Tag 2: Fortgeschrittene OOP-Konzepte in Python

Ziel: Vertiefung der objektorientierten Programmierung (OOP) durch **Encapsulation, abstrakte Klassen, statische Methoden und realistische Modellierung**.

Tagesablauf

Zeit	Thema	Beschreibung
9:00 - 9:20	**Interaktive Wiederholung**	Kurzes OOP-Quiz, Fehleranalyse in Code, Diskussion.
9:20 - 9:50	**Encapsulation & `@property` Dekorator**	Theorie & Anwendung von privaten Attributen.
9:50 - 10:20	**Abstrakte Klassen und `abc` Modul**	Einführung in abstrakte Methoden, praktische Übung.
10:20 - 10:50	**`@staticmethod` vs. `@classmethod`**	Erklärung, wann man sie nutzt, praktische Übung.
10:50 - 11:20	**OOP-Design mit UML-Diagramm**	Planung eines realen Systems mit UML.
11:20 - 11:50	**Code-Implementierung des Systems**	Umsetzung der Klassendefinitionen in Python.
11:50 - 12:20	**Debugging & Unit Testing mit `unittest`**	Einführung ins Testen, praktische Anwendung.
12:20 - 13:00	**Reflexion, Q&A und Ausblick auf Tag 3**	Zusammenfassung, Fragen, Vorschau auf NumPy & Pandas.

Detaillierter Plan

9:00 - 9:20: Interaktive Wiederholung

Ziel: Aktivieren des Vorwissens der Studierenden und Fehleranalyse in OOP-Code.

- **Aktivitäten:**
  - Kurzes **OOP-Quiz** mit Fragen zu:
    - Was sind Klassen und Objekte?
    - Erkläre Vererbung und Polymorphie.
  - Gruppenaufgabe: **Fehlersuche in OOP-Code**
    - Studierende finden und beheben Fehler in folgendem Code:

```
class Person:
def __init__(self name, age):  # Syntaxfehler
```

```
        self.name = name
        self.age = age

    def greet(self)
        print(f"Hallo, mein Name ist {self.name} und ich bin
    {self.age} Jahre alt.")
```

- **Diskussion:** Warum ist OOP wichtig in der Praxis?

9:20 - 9:50: Encapsulation & `@property` Dekorator

Ziel: Kapselung von Daten mithilfe von privaten Attributen und `@property`.

- **Theorie:**
  - Was ist Encapsulation?
  - Verwendung von **privaten Attributen (`__balance`)**.
  - Vorteile von `@property` gegenüber `getters` und `setters`.
- **Beispiel:**

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Privates Attribut

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            raise ValueError("Negativer Betrag nicht erlaubt!")
```

- **Übung:**
  - **Studierende implementieren eine `BankAccount` Klasse**, die auch eine Methode `withdraw()` mit Validierung enthält.

9:50 - 10:20: Abstrakte Klassen mit `abc` Modul

Ziel: Einführung in **abstrakte Methoden** zur Strukturierung großer Codebasen.

- **Theorie:**
  - Warum braucht man abstrakte Klassen?

    - Einführung in das `abc` Modul.

- **Beispiel:**

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)
```

- **Übung:**

    - **Studierende erweitern die `Shape` Klasse** um eine `Circle`-Klasse.

10:20 - 10:50: `@staticmethod` vs. `@classmethod`

Ziel: Erlernen des Unterschieds zwischen Instanzmethoden, `staticmethod` und `classmethod`.

- **Beispiel:**

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

    @classmethod
    def pi(cls):
        return 3.14159
```

- **Übung:**

- **Studierende erstellen eine `TemperatureConverter` Klasse**, die `@staticmethod` verwendet.

10:50 - 11:20: OOP-Design mit UML

Ziel: Planung eines realen Systems mit UML-Klassendiagrammen.

- **Beispiel: Bibliothekssystem oder Online-Shop**
  - Erstellung eines Klassendiagramms mit **Book**, `Member`, `Librarian`.
- **Übung:**
  - **Gruppenarbeit:**
    - Studierende entwickeln ein UML-Diagramm für ihr System.

11:20 - 11:50: Code-Implementierung

Ziel: Umsetzung der UML-Planung in Python.

- **Übung:**
  - Studierende schreiben eine **erste Version ihres Systems** in Code:

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
```

11:50 - 12:20: Debugging & Unit Testing mit `unittest`

Ziel: Einführung in **automatisierte Tests** für OOP-Klassen.

- **Theorie:**
  - Warum sind Unit Tests wichtig?
  - Einführung in `unittest`.
- **Beispiel:**

```
import unittest

class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

```
class TestBook(unittest.TestCase):
    def test_book_attributes(self):
        book = Book("1984", "George Orwell")
        self.assertEqual(book.title, "1984")
        self.assertEqual(book.author, "George Orwell")

if __name__ == '__main__':
    unittest.main()
```

- **Übung:**
  - Studierende schreiben **Unit-Tests für ihr eigenes OOP-Projekt**.

12:20 - 13:00: Reflexion, Q&A und Ausblick auf Tag 3

- **Kurze Wiederholung** der wichtigsten Konzepte.
- **Diskussion:** Was war die größte Herausforderung heute?
- **Ausblick auf Tag 3:** Einführung in **NumPy und Pandas für Datenanalyse**.