Python 3 - Fortgeschrittene Konzepte und Anwendungen

Angewandte OOP und Design Patterns

Hussam Alafandi

February 4, 2025

Einleitung

Ziele des Tages

- Praktische Anwendung von OOP in Python (Wiederholung & Vertiefung).
- Verständnis von Zusammensetzung (Composition) vs. Vererbung.
- Leichte Einführung in Design Patterns (Factory, Singleton).
- Implementierung eines kleinen Projekts.
- Verbesserte Debugging-Fähigkeiten.

Recap von Tag 2

Kurze Wiederholung

- OOP-Grundpfeiler: Klassen, Objekte, Vererbung, Kapselung, Polymorphie.
- Statische Methoden vs. Instanzmethoden.
- Abstrakte Klassen und deren Nutzen.

Kurze Wiederholung

- OOP-Grundpfeiler: Klassen, Objekte, Vererbung, Kapselung, Polymorphie.
- Statische Methoden vs. Instanzmethoden.
- Abstrakte Klassen und deren Nutzen.

Diskussion
Was war für euch am schwierigsten?

Zusammensetzung (Composition)

Warum Composition?

- Alternative oder Ergänzung zu Vererbung.
- Ein Objekt kann **mehrere** andere Objekte enthalten.
- Bessere Trennung der Verantwortlichkeiten in komplexen Systemen.

Beispiel: Auto und Motor

Code-Beispiel:

```python
class Motor:
    def __init__(self, leistung):
        self.leistung = leistung
class Auto:
    def __init__(self, marke, modell, motor):
        self.marke = marke
        self.modell = modell
        self.motor = motor  # Objekt in Objekt

    def anzeigen(self):
        print(f"{self.marke} {self.modell} mit {self.motor.leistung} PS")

motor1 = Motor(150)
auto1 = Auto("BMW", "X3", motor1)
auto1.anzeigen()
```

Übung: Bibliothek und Bücher

Aufgabe:

- Erstellt eine Klasse `Buch` mit `titel` und `autor`.
- Erstellt eine Klasse `Bibliothek`, die eine Liste von `Buch`-Objekten enthält.
- `Bibliothek` soll `buecher_hinzufuegen(buch)` und `anzeigen()` bereitstellen.

Einfache Design Patterns

Was sind Design Patterns?

- „Schablonen“ für wiederkehrende Probleme in der Softwareentwicklung.
- Hier nur sehr kurze Einführung: **Factory Method** und **Singleton**.

Factory Method

Idee: Eine spezielle „Fabrik“-Klasse erzeugt Objekte, ohne dass im Hauptcode konkrete Klassen instanziert werden müssen.

Beispiel:

```
class Fahrzeug:
    def __init__(self, typ):
        self.typ = typ
    def fahren(self):
        print(f"Das {self.typ} fährt los!")
class FahrzeugFabrik:
    @staticmethod
    def erstellen(typ):
        return Fahrzeug(typ)

auto = FahrzeugFabrik.erstellen("Auto")
auto.fahren()
```

Singleton

Idee: Nur eine Instanz einer Klasse im gesamten Programm.

Praxisbeispiel:

- Datenbankverbindung.
- Log-Instanz.

Mini-Projekt

Mini-Projekt: Online-Shop

Aufgabe:

- Klassen: `Produkt`, `Kunde`, `Bestellung`.
- `Bestellung` enthält `Produkte` und einen `Kunde`.
- Methoden: `produkt_hinzufuegen()`, `anzeigen()` usw.

Beispiel-Code für den Start

```
class Produkt:
    def __init__(self, name, preis):
        self.name = name
        self.preis = preis

class Kunde:
    def __init__(self, name, email):
        self.name = name
        self.email = email
```

Fortsetzung: Bestellung

```python
class Bestellung:
    def __init__(self, kunde):
        self.kunde = kunde
        self.produkte = []

    def produkt_hinzufuegen(self, produkt):
        self.produkte.append(produkt)

    def anzeigen(self):
        print(f"Bestellung für {self.kunde.name}:")
        for p in self.produkte:
            print(f"- {p.name} ({p.preis} €)")
```

Debugging & Code-Verbesserung

Debugging

Methoden:

- `print()` zur schnellen Fehlersuche.
- Breakpoints in VS Code oder PyCharm.
- `pdb`-Modul in Python.

Reflexion & Q&A

- Zusammensetzung vs. Vererbung: Wann nutze ich was?
- Wie kann ein einfaches Design Pattern helfen, den Code zu strukturieren?
- Fragen oder Schwierigkeiten?

Hausaufgabe (optional)
Erweitere das Online-Shop-Projekt:
Füge eine `gesamtsumme()`-Methode in `Bestellung` ein, die die Summe aller Produktpreise berechnet.