# Python Magische Methoden (Magic Methods)

Dunder-Methoden (Double Under)

# Magische Methoden

Magische Methoden, auch als Dunder-Methoden (Double Underscore) bekannt, sind spezielle Methoden in Python, die von der Sprache selbst aufgerufen werden, um grundlegende Operationen auf Objekten durchzuführen.

Diese Methoden ermöglichen es, benutzerdefiniertes Verhalten für eigene Klassen zu definieren. In Python sind magische Methoden durch doppelte Unterstriche am Anfang und am Ende ihres Namens gekennzeichnet, z.B. __init__, __str__ oder __repr__

# Objektinitialisierung und Aufräumarbeiten

__init__  beim Erstellen eines Objekts

__new__ beim Erstellen eines Objekts

__del__ beim Zerstören eines Objekts

# __init__

Die __init__ Methode wird aufgerufen, wenn ein neues Objekt der Klasse erstellt wird. Sie wird verwendet, um den Zustand des Objekts zu initialisieren. Sie hat keinen Rückgabewert.

```
class MyClass:
    def __init__(self, value) -> None:
        self.value = value

obj = MyClass(42)
```

# __new__

Die Methode __new__ in Python ist eine spezielle Methode, die aufgerufen wird, wenn eine neue Instanz einer Klasse erstellt wird.

Sie ist verantwortlich für das Erstellen und Zurückgeben einer neuen Instanz der Klasse.

Während die Methode __init__ für die Initialisierung der Attribute der Instanz verantwortlich ist, ist die Methode __new__ für die Erstellung der Instanz selbst verantwortlich.

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        instanz = super().__new__(cls)
        return instanz

obj = MyClass(42)
```

# Container und Iteration

__len__(self): Definiert die Länge des Objekts (verwendet mit len()).

__getitem__(self, key): Definiert das Verhalten für den Zugriff auf Elemente mit eckigen Klammern, z.B. obj[key].

__setitem__(self, key, value): Definiert das Verhalten für das Setzen von Elementen mit eckigen Klammern, z.B. obj[key] = value.

__iter__(self): Gibt ein Iterator-Objekt zurück, um die Iteration zu unterstützen.

__next__(self): Definiert das Verhalten der next()-Funktion bei der Iteration über ein Objekt.

# ein Custom Dictionary

```
class MyDictionary:
    def __init__(self):
        self.data = {}

    def __setitem__(self, key, value):
        self.data[key] = value

    def __getitem__(self, key):
        return self.data[key]

my_dict = MyDictionary()
my_dict["name"] = "Bob"
print(my_dict["name"])
```

# Vergleichsoperatoren

__eq__(self, other): Definiert das Verhalten des Gleichheitsoperators ==

__ne__(self, other): Definiert das Verhalten des Ungleichheitsoperators !=

__lt__(self, other): Definiert das Verhalten des Kleiner-als-Operators <

__le__(self, other): Definiert das Verhalten des Kleiner-als-oder-gleich-Operators <=

__gt__(self, other): Definiert das Verhalten des Größer-als-Operators >

__ge__(self, other): Definiert das Verhalten des Größer-als-oder-gleich-Operators >=

# __eq__ Beispiel Implementierung

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

    def __eq__(self, other: Point):
        if isinstance(other, Point):
            return self.x == other.x and self.y == other.y
        return False

point1 = Point(1, 2)
point2 = Point(1, 2)

print(point1 == point2)
```

# Arithmetische Operationen

Arithmetische Operatoren sind ebenfalls als Dunder-Methoden implementiert.

__add__(self, other): Definiert das Verhalten des Additionoperators +.
__sub__(self, other): Definiert das Verhalten des Subtraktionsoperators -.
__mul__(self, other): Definiert das Verhalten des Multiplikationsoperators *.
__truediv__(self, other): Definiert das Verhalten des Divisionsoperators /.
__floordiv__(self, other): Definiert das Verhalten des Ganzzahldivisionsoperators //.
__mod__(self, other): Definiert das Verhalten des Moduloperators %.
__pow__(self, other[, modulo]): Definiert das Verhalten des Potenzoperators **.

# __add__

Operatoren sind als Dunder-Methoden implementiert. So lässt sich mit __add__ die Addition in einer Klasse implementieren.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector2D):
            result_x = self.x + other.x
            result_y = self.y + other.y
            return Vector2D(result_x, result_y)

vector1 = Vector2D(1, 2)
vector2 = Vector2D(3, 4)
result_vector = vector1 + vector2
```