PS Angewandte Systemwissenschaften 1

Einheit 4: Python-Grundlagen & erstes Modell

Vom ODD-Protokoll zum funktionierenden Code

David Maier
Institut für Umweltsystemwissenschaften
Universität Graz

Kurzer Rückblick: Einheiten 1–3

Systeme & Komplexität

Systemdefinition, Emergenz, Feedback, Weaver's Problemklassen, Cilliers' Merkmale.

Modelle & ABM

Modellierungszyklus, vier Ansätze, ABM-Definition, Agenten als autonome Einheiten.

ODD & Staumodell

7 ODD-Elemente, Butterfly-Modell, Sugiyama-Staumodell als ODD beschrieben.

Wir haben beschrieben, **was** wir modellieren wollen (ODD). Heute übersetzen wir es in **Code**.

Teil 1: Python-Refresher

Tooling, Sprache, moderner Stil

VS Code & Python ausführen

VS Code einrichten

- Python-Extension installieren
- Terminal öffnen: `Ctrl+`
- Neue Datei: `hello.py`
- Rechtsklick → *Run Python File*

Python im Terminal

```
python hello.py          # system Python
uv run hello.py          # with uv (recommended)
```

Im Zweifelsfall prüfen: welches Python?

```
which python
```

Virtuelle Umgebungen & `uv`

Was ist eine virtuelle Umgebung?

Ein **isolierter Ordner** mit eigenem Python und eigenen Paketen. Jedes Projekt hat seine eigene Umgebung, so gibt es keine Konflikte zwischen Projekten.

`uv` ist ein moderner, schneller Ersatz für `pip`, `venv`, `conda` und ähnliche Tools. Ein einziges Werkzeug für alles.

Wichtige Befehle

```
# Create new project
uv init my-project
cd my-project

# Install packages
uv add numpy matplotlib

# Run script
uv run script.py
```

`uv` erstellt automatisch eine `.venv` und eine `uv.lock`-Datei. Ihr müsst euch nicht selbst um virtuelle Umgebungen kümmern.

Die `pyproject.toml`

Was steht da drin?

```toml
[project]
name = "traffic-jam"
version = "0.1.0"
requires-python = "⩾3.12"
dependencies = [
    "matplotlib⩾3.10",
]
```

`uv init` erstellt diese Datei, `uv add` trägt Pakete ein.

Warum ist das wichtig?

- **Python-Version:** `requires-python` legt fest, welche Version installierte Packages unterstützen müssen
- **Abhängigkeiten:** alle Pakete sind dokumentiert und reproduzierbar installierbar
- **Projektinfo:** Name und Version eures Projekts

`uv` **installiert und verwaltet Python selbst**. Wenn `requires-python = "⩾3.12"` gesetzt ist und ihr kein passendes Python habt, lädt `uv` es automatisch herunter.

Von Jupyter zu `.py`-Dateien

Jupyter Notebook

- Code in einzelnen **Zellen**
- Zellen in beliebiger Reihenfolge ausführbar
- Versteckter Zustand: Variablen leben im Hintergrund
- Schwer reproduzierbar: Ergebnis hängt von der Reihenfolge der Ausführung ab

Python-Script (`.py`)

- Code in **einer Datei**, von oben nach unten
- Jeder Lauf startet bei null: alle Variablen werden neu initialisiert
- Reproduzierbar: `uv run script.py` liefert immer dasselbe Ergebnis
- Besseres Tooling: Autocomplete, Linting, AI-Coding-Assistenten funktionieren zuverlässiger

Jupyter ist nur eine **Schicht über Python**. Darunter läuft derselbe Interpreter. Wir arbeiten direkt mit dem, was Jupyter im Hintergrund tut.

Wie funktioniert Python?

Interpretierte Sprache

```
x = 10          # ✓ runs
y = x + 5       # ✓ runs
print(y)        # ✓ output: 15
print(z)        # × NameError! Only HERE.
print("done")   # never reached
```

Zeilen 1 bis 3 laufen erfolgreich. Der Fehler in Zeile 4 fällt erst auf, wenn Python dort ankommt.

Konsequenzen

- **Dynamische Typisierung:** Typen werden erst zur Laufzeit geprüft, nicht vorher
- **Fehler erst beim Erreichen:** Ein Bug in Zeile 50 fällt erst auf, wenn Zeile 50 ausgeführt wird
- **Kein Kompilieren nötig:** Datei speichern, `uv run script.py`, fertig

Einrückung ist Logik: Python verwendet Einrückung statt `{}`, um Codeblöcke zu definieren. Falsche Einrückung führt zu Fehlern.

Variablen & Typen

Grundtypen

Typ	Beispiel
`int`	`x = 42`
`float`	`pi = 3.14`
`str`	`name = "Hallo"`
`bool`	`aktiv = True`
`list`	`zahlen = [1, 2, 3]`

In der Praxis

```python
name = "Sugiyama"
n_vehicles = 22
road_length = 230.0
is_jammed = False

# f-strings for text formatting
print(f"{n_vehicles} Fahrzeuge auf {road_length} m")
```

Python erkennt den Typ automatisch aus dem zugewiesenen Wert. Keine Typdeklaration nötig, anders als in Java oder C.

Gute Variablennamen

Schlecht ✗

```
a = 230.0
b = 22
x = a / b
l = []
for i in range(b):
    l.append(i * x)
```

Gut ✓

```
road_length = 230.0
n_vehicles = 22
spacing = road_length / n_vehicles
positions = []
for i in range(n_vehicles):
    positions.append(i * spacing)
```

Alles auf Englisch: Variablen, Funktionen, Klassen, Kommentare, Docstrings. Code wird international gelesen, geteilt und von englischsprachigen Tools verarbeitet.

Kontrollstrukturen

Verzweigungen

```python
speed = 25.0
if speed > 30.0:
    print("Too fast!")
elif speed < 5.0:
    print("Almost stopped")
else:
    print("Normal flow")

# Ternary (compact if/else)
label = "fast" if speed > 30.0 else "ok"
```

Schleifen

```python
# for loop
for i in range(5):
    print(f"Step {i}")

# while loop
t = 0.0
while t < 10.0:
    t += 0.1
```

List Comprehensions

Klassisch mit Schleife

```
quadrate = []
for x in range(10):
    quadrate.append(x ** 2)
```

Als Comprehension

```
quadrate = [x ** 2 for x in range(10)]

# With condition:
gerade = [x for x in range(20)
          if x % 2 == 0]
```

> Kompakter, lesbarer, und oft schneller. Wir verwenden Comprehensions häufig in unseren Modellen.

Funktionen

```python
def optimal_velocity(headway: float, v_max=9.25, d_min=4.0, d_max=25.0) -> float:
    """Calculate desired velocity based on headway distance."""
    if headway ⩽ d_min:
        return 0.0
    elif headway ⩾ d_max:
        return v_max
    else:
        return v_max * (headway - d_min) / (d_max - d_min)
```

Type Hints bei Parametern

`headway: float` dokumentiert den erwarteten Typ, wenn kein Defaultwert vorhanden ist.

Default-Parameter

Standardwerte wie `v_max=9.25` machen Aufrufe flexibel. Der Typ ist hier offensichtlich.

Docstrings (in `"""`) beschreiben, was die Funktion tut. Das ist Pflicht in unseren Projekten.

Klassen

Definition

```python
class Dog:
    """A dog with a name and energy level."""

    def __init__(self, name: str, energy=100):
        self.name = name
        self.energy = energy

    def bark(self):
        """Bark and lose some energy."""
        self.energy -= 10
        print(f"{self.name}: Woof!")
```

Verwendung

```python
rex = Dog("Rex")
rex.bark()          # Rex: Woof!
print(rex.energy)   # 90

luna = Dog("Luna", energy=50)
luna.bark()         # Luna: Woof!
print(luna.energy) # 40
```

Jedes Objekt hat seinen eigenen Zustand. `rex` und `luna` sind unabhängig.

Klasse = Bauplan | **Objekt** = konkretes Exemplar | **Methode** = eine Funktion, die zu einer Klasse gehört

Klassen: `@dataclass`

Klassisch

```
class Vehicle:
    def __init__(self,
                 position: float,
                 velocity: float) → None:
        self.position = position
        self.velocity = velocity
```

Mit @dataclass

```
from dataclasses import dataclass

@dataclass
class Vehicle:
    position: float
    velocity: float
```

`@dataclass` generiert `__init__`, `__repr__` und `__eq__` automatisch. Weniger Boilerplate, gleiche Funktionalität.

Vererbung (kurz)

```python
from dataclasses import dataclass

@dataclass
class Agent:
    """Base class for all agents."""
    position: float

@dataclass
class Vehicle(Agent):
    """Vehicle agent with velocity."""
    velocity: float = 0.0

car = Vehicle(position=10.5, velocity=8.0)
print(car.position)   # 10.5 (inherited from Agent)
print(car.velocity)   # 8.0
```

Vererbung braucht man selten in einfachen ABMs. Gut zu kennen, aber nicht übernutzen.

Module & Imports

Import-Arten

```python
# Standard library
import math
from pathlib import Path
from dataclasses import dataclass

# External packages (uv add ...)
import numpy as np
import matplotlib.pyplot as plt
```

Eigene Module

```
my-project/
├── main.py           # entry point
├── model.py          # model classes
└── pyproject.toml    # project file (uv)
```

```python
# in main.py:
from model import Vehicle, TrafficModel
```

Immer `uv add` statt `pip install` verwenden, damit Abhängigkeiten in `pyproject.toml` landen.

Moderner Python-Stil

Verwenden ✓

- Type Hints wo der Typ nicht offensichtlich ist
- f-Strings statt `.format()`
- `@dataclass` für Datenklassen
- `pathlib.Path` statt `os.path`
- List Comprehensions
- Docstrings für Funktionen, Methoden und Klassen

Vermeiden ✗

- Globale Variablen
- `from module import *`
- Verschachtelte Klassen ohne Grund
- Magic Numbers ohne Namen
- `print()`-Debugging statt Breakpoints
- Redundante Type Hints: `x: int = 5`

> Ziel: Code, den man in 6 Monaten noch versteht. Docstrings + klare Namen + sparsame Type Hints = selbstdokumentierender Code.

Teil 2: Vom ODD zum Code

Das Sugiyama-Staumodell in Python

Wir nehmen das ODD aus Einheit 3 und übersetzen es Schritt für Schritt in funktionierenden Python-Code.

Pair Programming

Wir bauen das Modell gemeinsam

Öffnet VS Code und erstellt ein neues Projekt mit `uv init traffic-jam`. Wir gehen den Code Schritt für Schritt durch.

Ergebnisse diskutieren

Was sehen wir?

- Bei niedrigem Noise: stabiler Fluss
- Bei ausreichendem Noise + Dichte: **Staubildung**
- Rückwärts wandernde Welle (~20 km/h)

Vergleich mit dem Experiment

- Sugiyama: Stau entsteht nach ~1 Minute
- Unser Modell: ähnliches Verhalten
- Gleiche qualitative Dynamik

Emergenz: Kein Fahrzeug ist programmiert anzuhalten. Der Stau entsteht aus der *Interaktion*, genau wie im echten Straßenverkehr.

Experimentieren

Parameter variieren

Parameter	Werte zum Testen
`n_vehicles`	10, 15, 22, 30
`sensitivity`	1.0, 2.0, 3.0
`noise`	0.0, 0.1, 0.5

Fragen

- Ab welcher Dichte entsteht ein Stau?
- Was passiert *ohne* Noise (`0.0`)?
- Was passiert bei sehr hoher Sensitivity?
- Wie ändert sich die Staugeschwindigkeit?

Das ist der Kern von ABM: **einfache Regeln, komplexe Ergebnisse**. Parameter-Variation zeigt, wann Phasenübergänge auftreten.

Teil 4: Git & GitHub

Versionskontrolle für reproduzierbare Forschung

Warum Versionskontrolle?

Ohne Git

```
modell_final.py
modell_final_v2.py
modell_final_v2_wirklich_final.py
modell_backup_alt.py
modell_NICHT_LOESCHEN.py
```

Wer kennt das nicht?

Mit Git

- Jede Änderung = **Commit** mit Nachricht & Zeitstempel
- Jederzeit zu einem früheren Zustand zurückkehren
- Mehrere Personen arbeiten gleichzeitig
- Ideal für wissenschaftliche Reproduzierbarkeit

Git-Konzepte

Repository

Ein Ordner, dessen Änderungsverlauf Git verfolgt. Existiert **lokal** auf eurem Rechner *und* **remote** auf GitHub.

Commit

Ein **Snapshot** aller Dateien zu einem Zeitpunkt. Jeder Commit hat eine Nachricht: *Was* wurde geändert und *warum*?

Branch

Ein paralleler Entwicklungsstrang. `main` ist der Hauptbranch. Für den Kurs reicht `main`.

Git vs. GitHub

Git

- Ein **Programm** auf eurem Rechner
- Verwaltet die Versionsgeschichte lokal
- Funktioniert komplett offline
- Kommandozeile: `git add`, `git commit`, …

GitHub

- Eine **Cloud-Plattform** für Git-Repositories
- Speichert eine Kopie eures Repos online
- Ermöglicht Zusammenarbeit im Team
- `git push` lädt eure Commits auf GitHub hoch

> Git ist das Werkzeug, GitHub ist der Speicherort. Ihr arbeitet lokal mit Git und synchronisiert mit GitHub.

Git-Workflow

Lokal arbeiten

```
# Create new repo
git init

# Stage changes
git add model.py

# Save snapshot
git commit -m "Initial traffic model"

# Upload to GitHub
git push
```

Von GitHub starten

```
# Clone repo from GitHub
git clone https://github.com/user/repo.git
cd repo

# ... work ...

# Upload changes
git add .
git commit -m "Add OVM function"
git push
```

Workflow: Ändern → `git add` → `git commit` → `git push`. Das ist 90% eurer Git-Nutzung im Kurs.

GitHub Copilot

Was ist das?

- AI-Coding-Assistent direkt in VS Code
- **Kostenlos** für Studierende (GitHub Education)
- Schlägt Code-Vervollständigungen vor
- Kann ganze Funktionen generieren

Richtig nutzen

- Vorschläge **prüfen**, nicht blind übernehmen
- Copilot macht Fehler, ihr müsst den Code verstehen
- Gut für Boilerplate und Standardmuster
- Schlecht für domänenspezifische Logik

Copilot ist ein **Werkzeug**, kein Ersatz für Verständnis. Ihr seid verantwortlich für euren Code, auch wenn Copilot ihn vorgeschlagen hat.

Zusammenfassung & Ausblick

Was wir heute gelernt haben

Heute

- Python-Refresher: Typen, Funktionen, Klassen, `@dataclass`
- Tooling: VS Code, `uv`, virtuelle Umgebungen
- Git & GitHub Basics
- **ODD → Code:** Entity → Klasse, State Variable → Attribut, Process → Methode
- Erstes funktionierendes ABM: Staumodell

Nächstes Mal

Einheit 5: Wir vertiefen die Modellierung und lernen, wie man ABMs systematisch analysiert.

Bis dahin:

- Staumodell lokal zum Laufen bringen
- Mit Parametern experimentieren
- Code auf GitHub pushen