

# Tag 1: Fortgeschrittene objektorientierte Programmierung in Python

---

Ihr Name

January 29, 2025

# Einleitung

---

## 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:

```
1 class BankAccount:
2     def __init__(self, owner, balance):
3         self.owner = owner
4         self.__balance = balance # Privates Attribut
5
6     def deposit(self, amount):
7         if amount > 0:
8             self.__balance += amount
9
10    def get_balance(self):
11        return self.__balance
12
13 account = BankAccount("Alice", 1000)
14 print(account.get_balance()) # Zugriff auf private Daten über Methode
```

## 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:

```
1 from abc import ABC, abstractmethod
2
3 class Shape(ABC):
4     @abstractmethod
5     def area(self):
6         pass
7
8     @abstractmethod
9     def perimeter(self):
10        pass
11
12 class Rectangle(Shape):
13     def __init__(self, width, height):
14         self.width = width
15         self.height = height
16
```

# 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:

```
1 class MathUtils:
2     @staticmethod
3     def add(a, b):
4         return a + b
5
6     @classmethod
7     def pi(cls):
8         return 3.14159
9
10 print(MathUtils.add(5, 7))    # 12
11 print(MathUtils.pi())        # 3.14159
```

## Diskussion:

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

## Fazit

---

## 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.