Laborbericht: **Magic Methods**

Bierl Patrick
22412802
Dpc-2
Dozent: Prof. Tobias Schaffer

11.07.2025

1 Einleitung

Bei der Objektorientierten Programmierung handelt es sich um ein Paradigma, einen Programmierstil der mit Objekten arbeitet, die Daten und Verhalten miteinander verbinden. Objektorientierte Programmierung (OOP) ist auch heute noch eine zentrale Methode in der Softwareentwicklung, da sie eine klare Strukturierung und Organisation komplexer Systeme ermöglicht. Durch die Kapselung von Daten und Verhalten in Klassen lassen sich wiederverwendbare, modulare und gut wartbare Komponenten entwickeln. Dieses Prinzip fördert die Verständlichkeit des Codes und erleichtert es, Änderungen vorzunehmen, ohne dass andere Teile des Programms beeinträchtigt werden. Darüber hinaus orientiert sich OOP an der realen Welt – durch Konzepte wie Vererbung, Polymorphie und Abstraktion lassen sich reale Zusammenhänge intuitiv in Software abbilden. Das ist besonders in großen Projekten von Vorteil, in denen viele Entwickler und Entwicklerinnen gleichzeitig an verschiedenen Teilen des Systems arbeiten. Kein Wunder also, dass viele moderne Programmiersprachen wie Python, Java oder C OOP umfassend unterstützen und dass zahlreiche etablierte Frameworks darauf aufbauen.
Im Kontext von Python spielen sogenannte Magic Methods (magische Methoden) eine besonders wichtige Rolle, um die Möglichkeiten der objektorientierten Programmierung voll auszuschöpfen. Dabei handelt es sich um spezielle Methoden, die durch doppelte Unterstriche gekennzeichnet sind. Sie ermöglichen es, benutzerdefinierte Klassen so zu gestalten, dass sie sich wie eingebaute Datentypen verhalten und sich nahtlos in die Python-typische Arbeitsweise integrieren.
In Kombination sorgen OOP und Magic Methods dafür, dass moderne Software nicht nur funktional, sondern auch verständlich, flexibel und wartbar bleibt. Sie sind ein Grund dafür, dass Python und andere objektorientierte Sprachen sowohl im professionellen Umfeld als auch im Bildungsbereich so beliebt sind. Selbst in Zeiten, in denen funktionale oder komponentenbasierte Ansätze an Bedeutung gewinnen – etwa im Kontext von Data Science oder Webentwicklung – behalten objektorientierte Konzepte ihren Stellenwert. Denn sie bieten eine bewährte Grundlage, auf der auch neue Paradigmen aufbauen können. Magic Methods erweitern dabei die Ausdruckskraft der OOP und geben Entwicklern und Entwicklerinnen die Möglichkeit, eigene Klassen mit minimalem Aufwand so zu gestalten, dass sie sich wie natürliche Teile der Sprache verhalten – ein starker Beweis für ihre anhaltende Relevanz.

2 Methodik

- Gewählter Ansatz:

  In diesem Projekt wurde eine Vektor-Klasse in Python implementiert, die grundlegende mathematische Operationen mit zweidimensionalen Vektoren unterstützt. Ziel ist es, die Handhabung von Vektoren durch Magic Methods (auch dunder methods genannt) zu vereinfachen, sodass intuitive Operatoren wie +, -, *, ==, @ etc. verwendet werden können.

- Verwendete Programmiersprache:

  Als Programmiersprache haben wir Python verwendet, welches die Objektorientierte Programmierung hervorragend unterstützt.

- Verwendete Algorithmen und Methoden:

  Es wurden grundlegende arithmetische Operationen für Vektoren verwendet, um die Magic Methods zu demonstrieren:

```
__add__ Addiert zwei Vektoren komponentenweise
__sub__ Subtrahiert zwei Vektoren komponentenweise
__mul__ Multipliziert einen Vektor mit einem Skalar
__eq__  Vergleicht zwei Vektoren auf gleiche Komponenten
__pow__ Hebt jede Komponente eines Vektors um eine Potenz
__matmul__      Berechnet das Skalarprodukt zweier Vektoren
__str__ Gibt den Vektor in lesbarer Form aus
```

- Eingesetzte Tools:

  Zur Bearbeitung des Projekts wurde Vs Code verwendet. Ausprobiert wurde der Code dann in Jupiter Notebook und Colab.

2.1 Verwendete Software und Hardware

- Programmiersprache: Python
- Hardware:
  - CPU - 13th Gen Intel(R) Core(TM) i7-1355U
  - GPU - Intel(R) Iris(R) Xe Graphics

2.2 Code-Repository

Der vollständige Quellcode dieses Projekts ist auf GitHub verfügbar unter:

`https://github.com/Done09/neue-repository`

Das Repository enthält:

- Quellcode

* Installationsanweisungen
* Beispieldaten (falls vorhanden)
* Dokumentation und Nutzungshinweise

2.3 Code-Implementierung

Vektor Klasse mit Magic Methods:

```
class Vektor:

    def __init__(self, x, y):
        """Initialisiert einen Vektor mit x- und y-Koordinate.
           """
        self.x = x
        self.y = y

    # TODO: Implementieren Sie die Methode zur Subtraktion
       zweier Vektoren
    def __sub__(self, anderer):
    #   """Erm glicht die Subtraktion zweier Vektoren."""
        return Vektor(self.x - anderer.x, self.y - anderer.y)

    # TODO: Implementieren Sie die Methode zur Subtraktion
       zweier Vektoren
    def __sub__(self, anderer):
    #   """Erm glicht die Subtraktion zweier Vektoren."""
        return Vektor(self.x - anderer.x, self.y - anderer.y)

    # TODO: Implementieren Sie die Methode zur skalaren
       Multiplikation
    def __mul__(self, skalar):
    #   """Erm glicht die skalare Multiplikation."""
        return Vektor(self.x * skalar, self.y * skalar)

    # TODO: Implementieren Sie die Vergleichsmethode f r zwei
       Vektoren
    def __eq__(self, anderer):
    #   """Vergleicht zwei Vektoren auf Gleichheit."""
        return self.x == anderer.x and self.y == anderer.y

    # TODO: Implementieren Sie die Methode zur elementweisen
       Potenzierung
    def __pow__(self, exponent):
    #   """Erm glicht die elementweise Potenzierung des
       Vektors."""
        return Vektor(self.x ** exponent, self.y ** exponent)

    # TODO: Implementieren Sie die Methode f r das Skalarprod.
        zweier Vektoren
    def __matmul__(self, anderer):
    #   """Erm glicht das Skalarprodukt zweier Vektoren."""
        return self.x * anderer.x + self.y * anderer.y
```

Testfälle für die Vektoren:

```
f main():

    v1 = Vektor(3, 4)
    v2 = Vektor(1, 2)

    # TODO: Aktivieren Sie die folgenden Zeilen nach
       Implementierung der Methoden
    print("Vektor 1:", v1)
    print("Vektor 2:", v2)

    print("Addition:", v1 + v2)

    # TODO: Aktivieren Sie die folgenden Zeilen nach
       Implementierung der Methoden
    print("Subtraktion:", v1 - v2)
    print("Multiplikation mit Skalar 2:", v1 * 2)
    print("Vergleich (sollte False sein):", v1 == v2)
    print("Vergleich (sollte True sein):", v1 == Vektor(3, 4))
    print("Elementweise Potenzierung mit Exponent 2:", v1 ** 2)
    print("Skalarprodukt (Matmul-Operator):", v1 @ v2)
```

3 Ergebnisse

Die verschiedenen Operationen der einzelnen Vektoren ergab folgende Ergebnisse:

```
    Vektor 1: (3, 4)
    Vektor 2: (1, 2)
    Addition: (4, 6)
    Subtraktion: (2, 2)
    Multiplikation mit Skalar 2: (6, 8)
    Vergleich (sollte False sein): False
    Vergleich (sollte True sein): True
    Elementweise Potenzierung mit Exponent 2: (9, 16)
    Skalarprodukt (Matmul-Operator): 11
```

4 Herausforderungen, Einschränkungen und Fehleranalyse

4.1 Aufgetretene Herausforderungen

* Neben allgemeiner Herausforderungen wie z.B. Syntaxfehler gab es eigentlich wenig Probleme, da die Iplementierung doch sehr intuitiv ist.

4.2 Fehleranalyse

Fehler, die während der Entwicklung auftraten:

* Syntaxfehler
* TypeError
* NameError

4.3 Einschränkungen der Implementierung

Das Modell weist Einschränkungen in der Typprüfung auf, es wird vorausgesetzt, dass der übegebene Paraeter immer ein Vektor oder ein Skalar ist, dies kann zu Laufzeitfehlern führen. Daher könnte eine Fehlerbehandlung(wie in 6. Fazit gezeigt) eingeführt werden.

5 Diskussion

Die Umsetzung der Vektor-Klasse mit sogenannten Magic Methods zeigt sehr gut, wie man in Python eigenen Code schreiben kann, der sich genauso verhält wie eingebaute Datentypen. Dadurch wird der Code nicht nur kürzer, sondern auch leichter zu lesen und zu verstehen. Die eingesetzten Methoden wie `__add__` für Addition oder `__matmul__` für das Skalarprodukt machen es möglich, mit einfachen Rechenzeichen direkt auf Objekte zuzugreifen, als wären es ganz normale Zahlen oder Listen.

Alle getesteten Rechenoperationen liefern die erwarteten Ergebnisse. Zum Beispiel funktioniert die Addition zweier Vektoren genauso, wie man es aus der Mathematik kennt. Das zeigt, dass die Methoden korrekt programmiert wurden.

Was noch fehlt, ist eine Überprüfung, ob die Eingaben wirklich passen. Aktuell wird z.B. nicht geprüft, ob bei einer Rechenoperation auch wirklich ein Vektor oder ein Skalar übergeben wird. Wenn man aus Versehen einen falschen Typ übergibt, kann es zu Fehlermeldungen kommen. Das könnte man verbessern, indem man z.B. mit `isinstance` vorher kontrolliert, ob der richtige Typ übergeben wurde.

Insgesamt zeigt das Projekt, wie hilfreich Magic Methods sind. Man kann damit eigene Klassen schreiben, die sich sehr natürlich und einfach bedienen lassen. Das ist nicht nur für den Unterricht nützlich, sondern auch in echten Projekten hilfreich.

6 Fazit

Die Implementierung ist übersichtlich, korrekt und deckt die Grundoperationen akkurat ab.

Es gibt jedoch auch Verbesserungspotential:

---

z.B.: Fehlende Fehlerbehandlung

* Es wird nicht überprüft, ob die Argumente bei Operationen vom richtigen Typ sind

```
def__add__(self,anderer):
    if not isinstance(anderer, Vektor):
        return Not Implemented
    return Vektor(self.x + anderer.x, self.y + anderer.y
```

---

7 Literaturverzeichnis

Verwendete Referenzen:

* Herr Schaffer Tobias, Google Drive, Laborübungen