SAE · optional

Python Grundlagen

Kein eigener KA-Stoff laut Arbeitsblättern. Zum Auffrischen, falls dir beim KA-Stoff etwas fehlt. Fehlerbehandlung steht im KA-Lernzettel, weil AB03 bis AB05 sie ausdrücklich verlangen.

Legende:

grün = Definition rot = Achtung / typischer Fehler orange = Merke lila = Erklärung / warum?

AB04 = Bezug zum Arbeitsblatt

Inhalt:

Teil 1 · Python-Grundlagen

1. Datentypen & Variablen
2. Bedingungen & Operatoren
3. Schleifen
4. Listen
5. Tupel
6. Dictionary
7. Set
8. Strings & f-Strings
9. Funktionen
10. Module & Pakete

Teil 2 · OOP (Wiederholung KA 2)

11. Klassen bis abstrakte Klassen

Teil 3 · Hintergrund

12. REST-Prinzipien
13. Netzwerk & WebSocket-Handshake

Teil 1

Python-Grundlagen

Datentypen & Variablen

Eine Variable ist ein Name für einen Wert. Python ist dynamisch typisiert, d.h. der Typ hängt am Wert, nicht an der Variable.

Typ	Bsp	Bedeutung
int	42	Ganzzahl
float	3.14	Kommazahl (Punkt!)
str	"Hallo"	Text
bool	True / False	Wahrheitswert
None	None	„kein Wert"
list	[1, 2, 3]	veränderbar
tuple	(1, 2)	unveränderbar
dict	{"a": 1}	Schlüssel-Wert
set	{1, 2}	ohne Duplikate

Typumwandlung:

`int("42")` → 42 `float("3.5")` → 3.5 `str(42)` → "42"

! `int("abc")` und `float("3,5")` → ValueError

↳ deshalb vorher `.replace(",", ".")`

ABER `input()` gibt immer einen String zurück!

↳ `betrag = float(input("Betrag: "))`

Typ prüfen: `type(x)` oder besser `isinstance(data, list)`

Type Hints:

Hinweise für Menschen und den Editor, welcher Typ erwartet wird. Python selbst prüft sie nicht.

```python
alter: int = 18
def load_csv(path: str) -> list[dict]:      # gibt Liste von Dicts zurück
def parse_log_line(line: str) -> dict | None:   # dict ODER None
```

Bedingungen & Operatoren

Vergleich: `==` `≠` `<` `>` `≤` `≥`

Logisch: `and` `or` `not`

Enthalten: `in` / `not in`

! `=` weist zu, `==` vergleicht

„Falsy" Werte:

↳ 0, "", [], {}, None zählen als False

↳ `if not fehlend:` = „wenn Liste leer"

```python
if betrag < 0:
    raise ValueError("negativ!")
elif betrag > 500:
    rabatt += 5
else:
    pass

# Kurzform (ternärer Ausdruck)
anrede = "Herr" if male else "Frau"

# statt if: kleinster Wert
return min(rabatt, MAX_RABATT)
```

Schleifen

for:

Läuft über jedes Element einer Sammlung (Liste, String, Dict, Datei).

`range(start, stop, schritt)`

! stop ist exklusiv: `range(2, 9, 2)` → 2, 4, 6, 8

while:

Läuft, solange die Bedingung True ist.

- `break` → Schleife sofort beenden
- `continue` → nächster Durchlauf

```python
for user in users:
    print(user["name"])

for i, user in enumerate(users):    # mit Index
    print(i, user["name"])

while True:                         # MQTT-Publisher
    value = input("Temp oder 'ende': ")
    if value.lower() == "ende":
        break

for line in file:                   # AB02
    entry = parse_log_line(line)
    if entry is None:
        statistics["INVALID"] += 1
        continue                    # Rest überspringen
    statistics[entry["level"]] += 1
```

Listen

Geordnete, veränderbare Sammlung. Zugriff über den Index (beginnt bei 0).

Befehl	Wirkung
`lst[0] / lst[-1]`	erstes / letztes Element
`lst[1:3]`	Slicing: Index 1 und 2
`len(lst)`	Anzahl
`lst.append(x)`	hinten anhängen
`lst.remove(x)`	Wert entfernen
`lst.pop()`	letztes entfernen + zurückgeben
`x in lst`	enthalten?
`sorted(lst)`	sortierte Kopie

Merke: JSON und CSV werden meist zu einer Liste von Dicts.

List Comprehension:

Kurzform für „neue Liste aus alter Liste bauen".

```python
# lang
ergebnis = []
for u in users:
    ergebnis.append(simplify_user(u))

# kurz (AB04)
ergebnis = [simplify_user(u) for u in users]

# mit Filter (AB03)
volljaehrig = [k for k in kunden if int(k["alter"]) ⩾ 18]
```

Tupel Wdh. KA 2

Ein Tupel repräsentiert einen Datensatz mit fester Länge und Werten mit potenziell unterschiedlichen Datentypen.

Bsp: `produkt = ("Tesla", "Modell X", 100000)`

Wichtige Eigenschaften:

- Unveränderlich (immutable)
- Unterschiedliche Datentypen möglich
- Zugriff wie Liste über Index

Neu in AB02: Funktionen geben mehrere Werte als Tupel zurück!

```python
# Destrukturierung (Entpacken)
date, time, level, message = parts

# Funktion gibt Tupel zurück (AB02)
def analyze_logfile(path: str) → tuple[dict, list]:
    ...
    return statistics, errors

stats, errors = analyze_logfile("server.log")

# erlaubte Eingaben (AB01)
if wert in ("ja", "j", "yes", "y"):
    return True
```

Dictionary

Datenstruktur, in der man Daten in Form von Schlüssel-Wert-Paaren speichern kann.

Befehl	Wirkung
`d["name"]`	Wert holen, KeyError wenn fehlt
`d.get("name")`	Wert holen, sonst None
`d.get("x", 0)`	sonst Standardwert 0
`d["neu"] = 1`	hinzufügen / ändern
`"k" in d`	Schlüssel vorhanden?
`d.items()`	Paare für for-Schleife
`d.keys()` / `d.values()`	nur Schlüssel / Werte

```python
# Dict statt if/elif-Kette (AB01)
KUNDENTYP_RABATTE = {"normal": 0.0, "silber": 5.0, "gold": 10.0}
rabatt = KUNDENTYP_RABATTE[typ]

# Zähler (AB02)
statistics = {"INFO": 0, "WARNING": 0, "ERROR": 0, "INVALID": 0}
statistics[level] += 1

# verschachtelt (AB04)
user["company"]["name"]                    # KeyError möglich
user.get("company", {}).get("name")        # sicher

for level, anzahl in statistics.items():
    print(level, anzahl)
```

Set (Menge)

Ungeordnete Sammlung ohne Duplikate. Prüfen mit `in` ist sehr schnell.

- `VALID_LEVELS = {"INFO", "WARNING", "ERROR"}`
- `s.add(x)` → hinzufügen (doppelt wird ignoriert)
- `{}` ist ein leeres Dict! Leeres Set = `set()`
- MQTT: schon verarbeitete `message_id`s merken → Duplikate erkennen

```python
processed_message_ids = set()

if msg_id in processed_message_ids:
    print("Duplikat ignoriert")
    return
processed_message_ids.add(msg_id)
```

Strings & f-Strings

Ein String ist eine Zeichenkette, deshalb verhält er sich wie eine Liste von Buchstaben. Strings sind unveränderlich.

Methode	Wirkung
`" ja\n".strip()`	Leerzeichen/Zeilenumbruch am Rand weg → "ja"
`"Gold".lower()`	alles klein → "gold"
`"a b c".split(" ")`	→ ["a", "b", "c"]
`s.split(" ", 3)`	höchstens 3× trennen → 4 Teile
`", ".join(lst)`	Liste → ein String
`s.replace(",", ".")`	ersetzen
`s.startswith("ERR")`	True/False
`"ERROR" in s`	enthält Teilstring?
`s[0:10]`	Slicing (Zeichen 0–9)

f-Strings:

```python
name = "Ali"; betrag = 89.1
print(f"Kunde: {name}")
print(f"Betrag: {betrag:.2f} €")   # 89.10 €
print(f"{len(errors)} Fehler")     # Ausdrücke erlaubt
```

Bytes ↔ String:

Über das Netzwerk kommen Bytes, nicht Text!

`text = message.payload.decode("utf-8")`

`daten = text.encode("utf-8")`

! Feste Positionen wie `line[20:25]` sind fragil → lieber `split()`

Funktionen / Methoden

Schlüsselwort Name (Parameterliste) → Rückgabetyp :

```python
def berechne_endpreis(betrag: float, rabatt: float) -> float:
    """Berechnet den Endpreis nach Rabatt."""    # Docstring
    return betrag - betrag * rabatt / 100

# Standardwert für Parameter (AB04)
def fetch_users(url: str = URL) -> list[dict]:
    ...

# Einstiegspunkt
def main() -> None:
    ...

if __name__ == "__main__":
    main()
```

- **Parameter** = Platzhalter in der Definition
- **Argument** = echter Wert beim Aufruf
- **return** gibt einen Wert zurück und beendet die Funktion
- ohne return → Funktion gibt `None` zurück
- mehrere Werte → `return a, b` (Tupel)
- Variablen in der Funktion sind lokal

Warum Rückgabewert? Damit man das Ergebnis weiterverwenden kann (speichern, testen, weiterrechnen). `print` zeigt nur an.

`if __name__ == "__main__":` → Code läuft nur beim direkten Start, nicht beim `import`.

Konstanten: GROSS_GESCHRIEBEN, oben im Modul.

Module & Pakete

Ein Modul ist eine Python-Datei mit Funktionen/Klassen, die man mit `import` nutzen kann.

- `import json` → `json.load(...)`
- `from pathlib import Path` → direkt `Path(...)`
- `import paho.mqtt.client as mqtt` → Alias

↳ **Standardbibliothek** = schon dabei

↳ **Externe Pakete** = installieren:

```
pip install requests flask paho-mqtt websockets
```

Modul	wofür?
`json, csv`	Dateiformate
`pathlib`	Pfade
`argparse`	Kommandozeilen-Parameter
`uuid, time, datetime`	IDs, Zeitstempel, Datum
`urllib.request`	HTTP ohne Zusatzpaket
`asyncio`	asynchroner Code
`requests extern`	HTTP-Client
`flask extern`	Webserver / REST-API
`paho-mqtt extern`	MQTT-Client
`websockets extern`	WebSocket

OOP Wiederholung KA 2

Klasse & Objekt

Klasse = abstrakter Bauplan. Definiert Attribute (Daten) und Methoden (Verhalten).

Objekt = konkretes Exemplar (Instanz) einer Klasse.

Objektkarte:

person1
name = "Uta" alter = 54

Klassenkarte:

Person
name, alter
gehen()

Konstruktor:

Der Konstruktor `__init__` wird automatisch aufgerufen, wenn ein Objekt erstellt wird.

self ist eine Referenz auf das aktuelle Objekt. Damit greift man auf dessen Attribute und Methoden zu.

```python
class Person:
    def __init__(self, name: str, alter: int):
        self.name = name
        self.alter = alter

    def gehen(self):
        print("taps, taps")

p = Person("Uta", 54)     # Objekt erzeugen
p.gehen()
```

Datenkapselung

- \+ Öffentlich (public): von außen lesbar und änderbar
- \- Privat (private): nur innerhalb der Klasse → `self.__attribut`

Getter: holt den Wert eines privaten Attributs

Setter: ändert den Wert, kontrolliert z.B. mit Bedingungen

Warum? Daten werden geschützt, ungültige Werte (Alter -5) werden verhindert.

Genau genommen: Python kennt kein echtes „privat". `__alter` wird nur umbenannt (Name Mangling → `_Person__alter`), `_alter` heißt per Konvention „bitte nicht von außen benutzen".

```python
class Person:
    def __init__(self):
        self.__alter = 0           # privat

    def get_alter(self):           # Getter
        return self.__alter

    def set_alter(self, wert):     # Setter
        if wert ≥ 0:
            self.__alter = wert
        else:
            print("Ungültiges Alter!")
```

Vererbung · Polymorphie · Abstrakte Klassen

Vererbung: Kindklasse erbt Attribute und Methoden der Elternklasse.

↳ `class Auto(Fahrzeug):` · `super().__init__(...)` = Konstruktor der Oberklasse

Polymorphie: gleiche Methode, unterschiedliches Verhalten je nach Klasse.

↳ Methode in Kindklasse neu definieren (überschreiben)

↳ alle Objekte einheitlich behandeln, z.B. in einer Liste

Abstrakte Klasse: Vorlage, von der man keine Objekte direkt erstellen kann. Abstrakte Methoden müssen in den Unterklassen umgesetzt werden.

```python
from abc import ABC, abstractmethod

class Tier(ABC):
    def __init__(self, name):
        self.name = name
    @abstractmethod
    def gib_laut(self): pass

class Hund(Tier):
    def gib_laut(self): print("Wuff")

class Katze(Tier):
    def gib_laut(self): print("Miau")

for t in [Hund("Buddy"), Katze("Garfield")]:
    t.gib_laut()        # Polymorphie
```

Spezielle Methoden („Dunder"):

Methode	wird benutzt bei
`__init__`	Objekt erstellen
`__str__`	`print(obj)` / `str(obj)`, ohne → `<Person object at 0x…>`
`__gt__` / `__lt__`	`p1 > p2` / `p1 < p2`
`__eq__`	`p1 == p2`

Teil 3

Hintergrund Schnittstellen

Steht so nicht in den Arbeitsblättern. Hilft beim Verstehen, ist aber kein eigener KA-Stoff.

REST-Prinzipien

REST ist ein Architekturstil für Web-APIs: Daten sind Ressourcen mit eigener URL, die man über HTTP-Methoden bearbeitet.

Prinzipien:

- Client-Server-Trennung
- Zustandslos: jede Anfrage enthält alle nötigen Infos
- Ressourcen haben URIs: `/users` , `/users/1`
- HTTP-Methode sagt, was passiert
- Daten meist als JSON

! URL = Nomen, nicht Verb: `/users` statt `/getUsers`

Idempotent = mehrmals ausführen hinterlässt denselben Zustand wie einmal.

↳ GET, PUT, DELETE: ja · POST: nein (legt jedes Mal neu an)

PUT ersetzt die Ressource vollständig, PATCH ändert nur die mitgeschickten Felder.

Zustandslos: Der Server merkt sich zwischen zwei Anfragen nichts, jede Anfrage bringt alle Infos selbst mit.

Netzwerk & WebSocket-Handshake

- HTTP, MQTT und WebSocket laufen über **TCP** (Ausnahme: HTTP/3 nutzt QUIC über UDP).
- WebSocket startet als HTTP-Request mit `Upgrade: websocket` , der Server antwortet `101 Switching Protocols` . Danach bleibt die Verbindung offen.
- „Alle 2 Sek. per GET nachfragen" (Polling) funktioniert, verschwendet aber Ressourcen. Deshalb WebSocket für Live-Daten.
- In JSON sind Schlüssel immer Strings: `json.dumps({1: "a"})` → `'{"1": "a"}'`