OOP Cheat Sheet (Python) – Theorie & Code-Schemata

1 OOP-Grundlagen

1.1 Objektorientierte Programmierung

- Programme werden in **Objekte** strukturiert.
- Objekte besitzen **Attribute** (Daten) und **Methoden** (Funktionen).
- Objekte werden aus **Klassen** (Bauplänen/Vorlagen) erzeugt.
- Typische Vorteile:
  - Strukturierung und bessere Übersichtlichkeit
  - Wiederverwendbarkeit von Code
  - bessere Wartbarkeit und Erweiterbarkeit

1.2 Klasse und Objekt

- **Klasse**: allgemeiner Bauplan, definiert Attribute und Methoden.
- **Objekt**: konkrete Instanz einer Klasse mit eigenen Attributwerten.

1.3 Attribute und Methoden

- **Attribute**: Variablen, die den Zustand eines Objekts beschreiben.
- **Methoden**: Funktionen innerhalb einer Klasse, die Verhalten definieren.
- Attribute werden häufig im Konstruktor initialisiert.

1.4 `self`

- `self` referenziert das aktuelle Objekt.
- Wird als erster Parameter in Methoden einer Klasse verwendet.
- Zugriff auf Attribute und Methoden des Objekts z. B. über `self.attributname`.

1.5 Konstruktor `__init__`

- Spezielle Methode, die beim Erzeugen eines Objekts automatisch ausgeführt wird.
- Signatur in Python: `def __init__(self, ...):`
- Typische Aufgabe: Initialisierung der Attribute.

2 Vererbung und Beziehungen

2.1 Vererbung

- **Superklasse**: allgemeine Klasse mit gemeinsamen Attributen/Methoden.
- **Unterklasse**: spezialisierte Klasse, die von der Superklasse erbt.
- Unterklassen:
  - **erben** Attribute und Methoden der Superklasse.
  - können zusätzliche Attribute/Methoden definieren.
  - können Methoden der Superklasse überschreiben.

- `super()` ermöglicht Aufruf von Methoden der Superklasse, z. B. deren Konstruktor.

2.2 Assoziation

- Beziehungstyp „hat ein“ oder „kennt ein“.
- Eine Klasse enthält ein Attribut, das auf ein Objekt einer anderen Klasse verweist.
- Keine Vererbung, sondern Nutzung einer anderen Klasse.

2.3 Vererbung vs. Assoziation

- Vererbung: „ist ein“-Beziehung.
- Assoziation: „hat ein“ / „verwendet ein“-Beziehung.

3 Kapselung

- Zusammenfassung von Daten (Attributen) und Operationen (Methoden) in einer Klasse.
- Direkter Zugriff auf interne Daten soll möglichst eingeschränkt werden.
- Zugriff idealerweise nur über definierte Methoden (Schnittstellen).
- Ziel: Schutz vor unkontrollierten Änderungen und Fehlern, bessere Wartbarkeit.
- In Python: „private“ Attribute durch Namenskonvention mit doppeltem Unterstrich (z. B. `__attribut`).

4 Code-Schemata (abstrakt)

4.1 Allgemeines Klassenschema

```
class NameDerKlasse:
    def __init__(self, parameter1, parameter2):
        self.attribut1 = parameter1
        self.attribut2 = parameter2

    def name_der_methode(self, parameter):
        # Zugriff auf Attribute und Verarbeitung
        # ...
        pass
```

4.2 Objekterzeugung und Zugriff

```
# Erzeugen eines Objekts der Klasse
objektname = NameDerKlasse(wert1, wert2)

# Zugriff auf Attribute
ausdruck1 = objektname.attribut1

# Aufruf einer Methode
objektname.name_der_methode(wert)
```

4.3 Schema: Superklasse

```
class Superklasse:
    def __init__(self, parameter1, parameter2):
        self.attribut1 = parameter1
```

```
        self.attribut2 = parameter2

    def info_methode(self):
        # Ausgabe oder Verarbeitung von Attributen
        # ...
        pass
```

4.4 Schema: Unterklasse ohne eigenen Konstruktor

```
class Unterklasse(Superklasse):
    def eigene_methode(self):
        # Zugriff auf geerbte Attribute und Methoden
        # ...
        pass
```

4.5 Schema: Unterklasse mit eigenem Konstruktor und `super()`

```
class Unterklasse(Superklasse):
    def __init__(self, parameter1, parameter2, weiterer_parameter):
        # Aufruf des Konstruktors der Superklasse
        super().__init__(parameter1, parameter2)

        # Initialisierung eines neuen Attributs
        self.weitere_attribut = weiterer_parameter

    def weitere_methode(self):
        # Nutzung von geerbten und neuen Attributen
        # ...
        pass
```

4.6 Schema: Vererbung verwenden

```
# Erzeugen von Objekten der Unterklasse
objekt_unterklasse = Unterklasse(wert1, wert2, wert3)

# Aufruf geerbter Methode
objekt_unterklasse.info_methode()

# Aufruf eigener Methode
objekt_unterklasse.weitere_methode()
```

4.7 Schema: Assoziation (eine Klasse kennt eine andere)

```
class Klasse1:
    def __init__(self, name):
        self.name = name

    def aktion(self, anderes_objekt):
        # Zugriff auf Objekt einer anderen Klasse
        # ...
        pass


```

```python
class Klasse2:
    def __init__(self, name, referenz_auf_klasse1):
        self.name = name
        self.referenz = referenz_auf_klasse1  # Assoziation: kennt ein
            Objekt von Klasse1

    def methode_mit_verwendung(self):
        # Zugriff auf Attribute/Methoden von Klasse1  ber  self.referenz
        # ...
        pass
```

4.8 Schema: Verwendung einer Assoziation

```python
objekt1 = Klasse1("name1")
objekt2 = Klasse2("name2", objekt1)

# Klasse2 nutzt Objekt von Klasse1
objekt2.methode_mit_verwendung()
```

4.9 Schema: Kapselung mit internem Attribut

```python
class BeispielMitKapselung:
    def __init__(self, startwert):
        self.__internes_attribut = startwert  # gekapseltes Attribut

    def erhoehen(self, betrag):
        self.__internes_attribut += betrag

    def aktueller_wert(self):
        return self.__internes_attribut
```

4.10 Schema: Klasse ohne Kapselung (nur als Kontrast)

```python
class BeispielOhneKapselung:
    def __init__(self, startwert):
        self.oeffentliches_attribut = startwert  # frei von au en
            nderbar
```