umwandeln. Texte vergleichst du wie Zahlen mit `==` und `!=`. Das richtige Passwort steht im Programm in Anführungszeichen.

Begründung:

Primzahl-Test

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

Ob eine Zahl durch eine andere teilbar ist, prüfst du mit dem Rest-Operator `%`: Ist `zahl % teiler == 0`, geht die Division ohne Rest auf.

Arbeitsauftrag 82: Primzahl-Test

Prüfe, ob eine eingegebene Zahl eine Primzahl ist. Tipp: Teste, ob sie durch eine Zahl zwischen 2 und sich selbst minus 1 teilbar ist.

Tipp

Arbeite mit einer Variablen, die sich ein Ergebnis merkt: Nimm zuerst an, die Zahl **ist** eine Primzahl (`ist_primzahl = True`). Probiere dann mit einer Schleife alle möglichen Teiler aus. Findest du einen, der ohne Rest aufgeht, war die Annahme falsch: `ist_primzahl = False`. Denke an den Sonderfall 1.

Arbeitsauftrag 83 (Zusatz): Sparziel

Auf einem Sparkonto liegen 100 €. Jedes Jahr kommen 5 % Zinsen dazu, das Guthaben wird also mit 1,05 multipliziert. Schreibe ein Programm, das berechnet, nach wie vielen Jahren das Guthaben zum ersten Mal mindestens 200 € beträgt. Gib die Anzahl der Jahre und das Guthaben (auf Cent gerundet) aus. Begründe deine Wahl der Schleifenart.

Begründung:

1.7.3 Weitere Übungen

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.

Arbeitsauftrag 84: 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	

Begründungen zu a) bis f):

Arbeitsauftrag 85: 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)
```

Arbeitsauftrag 86: 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.

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

Meine Antworten:

Arbeitsauftrag 87: 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).

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

Arbeitsauftrag 88: 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)
```

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

Die Fehler:

Arbeitsauftrag 89: 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
```

Begründung:

Gymnasium Ottweiler

Arbeitsauftrag 90: 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`.

Arbeitsauftrag 91: 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.

1.7.4 Das Wichtigste in Kürze

- Vor dem Programmieren entscheiden: Steht die Anzahl der Durchläufe beim Start fest → `for`. Hängt das Ende von einer Bedingung oder Eingabe ab → `while`.
- Summen starten bei 0, Produkte bei 1. Der Startwert gehört **vor** die Schleife.
- Bei while-Schleifen muss sich die Bedingung im Rumpf ändern können, sonst terminiert die Schleife nicht.
- Teste Programme mit Grenzfällen, besonders mit Eingaben, bei denen die Schleife gar nicht oder nur einmal läuft.

Selbstcheck

- ☐ Ich kann begründen, ob für eine Aufgabe eine for- oder eine while-Schleife besser passt.
- ☐ Ich kann die Ausgabe eines Programms mit Schleife vorhersagen und mit einem Ablaufprotokoll überprüfen.
- ☐ Ich kann mit einer Schleife Summen und Produkte berechnen und kenne den richtigen Startwert.
- ☐ Ich kann eine Eingabe mit einer while-Schleife so lange wiederholen lassen, bis sie stimmt.
- ☐ Ich kann zu einem Programm sinnvolle Testfälle angeben, auch Grenzfälle.

1.8 Funktionen: Programme aus Bausteinen

Leitidee

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

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.8.1 Eine eigene Funktion definieren

Stell dir vor, dein Programm soll an mehreren Stellen eine Trennlinie ausgeben. Statt die Linie jedes Mal neu 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()
```

AUSGABE

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

- **Zeile 1:** Das Wort `def` (von englisch *define*, „festlegen“) beginnt eine **Funktionsdefinition**. Danach folgt der Name der Funktion, hier `trennlinie`, dann runde Klammern und ein **Doppelpunkt**. Die Klammern sind leer, weil diese Funktion keine Werte braucht.
- **Zeile 2:** Diese Zeile ist um 4 Leerzeichen **eingerückt**. Alle eingerückten Zeilen unter `def` bilden den **Funktionsrumpf**. Das kennst du von `if` und `for`: Was eingerückt ist, gehört dazu.
- **Zeile 3:** Die leere Zeile ist nicht nötig, macht das Programm aber übersichtlicher. Die Funktion endet, wo die Einrückung endet.
- **Zeile 4:** Hier beginnt das eigentliche Programm, das **Hauptprogramm**. Wichtig: Beim Lesen der Zeilen 1 und 2 hat Python noch **nichts ausgegeben**. Es hat sich nur gemerkt, was `trennlinie` bedeuten soll.
- **Zeile 5:** `trennlinie()` ist ein **Funktionsaufruf**. Jetzt führt Python den Funktionsrumpf aus und gibt die Linie aus.

- **Zeilen 6–7:** Noch eine Ausgabe und noch ein Aufruf. Die Linie erscheint ein zweites Mal, obwohl sie nur einmal im Programm steht.

Du kannst dir eine Funktion wie ein **Rezept** vorstellen. Die Definition schreibt das Rezept auf. Davon wird noch niemand satt. Erst beim Aufruf wird nach dem Rezept gekocht – und das so oft, wie man möchte.

Definition: Funktion

Eine **Funktion** ist eine Folge von Anweisungen, die unter einem Namen zusammengefasst ist. Über diesen Namen kann man sie beliebig oft aufrufen, also ausführen lassen.

Merke

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

Was `parameter1` und `parameter2` in den Klammern bedeuten, lernst du gleich im Abschnitt über Parameter. Bei `trennlinie()` bleiben die Klammern leer.

Stolperstein

Vergisst du die Einrückung, meldet Python einen **IndentationError** („Einrückungsfehler“):

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

FEHLERMELDUNG

```
  File "programm.py", line 2
    print("--------------------")
    ^^^^^
IndentationError: expected an indented block after function definition on line 1
```

Die letzte Zeile heißt: „Nach der Funktionsdefinition in Zeile 1 wurde ein eingerückter Block erwartet.“ Gefunden hat Python das Problem in Zeile 2 (*line 2*). Rücke diese Zeile mit 4 Leerzeichen (oder der Tabulatortaste) ein. Fehlt der Doppelpunkt am Ende von Zeile 1, kommt ein `SyntaxError: expected ':'`.

Stolperstein

Eine Funktion muss **definiert sein, bevor** sie aufgerufen wird, denn Python liest das Programm von oben nach unten. Steht `trennlinie()` vor dem `def`, meldet Python `NameError: name 'trennlinie' is not defined`. Schreibe deshalb alle Funktionsdefinitionen an den Anfang des Programms. Und vergiss beim Aufruf die Klammern nicht: `trennlinie` ohne `()` ruft die Funktion nicht auf.

1.8.2 Parameter: der Funktion Werte mitgeben

Die Trennlinie sieht immer gleich aus. Oft soll eine Funktion aber mit **verschiedenen Werten** arbeiten. Eine Begrüßung zum Beispiel soll jedes Mal einen anderen Namen enthalten. Dafür schreibst du in die Klammern hinter dem Funktionsnamen einen **Parameter**:

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

AUSGABE

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

- **Zeile 1:** In den Klammern steht jetzt `name`. Das ist ein **Parameter**: eine Variable der Funktion, die ihren Wert erst beim Aufruf bekommt. Er ist wie ein Platzhalter, eine noch leere Schachtel.
- **Zeile 2:** Im Funktionsrumpf benutzt du `name` wie jede andere Variable.
- **Zeile 4:** Beim Aufruf steht in den Klammern der Wert `"Alex"`. Diesen Wert nennt man **Argument**. Python legt ihn in den Parameter, als stünde dort `name = "Alex"`, und führt dann den Rumpf aus.
- **Zeile 5:** Beim zweiten Aufruf bekommt `name` den Wert `"Sam"`.

Eine Funktion darf auch **mehrere Parameter** haben. Sie werden durch Kommas getrennt. Beim Aufruf werden die Argumente **der Reihe nach** auf die Parameter verteilt: das erste Argument in den ersten Parameter, das zweite in den zweiten.

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

AUSGABE

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

In Zeile 6 sind die Argumente vertauscht. Python bemerkt das nicht: `gruss` bekommt `"Sam"`, `name` bekommt `"Moin"`. Achte deshalb beim Aufruf auf die Reihenfolge.

Definition: Parameter und Argument

Ein **Parameter** ist eine Variable in der Funktionsdefinition, die beim Aufruf einen Wert bekommt. Der Wert, der beim Aufruf übergeben wird, heißt **Argument**.

Merke

Parameter sind die „Eingabewerte“ einer Funktion – beim Aufruf werden ihnen konkrete **Argumente** übergeben, und zwar in derselben Reihenfolge.

Stolperstein

Die Anzahl der Argumente muss zur Anzahl der Parameter passen:

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

FEHLERMELDUNG

```
TypeError: gruesse() missing 1 required positional argument: 'name'
```

Übersetzt: „Der Funktion `gruesse()` fehlt 1 benötigtes Argument: `name`." Gibst du zu viele Argumente an, z. B. `gruesse("Alex", "Sam")`, heißt es: `takes 1 positional argument but 2 were given` – „nimmt 1 Argument, aber 2 wurden übergeben".

Versuch: Probiere es aus

Rufe `gruesse` mit deinem eigenen Namen auf. Was passiert bei `gruesse(15)`? Lies die Fehlermeldung und erkläre sie mit dem, was du über das Verketten mit `+` weißt.

1.8.3 Rückgabewert: die Funktion liefert ein Ergebnis

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

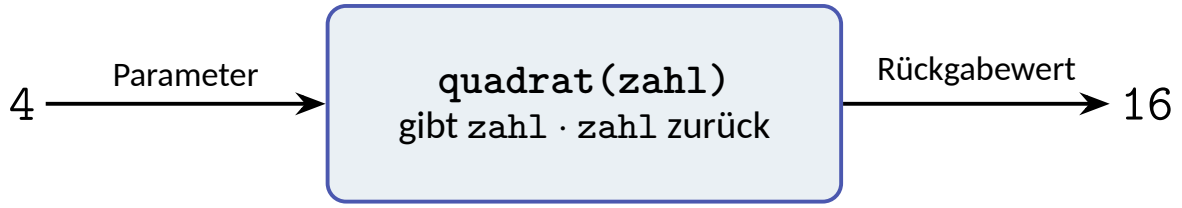
```
1 def quadrat(zahl):
2     ergebnis = zahl * zahl
3     return ergebnis
4
5 print(quadrat(4))
6 print(quadrat(7))
7
8 x = quadrat(10)
9 print("x ist jetzt:", x)
```

AUSGABE

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

- **Zeile 1:** Die Funktion `quadrat` hat einen Parameter `zahl`.
- **Zeile 2:** Im Rumpf wird `zahl * zahl` ausgerechnet und in `ergebnis` gespeichert.
- **Zeile 3:** `return ergebnis` gibt den Wert von `ergebnis` an die Stelle zurück, an der die Funktion aufgerufen wurde. Diesen Wert nennt man **Rückgabewert**.
- **Zeile 5:** Lies die Zeile von innen nach außen. Zuerst wird `quadrat(4)` ausgeführt und liefert `16`. Der Aufruf wird dann durch diesen Wert ersetzt – es steht also gewissermaßen `print(16)` da.
- **Zeile 6:** Genauso: `quadrat(7)` liefert `49`, das ausgegeben wird.
- **Zeile 8:** Diesmal wird der Rückgabewert `100` in der Variablen `x` gespeichert.
- **Zeile 9:** `x` wird ausgegeben.

Eine Funktion mit Parametern und Rückgabewert funktioniert wie eine **Maschine**: Oben wirft man etwas hinein, die Maschine arbeitet, und unten kommt ein Ergebnis heraus.



Die Funktion `quadrat` als Maschine: Das Argument 4 geht als Parameter hinein, der Rückgabewert 16 kommt heraus

Definition: Rückgabewert

Der **Rückgabewert** ist das Ergebnis, das eine Funktion mit `return` an die Stelle zurückgibt, an der sie aufgerufen wurde.

Weil ein Aufruf wie `quadrat(3)` am Ende einfach für einen Wert steht, kannst du ihn überall einsetzen, wo ein Wert stehen darf – in Rechnungen, Bedingungen oder als Argument einer anderen Funktion:

```
1 def quadrat(zahl):
2     ergebnis = zahl * zahl
3     return ergebnis
4
5 summe = quadrat(3) + quadrat(4)
6 print("3*3 + 4*4 =", summe)
7 if quadrat(5) > 20:
8     print("5*5 ist größer als 20.")
```

AUSGABE

```
3*3 + 4*4 = 25
5*5 ist größer als 20.
```

Wichtig: Mit `return` **endet die Funktion sofort**. Steht danach im Rumpf noch etwas, wird es nicht mehr ausgeführt.

Ein Rückgabewert muss keine Zahl sein. Diese Funktion setzt einen Gruß zusammen und gibt ihn als Zeichenkette zurück:

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

AUSGABE

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

Merke

- 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).

Arbeitsauftrag 92: Vorhersage

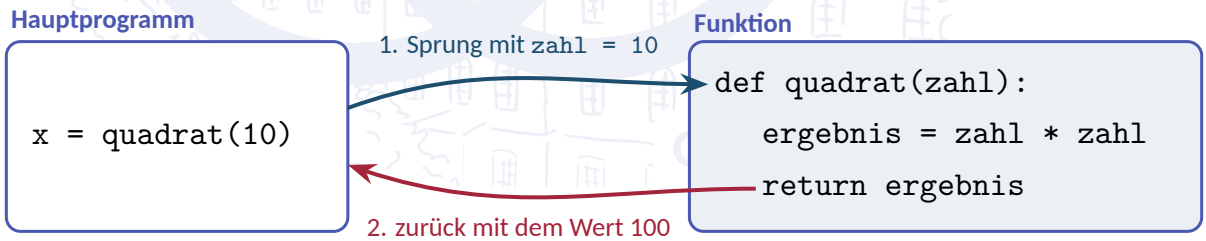
Lies das Programm, **ohne** es auszuführen. Schreibe auf, was es ausgibt. Überprüfe deine Vorhersage danach am Computer.

```
1 def verdopple(zahl):
2     return 2 * zahl
3
4 a = verdopple(3)
5 b = verdopple(a) + 1
6 print(a, b)
7 print(verdopple(verdopple(5)))
```

Meine Vorhersage:

1.8.4 Was beim Aufruf passiert

Ohne Funktionen läuft ein Programm einfach von oben nach unten. Ein Funktionsaufruf unterbricht diese Reihenfolge: Python **springt** in die Funktion, führt ihren Rumpf aus und **springt** mit dem Rückgabewert **zurück** an die Stelle des Aufrufs. Dort geht es weiter.



Beim Aufruf `quadrat(10)` springt Python in die Funktion, der Parameter `zahl` bekommt den Wert 10. Mit `return` springt Python zurück, und der Aufruf wird durch den Rückgabewert 100 ersetzt

Verfolge den Ablauf an diesem Programm:

```
1 def quadrat(zahl):
2     ergebnis = zahl * zahl
3     return ergebnis
4
5 x = quadrat(10)
6 y = quadrat(3) + 1
7 print(x, y)
```

AUSGABE
```
100 10
```

1. **Zeile 1:** Python liest die Definition und merkt sich die Funktion. Die Zeilen 2 und 3 werden jetzt **nicht** ausgeführt.
2. **Zeile 5:** Aufruf `quadrat(10)`. Python springt in die Funktion, der Parameter `zahl` bekommt den Wert `10`. Zeile 2 rechnet `ergebnis = 100`, Zeile 3 gibt `100` zurück. Zurück in Zeile 5 bekommt `x` den Wert `100`.
3. **Zeile 6:** Aufruf `quadrat(3)`. Wieder geht es in die Funktion, diesmal mit `zahl = 3`. Der Rückgabewert ist `9`. Zurück in Zeile 6 wird `9 + 1 = 10` gerechnet und in `y` gespeichert.
4. **Zeile 7:** Ausgabe von `x` und `y`.

Im Ablaufprotokoll des Hauptprogramms sieht es so aus, als würde Zeile 5 in einem einzigen Schritt erledigt:

Schritt	Zeile	Anweisung	x	y
1	1	`def quadrat(zahl):`	–	–
2	5	`x = quadrat(10)`	100	–
3	6	`y = quadrat(3) + 1`	100	10
4	7	`print(x, y)`	100	10

In Wirklichkeit läuft bei jedem Aufruf die Funktion einmal ab. Das Ablaufprotokoll **innerhalb der Funktion** zeigt die beiden Durchläufe:

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

Die Variablen `zahl` und `ergebnis` gibt es nur, solange die Funktion läuft. Man nennt sie **lokale Variablen**. Von außen kann man nicht auf sie zugreifen: Schreibt man im Programm oben am Ende noch `print(ergebnis)`, meldet Python `NameError: name 'ergebnis' is not defined`.

Aus der Maschine kommt nur heraus, was mit `return` zurückgegeben wird. Genau deshalb braucht man `return`: Mit dem, was du bisher kennst, ist es der einzige Weg, auf dem das Ergebnis aus der Funktion zum Hauptprogramm gelangt.

1.8.5 `print` oder `return`?

Der häufigste Fehler mit Funktionen: `print` statt `return` zu schreiben. Auf den ersten Blick sieht beides gleich aus, denn in beiden Fällen erscheint am Ende eine Zahl. Vergleiche die beiden Funktionen:

```
1  def quadrat_mit_print(zahl):
2      print(zahl * zahl)
3
4  def quadrat_mit_return(zahl):
5      return zahl * zahl
6
7  a = quadrat_mit_print(4)
8  b = quadrat_mit_return(4)
9  print("a ist", a)
10 print("b ist", b)
```

AUSGABE

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

- **Zeile 7:** `quadrat_mit_print(4)` gibt sofort `16` auf dem Bildschirm aus – das ist die erste Ausgabezeile. Die Funktion hat aber kein `return`. Deshalb gibt sie dem Hauptprogramm nichts zurück, und `a` bekommt den besonderen Wert **`None`** (englisch für „nichts“).
- **Zeile 8:** `quadrat_mit_return(4)` gibt nichts aus. Sie liefert aber `16` zurück, und dieser Wert landet in `b`.
- **Zeilen 9–10:** Jetzt sieht man den Unterschied: In `a` steht `None`, in `b` steht `16`.

Will man mit dem Ergebnis weiterrechnen, bricht das Programm mit der `print`-Variante sogar ab:

```
1  def quadrat_mit_print(zahl):
2      print(zahl * zahl)
3
4  summe = quadrat_mit_print(3) + quadrat_mit_print(4)
```

AUSGABE

```
9
16
```

FEHLERMELDUNG

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

Die Funktion hat zwar `9` und `16` angezeigt, aber beide Aufrufe liefern `None`. Und `None + None` kann Python nicht rechnen. Die Meldung heißt: „Der Operator `+` ist für die Typen `NoneType` und `NoneType` nicht möglich.“ Wenn in einer Fehlermeldung `NoneType` auftaucht, 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

Merke

- Eine Funktion ohne `return` liefert `None` zurück.
- `print` in einer Funktion zeigt einen Wert nur an. Soll das Programm mit dem Ergebnis weiterarbeiten, muss die Funktion es mit `return` zurückgeben.

Tipp

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

Arbeitsauftrag 93: print oder return?

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

```
1 def minuten_zu_sekunden(minuten):
2     print(minuten * 60)
3
4 dauer = minuten_zu_sekunden(5)
5 print("Das sind", dauer, "Sekunden.")
```

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

Ausgabe und Erklärung:

Gymnasium Ottweiler

1.8.6 Funktionen testen

Eine Funktion kann man beliebig oft mit verschiedenen Argumenten aufrufen. Das ist praktisch, um sie zu **testen**: Du rufst sie mit Werten auf, bei denen du das richtige Ergebnis schon kennst, und vergleichst. So gehst du vor:

1. **Vorher** von Hand ausrechnen, was herauskommen muss (erwartetes Ergebnis).
2. Die Funktion mit diesen Werten aufrufen und das tatsächliche Ergebnis ausgeben.
3. Vergleichen. Weicht ein Ergebnis ab, steckt ein Fehler in der Funktion – oder in deiner Rechnung.

Wähle die Testwerte mit Bedacht: einen ganz normalen Wert, aber auch Sonderfälle wie `0`, negative Zahlen oder Kommazahlen. Genau dort verstecken sich Fehler am häufigsten.

```
1 def quadrat(zahl):
2     return zahl * zahl
3
4 print(quadrat(4), "erwartet: 16")
5 print(quadrat(0), "erwartet: 0")
6 print(quadrat(-3), "erwartet: 9")
7 print(quadrat(1.5), "erwartet: 2.25")
```

AUSGABE

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

Alle vier Tests stimmen. Den Vergleich kann Python auch selbst übernehmen: `print(quadrat(-3) == 9)` gibt `True` aus, wenn der Test stimmt, und sonst `False`.

So kannst du es sagen

So beschreibst du, was eine Funktion tut. Als Beispiel dient 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.“*
- *„Eine Funktion liefert je nach Parameter ein spezifisches Ergebnis, zum Beispiel 17 für die Eingabe 4.“*
- *„Der Funktion wird der Wert 4 übergeben. Die Funktion berechnet $4 \cdot 4 + 1$ und gibt 17 zurück.“*

1.8.7 Übungen

So gehst du beim Schreiben einer Funktion vor: Überlege zuerst, **welche Werte** die Funktion braucht (Parameter) und **was sie zurückgeben** soll (Rückgabewert). Schreibe dann den Rumpf. Teste die Funktion zum Schluss mit mehreren Aufrufen, deren Ergebnis du vorher ausgerechnet hast.

Arbeitsauftrag 94: Celsius in Fahrenheit

In den USA misst man Temperaturen in Grad Fahrenheit (°F). Die Umrechnung lautet: °F = °C · 9/5 + 32.

1. Implementiere eine Funktion `celsius_zu_fahrenheit(grad_celsius)`, die eine Temperatur in Grad Celsius entgegennimmt und den Wert in Fahrenheit **zurückgibt**.

2. Teste sie mit mehreren Werten. Zur Kontrolle: Wasser gefriert bei 0 °C = 32 °F und siedet bei 100 °C = 212 °F.

Arbeitsauftrag 95: Maximum von zwei Zahlen

Implementiere eine Funktion `maximum(a, b)`, die die größere von zwei Zahlen zurückgibt. Verwende dafür **nicht** die eingebaute Funktion `max()`, sondern eine Verzweigung. Teste deine Funktion – auch mit zwei gleichen Zahlen.

Arbeitsauftrag 96 (Zusatz): Gerade Zahlen

1. Implementiere eine Funktion `ist_gerade(zahl)`, die `True` zurückgibt, wenn die Zahl gerade ist, und sonst `False`. Der Rückgabewert ist also vom Typ `bool`.
2. Nutze die Funktion anschließend in einer `for`-Schleife, um alle geraden Zahlen von 1 bis 20 auszugeben.

1.8.8 Weitere Übungen

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.

Arbeitsauftrag 97: Fachbegriffe am Beispiel ●○○

Am Kiosk kostet jede Pfandflasche 25 Cent Pfand zusätzlich zum Preis. Lies das Programm:

```
1 def preis_mit_pfand(preis, flaschen):
2     return preis + flaschen * 0.25
3
4 gesamt = preis_mit_pfand(3.5, 6)
5 print(gesamt)
```

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.

Meine Antworten:

Arbeitsauftrag 98: 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))`	

Arbeitsauftrag 99: 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)`		

Arbeitsauftrag 100: 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.

```
1 def flaeche_dreieck(g, h)
2 return g * h / 2
3
4 print(flaeche_dreieck(4))
```

Die drei Fehler:

Arbeitsauftrag 101: 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.

Meine Antworten:

Arbeitsauftrag 102: 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.

Arbeitsauftrag 103: 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.

```
1 def dreifach(zahl):
2     return 3 * zahl
3
4 summe = 0
5 for i in range(1, 4):
6     summe = summe + dreifach(i)
7 print("Summe:", summe)
```

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)`		

Ausgabe:

Arbeitsauftrag 104: 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*.

1.8.9 Das Wichtigste in Kürze

- Eine **Funktion** fasst Anweisungen unter einem Namen zusammen. Ihre **Definition** beginnt mit `def`, der Rumpf ist eingerückt. Die Definition legt nur fest, was die Funktion tut. Ausgeführt wird sie erst beim **Aufruf**.
- **Parameter** sind die Eingabewerte der Funktion. Beim Aufruf bekommen sie der Reihe nach die **Argumente**.
- `return wert` gibt einen **Rückgabewert** zurück und beendet die Funktion. Der Aufruf steht danach für diesen Wert.
- Ohne `return` liefert eine Funktion `None`. `print` in der Funktion zeigt nur an – weiterrechnen kann man damit nicht.
- Beim Aufruf springt Python in die Funktion und mit dem Rückgabewert zurück. Variablen in der Funktion sind **lokal**.
- Funktionen **testet** man mit mehreren Aufrufen, deren Ergebnis man vorher kennt – auch mit Sonderfällen.

Selbstcheck

- ☐ Ich kann eine Funktion mit `def` definieren und aufrufen.
- ☐ Ich kann den Unterschied zwischen Funktionsdefinition und Funktionsaufruf erklären.
- ☐ Ich kann erklären, was ein Parameter, ein Argument und ein Rückgabewert ist.
- ☐ Ich kann eine Funktion schreiben, die mit `return` ein Ergebnis zurückgibt.
- ☐ Ich kann erklären, warum eine Funktion mit `print` statt `return` den Wert `None` liefert.
- ☐ Ich kann den Ablauf eines Funktionsaufrufs Schritt für Schritt beschreiben.
- ☐ Ich kann eine Funktion mit sinnvoll gewählten Testwerten testen.

Gymnasium Ottweiler

1.9 Übung: Funktionen lesen, testen und erweitern

Leitidee

Im Abschnitt über Funktionen hast du gelernt, eigene Funktionen zu schreiben. Jetzt übst du, was Programmiererinnen und Programmierer jeden Tag tun: fremden Code **lesen** und verstehen, **Fehler finden**, Funktionen **testen** und aus kleinen Funktionen ein größeres Programm **zusammensetzen**. Dabei lernst du eine wichtige neue Idee kennen: Eine Funktion darf andere Funktionen aufrufen. So entstehen Programme aus Bausteinen, die man einzeln prüfen kann.

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.9.1 Das brauchst du

Wiederhole kurz das Wichtigste über Funktionen:

```
1 def quadrat(zahl):
2     return zahl * zahl
3
4 print(quadrat(3))
5 print(quadrat(10))
```

AUSGABE

```
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

Neu ist nur eine Kleinigkeit: Im Rumpf einer Funktion darf man auch **andere Funktionen** aufrufen. So kann eine größere Funktion kleinere **Hilfsfunktionen** benutzen:

```
1 def quadrat(zahl):
2     return zahl * zahl
3
4 def summe_der_quadrate(a, b):
5     return quadrat(a) + quadrat(b)
6
7 print(summe_der_quadrate(3, 4))
```

AUSGABE

```
25
```

- **Zeilen 1–2:** die bekannte Funktion `quadrat`.
- **Zeilen 4–5:** Die Funktion `summe_der_quadrate` rechnet selbst nichts mit `*`. Sie ruft zweimal `quadrat` auf und addiert die beiden Rückgabewerte.
- **Zeile 7:** Der Aufruf `summe_der_quadrate(3, 4)` springt in die Funktion mit `a = 3` und `b = 4`. Dort springt `quadrat(a)` in die Funktion `quadrat` und kommt mit `9` zurück. Dann liefert `quadrat(b)` die `16`. Zum Schluss gibt `summe_der_quadrate` den Wert `9 + 16 = 25` zurück.

Der Vorteil: Jede Funktion erledigt eine kleine, klar umrissene Aufgabe. Ist `quadrat` einmal getestet, kann man sich darauf verlassen, wo immer man sie benutzt.

Merke

- `def name(parameter):` definiert eine Funktion, `return wert` gibt einen Wert zurück.
- Eine Funktion kann **mehrfach** mit unterschiedlichen Argumenten aufgerufen werden.
- Funktionen können **andere Funktionen aufrufen** (z. B. eine Hauptfunktion nutzt Hilfsfunktionen).

1.9.2 Code lesen und erklären

Bevor du ein fremdes Programm ausführst, lohnt es sich, es zu **lesen** und vorherzusagen, was passiert. So trainierst du, Programme wirklich zu verstehen – und nicht nur auszuprobieren.

Tipp

Erinnere dich an die Rechenzeichen aus dem Abschnitt über Operatoren: `a % b` ist der **Rest** der Division (`7 % 3` ergibt `1`), `a // b` ist 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.

Arbeitsauftrag 105: Code lesen

Lies den folgenden Code, **ohne** ihn auszuführen.

```
1  def ist_teilbar(a, b):
2      return a % b == 0
3
4  def verarbeite(x, y):
5      if ist_teilbar(x, y):
6          return x // y
7      else:
8          return x * y
9
10 print(verarbeite(6, 3))
11 print(verarbeite(5, 3))
```

1. Beschreibe in eigenen Worten, was die Funktion `verarbeite` macht.
2. Gib an, was `print(verarbeite(5, 3))` ausgibt – und was `print(verarbeite(6, 3))` ausgibt.

Führe den Code danach aus und überprüfe deine Vermutung.

Meine Erklärung und Vorhersage:

1.9.3 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. Für die nächste Aufgabe brauchst du die Kreiszahl π.

Werkzeug

Python bringt viele mathematische Werte und Funktionen in einem **Modul** namens `math` mit. Ein Modul ist eine Sammlung fertiger Bausteine. Damit du es benutzen kannst, schreibst du 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)
```

AUSGABE

```
3.141592653589793
```

Arbeitsauftrag 106: Kreisumfang

Die folgende 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.

```
1 import math
2
3 def kreisumfang(radius):
4     umfang = 2 * radius
5     return umfang
6
7 # Test: sollte bei radius = 5 etwa 31.4 ergeben
8 print(kreisumfang(5))
```

AUSGABE

```
10
```

Der Fehler:

Arbeitsauftrag 107: Grenzfälle testen

Ein Online-Shop gibt **ab** einem Einkaufswert von 50 Euro 10 % Rabatt. Die folgende Funktion soll den Preis nach Abzug des Rabatts zurückgeben:

```
1 def rabatt_preis(preis):
2     if preis > 50:
3         return preis * 0.9
4     else:
5         return preis
```

1. Rechne von Hand aus, was für die Einkaufswerte 20, 50 und 100 Euro herauskommen **muss**.
2. Teste die Funktion mit diesen drei Werten. Welcher Test schlägt fehl?
3. Erkläre den Fehler und verbessere die Funktion.

Gymnasium Ottweiler

Erwartete Werte, Testergebnis und Verbesserung:

1.9.4 Ein Programm aus mehreren Funktionen

Tipp

Bei der nächsten Aufgabe schreibst du zuerst die beiden kleinen Funktionen und testest jede einzeln. Erst wenn sie stimmen, schreibst du `beschreibe_rechteck`. Diese Funktion ruft die beiden anderen auf, so wie `summe_der_quadrate` oben `quadrat` aufruft. Den Satz setzt du mit `+` zusammen. Denk daran: Zahlen müssen dafür mit `str(...)` in Text umgewandelt werden.

Arbeitsauftrag 108: Rechteck

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 zusammenfassenden 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.“

Arbeitsauftrag 109 (Zusatz): Maximum von drei Zahlen

Im Abschnitt über Funktionen hast du `maximum(a, b)` geschrieben:

```python
def maximum(a, b):
    if a > b:
        return a
    else:
        return b
```

Erweitere das Programm um eine Funktion `maximum3(a, b, c)`, die die größte von **drei** Zahlen zurückgibt. Benutze dafür die Funktion `maximum` – `maximum3` selbst soll ohne `if` auskommen. Teste mit mindestens drei Aufrufen, bei denen die größte Zahl jeweils an einer anderen Stelle steht.

1.9.5 Weitere Übungen

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.

Arbeitsauftrag 110: 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))))
```

Meine Vorhersage:

Arbeitsauftrag 111: 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.

Arbeitsauftrag 112: Gute Testwerte wählen ●○○

Die Funktion `ist_teenager(alter)` soll `True` zurückgeben, wenn jemand 13 bis einschließlich 19 Jahre alt ist, sonst `False`.

```
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.

Testwerte und Begründung:

Arbeitsauftrag 113: Ungerade mit Hilfsfunktion ●○○

Im Abschnitt über Funktionen hast du `ist_gerade(zahl)` geschrieben:

```
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.

Arbeitsauftrag 114: 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.

```
1 def stunden_zu_minuten(stunden):
2     return stunden * 100
3
4 def fahrzeit_minuten(stunden, minuten):
5     return stunden_zu_minuten(stunden) + minuten
6
7 print(fahrzeit_minuten(2, 15))
```

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.

Mein Vorgehen und der Fehler:

Arbeitsauftrag 115: 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.

Arbeitsauftrag 116: 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.

Arbeitsauftrag 117: 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`.

1.9.6 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.

Selbstcheck

- ☐ Ich kann ein Programm mit Funktionen lesen und seine Ausgabe vorhersagen.
- ☐ Ich kann erklären, was eine Funktion tut, die eine andere Funktion aufruft.
- ☐ Ich kann einen Fehler in einer Funktion durch gezieltes Testen finden und beheben.
- ☐ Ich kann Testwerte so wählen, dass auch Grenzfälle geprüft werden.
- ☐ Ich kann ein Programm aus mehreren Funktionen zusammensetzen.
- ☐ Ich kann `math.pi` in einer Rechnung verwenden.

Gymnasium Ottweiler

1.10 Zeichenketten: Texte Zeichen für Zeichen

Leitidee

Namen, Passwörter, Nachrichten – sehr viele Daten in Programmen sind Texte. Bisher hast du Texte nur als Ganzes benutzt: eingelesen, ausgegeben, mit `+` zusammengesetzt. Oft muss ein Programm aber **in einen Text hineinschauen**: Wie lang ist das Passwort? Enthält es eine Ziffer? Mit welchem Buchstaben beginnt der Name? Wie viele Vokale stecken in einem Satz? Dafür lernst du in diesem Abschnitt, wie Python einen Text sieht: als eine Reihe einzelner Zeichen, jedes an einer festen Position. Mit diesem Wissen kannst du später sogar Texte verschlüsseln.

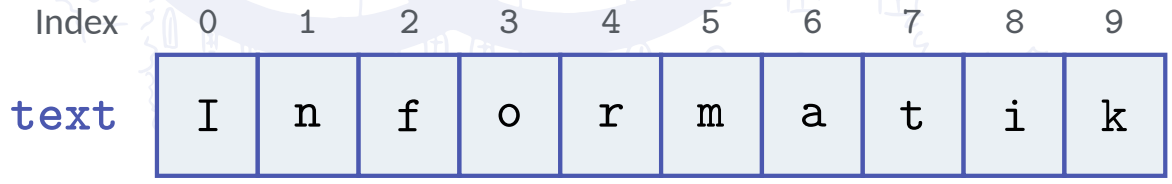
Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.10.1 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. Jedes Zeichen hat eine feste Position. Diese Positionsnummer heißt **Index**. Gezählt wird ab **0**, nicht ab 1:



Die Zeichenkette `"Informatik"` in der Variablen `text`: Jedes Zeichen hat eine Position (Index), gezählt ab 0

Das erste Zeichen `I` hat also den Index 0, das zweite Zeichen `n` den Index 1 und das letzte Zeichen `k` den Index 9. Die Zeichenkette hat 10 Zeichen, aber der größte Index ist 9. Warum ab 0? Man kann sich den Index als Abstand vom Anfang vorstellen: Das erste Zeichen ist 0 Schritte vom Anfang entfernt.

Definition: Zeichenkette

Eine **Zeichenkette** (englisch *string*, Datentyp `str`) ist eine Folge (**Sequenz**) von Zeichen, z. B. `"Informatik"`. Die Zeichen stehen in einer festen Reihenfolge.

Definition: Index

Der **Index** gibt die Position eines Zeichens in der Zeichenkette an. Das erste Zeichen hat den Index 0, das zweite den Index 1 und so weiter.

Merke

- Eine **Zeichenkette** (`str`) ist eine **Sequenz von Zeichen**, z. B. `"Informatik"`.
- Jedes Zeichen hat eine **Position (Index)**, beginnend bei `0`.

1.10.2 Auf einzelne Zeichen zugreifen

Um ein einzelnes Zeichen zu bekommen, schreibst du hinter die Zeichenkette den Index in **eckigen Klammern**:

```
1 text = "Informatik"
2 print(text[0])
3 print(text[2])
4 print(text[9])
```

AUSGABE

```
I
f
k
```

- **Zeile 1:** Die Variable `text` bekommt die Zeichenkette `"Informatik"`.
- **Zeile 2:** `text[0]` ist das Zeichen mit dem Index 0, also das erste Zeichen: `I`.
- **Zeile 3:** `text[2]` ist das **dritte** Zeichen, `f`. Schau in der Abbildung oben nach.
- **Zeile 4:** `text[9]` ist das letzte Zeichen, `k`.

In den eckigen Klammern darf auch eine Variable oder eine Rechnung stehen, zum Beispiel `text[i]` oder `text[i + 1]`. Das Ergebnis ist selbst wieder eine Zeichenkette, die aus genau einem Zeichen besteht.

Merke

Zugriff auf ein einzelnes Zeichen: `text[index]`.

1.10.3 Die Länge einer Zeichenkette

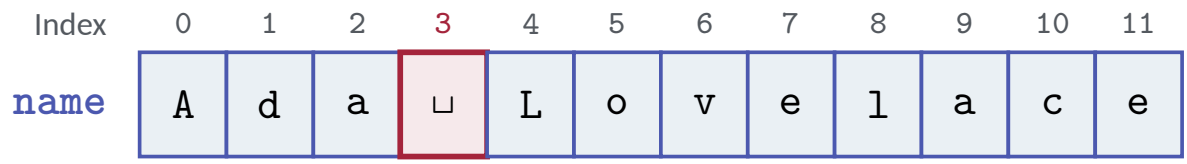
Wie viele Zeichen eine Zeichenkette hat, verrät dir die Funktion `len` (von englisch *length*, „Länge“):

```
1 text = "Informatik"
2 print(len(text))
3 print(len("Ada Lovelace"))
4 print(len(""))
```

AUSGABE

```
10
12
0
```

- **Zeile 2:** `"Informatik"` hat 10 Zeichen.
- **Zeile 3:** `"Ada Lovelace"` hat 12 Zeichen – das Leerzeichen zählt mit (siehe Abbildung). Ada Lovelace gilt übrigens als erste Programmiererin der Welt.
- **Zeile 4:** `""` ist die **leere Zeichenkette**. Sie enthält kein einziges Zeichen und hat die Länge 0.



Auch das Leerzeichen ist ein Zeichen: In `"Ada Lovelace"` steht es am Index 3. Die Zeichenkette hat 12 Zeichen

Weil der Index bei 0 beginnt, hat das **letzte Zeichen** immer den Index `len(text) - 1`. So findest du es, auch wenn du nicht weißt, wie lang der Text ist – zum Beispiel bei einer Eingabe:

```
1 text = "Informatik"
2 letzter_index = len(text) - 1
3 print(letzter_index)
4 print(text[letzter_index])
```

AUSGABE

```
9
k
```

Merke

Die **Länge** einer Zeichenkette erhält man mit `len(text)`. Deshalb hat das letzte Zeichen immer den Index `len(text) - 1`.

Von hinten zählen

Python erlaubt auch **negative Indizes**. Sie zählen von hinten: `text[-1]` ist das letzte Zeichen, `text[-2]` das vorletzte. Bei `"Informatik"` ist `text[-1]` also `k` und `text[-2]` ist `i`. `text[-1]` ist eine kurze Schreibweise für `text[len(text) - 1]`.

Wenn der Index zu groß ist

Was passiert, wenn du auf ein Zeichen zugreifst, das es gar nicht gibt?

```
text = "Informatik"
print(text[10])
```

FEHLERMELDUNG

```
IndexError: string index out of range
```

Stolperstein

Ein **IndexError** bedeutet: „Der Index liegt außerhalb des gültigen Bereichs.“ `"Informatik"` hat 10 Zeichen, gültig sind also nur die Indizes 0 bis 9 (und −1 bis −10). Der Index 10 gehört schon nicht mehr dazu. Das ist ein sehr häufiger Fehler: Man denkt an die Länge 10 und vergisst, dass ab 0 gezählt wird. Prüfe bei einem IndexError deshalb: Wie lang ist der Text, und welcher Index wurde benutzt? Auch bei der leeren Zeichenkette `""` führt schon `text[0]` zu diesem Fehler.

Merke

Ein Zugriff auf einen ungültigen Index (z. B. `text[100]` bei kurzem Text) führt zu einer **Fehlermeldung** (`IndexError`).

Arbeitsauftrag 118: Zeichen bestimmen

Gegeben ist das folgende Programm. Gib **ohne** Computer an, was jede Zeile ausgibt. Eine Zeile führt zu einem Fehler – welche, und warum? Überprüfe danach am Computer.

```
1 wort = "Python"
2 print(wort[1])
3 print(len(wort))
4 print(wort[len(wort) - 1])
5 print(wort[-2])
6 print(wort[6])
```

Meine Vorhersage:

1.10.4 Zeichenketten verketten

Aus dem Abschnitt über Variablen kennst du schon das **Verketten** mit `+`: Zwei Zeichenketten werden zu einer neuen, längeren Zeichenkette aneinandergehängt.

```
1 vorname = "Ada"
2 nachname = "Lovelace"
3 name = vorname + " " + nachname
4 print(name)
5 initialen = vorname[0] + nachname[0]
6 print(initialen)
```

AUSGABE

```
Ada Lovelace
AL
```

- **Zeile 3:** Drei Zeichenketten werden verkettet: der Vorname, ein Leerzeichen und der Nachname. Ohne das `" "` in der Mitte stünde `AdaLovelace` da.
- **Zeile 5:** Verkettung und Index-Zugriff lassen sich kombinieren: Das erste Zeichen des Vornamens und das erste Zeichen des Nachnamens ergeben die Initialen.

Merke

Die **Verkettung** (Konkatenation) zweier Zeichenketten erfolgt mit `+`.

Werkzeug: f-Strings

Texte mit Zahlen mit `+` zusammenzusetzen ist mühsam: Jede Zahl muss mit `str(...)` umgewandelt werden, und die Leerzeichen vergisst man leicht. Bequemer geht es mit einem **f-String**. Du schreibst ein `f` direkt vor das erste Anführungszeichen. Im Text darfst du dann Variablen oder Rechnungen in **geschweifte Klammern** `{...}` setzen. Python ersetzt sie durch ihren Wert:

```
1 name = "Alex"
2 alter = 15
3 print(f"{name} ist {alter} Jahre alt.")
4 print(name + " ist " + str(alter) + " Jahre alt.")
5 print(f"{name} hat {len(name)} Buchstaben.")
```

AUSGABE

```
Alex ist 15 Jahre alt.
Alex ist 15 Jahre alt.
Alex hat 4 Buchstaben.
```

Zeile 3 und Zeile 4 erzeugen dieselbe Ausgabe, aber der f-String kommt ohne `str(...)` aus. Vergisst du das `f`, erscheinen die geschweiften Klammern unverändert in der Ausgabe: `{name} ist {alter} Jahre alt.`

1.10.5 Alle Zeichen durchlaufen

Mit einer `for`-Schleife kannst du nacheinander **jedes Zeichen** einer Zeichenkette bearbeiten. Statt `range(...)` schreibst du hinter `in` einfach die Zeichenkette:

```
for zeichen in "Bit":
    print(zeichen)
```

AUSGABE

```
B
i
t
```

Die Schleifenvariable `zeichen` bekommt im ersten Durchlauf den Wert `"B"`, im zweiten `"i"` und im dritten `"t"`. Danach gibt es keine Zeichen mehr, und die Schleife endet. Der Name `zeichen` ist frei gewählt – du könntest auch `buchstabe` schreiben.

Zeichen zählen

Eine typische Aufgabe: Zähle, wie oft ein bestimmtes Zeichen vorkommt. Dazu verbindest du die Schleife mit einer Verzweigung und einer Zählvariablen:

```
1 wort = "Anna"
2 anzahl = 0
3 for zeichen in wort:
4     if zeichen == "n":
5         anzahl = anzahl + 1
6 print("Anzahl n:", anzahl)
```

AUSGABE
```
Anzahl n: 2
```

- **Zeile 2:** Die Zählvariable `anzahl` startet bei 0.
- **Zeile 3:** Die Schleife läuft über alle vier Zeichen von `"Anna"`.
- **Zeilen 4–5:** 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 Ablaufprotokoll zeigt, wie `zeichen` nacheinander jeden Wert annimmt und `anzahl` nur bei einem `n` wächst:

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

Beachte: Das große `A` am Anfang wird **nicht** gezählt, denn `"A"` und `"a"` sind für Python verschiedene Zeichen.

Einen neuen Text aufbauen

In einer Schleife kann man auch Zeichen für Zeichen eine **neue** Zeichenkette zusammensetzen. Man beginnt mit der leeren Zeichenkette `""` und hängt in jedem Durchlauf etwas an:

```
1 neu = ""
2 for zeichen in "Bit":
3     neu = neu + zeichen + zeichen
4 print(neu)
```

AUSGABE
```
BBiitt
```

Jedes Zeichen wird doppelt angehängt: Aus `""` wird `"BB"`, dann `"BBii"` und schließlich `"BBiitt"`.

Schleife über die Indizes

Manchmal brauchst du nicht nur das Zeichen, sondern auch seine **Position**. Dann lässt du die Schleife über alle Indizes laufen. `range(len(text))` liefert genau die gültigen Indizes 0, 1, ..., `len(text) - 1`:

```
1 text = "Bit"
2 for i in range(len(text)):
3     print(i, text[i])
```

AUSGABE

```
0 B
1 i
2 t
```

- **Zeile 2:** `len(text)` ist 3, also durchläuft `i` die Werte 0, 1 und 2.
- **Zeile 3:** Mit `text[i]` holst du das Zeichen an der Position `i`.

Merke

- `for zeichen in text:` durchläuft nacheinander alle **Zeichen** der Zeichenkette.
- `for i in range(len(text)):` durchläuft alle **Indizes**. Das Zeichen dazu ist `text[i]`.

1.10.6 Ist ein Zeichen enthalten?

Mit dem Wort `in` prüfst du, ob ein Zeichen – oder ein ganzes Textstück – in einer Zeichenkette vorkommt. Das Ergebnis ist `True` oder `False`:

```
1 print("a" in "Banane")
2 print("x" in "Banane")
3 print("nan" in "Banane")
4 print("B" in "banane")
```

AUSGABE

```
True
False
True
False
```

Die letzte Zeile zeigt wieder: Groß- und Kleinbuchstaben sind verschieden. Besonders praktisch ist `in` mit einer Zeichenkette, die alle „erlaubten“ Zeichen enthält. So prüfst du zum Beispiel, ob ein Buchstabe ein Vokal ist:

```
1 buchstabe = "e"
2 if buchstabe in "aeiou":
3     print(buchstabe, "ist ein Vokal.")
```

AUSGABE

```
e ist ein Vokal.
```

1.10.7 Groß- und Kleinschreibung

Für Python sind `"Ja"`, `"ja"` und `"JA"` drei verschiedene Zeichenketten. Oft ist die Schreibweise aber egal. Dann hilft es, alles in Klein- oder Großbuchstaben umzuwandeln:

```
1 text = "Hallo Welt"
2 print(text.upper())
3 print(text.lower())
4 print(text)
```

AUSGABE

```
HALLO WELT
hallo welt
Hallo Welt
```

- **Zeile 2:** `text.upper()` liefert den Text in Großbuchstaben (englisch *upper case*).
- **Zeile 3:** `text.lower()` liefert ihn in Kleinbuchstaben (*lower case*).
- **Zeile 4:** `text` selbst ist **unverändert**. `.upper()` und `.lower()` erzeugen eine neue Zeichenkette und lassen die alte, wie sie ist.

Die Schreibweise mit dem Punkt ist neu: `upper` wird nicht wie `len(text)` mit dem Text in Klammern aufgerufen, sondern **hinter** dem Text, verbunden durch einen Punkt. Solche Funktionen nennt man **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 auch leer sein, aber sie dürfen nicht fehlen.

Praktisch ist das bei Eingaben: Mit `if antwort.lower() == "ja":` erkennt ein Programm die Antwort, egal ob jemand `ja`, `Ja` oder `JA` eintippt.

Stolperstein

Eine Zeichenkette lässt sich nicht verändern. `text.lower()` allein ändert `text` deshalb **nicht**:

```
1 text = "Hallo"
2 text.lower()
3 print(text)
```

AUSGABE

```
Hallo
```

Das Ergebnis von `text.lower()` wurde nirgends gespeichert und ist verloren. Richtig ist `text = text.lower()`: Die Variable `text` bekommt dann die neue, kleingeschriebene Zeichenkette.

1.10.8 Zeichen prüfen: Buchstabe oder Ziffer?

Zwei weitere Methoden prüfen, woraus eine Zeichenkette besteht. Sie liefern `True` oder `False`:

- `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())
```

AUSGABE
```
True
False
True
False
False
```

`"Hallo Welt"` enthält ein Leerzeichen, und das ist kein Buchstabe. `"-5"` und `"3.5"` sind zwar Zahlen, aber das Minuszeichen und der Punkt sind keine Ziffern.

Damit kannst du den `ValueError` vermeiden, der bei `int(...)` mit einer falschen Eingabe entsteht: Das Programm wandelt die Eingabe nur um, wenn sie aus Ziffern besteht.

```
1 eingabe = input("Wie alt bist du? ")
2 if eingabe.isdigit():
3     print("In fünf Jahren bist du", int(eingabe) + 5)
4 else:
5     print("Bitte gib eine ganze Zahl mit Ziffern ein.")
```

AUSGABE
```
Wie alt bist du? fünfzehn
Bitte gib eine ganze Zahl mit Ziffern ein.
```

1.10.9 Zusatzwissen: Ausschnitte (Slicing)

Werkzeug: Slicing

Diesen Teil brauchst du für die Aufgaben nicht – er zeigt dir eine praktische Abkürzung. Mit zwei Indizes und einem Doppelpunkt in den eckigen Klammern schneidest du einen **Ausschnitt** (englisch *slice*) aus einer Zeichenkette heraus: `text[start:ende]`. Das Zeichen am Index `start` gehört dazu, das Zeichen am Index `ende` **nicht** mehr.

```
1 text = "Informatik"
2 print(text[0:4])
3 print(text[2:6])
4 print(text[::-1])
```

AUSGABE
```
Info
form
kitamrofnI
```

`text[0:4]` enthält die Zeichen mit den Indizes 0, 1, 2 und 3. Die Schreibweise `text[::-1]` ist ein Trick: Sie durchläuft den ganzen Text mit der Schrittweite −1, also rückwärts.

Merke: Zusatzwissen

Ein **Ausschnitt** (Slice) wird mit `text[start:ende]` erzeugt (das Zeichen am Index `ende` ist **nicht** mehr enthalten).

1.10.10 Übungen

Arbeitsauftrag 119: Zeichen-Steckbrief

Schreibe ein Programm, das ein Wort einliest und Folgendes ausgibt:

- die Länge des Wortes,
- das erste und das letzte Zeichen,
- das Wort rückwärts.

So könnte es aussehen:

AUSGABE

```
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 im Abschnitt *Einen neuen Text aufbauen*. Der Trick: Hänge jedes Zeichen **vorne** an, nicht hinten. Also nicht `rueckwaerts + zeichen`, sondern `zeichen + rueckwaerts`.

Arbeitsauftrag 120: Vokale zählen

Schreibe ein Programm, das einen Satz einliest und mit einer Schleife zählt, wie viele Vokale (a, e, i, o, u) darin vorkommen. Gib die Anzahl aus. Auch großgeschriebene Vokale sollen mitgezählt werden.

Tipp

Kombiniere drei Dinge aus diesem Abschnitt: 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()`.

Arbeitsauftrag 121: Passwort prüfen

Ein Passwort gilt hier als sicher, wenn es **mindestens 8 Zeichen** lang ist und **mindestens eine Ziffer** enthält. Implementiere eine Funktion `ist_sicher(passwort)`, die `True` oder `False` zurückgibt. Teste sie mit einem zu kurzen Passwort, einem langen Passwort ohne Ziffer und einem sicheren Passwort.

Arbeitsauftrag 122 (Zusatz): Palindrom

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 dabei keine Rolle spielen (Tipp: `text.lower()`). Teste mit „Anna“, „Rentner“ und „Informatik“.

1.10.11 Weitere Übungen

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.

Arbeitsauftrag 123: 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.

Das Zeichen b steht am Index:

Arbeitsauftrag 124: Ausgabe vorhersagen ●○○

Gib **ohne Computer** an, was das Programm ausgibt. Überprüfe danach am Computer.

```
1 text = "Klasse 10a"
2 print(len(text))
3 print(text[7] + text[8])
4 print("s" in text)
5 print("S" in text)
6 print(text.lower())
```

Meine Vorhersage:

Arbeitsauftrag 125: 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.

Arbeitsauftrag 126: 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	

Arbeitsauftrag 127: 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)
```

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.

Die beiden Fehler:

Arbeitsauftrag 128: 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 `" "`.

Arbeitsauftrag 129: 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"`.

Arbeitsauftrag 130: 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.

1.10.12 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.

Selbstcheck

- ☐ Ich kann erklären, warum eine Zeichenkette eine Sequenz von Zeichen ist, und zu jedem Zeichen den Index angeben.
- ☐ Ich kann mit `text[i]` und `len(text)` das erste, ein beliebiges und das letzte Zeichen ausgeben.
- ☐ Ich kann erklären, wann ein `IndexError` entsteht, und ihn vermeiden.
- ☐ Ich kann Zeichenketten verketten und einen f-String schreiben.
- ☐ Ich kann mit einer Schleife alle Zeichen durchlaufen, z. B. um Zeichen zu zählen oder einen Text umzudrehen.
- ☐ Ich kann `in`, `.upper()`, `.lower()`, `.isalpha()` und `.isdigit()` sinnvoll einsetzen.

Gymnasium Ottweiler

1.11 Listen: viele Werte unter einem Namen

Leitidee

Stell dir vor, du willst die Noten deiner letzten fünf Klassenarbeiten in einem Programm speichern. Mit dem, was du bisher kennst, brauchst du dafür fünf Variablen: `note1`, `note2` und so weiter. Bei den Noten einer ganzen Klasse oder bei den Temperaturen eines ganzen Jahres wird das schnell unübersichtlich. Programme brauchen deshalb eine Möglichkeit, **viele Werte unter einem Namen** zu speichern: die **Liste**. Vieles daran kennst du schon von den Zeichenketten – auch in einer Liste hat jeder Wert eine Position, und gezählt wird ab 0. Am Ende dieses Abschnitts programmierst du eine Einkaufsliste.

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.11.1 Warum Listen?

Speicherst du fünf Noten in fünf Variablen `note1` bis `note5`, hat das zwei Schwächen. Kommt eine sechste Note dazu, brauchst du eine neue Variable **und** musst jede Rechnung ändern, die alle Noten benutzt. Außerdem kannst du keine Schleife über `note1` bis `note5` laufen lassen: Für Python sind das fünf völlig unabhängige Namen.

Mit einer **Liste** speicherst du alle Noten in **einer** Variablen:

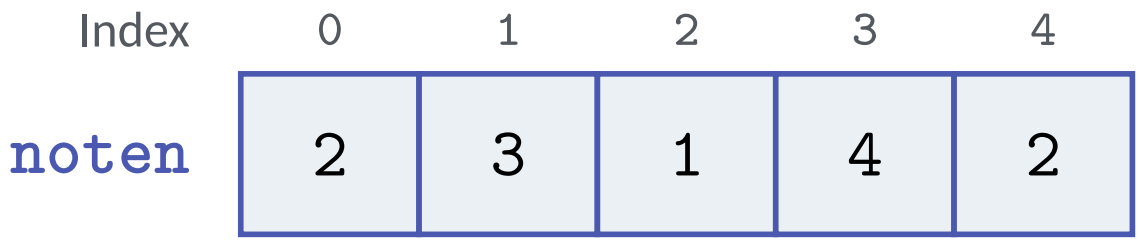
```
1 noten = [2, 3, 1, 4, 2]
2 print(noten)
3 print(type(noten))
```

AUSGABE

```
[2, 3, 1, 4, 2]
<class 'list'>
```

- **Zeile 1:** Eine Liste schreibt man in **eckigen Klammern** `[ ]`. Die einzelnen Werte werden durch Kommas getrennt. Die ganze Liste wird in der Variablen `noten` gespeichert.
- **Zeile 2:** `print` gibt die Liste mit ihren Klammern und Kommas aus.
- **Zeile 3:** Der Datentyp einer Liste heißt `list` (englisch für „Liste“).

Die Werte in einer Liste heißen **Elemente**. Sie stehen in einer festen Reihenfolge hintereinander, wie die Waggons eines Zuges: Es gibt ein erstes und ein letztes Element, und jedes andere Element hat genau einen Vorgänger und einen Nachfolger. Man sagt deshalb: Eine Liste ist eine **lineare** Datenstruktur.



Die Liste `noten` mit fünf Elementen. Über jedem Element steht seine Position, der Index

Definition: Liste bzw. Feld

Eine **Liste** (auch **Feld** genannt) ist eine Datenstruktur, die mehrere Werte – die **Elemente** – in einer festen Reihenfolge zu einer Einheit zusammenfasst. In Python schreibt man sie in eckigen Klammern, z. B. `[2, 3, 1, 4, 2]`.

Liste oder Feld?

In Python heißt diese Datenstruktur **Liste** (`list`). In anderen Programmiersprachen wie Java oder C begegnet dir dafür das Wort **Feld** (englisch *array*). Ein Feld hat dort meist eine feste Länge. Eine Python-Liste kann dagegen wachsen und schrumpfen. Die Grundidee ist dieselbe: viele Elemente in einer Reihe, durchnummeriert ab 0.

Weitere Listen

In einer Liste können auch Zeichenketten oder Kommazahlen stehen, z. B. `[1.99, 0.49, 2.5]`. Eine Liste kann sogar leer sein:

```
1 staedte = ["Saarbrücken", "Trier", "Mainz"]
2 leer = []
3 print(staedte)
4 print(leer)
```

AUSGABE

```
['Saarbrücken', 'Trier', 'Mainz']
[]
```

Beachte die Ausgabe in der ersten Zeile: Python zeigt Zeichenketten **in einer Liste** mit einfachen Anführungszeichen `'…'` an. Das ist nur die Art der Anzeige. Im Programm darfst du `"…"` oder `'…'` schreiben. In diesem Skript verwenden wir immer `"…"`.

Die leere Liste `[]` in Zeile 2 sieht nutzlos aus. Sie ist aber praktisch: Man legt sie an und füllt sie später nach und nach.

Merke

Eine **Liste** (in Python: `list`) fasst mehrere **Elemente zu einer Einheit** zusammen, z. B. `noten = [2, 3, 1, 4, 2]`.

1.11.2 Index und Element

Denk an die Schließfächer in deiner Schule. Jedes Fach hat eine **Nummer** an der Tür, und in jedem Fach liegt ein **Inhalt**. Wer „Fach 2“ sagt, meint die Nummer – nicht das, was darin liegt.

Bei einer Liste ist es genauso. Jedes Element hat eine Positionsnummer, den **Index**. Wie bei Zeichenketten beginnt die Zählung bei **0**: Das erste Element hat den Index 0, das zweite den Index 1 und so weiter.

Definition: Index und Element

Der **Index** ist die Positionsnummer eines Elements in einer Liste. Er beginnt bei 0. Das **Element** (Listenelement) ist der Wert, der an dieser Position gespeichert ist.

Lesender Zugriff

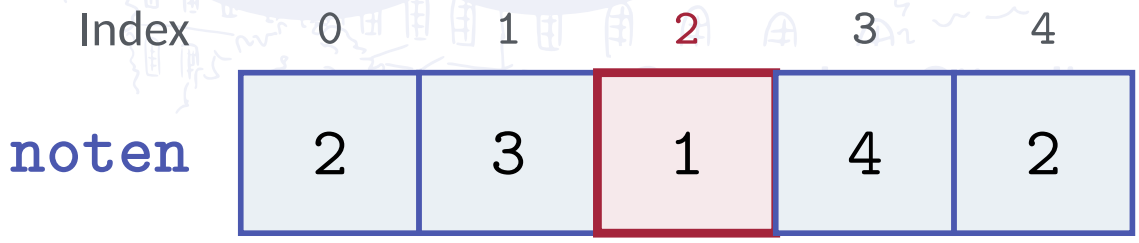
Um ein einzelnes Element zu lesen, schreibst du den Index in eckigen Klammern hinter den Namen der Liste – genau wie bei Zeichenketten:

```
1 noten = [2, 3, 1, 4, 2]
2 print(noten[0])
3 print(noten[2])
4 erste = noten[0]
5 print("Die erste Note ist", erste)
```

AUSGABE

```
2
1
Die erste Note ist 2
```

- **Zeile 2:** `noten[0]` liefert das Element mit dem Index 0. Das ist das erste Element, die `2`.
- **Zeile 3:** `noten[2]` liefert das Element mit dem Index 2. Das ist die `1` – also das **dritte** Element. Die Zahl in den eckigen Klammern ist der Index, nicht der Wert!
- **Zeilen 4–5:** Ein gelesenes Element kann man wie jeden anderen Wert in einer Variablen speichern.



Der Ausdruck `noten\[2\]` liefert das Element am Index 2. Das ist die 1, nicht die 2

Gerade bei Listen mit kleinen Zahlen verwechselt man Index und Element leicht. In der Liste `noten` kommt das Element `2` zweimal vor: am Index 0 und am Index 4. Am Index 2 steht dagegen das Element `1`.

Die Länge einer Liste

Wie viele Elemente eine Liste hat, verrät dir `len(...)` – wie bei Zeichenketten:

```
1 noten = [2, 3, 1, 4, 2]
2 print(len(noten))
3 print(noten[len(noten) - 1])
```

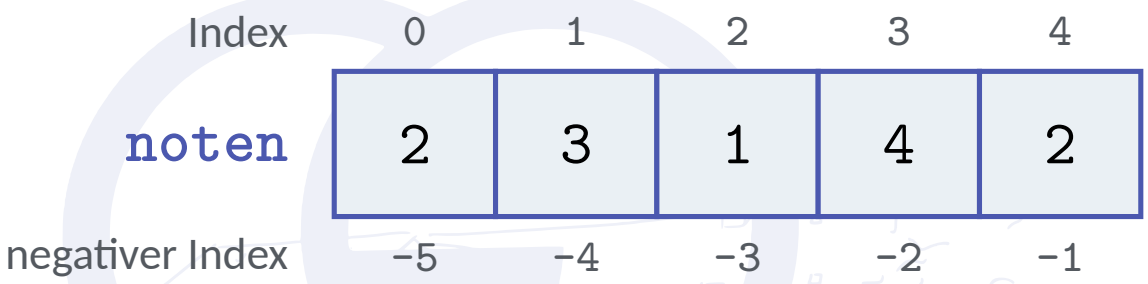
AUSGABE

```
5
2
```

Die Liste hat fünf Elemente. Weil die Zählung bei 0 beginnt, hat das letzte Element aber den Index **4**, also `len(noten) - 1`. Zeile 3 liest deshalb das letzte Element. Das funktioniert für jede Liste, egal wie lang sie ist.

Negative Indizes

Von den Zeichenketten kennst du schon eine Abkürzung: Ein **negativer Index** zählt von hinten. `noten[-1]` ist das letzte Element (die 2), `noten[-2]` das vorletzte (die 4).



Unter jedem Element steht der negative Index. Er zählt von hinten, beginnend bei -1

Merke

- Jedes Element besitzt einen **Index**, beginnend bei `0`. Der Index ist **nicht** dasselbe wie der gespeicherte Wert!
- **Lesender Zugriff:** `liste[index]` liefert das Element an dieser Position.
- Die Anzahl der Elemente erhält man mit `len(liste)`. Das letzte Element hat den Index `len(liste) - 1` (oder kurz `-1`).

Arbeitsauftrag 131: Index oder Element?

Gegeben ist die Liste `zahlen = [3, 0, 4, 1, 2]`. Bestimme **ohne Computer**, welchen Wert die Ausdrücke haben. Prüfe danach mit `print(...)` am Computer.

`zahlen[1]`	`len(zahlen)`	`zahlen[-1]`	`zahlen[zahlen[0]]`

Gib außerdem an: An welchem Index steht das Element `4`?

Das Element 4 steht am Index:

1.11.3 Schreibender Zugriff: ein Element ändern

Ein Schließfach kann man ausräumen und neu füllen – die Nummer bleibt gleich. Genauso kannst du ein Listenelement **überschreiben**. Der Zugriff mit Index steht dann **links** vom =:

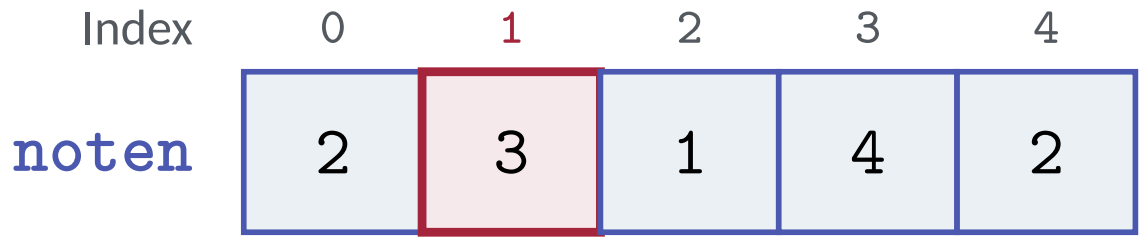
```
1 noten = [2, 3, 1, 4, 2]
2 noten[1] = 5
3 print(noten)
4 noten[2] = noten[2] + 1
5 print(noten)
```

AUSGABE

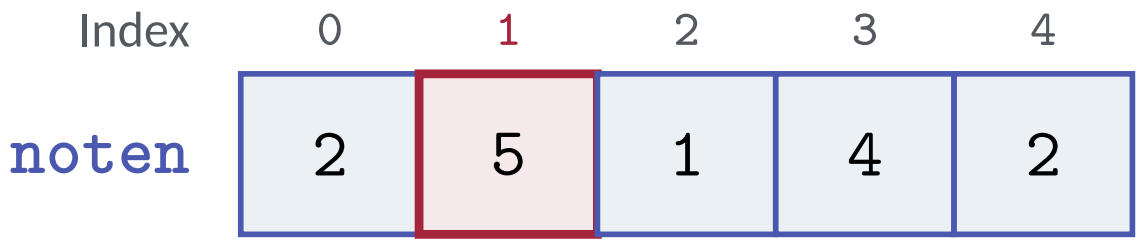
```
[2, 5, 1, 4, 2]
[2, 5, 2, 4, 2]
```

- **Zeile 2:** Das Element am Index 1 bekommt den Wert 5. Der alte Wert 3 ist weg. Das funktioniert wie eine Zuweisung an eine Variable – nur eben an **eine Stelle** der Liste.
- **Zeile 4:** Wie bei `x = x + 1` wird zuerst die rechte Seite ausgerechnet: `noten[2]` ist 1, plus 1 ergibt 2. Dieser Wert wird dann am Index 2 gespeichert.

Die Länge der Liste ändert sich dabei nicht. Es werden nur Inhalte ausgetauscht.



Vorher: Am Index 1 steht die 3



Nachher (nach `noten\[1\] = 5`): Am Index 1 steht die 5. Alle anderen Elemente sind unverändert

Merke

Schreibender Zugriff: `liste[index] = neuer_wert` überschreibt das Element an dieser Position.

Stolperstein

Bei **Zeichenketten** geht das nicht. Eine Zeichenkette kann man nur lesen, aber nicht stellenweise ändern:

```
wort = "Hallo"
wort[0] = "B"
```

FEHLERMELDUNG

```
TypeError: 'str' object does not support item assignment
```

Die Meldung heißt: „Ein `str`-Objekt unterstützt keine Zuweisung an einzelne Stellen.“ Listen sind dagegen **veränderbar**. Das ist ein wichtiger Unterschied zwischen den beiden Datenstrukturen.

1.11.4 Ungültige Position: der `IndexError`

Was passiert, wenn du auf einen Index zugreifst, den es gar nicht gibt?

```
noten = [2, 3, 1, 4, 2]
print(noten[5])
```

FEHLERMELDUNG

```
Traceback (most recent call last):
  File "noten.py", line 2, in <module>
    print(noten[5])
          ~~~~~^^^
IndexError: list index out of range
```

Lies die Meldung wieder von unten nach oben: **IndexError** heißt „Fehler beim Index“. *list index out of range* bedeutet: „Der Listenindex liegt außerhalb des gültigen Bereichs.“ Die Zeilen darüber zeigen, dass der Fehler in Zeile 2 passiert ist, und die Zeichen `^^^` zeigen auf `[5]`.

Die Liste hat fünf Elemente. Gültig sind also nur die Indizes 0 bis 4 (und von hinten −1 bis −5). Einen Index 5 gibt es nicht.

Auch beim **Schreiben** an eine Stelle, die es nicht gibt, meldet Python einen Fehler. Eine Liste wird durch eine Zuweisung nicht automatisch länger:

```
noten = [2, 3, 1, 4, 2]
noten[5] = 1
```

FEHLERMELDUNG

```
IndexError: list assignment index out of range
```

list assignment index out of range heißt: „Der Index für die Zuweisung liegt außerhalb des gültigen Bereichs." Um ein Element **hinzuzufügen**, brauchst du `append` aus dem nächsten Abschnitt.

Merke

Ein Zugriff auf einen ungültigen Index führt zu einer Fehlermeldung (`IndexError`). Gültig sind bei einer Liste mit `n` Elementen nur die Indizes `0` bis `n - 1`.

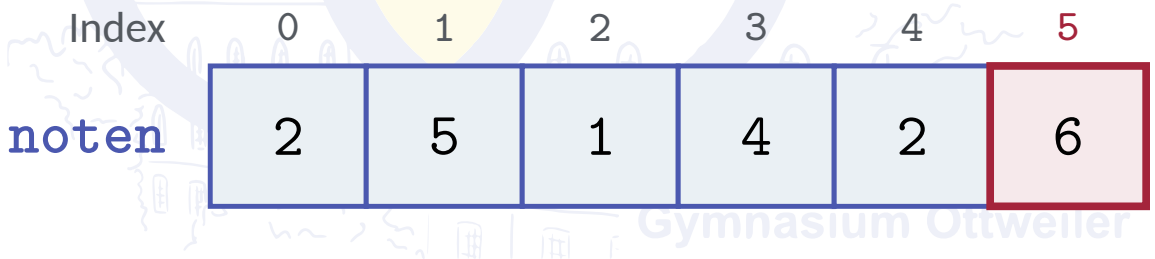
Die häufigste Ursache für einen `IndexError`: Man zählt aus Gewohnheit ab 1 und greift mit `liste[len(liste)]` auf das „letzte" Element zu. Der größte gültige Index ist aber `len(liste) - 1`. Prüfe bei einem `IndexError` immer: Wie lang ist die Liste gerade, und welchen Index habe ich benutzt?

1.11.5 Listen verändern: Elemente hinzufügen und entfernen

Eine Liste kann wachsen und schrumpfen. Dafür gibt es eigene **Methoden**, die man mit einem Punkt an den Namen der Liste hängt: `liste.methode(...)`. Methoden kennst du schon von den Zeichenketten, zum Beispiel `text.upper()`.

Ein Element anhängen: append

`append` heißt „anhängen". Die Methode fügt ein neues Element **am Ende** der Liste an. Nach `noten.append(6)` ist die Liste `noten` um ein Element länger. Das neue Element hat den Index 5:



Nach `noten.append(6)`: Das neue Element steht am Ende, am Index 5

Merke

Mit `liste.append(wert)` wird ein neues Element ans Ende der Liste angehängt.

Das Beispiel im Ganzen

Das folgende Programm fasst alles zusammen, was du bis jetzt über Listen gelernt hast – auch `append`. Überlege zuerst, was es ausgibt. Lies dann weiter.

```
1 noten = [2, 3, 1, 4, 2]
2
3 print(noten)
4 print("Anzahl Noten:", len(noten))
5 print("Erste Note (Index 0):", noten[0])
6 print("Dritte Note (Index 2):", noten[2])
7
8 noten[1] = 5
9 print("Nach Änderung:", noten)
10
11 noten.append(6)
12 print("Nach append:", noten)
```

AUSGABE

```
[2, 3, 1, 4, 2]
Anzahl Noten: 5
Erste Note (Index 0): 2
Dritte Note (Index 2): 1
Nach Änderung: [2, 5, 1, 4, 2]
Nach append: [2, 5, 1, 4, 2, 6]
```

- **Zeilen 3–6:** Lesender Zugriff: die ganze Liste, ihre Länge, das Element am Index 0 und das Element am Index 2. Beachte: Die **dritte** Note hat den Index **2**.
- **Zeile 8:** Schreibender Zugriff: Das Element am Index 1 (die 3) wird durch 5 ersetzt.
- **Zeile 11:** Die Note 6 wird ans Ende angehängt. Die Liste hat danach sechs Elemente.

Ein Element einfügen: `insert`

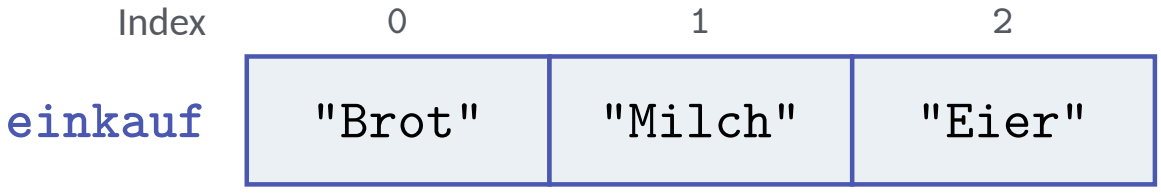
Mit `liste.insert(index, wert)` fügst du ein Element an einer **beliebigen Stelle** ein. Der erste Wert in den Klammern ist der Index, an dem das neue Element stehen soll, der zweite ist das Element selbst:

```
1 einkauf = ["Brot", "Milch", "Eier"]
2 einkauf.insert(1, "Käse")
3 print(einkauf)
```

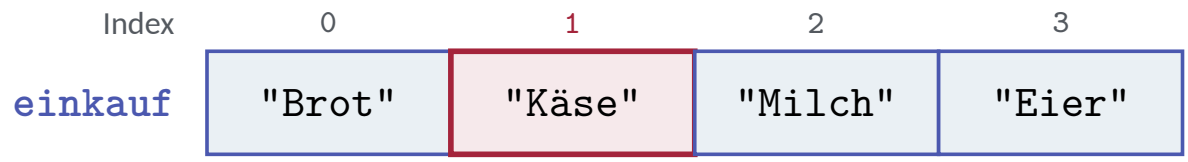
AUSGABE

```
['Brot', 'Käse', 'Milch', 'Eier']
```

Der Käse steht jetzt am Index 1. Alle Elemente, die vorher ab Index 1 standen, rücken eine Stelle nach rechts. Ihr Index wird dadurch um 1 größer: Die Milch hatte vorher den Index 1, jetzt hat sie den Index 2.



Vorher: drei Elemente mit den Indizes 0 bis 2



Nachher (nach `einkauf.insert(1, "Käse")`): Milch und Eier sind eine Stelle nach rechts gerückt

Elemente entfernen: `pop` und `remove`

Zum Entfernen gibt es zwei Methoden. Sie unterscheiden sich darin, was du in die Klammern schreibst: einen **Index** oder ein **Element**.

`liste.pop(index)` entfernt das Element **am angegebenen Index**. Ohne Index entfernt `pop()` das **letzte** Element. In beiden Fällen gibt `pop` das entfernte Element zurück – so wie eine Funktion mit `return` einen Wert zurückgibt. Du kannst es also in einer Variablen auffangen:

```
1 einkauf = ["Brot", "Käse", "Milch", "Eier"]
2 letztes = einkauf.pop()
3 print(letztes)
4 print(einkauf)
5 erstes = einkauf.pop(0)
6 print(erstes)
7 print(einkauf)
```

AUSGABE

```
Eier
['Brot', 'Käse', 'Milch']
Brot
['Käse', 'Milch']
```

- **Zeile 2:** `pop()` ohne Index entfernt das letzte Element (`"Eier"`) und gibt es zurück. Es wird in `letztes` gespeichert.
- **Zeile 5:** `pop(0)` entfernt das Element am Index 0 (`"Brot"`). Die übrigen Elemente rücken nach links. `"Käse"` hat jetzt den Index 0.

`liste.remove(wert)` sucht dagegen nach einem **Element mit diesem Wert** und entfernt es. Kommt der Wert mehrmals vor, wird nur das **erste** Vorkommen entfernt:

```
1 einkauf = ["Brot", "Käse", "Milch", "Käse"]
2 einkauf.remove("Käse")
3 print(einkauf)
```

AUSGABE

```
['Brot', 'Milch', 'Käse']
```

Die Tabelle fasst die Methoden zusammen:

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

Merke dir den Unterschied: `pop` erwartet einen **Index**, `remove` ein **Element**.

Stolperstein

Ist der Wert nicht in der Liste, kann `remove` nichts entfernen und meldet einen **ValueError** („falscher Wert“):

```
einkauf = ["Brot", "Milch"]
einkauf.remove("Käse")
```

FEHLERMELDUNG

```
ValueError: list.remove(x): x not in list
```

Die Meldung heißt: „`x` ist nicht in der Liste.“ Ähnlich bei `pop`: Aus einer leeren Liste kann nichts entfernt werden. Dann meldet Python `IndexError: pop from empty list`.

So kannst du es sagen

So beschreibst du Listenoperationen in Fachsprache – vom einfachen zum genauen Satz:

- *„Die 5 ist an Stelle 1.“ → „Das Listenelement am Index 1 hat den Wert 5.“*
- *„Ich mach die 3 zur 5.“ → „Das Listenelement an der Stelle 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`.“*

Arbeitsauftrag 132: Listen-Vorhersage

Lies das Programm, **ohne** es auszuführen. Zeichne die Liste `tiere` nach jeder Zeile als Reihe von Kästchen und schreibe den Index darüber. Gib dann an, was das Programm ausgibt. Überprüfe deine Vorhersage am Computer.

```
1 tiere = ["Hund", "Katze", "Maus"]
2 tiere.append("Igel")
3 tiere[0] = "Pferd"
4 tiere.insert(2, "Esel")
5 tiere.remove("Katze")
6 x = tiere.pop()
7 print(tiere)
8 print(x)
9 print(len(tiere))
```

Die Liste nach jeder Zeile und die Ausgabe:

1.11.6 Listen mit Schleifen durchlaufen

Oft soll ein Programm mit **jedem** Element einer Liste etwas tun. Dafür gibt es zwei Arten von `for`-Schleifen.

Schleife über die Elemente

Über die Zeichen einer Zeichenkette läufst du mit `for zeichen in text:`. Über die Elemente einer Liste geht es genauso:

```
1 staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]
2 for stadt in staedte:
3     print("Ich fahre nach", stadt)
```

AUSGABE

```
Ich fahre nach Saarbrücken
Ich fahre nach Trier
Ich fahre nach Mainz
Ich fahre nach Koblenz
```

- **Zeile 2:** Die Schleifenvariable `stadt` bekommt nacheinander jedes Element der Liste, von vorne nach hinten.
- **Zeile 3:** Der eingerückte Schleifenrumpf wird für jedes Element einmal ausgeführt – hier also viermal.

Schleife über die Indizes

Manchmal brauchst du nicht nur das Element, sondern auch seinen **Index**. Dann läufst du mit `range(len(liste))` über alle gültigen Indizes:

```
1 staedte = ["Saarbrücken", "Trier", "Mainz", "Koblenz"]
2 for i in range(len(staedte)):
3     print(i, staedte[i])
```

AUSGABE

```
0 Saarbrücken
1 Trier
2 Mainz
3 Koblenz
```

- **Zeile 2:** `len(staedte)` ist 4. `range(4)` erzeugt die Zahlen 0, 1, 2, 3 – genau die gültigen Indizes der Liste. Die Schleifenvariable `i` ist also ein **Index**.
- **Zeile 3:** Mit `staedte[i]` wird das Element am Index `i` gelesen.

Die Schleife über die Indizes brauchst du immer dann, wenn du die Position kennen musst – zum Beispiel, um Elemente zu verändern. Mehr dazu im nächsten Abschnitt.

Versuch: Fehler finden

Ändere die Schleife zu `for i in range(1, len(staedte) + 1):`. Was gibt das Programm jetzt aus, bevor es abbricht? Welche Fehlermeldung erhältst du? Erkläre: Welche Indizes erzeugt `range` jetzt, und welcher davon ist ungültig?

Eine Liste mit einer Schleife füllen

Ein sehr häufiges Muster: Man beginnt mit einer **leeren Liste** und hängt in einer Schleife ein Element nach dem anderen an.

```
1 quadratzahlen = []
2 for zahl in range(1, 6):
3     quadratzahlen.append(zahl * zahl)
4 print(quadratzahlen)
```

AUSGABE

```
[1, 4, 9, 16, 25]
```

- **Zeile 1:** Die Liste ist zuerst leer.
- **Zeilen 2–3:** Für jede Zahl von 1 bis 5 wird ihr Quadrat ans Ende der Liste angehängt.
- **Zeile 4:** Erst nach der Schleife wird die fertige Liste ausgegeben. Stünde `print` eingerückt in der Schleife, würde die Liste fünfmal ausgegeben – jedes Mal ein Stück länger.

1.11.7 Übungen

Arbeitsauftrag 133: Einkaufsliste

Lege eine leere Liste an. Frage nacheinander 5 Artikelnamen ab (mit `input()` in einer Schleife) und füge sie mit `append()` zur Liste hinzu. Gib die vollständige Liste am Ende aus.

Werkzeug

Mit `liste.index(wert)` erfährst du, an welchem **Index** ein Element steht. Kommt der Wert mehrmals vor, erhältst du den Index des ersten Vorkommens. Ist der Wert gar nicht in der Liste, meldet Python einen Fehler, z. B. `ValueError: 'Lea' is not in list`. Im nächsten Abschnitt programmierst du so eine Suche selbst (**lineare Suche**).

Arbeitsauftrag 134: Index vs. Wert

Gegeben ist die Liste `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 (Tipp: `liste.index(wert)`).

Tipp

Zur Erinnerung: Mit einer negativen Schrittweite zählt `range` rückwärts: `range(4, -1, -1)` erzeugt 4, 3, 2, 1, 0. Die Zahl in der Mitte (`-1`) ist wie immer die Grenze, die **nicht** mehr erreicht wird.

Arbeitsauftrag 135 (Zusatz): Liste umkehren

Gegeben ist die Liste `zahlen = [1, 2, 3, 4, 5]`. Erzeuge mit einer Schleife eine **neue** Liste, die die Elemente in umgekehrter Reihenfolge enthält, und gib sie aus. Verwende dafür keine eingebaute Funktion zum Umkehren.

1.11.8 Weitere Übungen

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.

Arbeitsauftrag 136: 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.

"Do" steht am Index:

Arbeitsauftrag 137: 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))
```

Meine Vorhersage:

Arbeitsauftrag 138: 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.

Arbeitsauftrag 139: IndexError erklären ●○○

Das Programm soll die letzte Farbe der Liste ausgeben. Es bricht aber ab.

```
farben = ["rot", "gelb", "grün"]
print(farben[3])
```

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.

Meine Antworten:

Arbeitsauftrag 140: 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*.

Arbeitsauftrag 141: 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
```

Arbeitsauftrag 142: 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)`.

Arbeitsauftrag 143: 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]`.

Warum braucht man die Hilfsvariable?

1.11.9 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`. Negative Indizes zählen von hinten: `liste[-1]` ist das letzte Element.
- `len(liste)` liefert die Anzahl der Elemente. Gültige Indizes sind `0` bis `len(liste) - 1`. Ein ungültiger Index führt zu einem `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.

Selbstcheck

- ☐ Ich kann erklären, was eine Liste ist und warum sie praktischer ist als viele einzelne Variablen.
- ☐ Ich kann bei einer Liste zwischen Index und Element unterscheiden.
- ☐ Ich kann ein Element über seinen Index lesen und überschreiben.
- ☐ Ich kann Elemente mit `append` und `insert` hinzufügen und mit `pop` und `remove` entfernen.
- ☐ Ich kann eine Liste mit einer Schleife durchlaufen und mit `append` in einer Schleife füllen.
- ☐ Ich kann erklären, wann ein `IndexError` auftritt, und den Fehler beheben.

Gymnasium Ottweiler

1.12 Algorithmen auf Listen: Summe, Minimum und Suche

Leitidee

In einer Liste stehen die Noten einer Klassenarbeit. Wie gut war die Klasse im Durchschnitt? Was war die beste Note? Hat jemand eine 6 geschrieben, und wenn ja, wie viele? Bei einer kurzen Liste siehst du das auf einen Blick. Ein Computer kann aber nicht „auf einen Blick“ sehen. Er muss die Liste **Element für Element** durchgehen. Ein **Algorithmus** ist eine genaue Schritt-für-Schritt-Anleitung, mit der man eine Aufgabe löst – so eindeutig, dass auch ein Computer sie ausführen kann. In diesem Abschnitt programmierst du einige grundlegende Algorithmen für Listen selbst: Summe und Mittelwert, Minimum und Maximum, die lineare Suche und das Zählen. Erst danach lernst du die fertigen Funktionen kennen, die Python dafür schon mitbringt.

Lernziele

Nach diesem Abschnitt kannst du

- 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,
- deine Ergebnisse mit `sum()`, `min()`, `max()` und `sorted()` kontrollieren.

1.12.1 Zwei Arten, eine Liste zu durchlaufen

Im Abschnitt über Listen hast du zwei Arten von `for`-Schleifen kennengelernt. Hier noch einmal beide nebeneinander:

Schleife	Schleifenvariable	geeignet, wenn …
`for element in liste:`	jedes **Element**	nur Werte nötig sind
`for i in range(len(liste)):`	jeder **Index**	Position nötig ist

Willst du Elemente einer Liste **verändern**, brauchst du die Schleife über die Indizes. Hier werden alle Preise verdoppelt:

```
1 preise = [2.0, 3.5, 1.0]
2 for i in range(len(preise)):
3     preise[i] = preise[i] * 2
4 print(preise)
```

AUSGABE

```
[4.0, 7.0, 2.0]
```

In Zeile 3 steht ein schreibender Zugriff: Das Element am Index `i` wird durch sein Doppeltes ersetzt.

Merke

Über eine Liste kann man auf zwei Arten iterieren (sie durchlaufen):

- **über die Elemente direkt:** `for element in liste:`
- **über die Indizes:** `for i in range(len(liste)):` – nützlich, wenn man den Index selbst braucht (z. B. für schreibenden Zugriff).

Stolperstein

Mit der Schleife über die Elemente kann man die Liste **nicht** verändern:

```
1 preise = [2.0, 3.5, 1.0]
2 for preis in preise:
3     preis = preis * 2
4 print(preise)
```

AUSGABE

```
[2.0, 3.5, 1.0]
```

Es gibt keine Fehlermeldung, aber die Liste ist unverändert. Die Variable `preis` bekommt in jedem Durchlauf den **Wert** eines Elements. Zeile 3 gibt nur der Variablen `preis` einen neuen Wert. Die Liste selbst bekommt davon nichts mit.

1.12.2 Summenbildung

An der Supermarktkasse werden die Preise nacheinander zu einer **Zwischensumme** addiert. Am Anfang steht die Zwischensumme auf 0. Bei jedem Artikel kommt sein Preis dazu. Nach dem letzten Artikel ist die Zwischensumme die Gesamtsumme.

Genauso arbeitet der Algorithmus zur **Summenbildung**. Die Zwischensumme ist eine **Hilfsvariable**, hier `summe`:

```
1 zahlen = [7, 2, 9, 4, 1, 8]
2
3 summe = 0
4 for zahl in zahlen:
5     summe = summe + zahl
6 print("Summe:", summe)
```

AUSGABE

```
Summe: 31
```

- **Zeile 3:** Die Hilfsvariable `summe` wird **vor** der Schleife auf 0 gesetzt. Man sagt: Sie wird mit 0 **initialisiert**.
- **Zeilen 4–5:** In jedem Durchlauf wird das aktuelle Element zu `summe` addiert.
- **Zeile 6:** Erst **nach** der Schleife steht in `summe` das Endergebnis. Deshalb ist `print` nicht eingerückt.

Wie sich `summe` Schritt für Schritt ändert, zeigt das Ablaufprotokoll. Damit es übersichtlich bleibt, steht hier eine kürzere Liste mit nur drei Zahlen:

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

Die `for`-Zeile wird viermal ausgeführt: dreimal mit einem neuen Element und ein viertes Mal, um festzustellen, dass die Liste zu Ende ist (Schritt 9).

Merke

Summenbildung: Eine Hilfsvariable wird vor der Schleife auf `0` gesetzt und in jedem Durchlauf um das aktuelle Element erhöht.

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`. Die Hilfsvariable muss deshalb immer **vor** der Schleife initialisiert werden.

Der Mittelwert

Den **Mittelwert** (Durchschnitt) erhältst du, wenn du die Summe durch die Anzahl der Elemente teilst. Die Anzahl liefert `len`:

```
1 noten = [2, 3, 1, 4, 2]
2 summe = 0
3 for note in noten:
4     summe = summe + note
5 mittelwert = summe / len(noten)
6 print("Durchschnitt:", mittelwert)
```

AUSGABE

```
Durchschnitt: 2.4
```

Das Programm funktioniert für jede Liste, egal wie viele Noten darin stehen. Kommt eine Note dazu, muss man nichts am Programm ändern.

Arbeitsauftrag 144: Durchschnitt berechnen

Schreibe eine Funktion `durchschnitt(liste)`, die den Durchschnitt aller Zahlen in einer Liste berechnet und zurückgibt. Teste sie mit der Liste `[7, 2, 9, 4, 1, 8]`.

1.12.3 Maximum und Minimum

Stell dir vor, du deckst Spielkarten mit Zahlen nacheinander auf. Die erste Karte behältst du in der Hand. Bei jeder weiteren Karte vergleichst du: Ist die neue Zahl größer als die in deiner Hand? Dann tauschst du. Sonst legst du die neue Karte weg. Wenn alle Karten aufgedeckt sind, hältst du die größte Zahl in der Hand.

Im Programm ist die „Karte in der Hand“ wieder eine Hilfsvariable. Das folgende Programm bestimmt das **Maximum** derselben Liste wie oben:

```
1 zahlen = [7, 2, 9, 4, 1, 8]
2 maximum = zahlen[0]
3 for zahl in zahlen:
4     if zahl > maximum:
5         maximum = zahl
6 print("Maximum:", maximum)
```

AUSGABE

```
Maximum: 9
```

- **Zeile 2:** Die Hilfsvariable `maximum` wird mit dem **ersten Element** der Liste initialisiert. Das ist die erste Karte in der Hand.
- **Zeile 4:** In jedem Durchlauf wird geprüft, ob das aktuelle Element größer ist als das bisherige Maximum.
- **Zeile 5:** Nur wenn das der Fall ist, wird `maximum` **aktualisiert**: Es bekommt den neuen, größeren Wert.

Für das **Minimum** geht man genauso vor. Man dreht nur den Vergleich um: < statt >.

Merke

Minimum/Maximum: Eine Hilfsvariable wird mit dem ersten Element initialisiert und bei jedem kleineren bzw. größeren Fund aktualisiert.

Das Ablaufprotokoll für die Liste `[7, 2, 9, 4]` zeigt, dass `maximum` sich nur einmal ändert – beim Element 9:

Schritt	Zeile	Anweisung	zahl	maximum
1	1	`zahlen = [7, 2, 9, 4]`	–	–
2	2	`maximum = zahlen[0]`	–	7
3	3	`for zahl in zahlen:`	7	7
4	4	`if zahl > maximum:`	7	7
5	3	`for zahl in zahlen:`	2	7
6	4	`if zahl > maximum:`	2	7
7	3	`for zahl in zahlen:`	9	7
8	4	`if zahl > maximum:`	9	7
9	5	`maximum = zahl`	9	9
10	3	`for zahl in zahlen:`	4	9
11	4	`if zahl > maximum:`	4	9
12	3	`for zahl in zahlen:`	4	9
13	6	`print("Maximum:", maximum)`	4	9

Stolperstein

Warum beginnt man mit dem ersten Element und nicht einfach mit 0? Weil 0 kein Element der Liste ist. Bei einer Liste aus lauter negativen Zahlen geht das schief:

```
1 temperaturen = [-5, -2, -8]
2 maximum = 0
3 for t in temperaturen:
4     if t > maximum:
5         maximum = t
6 print("Höchste Temperatur:", maximum)
```

AUSGABE

```
Höchste Temperatur: 0
```

Keine der Temperaturen ist größer als 0. Deshalb wird `maximum` nie aktualisiert, und das Programm gibt 0 aus – einen Wert, der gar nicht in der Liste steht. Richtig wäre −2. Mit `maximum = temperaturen[0]` stimmt das Ergebnis.

Arbeitsauftrag 145: Minimum finden

Schreibe eine Funktion `minimum(liste)`, die das kleinste Element einer Liste zurückgibt, ohne die eingebaute Funktion `min()` zu verwenden. Teste sie mit der Liste `[7, 2, 9, 4, 1, 8]`.

1.12.4 Lineare Suche

Du suchst in einem Stapel Klassenarbeiten die von Tom. Die Arbeiten sind nicht sortiert. Also nimmst du die oberste, schaust auf den Namen, dann die nächste – so lange, bis du Toms Arbeit gefunden hast oder der Stapel zu Ende ist. Dieses Vorgehen heißt **lineare Suche**: Man geht die Liste der Reihe nach durch, wie an einer Linie entlang.

Die folgende Funktion sucht einen Wert in einer Liste. Sie gibt den **Index** zurück, an dem der Wert steht. Kommt er nicht vor, gibt sie `-1` zurück:

```
1 def lineare_suche(liste, gesuchter_wert):
2     for i in range(len(liste)):
3         if liste[i] == gesuchter_wert:
4             return i
5     return -1
6
7 namen = ["Ben", "Ada", "Tom", "Mia"]
8 print(lineare_suche(namen, "Tom"))
9 print(lineare_suche(namen, "Lea"))
```

AUSGABE

```
2
-1
```

- **Zeile 2:** Weil die Funktion den Index zurückgeben soll, läuft die Schleife über die **Indizes**.
- **Zeile 3:** Das Element am Index `i` wird mit dem gesuchten Wert verglichen.
- **Zeile 4:** Bei einem Treffer gibt `return i` den Index zurück. `return` beendet die Funktion **sofort** – auch mitten in der Schleife. Die restlichen Elemente werden nicht mehr angesehen.
- **Zeile 5:** Diese Zeile wird nur erreicht, wenn die Schleife bis zum Ende gelaufen ist, ohne etwas zu finden. Dann gibt die Funktion `-1` zurück.
- **Zeilen 8–9:** `"Tom"` steht am Index 2. `"Lea"` kommt nicht vor, deshalb lautet das Ergebnis `-1`.

Das Ablaufprotokoll zeigt die Zeilen **innerhalb** der Funktion beim Aufruf `lineare_suche(namen, "Tom")`:

Schritt	Zeile	Anweisung	i
1	2	`for i in range(len(liste)):`	0
2	3	`if liste[i] == gesuchter_wert:`	0
3	2	`for i in range(len(liste)):`	1
4	3	`if liste[i] == gesuchter_wert:`	1
5	2	`for i in range(len(liste)):`	2
6	3	`if liste[i] == gesuchter_wert:`	2
7	4	`return i`	2

Nach drei Vergleichen ist Tom gefunden (Schritt 6). In Schritt 7 wird `return i` ausgeführt, und die Funktion ist beendet. Mia wird gar nicht mehr angeschaut.

Warum gerade `-1`? Ein Index ist nie negativ, wenn man von vorne zählt. Deshalb ist `-1` ein eindeutiges Zeichen für „nicht gefunden“. Aber Vorsicht: Als **negativer Index** ist `-1` in Python gültig. `namen[-1]` wäre `"Mia"`! Prüfe das Ergebnis der Suche also immer, bevor du es als Index benutzt:

```
position = lineare_suche(namen, "Lea")
if position == -1:
    print("Lea ist nicht in der Liste.")
else:
    print("Lea steht am Index", position)
```

AUSGABE

```
Lea ist nicht in der Liste.
```

Manchmal will man nur wissen, **ob** ein Wert vorkommt. Dann kann die Suche einen Wahrheitswert zurückgeben. Den Index braucht man dafür nicht, also genügt die Schleife über die Elemente:

```
def enthaelt(liste, gesuchter_wert):
    for element in liste:
        if element == gesuchter_wert:
            return True
    return False

namen = ["Ben", "Ada", "Tom", "Mia"]
print(enthaelt(namen, "Mia"))
print(enthaelt(namen, "Lea"))
```

AUSGABE

```
True
False
```

Merke

Lineare Suche: Man durchläuft die Liste Element für Element und vergleicht mit dem Suchwert, bis er gefunden wurde oder die Liste zu Ende ist.

Stolperstein

Ein sehr häufiger Fehler: `return -1` steht in einem `else` **innerhalb** der Schleife.

```
1 def lineare_suche(liste, gesuchter_wert):
2     for i in range(len(liste)):
3         if liste[i] == gesuchter_wert:
4             return i
5         else:
6             return -1
7 
8 namen = ["Ben", "Ada", "Tom", "Mia"]
9 print(lineare_suche(namen, "Tom"))
```

AUSGABE

```
-1
```

Tom steht in der Liste, trotzdem kommt `-1` heraus. Beim ersten Element (`"Ben"`) ist der Vergleich falsch, also wird der `else`-Teil ausgeführt – und `return` beendet die Funktion sofort. Die anderen Namen werden nie verglichen. „Nicht gefunden“ steht erst fest, wenn **alle** Elemente verglichen wurden, also **nach** der Schleife.

So kannst du es sagen

So beschreibst du einen Listen-Algorithmus in Fachsprache:

- *„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.“*

1.12.5 Zählen

Beim Zählen führt man eine **Strichliste**: Jedes Mal, wenn ein Element eine Bedingung erfüllt, kommt ein Strich dazu. Im Programm ist die Strichliste wieder eine Hilfsvariable, die mit 0 beginnt. Sie wird aber – anders als bei der Summe – nur um **1** erhöht, und nur dann, wenn die Bedingung erfüllt ist:

```
1 noten = [2, 3, 1, 4, 2, 5, 1]
2 anzahl = 0
3 for note in noten:
4     if note <= 2:
5         anzahl = anzahl + 1
6 print("Anzahl der Noten 1 oder 2:", anzahl)
```

AUSGABE

```
Anzahl der Noten 1 oder 2: 4
```

Die Noten 2, 1, 2 und 1 erfüllen die Bedingung `note <= 2`. Bei jeder von ihnen wird `anzahl` um 1 erhöht.

Arbeitsauftrag 146: Häufigkeit zählen

Schreibe eine Funktion `zaehle_vorkommen(liste, wert)`, die zählt, wie oft `wert` in `liste` vorkommt. Anders als die lineare Suche, die beim ersten Treffer aufhört, soll die Funktion **alle** Vorkommen zählen. Teste sie mit der Liste `[1, 2, 2, 3, 2, 4]` und dem Wert `2`.

1.12.6 Fertige Funktionen zur Kontrolle

Python bringt für viele dieser Aufgaben fertige Funktionen mit. Du kannst sie ab jetzt benutzen – zum Beispiel, um deine eigenen Funktionen zu überprüfen:

Funktion	Ergebnis
`sum(liste)`	die Summe aller Elemente
`min(liste)`	das kleinste Element
`max(liste)`	das größte Element
`sorted(liste)`	eine **neue**, aufsteigend sortierte Liste
`wert in liste`	`True`, wenn `wert` in der Liste vorkommt, sonst `False`

```
1 zahlen = [7, 2, 9, 4, 1, 8]
2 print(sum(zahlen))
3 print(min(zahlen), max(zahlen))
4 print(sorted(zahlen))
5 print(zahlen)
6 print(9 in zahlen)
```

AUSGABE

```
31
1 9
[1, 2, 4, 7, 8, 9]
[7, 2, 9, 4, 1, 8]
True
```

Beachte Zeile 5: `sorted` verändert die ursprüngliche Liste nicht. Es erzeugt eine neue, sortierte Liste. Den Operator `in` kennst du schon von den Zeichenketten. Bei Listen fragt er, ob ein Element mit diesem Wert vorkommt – im Hintergrund ist das eine lineare Suche.

Warum hast du die Algorithmen dann überhaupt selbst programmiert? Erstens verstehst du nun, was in den fertigen Funktionen passiert. Zweitens gibt es viele ähnliche Aufgaben, für die es keine fertige Funktion gibt: zum Beispiel den **Index** des größten Elements oder die zwei größten Werte. Die kannst du jetzt lösen, indem du die Algorithmen abwandelst.

Tipp

So testest du eine eigene Funktion mit einer fertigen: `print(minimum(zahlen) == min(zahlen))` muss `True` ausgeben. Probiere dabei verschiedene Listen aus, auch eine mit negativen Zahlen und eine mit nur einem Element.

1.12.7 Übungen

Arbeitsauftrag 147: Code lesen

Lies die Funktion, **ohne** sie auszuführen.

```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. Gib an, was das Programm ausgibt.
2. Beschreibe in einem Satz, was die Funktion `geheim` allgemein berechnet.
3. Erkläre, warum das Ergebnis nicht `3` ist, obwohl am Index 3 auch eine 8 steht.

Deine Antworten:

Arbeitsauftrag 148 (Zusatz): Zwei größte Werte

Finde mit einer Schleife die **zwei größten** Werte der Liste `zahlen = [4, 17, 9, 22, 3, 15]` und gib sie aus. Verwende dabei nicht `sorted()`.

Tipp

Merke dir in **zwei** Hilfsvariablen den größten und den zweitgrößten Wert. Initialisiere sie mit den ersten beiden Elementen (in der richtigen Reihenfolge!). Überlege dann für jedes weitere Element: Ist es größer als der bisher größte Wert? Oder nur größer als der zweitgrößte?

1.12.8 Weitere Übungen

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.

Arbeitsauftrag 149: 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)
```

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		

Anzahl der Änderungen:

Arbeitsauftrag 150: 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)
```

Ausgabe und Erklärung:

Arbeitsauftrag 151: 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.

Arbeitsauftrag 152: 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.

Arbeitsauftrag 153: 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.

Arbeitsauftrag 154: Zwei Fehler im Maximum ●●○

Die Funktion soll die beste Tordifferenz einer Saison zurückgeben. Sie liefert falsche Ergebnisse.

```
1 def beste_differenz(differenzen):
2     beste = 0
3     for d in differenzen:
4         if d > beste:
5             beste = d
6         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.

Testergebnisse und Fehler:

Arbeitsauftrag 155: 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).

Arbeitsauftrag 156: 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.

1.12.9 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.

Selbstcheck

- ☐ Ich kann entscheiden, ob ich über die Elemente oder über die Indizes einer Liste iterieren muss.
- ☐ Ich kann die Summe und den Mittelwert der Elemente einer Liste berechnen.
- ☐ Ich kann das Minimum und das Maximum einer Liste ohne `min` und `max` bestimmen und erklären, warum ich mit dem ersten Element beginne.
- ☐ Ich kann eine lineare Suche programmieren, die den Index oder `-1` zurückgibt.
- ☐ Ich kann erklären, warum `return -1` erst nach der Schleife stehen darf.
- ☐ Ich kann zählen, wie oft ein Wert in einer Liste vorkommt.
- ☐ Ich kann meine Ergebnisse mit `sum`, `min`, `max` und `sorted` überprüfen.

Gymnasium Ottweiler

1.13 Projekt Verschlüsselung (Teil 1): die Caesar-Chiffre

Leitidee

Wie verschickt man eine Nachricht so, dass nur Eingeweihte sie lesen können? Schon der römische Feldherr Julius Caesar soll dafür ein einfaches Verfahren benutzt haben: Jeder Buchstabe wird im Alphabet um ein paar Stellen verschoben. In diesem und im nächsten Abschnitt programmiert ihr dieses Verfahren **im Team**. Dabei braucht ihr fast alles, was ihr bisher gelernt habt: Zeichenketten, Schleifen, Verzweigungen, Funktionen und den Rest-Operator `%`. Ihr lernt außerdem, wie man ein Programmierprojekt plant, die Arbeit aufteilt, testet und die Teile am Ende zusammenfügt.

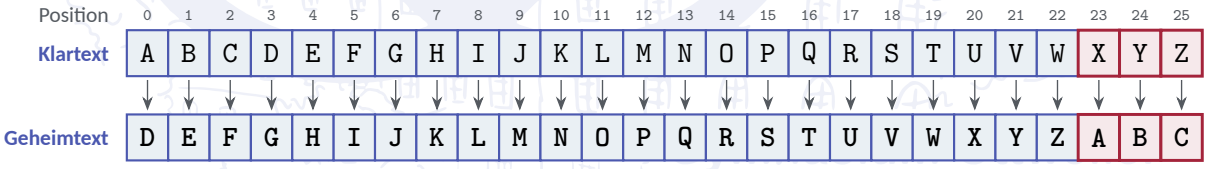
Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.13.1 Wie die Caesar-Chiffre funktioniert

Bei der **Caesar-Chiffre** wird jeder Buchstabe eines Textes im Alphabet um eine feste Anzahl von Stellen nach hinten verschoben. Diese Anzahl ist der **Schlüssel**. Mit dem Schlüssel 3 wird aus A ein D, aus B ein E, aus C ein F und so weiter. Am Ende des Alphabets geht es vorne weiter: Aus X wird A, aus Y wird B, aus Z wird C.



Caesar-Chiffre mit dem Schlüssel 3: Jeder Buchstabe der oberen Reihe wird durch den Buchstaben darunter ersetzt. Am Ende des Alphabets geht es vorne weiter (rot)

So wird aus dem **Klartext** `HALLO` mit dem Schlüssel 3 der **Geheimtext** `KDOOR`: H → K, A → D, L → O, L → O, O → R. Wer den Schlüssel kennt, kann den Geheimtext wieder entschlüsseln, indem er jeden Buchstaben um 3 Stellen **zurück** verschiebt.

Definition: Klartext, Geheimtext, Schlüssel

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.

Arbeitsauftrag 157: Von Hand verschlüsseln

Benutze die Abbildung oben (Schlüssel 3).

1. Verschlüssele das Wort `INFORMATIK`:

I	N	F	O	R	M	A	T	I	K

2. Entschlüssele den Geheimtext `ZDVVHU`.
3. Beschreibe in einem Satz, wie du beim Entschlüsseln vorgegangen bist.

Deine Antworten zu 2 und 3:

1.13.2 Das Projekt im Team planen

Ein Programm, an dem mehrere Personen arbeiten, entsteht nicht dadurch, dass alle gleichzeitig drauflos tippen. Professionelle Softwareteams gehen in Schritten vor. Für euer Projekt sieht das so aus:

1. **Verstehen und planen:** Was genau soll das Programm können? Wie löst man das Problem – erst einmal ohne Computer?
2. **Zerlegen und verteilen:** Das Problem wird in Teilaufgaben zerlegt. Für jede Teilaufgabe wird eine Funktion geplant. Jede Person im Team übernimmt eine Teilaufgabe.
3. **Umsetzen und testen:** Jede Person programmiert ihre Funktion und testet sie mit vorher festgelegten Beispielen.
4. **Zusammenführen:** Die Funktionen werden zu einem Programm zusammengesetzt und gemeinsam getestet.
5. **Dokumentieren und vorstellen:** Das Team hält fest, was das Programm kann, wie es aufgebaut ist und was offen geblieben ist.

Damit ihr gleichzeitig an verschiedenen Funktionen arbeiten könnt, müsst ihr **vorher** festlegen, wie jede Funktion heißt, welche Parameter sie bekommt und was sie zurückgibt. Das nennt man die **Schnittstelle** einer Funktion. Wer die Funktion benutzt, muss nur die Schnittstelle kennen – nicht, wie die Funktion innen arbeitet.

Definition: Schnittstelle einer Funktion

Die **Schnittstelle** einer Funktion legt fest, wie sie heißt, welche Parameter sie erwartet und welchen Rückgabewert sie liefert.

Für euer Projekt sind die Schnittstellen schon vorgegeben:

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)

Teilaufgabe B benutzt die Funktion aus Teilaufgabe A. Trotzdem kann man B schon programmieren, bevor A fertig ist: Man weiß ja, wie A heißt und was A zurückgibt.

Tipp

Du arbeitest den Abschnitt allein zu Hause durch? Dann übernimm alle Teilaufgaben nacheinander. Der Ablauf bleibt derselbe: erst planen, dann jede Funktion einzeln programmieren und testen, dann zusammenfügen.

Arbeitsauftrag 158: Projektplan

Plant gemeinsam auf Papier, **bevor** ihr programmiert. Beantwortet dazu die drei Fragen:

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. Tragt ein, wer welche Teilaufgabe übernimmt und wer sie testet. Zum Testen tauscht ihr: 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:

1.13.3 Werkzeuge für das Projekt

Das Alphabet als Zeichenkette

Das Alphabet speichert ihr am einfachsten als Zeichenkette. Den Index eines Zeichens findet ihr mit `.index()`. Diese Methode kennst du von den Listen. Sie funktioniert bei Zeichenketten genauso:

```
1 alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2 print(alphabet)
3 print("Position von C:", alphabet.index("C"))
4 print("Buchstabe an Position 5:", alphabet[5])
```

AUSGABE

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Position von C: 2
Buchstabe an Position 5: F
```

- **Zeile 1:** Das Alphabet besteht aus 26 Großbuchstaben. A hat den Index 0, Z den Index 25.
- **Zeile 3:** `alphabet.index("C")` geht vom **Zeichen** zur **Position**: Ergebnis 2.
- **Zeile 4:** `alphabet[5]` geht von der **Position** zum **Zeichen**: Ergebnis F.

Stolperstein

Kommt das gesuchte Zeichen nicht im Alphabet vor – zum Beispiel ein Leerzeichen oder ein Kleinbuchstabe –, meldet `.index()` einen Fehler:

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
print(alphabet.index("a"))
```

FEHLERMELDUNG

```
ValueError: substring not found
```

substring not found heißt „Teilzeichenkette nicht gefunden“. Prüft deshalb vorher mit `zeichen in alphabet`, ob das Zeichen überhaupt ein Großbuchstabe ist.

Am Ende des Alphabets: der Rest-Operator

Verschiebt man X (Position 23) um 3, ergibt sich die Position 26. Die gibt es nicht: `alphabet[26]` führt zu einem `IndexError`. Nach Z soll es wieder bei A weitergehen – wie bei einer Uhr, auf der nach 12 wieder die 1 kommt.

Genau das leistet der **Rest-Operator** `%`, den du von den Operatoren kennst. `zahl % 26` ist der Rest bei der Division durch 26. Für Zahlen von 0 bis 25 ändert sich nichts. Aus 26 wird 0, aus 27 wird 1, aus 28 wird 2:

```
1 for position in range(22, 26):
2     neue_position = (position + 3) % 26
3     print(position, "->", neue_position)
```

AUSGABE

```
22 -> 25
23 -> 0
24 -> 1
25 -> 2
```

W (22) wird zu Z (25). X (23) wird zu A (0), Y zu B, Z zu C – genau wie in der Abbildung oben. Die Klammern in Zeile 2 sind wichtig: Zuerst wird addiert, dann der Rest gebildet.

Merke

Mit `(position + schluessel) % 26` erhält man immer eine gültige Position von 0 bis 25. Am Ende des Alphabets geht es dadurch vorne weiter.

Werkzeug

Es geht auch ohne Alphabet-Zeichenkette. Der Computer speichert jedes Zeichen als Zahl. `ord(zeichen)` liefert die Zahl zu einem Zeichen, `chr(zahl)` das Zeichen zu einer Zahl. Die Großbuchstaben A bis Z haben die Zahlen 65 bis 90, der Reihe nach:

```
1 print(ord("A"), ord("B"), ord("Z"))
2 print(chr(68))
3 print(ord("C") - ord("A"))
```

AUSGABE

```
65 66 90
D
2
```

Die Position im Alphabet ist also `ord(buchstabe) - ord("A")`, und zurück geht es mit `chr(position + ord("A"))`. Entscheidet euch im Team für **einen** der beiden Wege.

1.13.4 Teilaufgabe A: einen Buchstaben verschlüsseln

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. Gerade dort stecken die meisten Fehler.

Arbeitsauftrag 159: Teilaufgabe A

Schreibt eine Funktion `verschluessle_buchstabe(buchstabe, schluessel)`, die **einen** Großbuchstaben um `schluessel` Stellen im Alphabet verschiebt und den neuen Buchstaben zurückgibt. Denkt an den Übertrag am Ende des Alphabets (`% 26`). Ihr könnt mit diesem Gerüst beginnen:

```
1 alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2
3 def verschluessle_buchstabe(buchstabe, schluessel):
4     # Position finden, verschieben, mit % 26 umlaufen lassen
5
6 print(verschluessle_buchstabe("A", 3))   # erwartet: D
7 print(verschluessle_buchstabe("X", 3))   # erwartet: A
8 print(verschluessle_buchstabe("Z", 1))   # erwartet: A
```

Das Gerüst läuft so noch nicht: Steht im Funktionsrumpf nur ein Kommentar, meldet Python einen `IndentationError`, denn Kommentare zählen nicht als Anweisungen. Ersetzt den Kommentar durch eure eingerückten Programmzeilen.

Testet die Funktion mit den Aufrufen in der Tabelle. Tragt ein, was eure Funktion zurückgibt, und ob es stimmt.

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		

1.13.5 Teilaufgabe B: einen ganzen Text verschlüsseln

Für Teilaufgabe B braucht ihr ein Muster, das ihr aus dem Abschnitt über Zeichenketten (*Einen neuen Text aufbauen*) kennt. Es funktioniert wie die Summenbildung – nur mit Text statt Zahlen. Man beginnt mit einer **leeren Zeichenkette** `""` und hängt in einer Schleife mit `+` Zeichen an:

```
1 text = "ABC"
2 ergebnis = ""
3 for zeichen in text:
4     ergebnis = ergebnis + zeichen + zeichen
5 print(ergebnis)
```

AUSGABE

```
AABBCC
```

- **Zeile 2:** `ergebnis` beginnt als leere Zeichenkette – so wie `summe` bei 0 beginnt.
- **Zeile 4:** In jedem Durchlauf wird das Zeichen zweimal angehängt. Der neue, längere Text wird wieder in `ergebnis` gespeichert.

Arbeitsauftrag 160: Teilaufgabe B

1. Schreibt eine Funktion `verschluessle_text(text, schluessel)`. Sie soll in einer Schleife jeden Buchstaben eines Textes mit `verschluessle_buchstabe` verschlüsseln und den Geheimtext als neue Zeichenkette zurückgeben. Testet mit `verschluessle_text("HALLO", 3)` (erwartet: `KDOOR`).
2. **Zusatz-Herausforderung:** Was passiert bei Leerzeichen oder Kleinbuchstaben, z. B. bei `"Hallo Welt!"`? Überlegt euch im Team eine Lösung. Zum Beispiel: Leerzeichen und Satzzeichen bleiben unverändert, und der Text wird vorher mit `.upper()` in Großbuchstaben umgewandelt.

1.13.6 Teilaufgabe C: das interaktive Programm

Arbeitsauftrag 161: Teilaufgabe C

Setzt beide Funktionen zu einem vollständigen Programm zusammen. Es soll den Klartext und den Schlüssel mit `input()` abfragen und den Geheimtext ausgeben. Testet das fertige Programm mit mindestens drei verschiedenen Eingaben.

1.13.7 Zusammenführen und dokumentieren

Wenn alle Teilaufgaben fertig sind, fügt ihr sie in **einer** Datei zusammen, und zwar in dieser Reihenfolge: das Alphabet, Funktion A, Funktion B, das Hauptprogramm. Eine Funktion muss definiert sein, bevor sie aufgerufen wird. Testet danach das **ganze** Programm noch einmal mit allen Tests aus den Teilaufgaben.

Ein Programm, das in ein paar Wochen niemand mehr versteht, ist wenig wert. Schreibt deshalb an den Anfang einen kurzen **Kommentar** mit dem Namen des Programms, den Namen des Teams und einem Satz dazu, was das Programm tut. Kommentare beginnen in Python mit `#` und werden beim Ausführen übersprungen.

Arbeitsauftrag 162: Projekttagebuch

Führt ein Projekttagebuch. Tragt am Ende jeder Arbeitsphase ein, was ihr geschafft habt, welche Probleme aufgetreten sind und was als Nächstes zu tun ist. Ihr braucht es im nächsten Abschnitt, um das Projekt fortzusetzen und vorzustellen.

Datum	erledigt	Probleme und wie wir sie gelöst haben	nächster Schritt

1.13.8 Weitere Übungen

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.

Arbeitsauftrag 163: 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.

Meine Lösungen:

Arbeitsauftrag 164: 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.

Meine Erklärung zur letzten Zeile:

Arbeitsauftrag 165: 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))
```

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.

Meine Antworten:

Gymnasium Ottweiler

Arbeitsauftrag 166: 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`.

1.13.9 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** der Funktionen fest, testet jede Funktion mit Normal- und Randfällen und führt die Teile am Ende zusammen.

Im nächsten Abschnitt entwickelt ihr die passende Entschlüsselung und schließt das Projekt ab.

Selbstcheck

- ☐ Ich kann einen kurzen Text von Hand mit der Caesar-Chiffre ver- und entschlüsseln.
- ☐ Ich kann erklären, wozu `% 26` bei der Caesar-Chiffre gebraucht wird.
- ☐ Ich kann die Funktionen `verschluessle_buchstabe` und `verschluessle_text` programmieren.
- ☐ Ich kann eine Funktion mit Normal- und Randfällen testen und das Ergebnis in einer Testtabelle festhalten.
- ☐ Ich kann erklären, was die Schnittstelle einer Funktion ist und warum sie für die Arbeit im Team wichtig ist.

Gymnasium Ottweiler

1.14 Projekt Verschlüsselung (Teil 2) und Berufe in der Softwareentwicklung

Leitidee

Euer Programm kann Texte verschlüsseln. Eine geheime Nachricht nützt aber nur etwas, wenn die Empfängerin oder der Empfänger sie auch wieder **entschlüsseln** kann. In diesem Abschnitt ergänzt ihr diese Funktion, prüft, wie sicher die Caesar-Chiffre wirklich ist, und schließt das Projekt ab: testen, dokumentieren, vorstellen. Im zweiten Teil schaut ihr über den Klassenraum hinaus: Wer entwickelt beruflich Software, welche Berufe gibt es, und wie kommt man dorthin?

Lernziele

Nach diesem Abschnitt kannst du

- 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.

1.14.1 Rückblick: der Stand aus Teil 1

Nehmt eure Ergebnisse aus dem ersten Projektteil und euer Projekttagebuch wieder zur Hand. Kopiert eure fertigen Funktionen an den Anfang eurer Programmdatei. Sie sollten ungefähr so aussehen:

```
1  alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2
3  def verschluessle_buchstabe(buchstabe, schluessel):
4      position = alphabet.index(buchstabe)
5      neue_position = (position + schluessel) % 26
6      return alphabet[neue_position]
7
8  def verschluessle_text(text, schluessel):
9      ergebnis = ""
10     for zeichen in text.upper():
11         if zeichen in alphabet:
12             neu = verschluessle_buchstabe(zeichen, schluessel)
13             ergebnis = ergebnis + neu
14         else:
15             ergebnis = ergebnis + zeichen
16     return ergebnis
17
18 print(verschluessle_text("HALLO WELT", 3))
```

AUSGABE

```
KDOOR ZHOW
```

- **Zeilen 3–6:** `verschluessle_buchstabe` bestimmt die Position des Buchstabens, verschiebt sie um den Schlüssel und lässt sie mit `% 26` am Ende des Alphabets umlaufen.
- **Zeilen 8–16:** `verschluessle_text` geht den Text Zeichen für Zeichen durch. Buchstaben werden verschlüsselt, alle anderen Zeichen (wie das Leerzeichen) unverändert angehängt.

Merke

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.

1.14.2 Teilaufgabe D: Entschlüsseln

Beim Entschlüsseln wird jeder Buchstabe um den Schlüssel **zurück** verschoben. Statt `+ schluessel` rechnet man also `- schluessel`. Was passiert dabei am **Anfang** des Alphabets? A hat die Position 0, und 0 − 3 = −3. Auch hier hilft der Rest-Operator:

```
print((0 - 3) % 26)
print((1 - 3) % 26)
```

AUSGABE

```
23
24
```

In Python liefert `% 26` immer ein Ergebnis von 0 bis 25 – auch bei negativen Zahlen. Aus A (0) wird mit dem Schlüssel 3 also X (23), aus B (1) wird Y (24). Genau so muss es sein.

Es gibt noch einen zweiten Weg. Stell dir das Alphabet als Kreis vor, wie das Zifferblatt einer Uhr mit 26 Stunden. 3 Schritte zurück landen an derselben Stelle wie 23 Schritte vorwärts. Zurückschieben um `s` ist also dasselbe wie Vorwärtsschieben um `26 - s`.

Arbeitsauftrag 167: Teilaufgabe D

Überlegt im Team: Wie hängt die Entschlüsselung mit der Verschlüsselung zusammen?

Schreibt dann eine Funktion `entschluessle_text(geheimtext, schluessel)`. **Tipp:** Entschlüsseln mit dem Schlüssel `s` entspricht dem Verschlüsseln mit dem Schlüssel `-s` (bzw. `26 - s`). Ihr könnt also die Funktion `verschluessle_text` wiederverwenden.

Testet mit einem **Hin-und-zurück-Test**: Verschlüsselt einen Text und entschlüsselt das Ergebnis wieder. Es muss der ursprüngliche Text herauskommen (in Großbuchstaben).

Klartext	Schlüssel	verschlüsselt	wieder entschlüsselt	✓ oder ✗
`HALLO WELT`	3			
`Treffen um acht`	5			

1.14.3 Erweiterungen für schnelle Teams

Arbeitsauftrag 168 (Zusatz): Brute-Force-Entschlüsselung

Baut eine **Brute-Force-Entschlüsselung**: Eine Funktion `brute_force(geheimtext)` probiert für einen gegebenen Geheimtext automatisch alle 26 möglichen Schlüssel durch und gibt alle Ergebnisse aus. So kann man den richtigen Klartext von Hand heraussuchen. Testet mit dem Geheimtext `KDOOR ZHOW`.

Arbeitsauftrag 169 (Zusatz): Menü

Erweitert euer Programm um ein kleines Text-Menü: Die Nutzerin oder der Nutzer wählt, ob ein Text verschlüsselt oder entschlüsselt werden soll oder ob das Programm beendet wird. Das Menü erscheint so lange immer wieder, bis „beenden“ gewählt wurde.

Arbeitsauftrag 170: Wie sicher ist die Caesar-Chiffre?

Beurteile die Sicherheit der Caesar-Chiffre. Gehe dabei auf diese Fragen ein:

1. Wie viele verschiedene Schlüssel gibt es, die den Text tatsächlich verändern?
2. Wie lange braucht ein Programm, das einfach **alle Schlüssel nacheinander ausprobiert** (man nennt das **Brute-Force**, „rohe Gewalt“, siehe die Zusatzaufgabe oben), um einen Geheimtext zu knacken?
3. Was müsste ein sicheres Verfahren anders machen?

Deine Beurteilung:

Gymnasium Ottweiler

1.14.4 Das Projekt abschließen

Zum Abschluss eines Projekts gehört mehr als ein laufendes Programm. Andere sollen verstehen, was es kann und wie es aufgebaut ist. Schreibt deshalb an den Anfang der Datei einen **Kommentarkopf**:

```
# 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.
```

Alle Zeilen beginnen mit `#`. Python überspringt sie beim Ausführen – sie sind nur für Menschen da.

Arbeitsauftrag 171: Projektabschluss

1. Ergänzt euer Programm um einen Kommentarkopf wie oben.
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. Beantwortet darin: 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 und füllt die Tabelle aus.

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:	

Stichpunkte für eure Vorstellung:

Gymnasium Ottweiler

1.14.5 Berufliche Orientierung: Berufe rund um die Softwareentwicklung

Euer Verschlüsselungsprogramm ist klein. Aber 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 Aufgaben und Ausbildungswegen. Manche haben eine Ausbildung gemacht, andere ein Studium, wieder andere sind aus einem ganz anderen Beruf eingestiegen.

Recherchiert in Partner- oder Kleingruppenarbeit zu **einem** Beruf bzw. Berufsfeld mit Bezug zur Softwareentwicklung, 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 (Datenwissenschaft)
- UX/UI-Designerin oder UX/UI-Designer (Gestaltung von Bedienoberflächen)

Definition: Berufsbiografie

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

Im Internet findest du zu jedem Beruf sehr viel – nicht alles davon stimmt, und manches ist Werbung. Beginne deshalb bei Quellen, die neutral informieren:

- **BERUFENET** der Bundesagentur für Arbeit (`berufenet.arbeitsagentur.de`): Steckbriefe zu sehr vielen Berufen mit Aufgaben, Zugangswegen und Ausbildung.
- **planet-beruf.de** und **abi.de**, ebenfalls von der Bundesagentur für Arbeit: Informationen zur Berufs- und Studienwahl für Schülerinnen und Schüler, auch mit Erfahrungsberichten.
- **Hochschulkompass** (`hochschulkompass.de`): Übersicht über Studiengänge an deutschen Hochschulen.
- Die **Berufsberatung** der Agentur für Arbeit und das **Berufsinformationszentrum (BiZ)**. Fragt auch an eurer Schule nach Beratungsterminen.
- Für Berufsbiografien: Interviews und Porträts, z. B. auf den Karriereseiten von Unternehmen. Bedenke: Unternehmen wollen dort auch für sich werben.

Tipp

Prüfe jede Information mit drei Fragen: **Wer** hat sie veröffentlicht? **Wann** wurde sie veröffentlicht oder zuletzt aktualisiert? Wird etwas **verkauft oder beworben**? Wichtige Angaben bestätigst du mit einer zweiten Quelle. Vorsicht bei Antworten von KI-Chatbots: Sie können überzeugend klingende, aber falsche Angaben erfinden (mehr dazu im Kapitel über Sprachmodelle). Schreibe nur auf, was du in einer Quelle gefunden hast, und notiere diese Quelle.

Leitfragen für die Recherche

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 (Berufsbiografie): Ausbildung oder Studium → Berufseinstieg → heutige Tätigkeit.

Arbeitsauftrag 172: Berufsfeld-Steckbrief

Recherchiert zu einem Beruf bzw. Berufsfeld mit Bezug zur Softwareentwicklung. Haltet die Antworten auf die Leitfragen 1 bis 5 in Stichpunkten fest und notiert zu jeder Angabe die Quelle. Für ausführlichere Notizen nutzt ein Extrablatt oder euer Berufsorientierungs-Portfolio.

Stichpunkte	Quelle
Beruf/Berufsfeld	
Aufgaben und Tätigkeiten	
Ausbildung oder Studium	
Werkzeuge und Programmiersprachen	
Veränderung des Berufsbilds	
Meine Einschätzung	–

Arbeitsauftrag 173: Berufsbiografie

Sucht den Werdegang einer **realen** Person aus eurem Berufsfeld, z. B. in einem Interview oder einem Porträt. Tragt die Stationen ein. 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)		

Arbeitsauftrag 174: Kurzbeitrag

Bereitet zu eurem Berufsfeld einen mündlichen Beitrag von etwa einer Minute vor. Er soll drei Teile haben: Was macht man in diesem Beruf? Wie kommt man dorthin? Was hat euch bei der Recherche überrascht?

Stichpunkte für euren Beitrag:

1.14.6 Weitere Übungen

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.

Arbeitsauftrag 175: 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?

Meine Lösungen:

Arbeitsauftrag 176: 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.

Projektschritt – Beruf – Begründung:

Arbeitsauftrag 177: 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.

Häufigster Buchstabe, Schlüssel, Klartext:

Gymnasium Ottweiler

Arbeitsauftrag 178: 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`.

1.14.7 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.

Selbstcheck

- ☐ Ich kann erklären, warum Entschlüsseln mit `s` dasselbe ist wie Verschlüsseln mit `26 - s`.
- ☐ Ich kann eine Funktion `entschluessle_text` schreiben und mit einem Hin-und-zurück-Test prüfen.
- ☐ Ich kann begründen, warum die Caesar-Chiffre unsicher ist.
- ☐ Ich kann ein Programm mit einem Kommentarkopf dokumentieren und kurz vorstellen.
- ☐ Ich kann mindestens einen Beruf mit Bezug zur Softwareentwicklung beschreiben und weiß, wo ich seriöse Informationen dazu finde.

Gymnasium Ottweiler

1.15 Kapitel 1 kompakt: der Lernzettel

Leitidee

Hier steht das Wichtigste aus Kapitel 1 auf wenigen Seiten, zum Wiederholen vor einem GLN. Wenn dir etwas unklar ist, lies im passenden Abschnitt nach: Dort ist alles ausführlich erklärt, mit Beispielen und Aufgaben.

1.15.1 Variablen, Ein- und Ausgabe (Abschnitt 1.1)

- Eine **Variable** ist ein benannter Speicherplatz. Mit `=` bekommt sie einen Wert: `alter = 15` (Zuweisung, gelesen „alter wird 15“).
- **Datentypen:** `int` (ganze Zahl, `15`), `float` (Kommazahl mit Punkt, `1.68`), `str` (Text in Anführungszeichen, `"Hallo"`), `bool` (`True`/`False`).
- `print(...)` gibt aus, mehrere Werte mit Komma getrennt: `print("Alter:", alter)`.
- `input(...)` liest ein. **Das Ergebnis ist immer ein Text (`str`)**, auch wenn eine Zahl eingegeben wird.
- **Typumwandlung:** Zum Rechnen umwandeln.

```
alter = int(input("Alter? "))        # ganze Zahl
groesse = float(input("Größe? "))    # Kommazahl
```

Typische Fehler: `int(...)` vergessen, dann gibt es beim Rechnen einen `TypeError`. Kommazahl mit Komma statt Punkt (`1,68`). Ein Variablenname mit anderer Groß-/Kleinschreibung (`Alter` statt `alter`) führt zum `NameError`.

1.15.2 Operatoren und Bedingungen (Abschnitt 1.2)

Art	Operatoren	Beispiel
Rechnen	`+ - * /`	`7 / 2` ergibt `3.5` (immer `float`)
ganzzahlig teilen, Rest	`//`, `%`	`7 // 2` ergibt `3`, `7 % 2` ergibt `1`
Potenz	`**`	`2 ** 3` ergibt `8`
Vergleich	`== != < > <= >=`	`3 != 4` ergibt `True`
logisch	`and`, `or`, `not`	`True and False` ergibt `False`

- `and`: **beide** Bedingungen müssen wahr sein. `or`: **mindestens eine**. `not`: kehrt um.
- **`=` weist zu, `==` vergleicht.**
- Umrechnen mit `//` und `%`: `200 // 60` ergibt 3 (volle Minuten), `200 % 60` ergibt 20 (Rest-Sekunden). Ob eine Zahl gerade ist: `zahl % 2 == 0`.
- **De Morgan** (eine verknüpfte Bedingung verneinen): Aus `and` wird `or` und umgekehrt, jeder Teil wird verneint.

statt ...	schreibt man ...
`not (A and B)`	`not A or not B`
`not (A or B)`	`not A and not B`
`not (alter >= 12 and groesse >= 140)`	`alter < 12 or groesse < 140`

Das Gegenteil von `>=` ist `<`, das Gegenteil von `<=` ist `>`, das von `==` ist `!=`.

1.15.3 Verzweigungen (Abschnitte 1.3 und 1.4)

```
if bedingung:            # Doppelpunkt!
    anweisungen          # eingerückt: 4 Leerzeichen
elif andere_bedingung:   # beliebig viele, freiwillig
    anweisungen
else:                    # freiwillig, ohne Bedingung
    anweisungen
```

- Es wird **nur der Block der ersten wahren Bedingung** ausgeführt, der Rest wird übersprungen.
- **Einseitig:** nur `if`. **Zweiseitig:** `if … else`. **Mehrseitig:** `if … elif … else`.
- Was eingerückt ist, gehört zum Block. Verzweigungen kann man **verschachteln**.
- Bei Bereichen (unter 1000, 1000 bis unter 2000, ab 2000) von unten nach oben prüfen: `if x < 1000:`, dann `elif x < 2000:`, dann `else:`.
- **Mehrfachauswahl (`CASE OF`)** vergleicht **einen** Wert mit **festen** Werten:

```
match wahl:
    case 1:
        print("Kurzstrecke")
    case 2:
        print("Einzelfahrt")
    case _:              # alle anderen Werte, steht immer zuletzt
        print("Gibt es nicht")
```

Typische Fehler: Doppelpunkt vergessen (`SyntaxError`). = statt == in der Bedingung. Mehrere `if` statt `elif`, dann können zwei Ausgaben kommen. Grenzwert falsch (<= statt <). Bei `match` ohne `int(...)` passt `case 1:` nie, weil die Eingabe ein Text ist.

1.15.4 Schleifen (Abschnitte 1.5 bis 1.7)

Schleife	wann?	Beispiel
`for`	Anzahl der Durchläufe steht **vorher fest**	`for i in range(5):`
`while`	Ende hängt von einer **Bedingung** ab, Anzahl vorher unbekannt	`while zahl != 0:`

- `range(5)` ergibt 0, 1, 2, 3, 4. `range(1, 5)` ergibt 1, 2, 3, 4. `range(0, 10, 3)` ergibt 0, 3, 6, 9. **Der Stoppwert gehört nie dazu.**
- Die **Zählvariable** (`i`) ändert sich bei jedem Durchlauf von selbst.
- In einer `while`-Schleife muss sich die Bedingung **im Rumpf ändern** können, sonst entsteht eine **Endlosschleife**.

Zwei Muster, die du sicher können solltest:

```
summe = 0                        # VOR der Schleife: einmal auf 0
for zahl in range(1, 11):
    summe = summe + zahl         # IN der Schleife: aufaddieren
print("Summe:", summe)           # NACH der Schleife: ausgeben
```

```python
summe = 0
anzahl = 0
wert = float(input("Wert (0 = Ende): "))      # erste Eingabe vor der Schleife
while wert != 0:
    summe = summe + wert
    anzahl = anzahl + 1
    wert = float(input("Wert (0 = Ende): "))  # neue Eingabe am Ende des Rumpfs
print(anzahl, "Werte, Summe:", summe)
```

Typische Fehler: `summe = 0` in die Schleife geschrieben, dann wird die Summe in jedem Durchlauf gelöscht. `range(1, 10)` statt `range(1, 11)`. In der `while`-Schleife die neue Eingabe vergessen (Endlosschleife). `summe + zahl` ohne `summe =` davor ändert nichts.

1.15.5 Funktionen (Abschnitte 1.8 und 1.9)

```python
def flaeche(laenge, breite):     # Definition mit zwei Parametern
    return laenge * breite       # Rückgabewert

ergebnis = flaeche(3, 4)         # Aufruf mit den Argumenten 3 und 4
print(ergebnis)                  # 12
```

- Eine **Funktion** fasst Anweisungen unter einem Namen zusammen und ist wiederverwendbar.
- **Parameter** sind die Platzhalter in der Definition, **Argumente** die Werte beim Aufruf.
- `return` gibt einen **Rückgabewert** zurück und **beendet** die Funktion. Ohne `return` liefert sie `None`.
- Die Definition allein tut nichts. Erst der **Aufruf** führt die Funktion aus.
- Funktionen dürfen andere Funktionen aufrufen.

Typische Fehler: `print` statt `return` in der Funktion, dann kann man mit dem Ergebnis nicht weiterrechnen. Funktion aufgerufen, bevor sie definiert ist. Falsche Anzahl von Argumenten.

1.15.6 Zeichenketten (Abschnitt 1.10)

Was	Beispiel mit `text = "Python"`	Ergebnis
Zeichen am Index	`text[0]`, `text[5]`	`"P"`, `"n"`
Länge	`len(text)`	6
Ausschnitt (Slice)	`text[0:3]`	`"Pyt"` (Index 3 nicht mehr dabei)
Verkettung	`text + "!"`	`"Python!"`
Kleinbuchstaben	`text.lower()`	`"python"`

- Eine Zeichenkette ist eine **Folge von Zeichen**. Der Index beginnt bei **0**, das letzte Zeichen hat den Index `len(text) - 1`.
- Durchlaufen: `for zeichen in text:` geht jedes Zeichen der Reihe nach durch.
- Ein Index, den es nicht gibt (`text[6]`), führt zum `IndexError`.

1.15.7 Listen und Listenalgorithmen (Abschnitte 1.11 und 1.12)

- Eine **Liste** fasst mehrere Elemente zusammen: `noten = [2, 3, 1, 4]`. Der **Index** ist die Position (ab 0), das **Element** der gespeicherte Wert.
- Lesen: `noten[0]` ergibt `2`. Ändern: `noten[1] = 2`. Anhängen: `noten.append(5)`. Länge: `len(noten)`.

- Durchlaufen **über die Elemente** (`for note in noten:`) oder **über die Indizes** (`for i in range(len(noten)):`, wenn man den Index braucht, zum Beispiel zum Ändern).

Die Standardalgorithmen:

```
groesste = zahlen[0]              # Maximum: mit dem ersten Element beginnen
for zahl in zahlen:
    if zahl > groesste:
        groesste = zahl

gefunden = False                  # lineare Suche
for zahl in zahlen:
    if zahl == gesucht:
        gefunden = True
```

Für die **Summe** gilt dasselbe Muster wie bei den Schleifen (vor der Schleife 0, in der Schleife aufaddieren). Für das **Minimum** dreht man den Vergleich um (<).

Typische Fehler: Index und Element verwechseln. Der letzte Index ist `len(liste) - 1`, nicht `len(liste)`. Das Maximum mit 0 statt mit dem ersten Element beginnen (falsch, wenn alle Zahlen negativ sind).

1.15.8 Projekt: Caesar-Verschlüsselung (Abschnitte 1.13 und 1.14)

- Jeder Buchstabe wird um den **Schlüssel** im Alphabet verschoben. Mit Schlüssel 3: A → D, X → A.
- Die Position eines Buchstabens liefert `alphabet.index(buchstabe)`, den Buchstaben an einer Position `alphabet[position]`:

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

- Neue Position: `(position + schluessel) % 26`. Das `% 26` sorgt dafür, dass es nach Z wieder bei A weitergeht.
- **Entschlüsseln** heißt um den Schlüssel zurückschieben: `(position - schluessel) % 26`.
- Ein Text wird Zeichen für Zeichen verschlüsselt, vorher mit `text.upper()` in Großbuchstaben umgewandelt. Eine Hilfsfunktion für einen Buchstaben, eine zweite für den ganzen Text, die die erste in einer Schleife aufruft.

Gymnasium Ottweiler

1.16 Rückblick auf Kapitel 1

Leitidee

Am Ende des Kapitels lohnt sich ein Blick zurück: Welche Fachbegriffe kennst du jetzt? Was kannst du schon sicher, was solltest du noch einmal üben? Die Tabellen auf den nächsten Seiten helfen dir dabei, zum Beispiel bei der Vorbereitung auf eine Klassenarbeit.

1.16.1 Die Basisbegriffe

Diese Begriffe nennt der Lehrplan als **Basisbegriffe**. Du solltest jeden davon mit eigenen Worten erklären und an einem kleinen Beispiel zeigen können.

Begriff	Bedeutung	Beispiel
Variable	benannter Speicherplatz für einen Wert	`alter = 15`
Operator	Zeichen für eine Rechen-, Vergleichs- oder Logikoperation	`+`, `%`, `==`, `and`
Eingabe	Daten, die das Programm von der Nutzerin oder dem Nutzer bekommt	`name = input("Name? ")`
Ausgabe	Daten, die das Programm anzeigt	`print("Hallo", name)`
Bedingung	Ausdruck, der wahr (`True`) oder falsch (`False`) ist	`alter >= 18`
Verzweigung	Je nach Bedingung werden unterschiedliche Anweisungen ausgeführt.	`if … elif … else`
for-Schleife	wiederholt Anweisungen für jeden Wert einer Folge (Zählschleife)	`for i in range(5):`
while-Schleife	wiederholt Anweisungen, solange eine Bedingung wahr ist	`while zahl > 0:`
Funktion	benannter Programmteil, der bei Aufruf ausgeführt wird	`def quadrat(x):`
Parameter	Platzhalter für den Wert, den man der Funktion beim Aufruf übergibt	`x` in `def quadrat(x):`
Rückgabewert	Ergebnis, das eine Funktion mit `return` zurückgibt	`return x * x`
Zeichenkette	Folge (Sequenz) von Zeichen	`"Hallo"`
Liste bzw. Feld	geordnete Zusammenfassung mehrerer Elemente zu einer Einheit	`[4, 7, 1]`
Index	Position eines Elements; die Zählung beginnt bei 0	`zahlen[0]`

Gymnasium Ottweiler

1.16.2 Kompetenz-Check

Kreuze ehrlich an, wie sicher du dich fühlst. Wo du „noch nicht“ ankreuzt, lies den passenden Abschnitt noch einmal und bearbeite seine Arbeitsaufträge.

Ich kann ...	sicher	teilweise	noch nicht
Variablen verwenden, um Daten unterschiedlicher Typen zu speichern.			
Ein- und Ausgabeanweisungen verwenden und Eingaben in Zahlen umwandeln.			
arithmetische, vergleichende und logische Operatoren verwenden.			
verknüpfte Bedingungen mit den Regeln von De Morgan verneinen.			
ein-, zwei- und mehrseitige Verzweigungen analysieren und programmieren.			
eine Mehrfachauswahl (`CASE OF`) mit `match/case` programmieren.			
for- und while-Schleifen analysieren, programmieren und passend auswählen.			
Funktionen mit Parametern und Rückgabewert verwenden und selbst schreiben.			
fremde Programme lesen, erklären, testen und erweitern.			
eigene Algorithmen entwerfen und als Programm umsetzen.			
Zeichenketten als Folge von Zeichen beschreiben und bearbeiten (Verkettung, Länge, Zugriff auf Zeichen).			
Listen anlegen, zwischen Index und Element unterscheiden und Elemente lesen und ändern.			
mit Schleifen über Zeichenketten und Listen laufen.			
Minimum, Maximum, Summe und lineare Suche auf Listen programmieren.			
im Team ein Programmierprojekt umsetzen.			
Berufe mit Bezug zur Softwareentwicklung recherchieren und vorstellen.			