Fortgeschrittene OOP-Konzepte in Python

Dunder-Methoden, Operator Overloading, Komposition vs. Vererbung, Polymorphismus, Duck Typing und Design Patterns

---

Ihr Name

4. Februar 2025

Agenda

Dunder-Methoden und Operator Overloading

Komposition vs. Vererbung (Favor Composition)

Polymorphismus & Duck Typing in Python

Design Patterns (Nur wesentliche)

Zusammenfassung und Ausblick

Dunder-Methoden und Operator Overloading

Einführung in Dunder-Methoden

- Dunder-Methoden (Double Underscore) sind spezielle Methoden in Python, deren Namen mit `__` beginnen und enden.
- Beispiele: `__init__`, `__str__`, `__repr__`, `__add__`, `__len__` usw.
- Sie erlauben es, benutzerdefinierte Klassen in Python nahtlos in Sprachkonstrukte zu integrieren.

Beispiel: `__init__` und `__str__`

```python
class Person:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter

    def __str__(self):
        return f"{self.name}, {self.alter} Jahre alt"

p = Person("Anna", 30)
print(p)  # Ausgabe: Anna, 30 Jahre alt
```

Operator Overloading: Grundlagen

- Operator Overloading erlaubt es, Standardoperatoren (z.B. `+`, `-`, `*`) mit eigenen Klassen zu verwenden.
- Python löst z.B. bei `obj1 + obj2` automatisch die Methode `__add__` auf.

Beispiel: Vektor-Klasse mit Operator Overloading

```python
class Vektor:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vektor(self.x + other.x, self.y + other.y
            )

    def __str__(self):
        return f"Vektor({self.x}, {self.y})"

v1 = Vektor(2, 3)
v2 = Vektor(4, 5)
v3 = v1 + v2
print(v3)  # Ausgabe: Vektor(6, 8)
```

Weitere nützliche Dunder-Methoden

- `__eq__`: Vergleicht Objekte (z.B. `==`)
- `__len__`: Ermöglicht die Verwendung von `len(obj)`
- `__getitem__` und `__setitem__`: Erlauben Indexzugriffe
- `__iter__`: Ermöglicht Iteration über das Objekt

Komposition vs. Vererbung (Favor Composition)

Vererbung: Grundlagen

- Vererbung erlaubt es, eine neue Klasse basierend auf einer bestehenden zu definieren.
- Ermöglicht Code-Wiederverwendung, kann jedoch zu starker Kopplung führen.

```python
class Fahrzeug:
    def starten(self):
        print("Fahrzeug startet")

class Auto(Fahrzeug):
    pass

a = Auto()
a.starten()  # Ausgabe: Fahrzeug startet
```

Komposition: Grundlagen

- Bei der Komposition enthält ein Objekt andere Objekte als Attribute.
- Dies führt zu lose gekoppelten und flexibleren Designs.

```
1  class Motor:
2      def starten(self):
3          print("Motor startet")
4
5  class Auto:
6      def __init__(self):
7          self.motor = Motor()
8
9      def starten(self):
10         self.motor.starten()
11
12 a = Auto()
13 a.starten()  # Ausgabe: Motor startet
```

Warum Komposition bevorzugen?

- Wiederverwendbarkeit: Einzelne Komponenten können in verschiedenen Kontexten genutzt werden.
- Flexibilität: Leichtere Anpassung und Austausch von Komponenten.
- Vermeidung von tiefen Vererbungsstrukturen: Erhöht die Wartbarkeit.

Polymorphismus & Duck Typing in Python

Polymorphismus: Prinzipien

- Erlaubt es, dass unterschiedliche Objekte auf denselben Methodenaufruf reagieren.
- Führt zu generischem Code, der mit verschiedenen Objekttypen arbeitet.

Duck Typing: Das Prinzip

- Wenn es aussieht wie eine Ente und quakt wie eine Ente, dann ist es eine Ente.
- Python überprüft nicht den Typ, sondern ob das Objekt die benötigte Methode besitzt.
- Keine Notwendigkeit, explizit Schnittstellen zu implementieren.

Beispiel: Duck Typing in Aktion

```
class Ente:
    def quaken(self):
        print("Quak, quak!")

class Hund:
    def quaken(self):
        print("Ich kann nicht quaken, aber ich belle!")

def tier_macht_geraeusch(tier):
    tier.quaken()

ente = Ente()
hund = Hund()
tier_macht_geraeusch(ente)  # Ausgabe: Quak, quak!
tier_macht_geraeusch(hund)  # Ausgabe: Ich kann nicht
    quaken, aber ich belle!
```

Vorteile von Polymorphismus und Duck Typing

- Flexibilität: Funktionen und Methoden können mit verschiedenen Objekttypen arbeiten.
- Einfachheit: Reduzierung von starren Typüberprüfungen.
- Ermöglicht generische Programmierung: Bessere Wiederverwendbarkeit des Codes.

Design Patterns (Nur wesentliche)

Einführung in Design Patterns

- Design Patterns sind bewährte Lösungen für wiederkehrende Probleme im Software-Design.
- Sie helfen dabei, den Code verständlicher und wartbarer zu gestalten.
- In Python sind nicht alle Patterns zwingend notwendig Fokus auf wesentliche.

Singleton Pattern

- Ziel: Sicherstellen, dass von einer Klasse nur eine einzige Instanz existiert.
- Nützlich, wenn ein globaler Zugriffspunkt benötigt wird.

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

# Test
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # Ausgabe: True
```

Factory Pattern

- Ziel: Erstellung von Objekten, ohne deren konkrete Klassen zu spezifizieren.
- Bietet Flexibilität bei der Objekterzeugung.

```
class Tier:
    def sprich(self):
        pass

class Katze(Tier):
    def sprich(self):
        print("Miau")

class Hund(Tier):
    def sprich(self):
        print("Wuff")

class TierFactory:
    @staticmethod
    def erstelle_tier(tier_typ):
```

Observer Pattern

- Ziel: Ermöglicht es, dass ein Objekt (Subject) mehrere andere Objekte (Observer) über Zustandsänderungen informiert.
- Besonders nützlich in ereignisgesteuerten Anwendungen.

```
class Observable:
    def __init__(self):
        self.beobachter = []

    def registriere_beobachter(self, beobachter):
        self.beobachter.append(beobachter)

    def benachrichtige_beobachter(self, nachricht):
        for b in self.beobachter:
            b.update(nachricht)

class Beobachter:
    def update(self, nachricht):
```

Strategy Pattern (optional)

- Ziel: Auswahl eines Algorithmus zur Laufzeit.
- Trennt den Algorithmus von der Anwendungslogik.

```
1  class Strategie:
2      def berechne(self, a, b):
3          pass
4
5  class Addition(Strategie):
6      def berechne(self, a, b):
7          return a + b
8
9  class Multiplikation(Strategie):
10     def berechne(self, a, b):
11         return a * b
12
13 class Kontext:
14     def __init__(self, strategie):
15         self.strategie = strategie
16
```

Wichtige Hinweise zu Design Patterns in Python

- Aufgrund der dynamischen Natur von Python sind manche Patterns weniger zwingend.
- Nutzen Sie Patterns als Richtlinien nicht als starre Regeln.
- Wesentlich ist das Verständnis des zugrunde liegenden Problems.

Zusammenfassung und Ausblick

Zusammenfassung

- Dunder-Methoden: Erlauben die Integration von benutzerdefinierten Klassen in die Python-Syntax.
- Operator Overloading: Macht den Einsatz von Operatoren bei eigenen Klassen intuitiver.
- Komposition vs. Vererbung: Komposition bietet oft flexiblere und weniger gekoppelte Designs.
- Polymorphismus & Duck Typing: Fördern generischen, dynamischen Code, der sich auf die Methodenfähigkeit eines Objekts konzentriert.
- Design Patterns: Bieten bewährte Lösungen für wiederkehrende Probleme in Python pragmatisch einsetzen.

Weiterführende Literatur und Ressourcen

- Bücher: Fluent Python von Luciano Ramalho
- Online-Ressourcen: Python-Dokumentation, Blogs zu Design Patterns in Python
- Übungen: Praktische Projekte und kleine Übungen, um die Konzepte zu festigen

Fragen & Diskussion

Vielen Dank für Ihre Aufmerksamkeit!

Haben Sie Fragen?