# Tag 1: Fortgeschrittene objektorientierte Programmierung in Python

Ihr Name

February 1, 2025

# Einleitung

# Willkommen und Rückblick

## Ziele für heute:

- Grundlagen der objektorientierten Programmierung (OOP) festigen.
- Fortgeschrittene Themen wie Kapselung, abstrakte Klassen und Methoden erlernen.
- OOP anwenden, um reale Probleme zu modellieren.
- Debugging und Testen von OOP-Code üben.

## Frage an die Teilnehmer:

- Was sind die Hauptvorteile der Verwendung von OOP in der Softwareentwicklung?

# Kapselung

# Was ist Kapselung?

**Definition:**

- Kapselung bedeutet, Daten und Methoden, die auf diese Daten zugreifen, in einer Einheit (Klasse) zu bündeln.
- Zugriff auf Daten durch private und öffentliche Attribute steuern.

**Schlüsselkonzepte:**

- Öffentliche Attribute: Von außerhalb der Klasse zugänglich.
- Private Attribute: Von außerhalb der Klasse verborgen (Verwendung des Präfixes ‘__‘).

**Vorteile der Kapselung:**

- Schutz der Datenintegrität.
- Reduzierung von Abhängigkeiten.
- Erleichterung von Änderungen und Erweiterungen.

# Beispiel für Kapselung

Codebeispiel:

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

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount("Alice", 1000)
print(account.get_balance())  # Zugriff auf private Daten über Methode
```

# Kapselung und Datenzugriff

Diskussion:

- Warum ist der direkte Zugriff auf ‘__balance‘ nicht erlaubt?
- Wie kann dies zu sicherem Programmieren beitragen?

# Abstrakte Klassen

# Was sind abstrakte Klassen?

**Definition:**

- Abstrakte Klassen dienen als Blaupausen für andere Klassen.
- Sie können nicht direkt instanziiert werden.
- Verwendung des ‘abc‘-Moduls in Python.

**Warum abstrakte Klassen verwenden?**

- Erzwingen die Implementierung bestimmter Methoden in Unterklassen.
- Stellen eine konsistente Schnittstelle über verschiedene Unterklassen sicher.

**Beispiele für den Einsatz:**

- Formen wie Rechtecke und Kreise.
- Fahrzeugtypen wie Autos und Motorräder.

# Beispiel für abstrakte Klassen

Codebeispiel:

```
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

```

# Statische und Klassenmethoden

# Statische und Klassenmethoden

Definitionen:

- **Statische Methoden:** Definiert mit ‘@staticmethod‘. Sie hängen nicht von Instanz- oder Klassenvariablen ab.
- **Klassenmethoden:** Definiert mit ‘@classmethod‘. Sie nehmen die Klasse als ersten Parameter.

Anwendungsfälle:

- Statische Methoden für Hilfsfunktionen (z.B. Berechnungen).
- Klassenmethoden für Factory-Methoden.

# Beispiel für statische und Klassenmethoden

Codebeispiel:

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

    @classmethod
    def pi(cls):
        return 3.14159

print(MathUtils.add(5, 7))  # 12
print(MathUtils.pi())       # 3.14159
```

Diskussion:

- Wann sollte eine statische Methode anstelle einer Instanzmethode verwendet werden?

# Fazit

# Zusammenfassung und Q&A

## Wichtige Erkenntnisse:

- Kapselung schützt Daten innerhalb einer Klasse.
- Abstrakte Klassen bieten eine Blaupause für konsistente Schnittstellen.
- Statische und Klassenmethoden ermöglichen Dienstprogramme und klassenbezogene Verhaltensweisen.
- OOP hilft effektiv bei der Modellierung realer Probleme.

## Nächste Schritte:

- Modellierung realer Probleme üben.
- Heutige Codebeispiele überprüfen und erweitern.
- Experimentieren Sie mit abstrakten Klassen und Methoden.