LU02e - By Value und By Reference bei Dataclasses und Objekten in Python

Einführung

In Python werden Objekte, einschließlich Dataclasses, immer By Reference übergeben. Das bedeutet, dass bei der Übergabe eines Objekts an eine Funktion eine Referenz auf das Objekt übergeben wird und keine Kopie davon erstellt wird. Änderungen, die innerhalb der Funktion an den Attributen des Objekts vorgenommen werden, wirken sich direkt auf das ursprüngliche Objekt aus.

Dataclasses

Dataclasses wurden in Python 3.7 eingeführt und bieten eine bequeme Möglichkeit, Klassen zu definieren, die hauptsächlich Daten speichern. Sie reduzieren den Boilerplate-Code, der normalerweise mit der Definition von Klassen einhergeht, und bieten gleichzeitig eine klare und strukturierte Art der Datenverwaltung.

Hier ein einfaches Beispiel für eine Dataclass:

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
```

Diese Dataclass Person hat zwei Attribute: name und age.

By Reference bei Dataclasses

Da Dataclasses Objekte in Python sind, werden sie By Reference übergeben. Das bedeutet, dass jede Veränderung der Attribute eines Dataclass-Objekts innerhalb einer Funktion direkt auf das Originalobjekt wirkt.

Beispiel: By Reference mit Dataclasses

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
```

```
def birthday(person: Person):
    """
    Increments the age of the person by 1.
    """
    person.age += 1

if __name__ == '__main__':
    p = Person(name='Alice', age=30)
    print(f'Vor der Änderung: {p}')  # Output: Vor der Änderung:
Person(name='Alice', age=30)
    birthday(p)
    print(f'Nach der Änderung: {p}')  # Output: Nach der Änderung:
Person(name='Alice', age=31)
```

In diesem Beispiel wird die Instanz der Dataclass `Person` an die Funktion `birthday()` übergeben. Da Objekte By `Reference` übergeben werden, wird das Attribut `age` der Originalinstanz `p` direkt in der Funktion verändert.

Mutable und Immutable Attribute in Dataclasses

Python-Dataclasses können sowohl mutable als auch immutable Attribute enthalten. Unabhängig davon, ob die Attribute mutable oder immutable sind, wird die Dataclass selbst immer By `Reference` übergeben.

Problem: Mutable Default Values

Ein häufiges Problem bei der Verwendung von mutable Objekten (wie Listen) in Dataclasses ist, dass alle Instanzen der Dataclass denselben Standardwert teilen können, wenn dieser direkt als Default-Wert gesetzt wird. Dies kann zu unerwartetem Verhalten führen.

```
from dataclasses import dataclass

@dataclass
class Student:
    name: str
    grades: list = []

# Problem: Alle Instanzen teilen sich dieselbe Liste.
if __name__ == '__main__':
    student1 = Student(name='Alice')
    student2 = Student(name='Bob')

    student1.grades.append(90)
    print(student2.grades)  # Output: [90] - Dies ist wahrscheinlich nicht
das gewünschte Verhalten
```

In diesem Beispiel teilen sich alle Instanzen der Dataclass `Student` die gleiche Liste `grades`, da der Standardwert direkt gesetzt wurde. Wenn eine Note zu einer Instanz hinzugefügt wird, beeinflusst dies auch alle anderen Instanzen.

Lösung: Verwendung von `field(default_factory=list)`

Um dieses Problem zu lösen, verwendet man in Dataclasses für mutable Standardwerte wie Listen die Funktion `field` mit dem Parameter `default_factory`. Dadurch wird sichergestellt, dass jede Instanz ihre eigene Liste erhält.

```
from dataclasses import dataclass, field

@dataclass
class Student:
    name: str
    grades: list = field(default_factory=list)

if __name__ == '__main__':
    student1 = Student(name='Alice')
    student2 = Student(name='Bob')

    student1.grades.append(90)
    print(student2.grades)  # Output: [] - Jede Instanz hat ihre eigene,
unabhängige Liste
```

Durch die Verwendung von `field(default_factory=list)` wird sichergestellt, dass jede Instanz von `Student` eine eigene Liste erhält, und Änderungen an einer Instanz wirken sich nicht auf andere Instanzen aus.

Immutable Dataclasses

Wenn Sie möchten, dass eine Dataclass unveränderlich ist, können Sie sie durch Setzen des `frozen`-Parameters in `@dataclass` unveränderlich machen. In einer gefrorenen (frozen) Dataclass sind alle Attribute unveränderlich, und jede versuchte Änderung führt zu einem Fehler. Dies ähnelt dem Verhalten von By Value, da die ursprüngliche Instanz nicht verändert werden kann.

Beispiel: Immutable Dataclass

```
from dataclasses import dataclass

@dataclass(frozen=True)
class ImmutablePerson:
    name: str
    age: int
```

In diesem Fall ist die Dataclass `ImmutablePerson` unveränderlich, was bedeutet, dass jedes Attribut,

sobald es gesetzt ist, nicht mehr geändert werden kann. Jeder Versuch, das Attribut age oder name zu ändern, führt zu einem Fehler.

Beispiel: Anwendung einer frozen Dataclass

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Person:
    name: str
    age: int

def birthday(person: Person) -> Person:
    """
    Returns a new Person instance with the age incremented by 1.
    """
    return Person(name=person.name, age=person.age + 1)

if __name__ == '__main__':
    p = Person(name='Alice', age=30)
    print(f'Vor der Änderung: {p}')  # Output: Vor der Änderung:
Person(name='Alice', age=30)
    new_p = birthday(p)
    print(f'Nach der Änderung: {p}')  # Output: Nach der Änderung:
Person(name='Alice', age=30)
    print(f'Neue Instanz: {new_p}')  # Output: Neue Instanz:
Person(name='Alice', age=31)
```

Erläuterung:

Die Dataclass `Person` ist als `frozen` markiert, was sie unveränderlich macht. Die Funktion `birthday` ist eine pure function. Sie nimmt eine Instanz der Dataclass `Person` als Eingabe und gibt eine neue Instanz zurück, wobei das age-Attribut um eins erhöht wird. Das Originalobjekt p bleibt unverändert. Stattdessen wird eine neue Instanz new_p mit dem geänderten Alter erstellt.

Vorteile:

- **Keine Seiteneffekte:** Da das Originalobjekt nicht verändert wird, gibt es keine ungewollten Änderungen am Zustand.
- **Vorhersehbarkeit:** Pure functions sind leichter zu verstehen und zu testen, da das Ergebnis nur von den Eingabewerten abhängt.
- **Immutable Data:** Die Verwendung von immutable Datenstrukturen verhindert unbeabsichtigte Modifikationen und erleichtert die Parallelisierung von Programmen.

Dieses Vorgehen ist besonders nützlich in Szenarien, in denen Datenintegrität und Vorhersagbarkeit des Programmverhaltens von hoher Bedeutung sind.

Zusammenfassung

In Python werden Objekte, einschließlich Dataclasses, immer By `Reference` übergeben. Änderungen an den Attributen eines Objekts innerhalb einer Funktion wirken sich direkt auf das Originalobjekt aus. Um ungewollte Änderungen zu vermeiden, kann eine Dataclass als `frozen` deklariert werden, was sie unveränderlich macht und vor unbeabsichtigten Modifikationen schützt. Zusätzlich ist es wichtig, bei der Verwendung von mutable Default-Werten, wie Listen, darauf zu achten, dass jede Instanz ihre eigene unabhängige Kopie erhält, indem `field(default_factory=list)` verwendet wird.

Tipp: Wenn Sie mutable Attribute in einer Dataclass verwenden, sollten Sie sorgfältig überlegen, ob die direkte Veränderung dieser Attribute innerhalb von Funktionen in Ihrem Programmdesign erwünscht ist. Verwenden Sie `frozen` Dataclasses oder `field(default_factory=list)`, um unbeabsichtigte Änderungen zu verhindern.