



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 01

MODERNE PYTHON- ENTWICKLUNG

Wo wir stehen

Herzlichen Glückwunsch! In Teil 1 haben Sie erreicht:

- ✓ Git & GitHub mit SSH-Authentifizierung eingerichtet
- ✓ VS Code mit wichtigen Erweiterungen konfiguriert
- ✓ GitHub Copilot für KI-Unterstützung aktiviert
- ✓ Erstes Repository erstellt und gepusht

Heute: Vom Skript-Schreiber zum professionellen Python-Entwickler!

Heutige Lernziele

Am Ende dieser Vorlesung werden Sie:

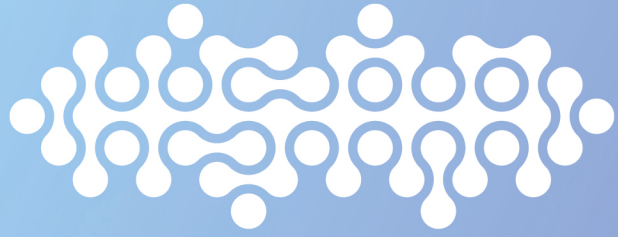
1.  **Verstehen**, warum Python die moderne Softwareentwicklung dominiert
2.  **Unterscheiden** zwischen Skript-Python und Produktions-Python
3.  **Einrichten** Ihres ersten professionellen Python-Projekts
4.  **Beherrschen** von `uv` - dem revolutionären Python-Paketmanager
5.  **Verstehen** von virtuellen Umgebungen und `pyproject.toml`
6.  **Erleben** des modernen Python-Entwicklungsworkflows

Kurze Umfrage: Ihre Python-Erfahrung

Heben Sie die Hand, wenn Sie...

- Python schon einmal verwendet haben (in beliebigem Kontext)?
- Mehr als 100 Zeilen Python geschrieben haben?
- `pip install` verwendet haben, um Pakete hinzuzufügen?
- Eine virtuelle Umgebung erstellt haben?
- Ein Python-Paket veröffentlicht haben?

Heute: Wir verbessern Ihre Python-Fähigkeiten, egal wo Sie starten!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Warum Python?

1

Warum Python? Die Zahlen

GitHub (2024-2025):

- **19,8 Millionen** Repositories verwenden Python
- **Zweitbeliebteste** Sprache (nach JavaScript)

Stack Overflow Developer Survey 2024:

- **51%** der professionellen Entwickler nutzen Python
- **#1 Sprache**, die Entwickler lernen wollen

TIOBE Index (2025):

- **#1 Programmiersprache** weltweit
- **15,2%** Marktanteil

Warum Python gewinnt

Lesbarkeit - Code, der sich wie Englisch liest:

```
def calculate_average(numbers):  
    """Calculate the average of a list of numbers."""  
    return sum(numbers) / len(numbers)  
  
students = [85, 92, 78, 90, 88]  
average = calculate_average(students)  
print(f"Average grade: {average}")
```

Vergleichen Sie das mit Java oder C++ - Python gewinnt bei der Klarheit!

Python: Eine Sprache, jede Domäne

Entwicklung:

- Web: Django, Flask, FastAPI
- Desktop: PyQt, Tkinter
- Spiele: Pygame

Daten & KI:

- Data Science: Pandas, NumPy
- ML: TensorFlow, PyTorch
- KI: LangChain, OpenAI SDK

Karriere-Auswirkung: Lernen Sie Python einmal, arbeiten Sie in jeder Domäne.

Die KI-Revolution ist Python

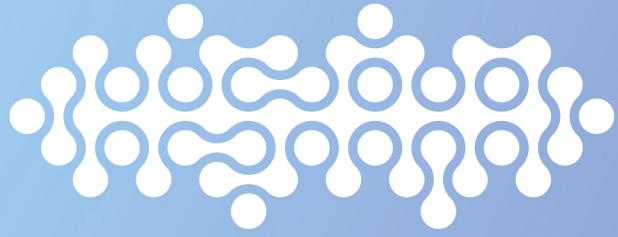
Wir erleben eine KI-Revolution, und Python ist die Sprache der KI:

- **PyTorch & TensorFlow** - Die dominierenden ML-Frameworks
- **Hugging Face** - Der KI-Modell-Hub (hauptsächlich Python)
- **LangChain, LlamaIndex** - LLM-Anwendungsframeworks
- **OpenAI API, Anthropic API** - Python SDKs sind erstklassig

Wenn Sie mit modernster KI arbeiten wollen, ist Python nicht optional.

Das Fazit: Warum Python

1.  **Allgegenwärtig** - Überall verwendet, von Startups bis NASA
2.  **Produktiv** - Weniger Code schreiben, mehr erreichen
3.  **Vielseitig** - Eine Sprache für Web, Daten, KI, Automatisierung
4.  **Gefragt** - Top-Gehälter, meiste Stellenangebote
5.  **Zukunftssicher** - Führende KI/ML-Sprache
6.  **Einsteigerfreundlich** - Konzepte lernen, nicht Syntax-Eigenheiten
7.  **Professionell** - Skaliert von Skripten zu Enterprise



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Skript vs. Produktions-Python

2

Die zwei Pythons

Skript-Python (Was Tutorials lehren):

```
# analysis.py
import pandas as pd
data = pd.read_csv("data.csv")
result = data.describe()
print(result)
```

```
python analysis.py # Funktioniert... vorerst
```

Frage: Was passiert, wenn Ihr Kollege das ausführen will?

Produktions-Python: Der professionelle Weg

```
# src/analytics/pipeline.py
import pandas as pd
from pathlib import Path

def analyze_data(file_path: Path) -> pd.DataFrame:
    """Analyze data from CSV file and return statistics."""
    data = pd.read_csv(file_path)
    return data.describe()
```

```
# Moderner uv-Workflow - ordentliche Paketierung
uv run analytics-cli analyze data.csv
```

Gleiche Funktionalität, professionelle Struktur!

Was ändert sich bei Skalierung?

Probleme:

- "Es läuft auf meinem Rechner"
- Wie Code verteilen?
- Bugs in Produktion kosten Geld
- 10 Entwickler, gleiche Codebasis

Lösungen:

- Reproduzierbare Umgebungen
- Pakete bauen & veröffentlichen
- Testing, Linting, Type Checking
- Standardisierte Projektstruktur

Heutiger Fokus: Reproduzierbare Umgebungen mit `uv`

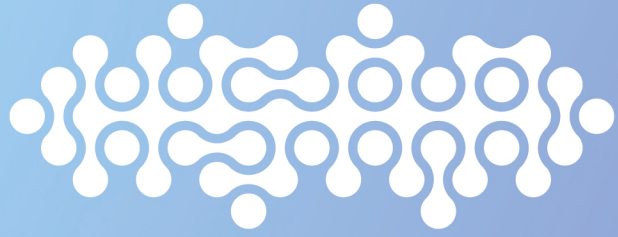
Diskussion: "Es läuft auf meinem Rechner"

Szenario: Sie schreiben ein Python-Skript, das perfekt funktioniert. Sie senden es an einen Kollegen. Es stürzt ab.

Was könnte schiefgehen?

- Andere Python-Version?
- Fehlende Pakete?
- Unterschiedliche Paketversionen?
- Betriebssystem-Unterschiede?

Wie würden Sie das lösen?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Ihr erstes professionelles Projekt

3

Hands-On: Ihr erstes professionelles Projekt

Wir transformieren Skript-Python in Produktions-Python!

Was Sie tun werden:

1. Starter-Code herunterladen (Road Profile Viewer)
2. Lokale Umgebung einrichten
3. Git-Repository initialisieren
4. Mit GitHub verbinden
5. Mit modernem `uv`-Tooling ausführen

Los geht's!

Schritt 1: Code herunterladen

Kursteilnehmer:

1. Zur Canvas-Kursseite gehen
2. `road-profile-viewer.7z` herunterladen
3. Archiv entpacken

Externe Nutzer:

1. GitHub Release-Seite besuchen
2. Quellcode-Archiv herunterladen
3. Entpacken

Was Sie finden: Python-Skripte zur Visualisierung von Straßenhöhenprofilen

Moment - Warum eine ZIP-Datei?

"Haben wir nicht gerade gesagt, Git ist besser als ZIP-Dateien?"

Ja! Und Sie haben recht, das zu bemerken.

Warum wir das tun:

1. Mit einer ZIP-Datei beginnen (der "alte Weg")
2. Git selbst initialisieren
3. Eigenes GitHub-Repository erstellen
4. Verbinden und pushen

Es einmal selbst zu tun lehrt Sie, wie Repositories entstehen.

Think of it as training wheels before the Tour de France!

Schritt 2-3: Einrichten und Anpassen

```
# Projektordner erstellen
mkdir road-profile-viewer
cd road-profile-viewer
# Entpackte Dateien hierher kopieren, dann in VS Code öffnen
code .
```

`pyproject.toml` bearbeiten:

```
# Diese Zeile auf IHRE Infos ändern:
authors = [{name = "IHR_GITHUB_USERNAME", email = "ihre@email.com"}]
```

Datei speichern! Das macht Sie zum offiziellen Autor.

Schritt 4: Git initialisieren

```
# Git im Projektordner initialisieren
git init

# Alle Dateien zum Staging hinzufügen
git add .

# Ersten Commit erstellen
git commit -m "Initial commit: Road profile viewer scripts"

# Status prüfen
git status
```

Checkpoint: Sie haben jetzt ein lokales Git-Repository mit Ihrem ersten Commit!

Schritt 5-6: Mit GitHub verbinden

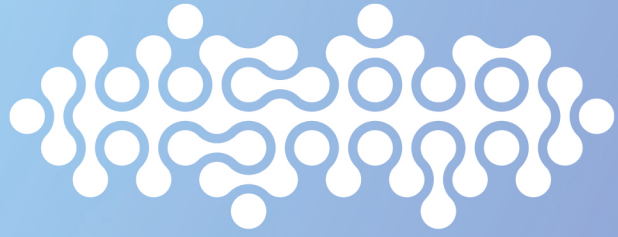
Auf GitHub:

1. "+" klicken → **"New repository"**
2. Name: `road-profile-viewer`
3. **NICHT** mit README oder .gitignore initialisieren

Im Terminal:

```
git remote add origin https://github.com/IHR_USERNAME/road-profile-viewer.git  
git branch -M main  
git push -u origin main
```

GitHub aktualisieren - Ihr Code ist da!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Modernes Python mit uv

4

uv installieren: Der moderne Weg

Windows (PowerShell):

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Ubuntu/Linux:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Überprüfen:

```
uv --version  
# Sollte uv 0.5.x oder höher anzeigen
```

Anwendung starten!

```
cd src/road-profile-viewer  
uv run main.py
```

Das war's! `uv` macht automatisch:

-  Erkennt Python-Version & installiert Abhängigkeiten
-  Erstellt isolierte Umgebung
-  Startet Ihre Anwendung

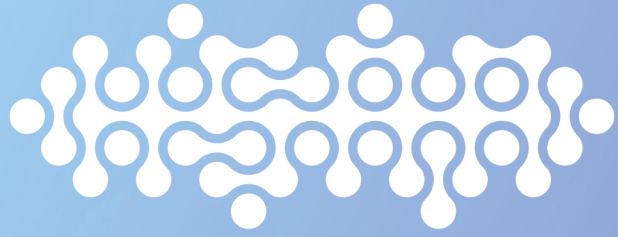
Browser öffnen: `http://127.0.0.1:8050`

Was ist gerade passiert?

Mit einem einzigen Befehl (`uv run main.py`) haben Sie:

- Die richtige Python-Version automatisch installiert
- Alle Projekt-Abhängigkeiten eingerichtet (Dash, Plotly, Pandas, etc.)
- Eine isolierte Umgebung erstellt (keine Konflikte!)
- Einen Webserver mit interaktiven Visualisierungen gestartet

Das ist der moderne Python-Workflow - kein manuelles Setup, keine "Es läuft auf meinem Rechner"-Probleme!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Virtuelle Umgebungen & Abhängigkeiten

5

Das Problem: Abhängigkeits-Chaos

Stellen Sie sich vor, Sie arbeiten an drei Projekten:

- **Projekt A:** Benötigt `pandas 1.5.0`
- **Projekt B:** Benötigt `pandas 2.1.0`
- **Ihr System:** Kann nur EIN globales pandas haben!

```
pip install pandas==1.5.0 # Für Projekt A  
pip install pandas==2.1.0 # Überschreibt! Projekt A kaputt!
```

Das ist die "Dependency Hell" - und deshalb gibt es virtuelle Umgebungen.

Virtuelle Umgebungen: Die Lösung

Denken Sie an Wohnungen in einem Gebäude:

- Jede Wohnung (virtuelle Umgebung) hat eigene Möbel (Pakete)
- Bewohner (Projekte) stören sich nicht gegenseitig
- Das Gebäude (Ihr Computer) beherbergt viele unabhängige Wohnungen
- Jede kann verschiedene Versionen derselben Möbel haben

Virtuelle Umgebung = Isolierte Python-Installation für jedes Projekt

Innerhalb einer virtuellen Umgebung

```
.venv/                                # Ordner der virtuellen Umgebung
├── bin/ (oder Scripts/ auf Windows)
│   ├── python                        # Python-Executable
│   ├── pip                          # Paket-Installer
│   └── activate                      # Aktivierungs-Skript
├── lib/
│   └── python3.12/
│       └── site-packages/           # HIER LEBEN IHRE PAKETE
│           ├── pandas/
│           ├── dash/
│           └── plotly/
└── pyvenv.cfg
```

Wichtig: Pakete gehen in DIESE Umgebung, nicht in Ihr globales Python!

Der alte Weg: Alles manuell

```
python -m venv .venv          # 1. Erstellen (langsam!)
.venv\Scripts\activate        # 2. Aktivieren (OS-spezifisch!)
pip install --upgrade pip      # 3. pip upgraden
pip install dash plotly pandas # 4. Deps installieren
python main.py                 # 5. Endlich ausführen!
```

5+ Schritte, OS-spezifische Befehle, fehleranfällig, langsam...

Der neue Weg: uv macht alles

```
uv run main.py # Das war's. Wirklich.
```

Was uv automatisch macht:

```
[1] pyproject.toml lesen  
  ↓  
[2] .venv/ erstellen (in Millisekunden!)  
  ↓  
[3] Abhängigkeiten auflösen & sperren  
  ↓  
[4] Nur Fehlendes installieren (gecacht!)  
  ↓  
[5] Ihr Skript ausführen
```

Ein Befehl ersetzt sechs manuelle Schritte!

Warum uv revolutionär ist

Geschwindigkeitsvergleich (venv + 10 Pakete installieren):

Tool	Zeit
Traditionell (<code>venv</code> + <code>pip</code>)	~30 Sekunden
<code>uv</code>	~1 Sekunde

Warum? uv ist in Rust geschrieben (kompiliert, parallel, optimiert)

Traditionelle Tools: Python-basiert, langsam, keine Parallelisierung

Was macht uv so schnell?

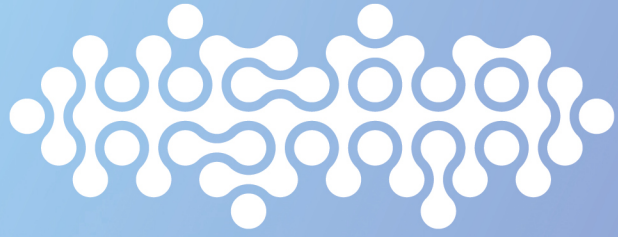
Traditionell (pip, poetry):

- In Python geschrieben
- Neuer Prozess für alles
- Keine parallelen Operationen
- Ineffiziente Datei-I/O

uv (Rust):

- Zu nativem Code kompiliert
- Parallele Downloads
- Optimierte Dateioperationen
- Globaler Paket-Cache

Ergebnis: 10-100x schneller als traditionelle Tools!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

pyproject.toml verstehen

6

pyproject.toml verstehen

Die einzige Quelle der Wahrheit für Ihr Python-Projekt:

```
[project]
name = "road-profile-viewer"
version = "0.1.0"
requires-python = ">=3.10"

dependencies = [
    "dash>=2.14.0",
    "plotly>=5.18.0",
    "pandas>=2.1.0",
]

[project.optional-dependencies]
dev = ["pytest>=7.4.0", "ruff>=0.1.0"]
```

Alles in einer Datei: Metadaten + Abhängigkeiten + Build-Config

Versions-Spezifizierer erklärt

```
dependencies = [  
    "dash>=2.14.0",      # Jede Version 2.14.0 oder höher  
    "plotly~=5.18.0",    # Nur 5.18.x (Patch-Updates OK)  
    "pandas>=2.1,<3.0",  # 2.1+ aber nicht 3.x  
    "numpy==1.24.0",     # Exakte Version (meist zu streng!)  
]
```

Best Practice: Verwenden Sie `>=` für die meisten Abhängigkeiten.

Lassen Sie `uv` kompatible Versionen automatisch auflösen!

Drei Wege, Ihre App auszuführen

Methode	Befehl	Beste für
uv run	<code>uv run main.py</code>	Schnelle Entwicklung
Manuell venv	<code>python main.py</code>	Debugging, IDE-Setup
Benanntes Skript	<code>road-viewer</code>	Produktion, Distribution

Für tägliche Entwicklung: Nutzen Sie `uv run` - am schnellsten und einfachsten!

Hands-On: Benanntes Skript hinzufügen

Zu Ihrer `pyproject.toml` hinzufügen:

```
[project.scripts]
road-viewer = "main:main"
```

`main.py` mit main-Funktion aktualisieren:

```
def main():
    """Main entry point for Road Profile Viewer."""
    app = create_app()
    app.run_server(debug=True)

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

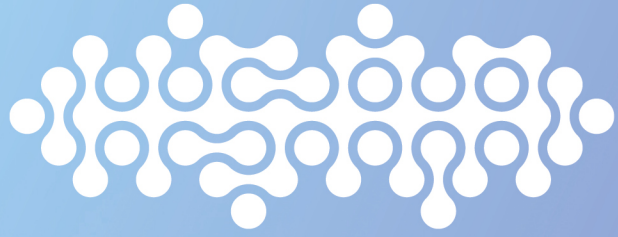
Benanntes Skript ausführen

```
# Projekt synchronisieren (registriert das Skript)
uv sync

# Jetzt mit dem sauberen Befehl ausführen!
uv run road-viewer
```

Sie haben einen professionellen CLI-Befehl erstellt!

Praxisbeispiele: `pytest`, `black`, `ruff`, `uvicorn` - alle nutzen dieses Muster!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

uv-Befehlsreferenz

7

Kurzreferenz: Wichtige uv-Befehle

```
# Code ausführen
uv run script.py          # Python-Skript ausführen
uv run command            # Benanntes Skript ausführen

# Paketverwaltung (moderner Weg)
uv add package            # Paket zum Projekt hinzufügen
uv add package --dev      # Als Dev-Abhängigkeit hinzufügen
uv sync                  # Umgebung mit pyproject.toml synchronisieren

# Umgebung
uv venv                  # Virtuelle Umgebung erstellen
uv venv --python 3.12    # Mit spezifischer Python-Version
```

Weitere uv-Befehle

```
# Projekt-Initialisierung
uv init                # Neues Python-Projekt erstellen
uv init --app          # Anwendungsprojekt erstellen

# Python-Versionsverwaltung
uv python install 3.12  # Python 3.12 installieren
uv python list         # Verfügbare Versionen auflisten

# Globale Tools
uv tool install ruff    # Tool global installieren
uv tool list           # Globale Tools auflisten
```

Wichtig: `uv add` aktualisiert `pyproject.toml` automatisch!

Das Gesamtbild

Sie tippen: `uv run main.py`

↓

[uv liest `pyproject.toml`]

↓

[Prüft/erstellt `.venv/`]

↓

[Löst Abhängigkeiten auf - alle kompatibel?]

↓

[Installiert aus Cache oder lädt herunter]

↓

[Führt Ihr Skript aus]

↓

Ihre App läuft!

All diese Komplexität ist hinter einem Befehl verborgen!

Kernpunkte: Virtuelle Umgebungen

1. Virtuelle Umgebungen:

- Isolierte Räume für Projekt-Abhängigkeiten
- Verhindern Konflikte zwischen Projekten
- Befinden sich im `.venv/`-Ordner

2. `pyproject.toml`:

- Einzige Quelle der Wahrheit für Ihr Projekt
- Ersetzt `requirements.txt` + `setup.py`
- Moderner Python-Standard

Kernpunkte: Warum uv wichtig ist

3. Warum uv:

- **Geschwindigkeit:** 10-100x schneller als traditionelle Tools
- **Einfachheit:** Ein Befehl ersetzt viele
- **Zuverlässigkeit:** Korrekte Abhängigkeitsauflösung
- **Modern:** Gebaut für 2025er Workflows

4. Professioneller Workflow:

```
uv run main.py # vs 6 manuelle Schritte mit pip/venv
```

Die Zukunft der Python-Entwicklung ist hier!

Testen Sie Ihr Verständnis

Bereit, Ihr Wissen zu prüfen?

- **20 Fragen** zu allen Themen
- **Sofortiges Feedback** auf Ihre Antworten
- **Nicht benotet** - rein zur Selbsteinschätzung
- Ähnliche Fragen **werden in der Prüfung erscheinen**

Zum Kapitel 01 (Moderne Python-Entwicklung) Quiz:

<https://forms.gle/WwnX3ytHAd53sZY39>

Zusammenfassung: Was Sie heute gelernt haben

1. **Python dominiert** - #1 Sprache, essentiell für KI/ML
2. **Skript vs Produktion** - Professioneller Code braucht Struktur
3. **Virtuelle Umgebungen** - Isolierte Abhängigkeiten pro Projekt
4. **pyproject.toml** - Einzige Quelle der Wahrheit
5. **uv-Revolution** - 10-100x schneller, ein Befehl macht alles
6. **Moderner Workflow** - `uv run` ersetzt 6 manuelle Schritte

Was kommt als Nächstes?

Kapitel 02: Code-Qualität in der Praxis

- PEP 8-Konformität mit Ruff
- Linting und Formatierung
- Type Hints und Type Checking
- Pre-Commit Hooks

Ihre Hausaufgabe: Road Profile Viewer Setup abschließen und Quiz machen!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

VIELEN DANK