SAE · 3. Lehrjahr · KA-Stoff

Python KA Datei-I/O & Schnittstellen

Nur der Stoff aus AB01 bis AB05 und den Dateien der MQTT/WebSocket-Übung vom 09.10. Python-Grundlagen und OOP stehen im separaten Lernzettel „Grundlagen" (optional).

Legende:

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

AB04 = Bezug zum Arbeitsblatt

Inhalt:

AB01 · Sauberer Code

1. Refactoring, Validierung, Tests
2. Fehlerbehandlung

AB02 · AB03 · Dateien

3. Dateien öffnen
4. CSV
5. JSON
6. Import mit Config (AB03)
7. Log-Analyse (AB02)

AB04 · AB05 · 09.10. · Schnittstellen

8. Grundbegriffe
9. HTTP
10. CRUD (AB05)
11. REST-Client (AB04)
12. REST-Server (AB05)
13. MQTT
14. WebSocket & asyncio
15. Vergleich + KA-Check

Sauberer Code & Tests

Refactoring AB01

Refactoring = Code verbessern (lesbarer, wartbarer, testbarer), ohne dass sich das Verhalten ändert.

Schwachstellen (rabatt_rechner_schlecht.py):

1. `input()` ist str → `betrag > 0` gibt TypeError
2. Magic Numbers (5, 10, 20, 500) mitten im Code
3. Logik nicht in Funktionen / Funktion ohne return
4. keine Validierung (negativ, unbekannter Kundentyp)
5. „Ja" / „ ja" wird nicht erkannt
6. Geldbeträge nicht formatiert (89.1000001)
7. Variable überschreibt Funktion (`rabatt_berechnung = int(rabatt_berechnung(...))`)
8. keine Tests, kein `main()`

Verbesserungen:

- Funktionen mit sprechenden Namen (`berechne_rabatt`)
- Konstanten: `MAX_RABATT = 20.0`
- Dict statt if/elif-Kette
- Validierung mit `raise ValueError`
- Eingabe normalisieren: `.strip().lower()`
- Ausgabe: `f"{x:.2f}"`
- Type Hints + Docstrings
- Eingabe/Ausgabe (main) getrennt von Logik

Warum trennen? Reine Logik-Funktionen ohne `input()` kann man automatisch testen.

Validierung

Validierung = prüfen, ob Daten gültig sind, bevor man mit ihnen weiterarbeitet.

Typ richtig? → Pflichtfeld da? → Wertebereich ok? → erlaubter Wert?

↳ Bsp: Betrag ≥ 0 · Kundentyp in {normal, silber, gold} · alter ist Zahl · email nicht leer

Tests mit assert

`assert bedingung` → wenn False, gibt es einen AssertionError. Kein Fehler = Test bestanden.

Was testen?

- **Normalfall:** `berechne_rabatt(100, "silber", False) == 5.0`
- **Grenzfall:** genau 500, 0 €, Maximum 20 %
- **Fehlerfall:** negativer Betrag muss ValueError werfen

In der KA: mind. 3 Tests, je einer pro Fall-Art.

```python
def teste_rabatt() -> None:
    assert berechne_rabatt(100, "normal", False) == 0.0
    assert berechne_rabatt(100, "gold", True) == 12.0
    assert berechne_rabatt(600, "gold", True) == 17.0
    assert berechne_rabatt(900, "gold", True) <= 20.0

    # Fehlerfall: MUSS ValueError auslösen
    try:
        berechne_rabatt(-1, "normal", False)
        raise AssertionError("Negativ nicht erkannt")
    except ValueError:
        pass
```

Fehlerbehandlung (Exceptions)

AB01 · AB03–AB05

Eine Exception ist ein Fehler zur Laufzeit. Wird sie nicht abgefangen, bricht das Programm ab.

```python
try:
    with open("config.json", encoding="utf-8") as f:
        config = json.load(f)
except FileNotFoundError:
    print("Datei fehlt")
except json.JSONDecodeError as error:
    print(f"Kein gültiges JSON: {error}")
else:
    print("läuft nur, wenn KEIN Fehler kam")
finally:
    print("läuft IMMER (z.B. Verbindung schließen)")
```

Selbst Fehler auslösen:

`raise ValueError("Betrag darf nicht negativ sein.")`

Fehler „übersetzen":

`raise RuntimeError("Timeout") from error`

Der Aufrufer muss nur noch RuntimeError abfangen, die Ursache bleibt erhalten.

Regeln:

1. Spezielle Fehler vor allgemeinen abfangen
2. Mehrere zusammen: `except (OSError, ValueError):`
3. Kein nacktes `except:`, das verschluckt alles
4. Nur da abfangen, wo man sinnvoll reagieren kann

Exception	Wann?	Bsp
ValueError	richtiger Typ, falscher Wert	`int("abc")`
TypeError	falscher Typ	`"5" > 0`
KeyError	Schlüssel fehlt im Dict	`d["fehlt"]`
IndexError	Index zu groß	`lst[99]`
FileNotFoundError	Datei existiert nicht	`open("x.csv")`
PermissionError	keine Rechte	`open("/x", "w")`
json.JSONDecodeError	kaputtes JSON	`json.loads("{a:")`
ZeroDivisionError	durch 0 teilen	`1 / 0`
AssertionError	assert schlägt fehl	`assert 1 == 2`

AB02 · AB03

Dateien (Datei-I/O)

Dateien öffnen AB02 + AB03

I/O = Input/Output. Programme lesen Daten aus Dateien (Input) und schreiben Ergebnisse in Dateien (Output).

```python
with open("server.log", "r", encoding="utf-8") as file:
    for line in file:            # zeilenweise
        print(line.strip())      # \n am Ende weg

inhalt = file.read()             # alles als ein String
zeilen = file.readlines()        # Liste von Zeilen

from pathlib import Path
pfad = Path("users.json")
if pfad.exists():
    with pfad.open("r", encoding="utf-8") as f:
        ...
```

Modus	Bedeutung
"r"	lesen (Standard)
"w"	schreiben, überschreibt!
"a"	anhängen

Warum with?

Die Datei wird automatisch geschlossen, auch wenn ein Fehler passiert.

Immer `encoding="utf-8"` angeben, sonst Probleme mit ä, ö, ü.

! Relative Pfade gelten ab dem Ordner, aus dem das Programm gestartet wird.

CSV

CSV = Comma Separated Values. Tabelle als Textdatei: 1. Zeile = Kopfzeile (Header), danach ein Datensatz pro Zeile, Werte durch Komma getrennt.

```
id,name,email,alter
1,Ali Demir,ali@example.com,24
3,Sam Keller,sam@example.com,16
```

- `csv.reader` → jede Zeile eine Liste
- `csv.DictReader` → jede Zeile ein Dict (Header = Schlüssel)
- `csv.writer` + `writerow([...])`
- `csv.DictWriter(f, fieldnames=[...])` + `writeheader()` + `writerows(liste)`

! Aus CSV kommt alles als String → `int(k["alter"])`

! Beim Schreiben `newline=""`, sonst Leerzeilen (Windows)

```python
import csv

# lesen
with open("kunden.csv", encoding="utf-8") as f:
    kunden = list(csv.DictReader(f))
# → [{"id": "1", "name": "Ali Demir", ...}, ...]

# schreiben
with open("users.csv", "w", encoding="utf-8",
          newline="") as f:
    writer = csv.DictWriter(
        f, fieldnames=["id", "name", "email"])
    writer.writeheader()
    writer.writerows(users)

# einfache Tabelle (AB02)
writer = csv.writer(f)
writer.writerow(["level", "anzahl"])
writer.writerow(["INFO", 3])
```

JSON

JSON = JavaScript Object Notation. Textformat für strukturierte (auch verschachtelte) Daten. Standardformat bei REST-APIs und MQTT-Payloads.

JSON	Python
`{ "a": 1 }` Objekt	`dict`
`[1, 2]` Array	`list`
`"Text"`	`str`
`1` / `2.5`	`int` / `float`
`true` / `false`	`True` / `False`
`null`	`None`

! In JSON nur doppelte Anführungszeichen, kein Komma nach dem letzten Element.

```python
import json

with open("config.json", encoding="utf-8") as f:
    config = json.load(f)
min_alter = config["min_alter"]

with open("errors.json", "w", encoding="utf-8") as f:
    json.dump(errors, f, indent=4, ensure_ascii=False)
```

	Datei	String
lesen	`json.load(f)`	`json.loads(s)`
schreiben	`json.dump(obj, f)`	`json.dumps(obj)`

Merke: **s** = string. MQTT-Payload → `loads` / `dumps`.

`indent=4` → schön eingerückt

`ensure_ascii=False` → ä bleibt ä (statt `\u00e4`)

Serialisierung = Python-Objekt → Text (dumps),
Deserialisierung = Text → Objekt (loads)

	CSV	JSON
Struktur	flache Tabelle	verschachtelt möglich
Datentypen	nur Text	Zahl, bool, null …
gut für	Excel, Listen	APIs, Config
Größe	kompakt	mehr Overhead

CSV/JSON-Import mit Config

AB03

Einstellungen (Grenzwerte, Dateinamen, Pflichtfelder) stehen in einer Konfigurationsdatei statt fest im Code. So kann man sie ändern, ohne zu programmieren.

kunden.csv lesen → config.json lesen → validieren → filtern (min_alter) → export_datei schreiben

dateiimport.py · Musterlösung

```python
import csv
import json


def load_csv(path: str) -> list[dict]:
    with open(path, encoding="utf-8") as f:
        return list(csv.DictReader(f))


def load_config(path: str) -> dict:
    with open(path, encoding="utf-8") as f:
        return json.load(f)


def ist_gueltig(kunde: dict, pflichtfelder: list[str]) -> bool:
    for feld in pflichtfelder:
        if not kunde.get(feld):               # fehlt oder leer
            return False
    return kunde["alter"].isdigit()           # Alter muss Zahl sein


def main() -> None:
    try:
        config = load_config("config.json")
        kunden = load_csv("kunden.csv")
    except FileNotFoundError as error:
        print(f"Datei fehlt: {error.filename}")
        return
    except json.JSONDecodeError:
        print("config.json ist kein gültiges JSON")
        return

    gueltig = [k for k in kunden if ist_gueltig(k, config["pflichtfelder"])]
    volljaehrig = [k for k in gueltig if int(k["alter"]) >= config["min_alter"]]

    with open(config["export_datei"], "w", encoding="utf-8") as f:
        json.dump(volljaehrig, f, indent=4, ensure_ascii=False)


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

Log-Analyse AB02

Ein Log ist eine Protokolldatei: jede Zeile ein Ereignis mit Zeitstempel, Level und Nachricht.

```
2026-09-01 08:17:02 ERROR Verbindung zur Datenbank fehlgeschlagen
FEHLERHAFTE ZEILE OHNE LEVEL
```

Ablauf:

1. Datei zeilenweise lesen
2. Zeile zerlegen: `split(" ", 3)` → Datum, Zeit, Level, Nachricht
3. ungültig (≠ 4 Teile oder unbekanntes Level) → INVALID zählen
4. Level im Dict hochzählen
5. ERROR-Einträge in Liste sammeln
6. Statistik → `log_statistik.csv`
7. Fehler → `errors.json`

python

```python
VALID_LEVELS = {"INFO", "WARNING", "ERROR"}

def parse_log_line(line: str) -> dict | None:
    parts = line.strip().split(" ", 3)
    if len(parts) != 4:
        return None
    date, time, level, message = parts
    if level not in VALID_LEVELS:
        return None
    return {"timestamp": f"{date} {time}",
            "level": level,
            "message": message}
```

Warum `split(" ", 3)` ? Die Nachricht selbst enthält Leerzeichen und soll ein Stück bleiben.

Bonus: argparse

Parameter beim Programmstart übergeben statt fest im Code: `python log.py --log server.log --csv out.csv`

```python
parser = argparse.ArgumentParser(description="Log-Analysator")
parser.add_argument("--log", required=True, help="Pfad zur Log-Datei")
args = parser.parse_args()
print(args.log)
```

Schnittstellen

Grundbegriffe

Eine Schnittstelle (API = Application Programming Interface) ist eine festgelegte Art, wie zwei Programme miteinander Daten austauschen.

- **Client**: stellt Anfragen (z.B. Python-Skript, Browser)
- **Server**: wartet auf Anfragen und antwortet
- **Protokoll**: Regeln der Kommunikation (HTTP, MQTT, WebSocket)
- **Endpunkt**: Adresse einer Funktion der API, z.B. `/users/1`
- **Payload**: die eigentlichen Nutzdaten
- **localhost** = `127.0.0.1` = der eigene Rechner

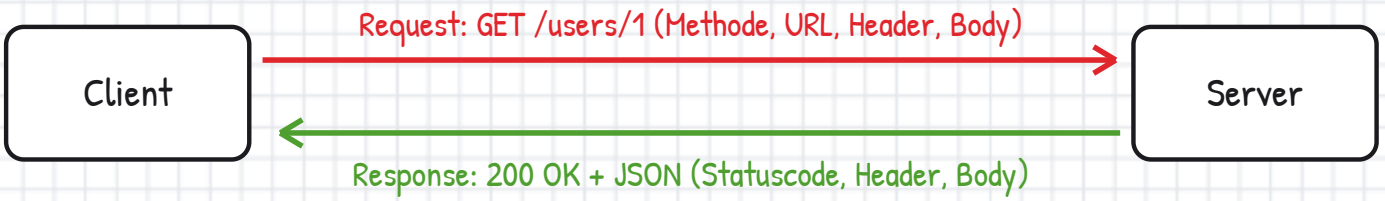
Port:

Nummer, die den Dienst auf einem Rechner identifiziert (IP = Haus, Port = Wohnungstür).

Port	Dienst
80 / 443	HTTP / HTTPS
1883	MQTT (unverschlüsselt)
8000	unser Flask-Server
8765	unser WebSocket-Server

HTTP

HTTP = Hypertext Transfer Protocol. Der Client schickt einen Request, der Server antwortet mit einer Response. Danach ist die Anfrage abgeschlossen.



danach ist die Anfrage abgeschlossen

URL-Aufbau:

`http://localhost:8000/users/1?sort=name`

Protokoll · Host · Port · Pfad · Query-Parameter

Header:

Zusatzinfos zur Nachricht.

`Content-Type: application/json` → „im Body steht JSON"

Statuscodes (AB05):

Klasse	Bedeutung
2xx	Erfolg 200 OK · 201 Created · 204 No Content
4xx	Fehler des Clients 400 · 404 · 405
5xx	Fehler des Servers 500

Merke: 4xx = „du hast was falsch gemacht", 5xx = „ich (Server) hab was falsch gemacht".

REST & CRUD

AB04 + AB05

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

CRUD = Create, Read, Update, Delete: grundlegende Operationen auf gespeicherte Daten (so in AB05).

CRUD	Methode	Endpunkt	Body?	Erfolg
Read alle	GET	/users	nein	200 OK
Read einen	GET	/users/{id}	nein	200 OK
Create	POST	/users	JSON	201 Created
Update	PUT	/users/{id}	JSON	200 OK
Delete	DELETE	/users/{id}	nein	204 No Content

Fehler	Situation
400 Bad Request	ungültiges JSON, Pflichtfeld fehlt, ID ist keine Zahl (/users/abc)
404 Not Found	User mit dieser ID gibt es nicht
405 Method Not Allowed	Methode nicht erlaubt (z.B. DELETE auf /users)
500 Internal Server Error	Server kann Datei nicht lesen/speichern

PUT = User vollständig ändern (AB05, Aufgabe 7).

REST-Client mit requests AB04

Befehl	Wirkung
requests.get(url, timeout=5)	GET-Anfrage
requests.post(url, json=d)	JSON im Body senden
r.status_code	z.B. 200
r.raise_for_status()	HTTPError bei ≥ 400
r.json()	Body → Python-Objekt
r.text / r.headers	Body als Text / Header

4 Fehlerarten (AB04):

1. `requests.Timeout` → Server antwortet nicht rechtzeitig
2. `requests.ConnectionError` → kein Netz / Host falsch
3. `requests.HTTPError` → Status ≥ 400
4. `ValueError` → `.json()` mit ungültigem JSON

↳ alle requests-Fehler erben von `requests.RequestException` → zuletzt abfangen

! Auch der JSON-Fehler (`requests.JSONDecodeError`) erbt seit requests 2.27 von `RequestException` und `ValueError` → `except ValueError` muss vor `RequestException` stehen.

! Ohne `timeout=` kann das Programm ewig hängen.

python

```python
import requests

URL = "https://jsonplaceholder.typicode.com/users"

def fetch_users(url: str = URL) -> list[dict]:
    try:
        response = requests.get(url, timeout=5)
        response.raise_for_status()
        data = response.json()
    except requests.Timeout as e:
        raise RuntimeError("Timeout") from e
    except requests.ConnectionError as e:
        raise RuntimeError("Netzwerkfehler") from e
    except requests.HTTPError as e:
        raise RuntimeError(f"HTTP-Fehler: {e}") from e
    except ValueError as e:          # vor RequestException!
        raise RuntimeError("Kein gültiges JSON") from e
    except requests.RequestException as e:
        raise RuntimeError(f"Request-Fehler: {e}") from e

    if not isinstance(data, list):
        raise RuntimeError("Liste erwartet")
    return data

def simplify_user(user: dict) -> dict:
    return {"id": user.get("id"),
            "name": user.get("name"),
            "email": user.get("email"),
            "company": user.get("company", {}).get("name")}
```

↳ Alternative ohne Zusatzpaket: `urllib.request.urlopen(url)` (AB05-Testclient), Fehler dort: `urllib.error.HTTPError` mit `e.code`

REST-Server mit Flask AB05

Flask ist ein kleines Web-Framework. Mit `@app.route` verbindet man eine URL + Methode mit einer Python-Funktion.

Bausteine:

- `app = Flask(__name__)`
- `@app.route("/users", methods=["GET"])`
- `/users/<raw_id>` → Teil der URL wird Parameter
- `request.get_json(silent=True)` → Body als dict (oder None)
- `return jsonify(daten), 201` → JSON + Header + Status
- `return "", 204` → leere Antwort
- `app.run(port=8000)`

405 erzeugt Flask automatisch, wenn die Methode nicht in `methods=[...]` steht.

Ablauf einer Anfrage:

1. ID prüfen (Zahl?) → sonst 400
2. Body prüfen (JSON? Pflichtfelder?) → sonst 400
3. Daten laden → Fehler: 500
4. User suchen → nicht da: 404
5. ändern / speichern
6. Erfolgscode + JSON zurück

Testen:

↳ GET → einfach im Browser `http://localhost:8000/users`

↳ POST/PUT/DELETE → Client-Skript, curl oder Postman

rest_server.py (Kern)

```python
from flask import Flask, jsonify, request
app = Flask(__name__)

@app.route("/users", methods=["GET"])
def get_all_users():
    return jsonify(load_users()), 200

@app.route("/users/<raw_id>", methods=["GET"])
def get_user(raw_id):
    try:
        user_id = int(raw_id)
    except ValueError:
        return jsonify({"error": "Ungültige ID"}), 400
    for user in load_users():
        if user["id"] == user_id:
            return jsonify(user), 200
    return jsonify({"error": "Nicht gefunden"}), 404

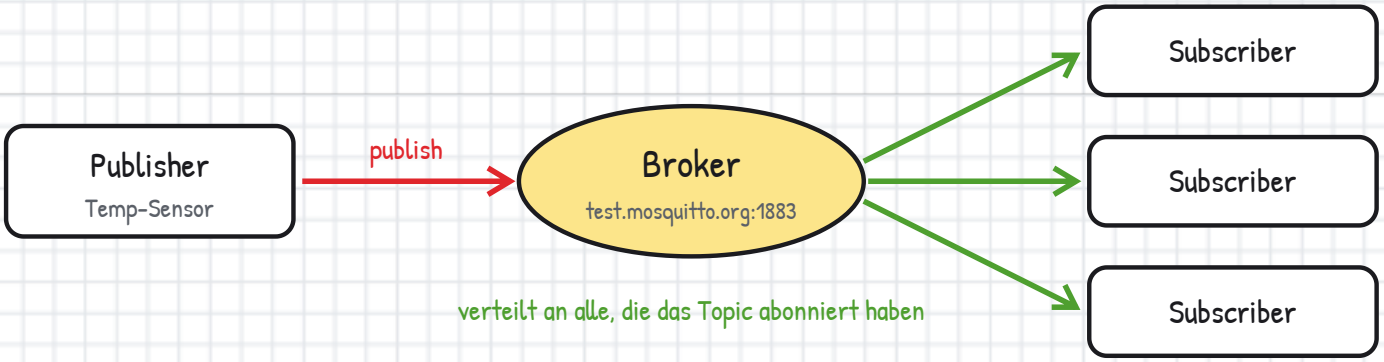
@app.route("/users", methods=["POST"])
def create_user():
    data = request.get_json(silent=True)
    if not isinstance(data, dict):
        return jsonify({"error": "Ungültiges JSON"}), 400
    for feld in ("name", "username", "email"):
        if not data.get(feld):
            return jsonify({"error": f"{feld} fehlt"}), 400
    users = load_users()
    neue_id = max((u["id"] for u in users), default=0) + 1
    user = {"id": neue_id, "name": data["name"],
            "username": data["username"],
            "email": data["email"]}
    users.append(user)
    save_users(users)
    return jsonify(user), 201

@app.route("/users/<raw_id>", methods=["DELETE"])
def delete_user(raw_id):
    ...                 # ID prüfen, suchen, entfernen
    return "", 204      # kein Body!

if __name__ == "__main__":
    app.run(port=8000)
```

MQTT 09.10.

MQTT (Message Queuing Telemetry Transport) ist ein leichtes Nachrichtenprotokoll nach dem Publish/Subscribe-Prinzip. Typisch für IoT und Sensoren.



Begriffe:

- **Broker**: zentraler Vermittler, empfängt und verteilt
- **Publisher**: sendet Nachrichten an ein Topic
- **Subscriber**: abonniert Topics, bekommt Nachrichten
- **Topic**: „Adresse" mit Ebenen, getrennt durch `/`
- **Client-ID**: muss eindeutig sein
- **Payload**: Inhalt (Bytes), meist JSON

Vorteil: Sender und Empfänger kennen sich nicht (entkoppelt). Ein Sensor, beliebig viele Empfänger.

Wildcards (nur beim Abonnieren):

`+` = genau eine Ebene

↳ `haus/+/temp` passt auf `haus/kueche/temp`, nicht `haus/eg/kueche/temp`

`#` = beliebig viele Ebenen, nur am Ende

↳ `gbsha/G305/87/#` passt auf alles darunter **und** auf `gbsha/G305/87` selbst

! Topics sind case-sensitive: `G305` ≠ `g305`

QoS (Quality of Service):

QoS	Garantie
0	höchstens einmal (kann verloren gehen)
1	mindestens einmal (Duplikate möglich)
2	genau einmal (am sichersten, am langsamsten)

Deshalb: jede Nachricht bekommt `message_id = str(uuid.uuid4())`, Subscriber merkt sich IDs im Set.

Callbacks:

Funktionen, die die Bibliothek automatisch aufruft, wenn etwas passiert.

- `on_connect` → verbunden: hier `subscribe()`
- `on_message` → Nachricht angekommen

`loop_start()` = Netzwerk läuft im Hintergrund (Publisher)

`loop_forever()` = blockiert, wartet für immer (Subscriber)

mqtt_publisher.py

```python
import json, uuid
import paho.mqtt.client as mqtt

client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2,
                     client_id="publisher-gruppe1")
client.connect("test.mosquitto.org", 1883, keepalive=60)
client.loop_start()
try:
    data = {"message_id": str(uuid.uuid4()),
            "sensor_id": "temp-101",
            "temperature": 22.0, "unit": "C"}
    result = client.publish("gbsha/G305/sensor",
                            json.dumps(data), qos=1)
    result.wait_for_publish()
finally:
    client.loop_stop()
    client.disconnect()
```

mqtt_subscriber.py

```python
def on_connect(client, userdata, flags,
               reason_code, properties):
    if reason_code == 0:
        client.subscribe("gbsha/G305/#", qos=1)

def on_message(client, userdata, message):
    try:
        text = message.payload.decode("utf-8")
        data = json.loads(text)
        print(message.topic, data.get("temperature"))
    except json.JSONDecodeError:
        print("Kein JSON")

client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2,
                     client_id="subscriber-gruppe1")
client.on_connect = on_connect
client.on_message = on_message
client.connect("test.mosquitto.org", 1883, 60)
client.loop_forever()
```

WebSocket & asyncio 09.10.

WebSocket ist ein Protokoll für eine dauerhafte, bidirektionale Verbindung zwischen Client und Server. Beide Seiten können jederzeit senden.

Ablauf:

1. Client verbindet sich: `websockets.connect("ws://localhost:8765")`
2. Verbindung bleibt offen → Nachrichten in beide Richtungen
3. Ende: eine Seite schließt. Normal geschlossen → `async for` endet einfach, bei Abbruch kommt `ConnectionClosedError`

Adresse: `ws://localhost:8765` (verschlüsselt `wss://`)

Wofür? Chat, Live-Ticker, Spiele, Dashboards: überall, wo der Server von sich aus Neuigkeiten schicken soll.

websocket_server.py

```python
import asyncio
import websockets

async def handle_client(websocket):
    try:
        async for message in websocket:
            await websocket.send(f"Echo: {message}")
    except websockets.ConnectionClosed:
        print("Client weg")

async def main():
    async with websockets.serve(handle_client,
                                "localhost", 8765):
        await asyncio.Future()    # läuft für immer

asyncio.run(main())
```

websocket_client.py

```python
async def main():
    async with websockets.connect("ws://localhost:8765") as
ws:
        await ws.send("Hallo")
        antwort = await ws.recv()
        print(antwort)          # Echo: Hallo
```

asyncio:

Asynchrone Programmierung: während auf etwas gewartet wird (Netzwerk), kann anderer Code laufen.

- `async def` → Koroutine (asynchrone Funktion)
- `await` → warten, ohne alles zu blockieren
- `asyncio.run(main())` → startet die Event-Loop
- `async with` / `async for`

Vergleich

	REST (HTTP)	MQTT	WebSocket
Prinzip	Request / Response	Publish / Subscribe	bidirektional
Verbindung	pro Anfrage	dauerhaft zum Broker	dauerhaft Client ↔ Server
Verbindung aufbauen	immer der Client	jeder verbindet sich als Client zum Broker	immer der Client (Handshake)
Wer sendet?	Client fragt, Server antwortet	jeder darf publishen	danach beide jederzeit
Vermittler	-	Broker	-
typisch	CRUD, Web-APIs	IoT, Sensoren, viele Empfänger	Chat, Live-Daten
Python	`requests / flask`	`paho-mqtt`	`websockets`

Welche Technik passt?

- Kundendaten anlegen, ändern, löschen → **REST**
- 200 Temperatursensoren senden alle 5 Sek. an mehrere Auswertungen → **MQTT**
- Chat im Browser, neue Nachrichten sofort anzeigen → **WebSocket**

Zum Schluss

KA-Check

Typische Aufgaben

1. Schwachstellen in Code finden und verbessern (wie AB01)
2. „Was gibt das Programm aus?" (Code lesen)
3. Lückencode: `with open` , `json.dump` , `DictWriter` , `raise_for_status` , `jsonify`
4. passende `except` -Blöcke ergänzen, richtige Reihenfolge
5. CRUD-Tabelle ausfüllen (Methode, Endpunkt, Statuscode)
6. Situation → Statuscode (z.B. `/users/abc` → 400)
7. einen Flask-Endpunkt schreiben (GET mit 400/404 oder POST mit 201)
8. 3 Tests mit `assert` schreiben (Normal-, Grenz-, Fehlerfall)
9. REST vs. MQTT vs. WebSocket erklären, Szenario zuordnen
10. Begriffe: Broker, Topic, Wildcard, QoS, Callback, CRUD, Serialisierung

Merksätze

- `input()` und CSV liefern immer Strings
- `load` / `dump` = Datei, `loads` / `dumps` = String
- CSV schreiben: `newline=""` · Dateien: `encoding="utf-8"` · immer `with`
- API-Aufruf: `timeout=` + `raise_for_status()`
- spezielle Exceptions vor allgemeinen
- POST → 201 · DELETE → 204 · nicht gefunden → 404 · kaputte Eingabe → 400 · Server kaputt → 500
- `+` = eine Ebene, `#` = alle darunter (nur am Ende)
- QoS 1 → Duplikate möglich → `message_id` + Set

Nach den Arbeitsblättern AB01–AB05, und den Dateien der MQTT/WebSocket-Übung vom 09.10. Python-Grundlagen und OOP: separater Lernzettel „Grundlagen".