LU13a - Spezielle Methoden

Notizen

Zerlegen in mehrere einzelne Seiten. Beispiele besser ordnen und dokumentieren.

print() von Objekten

Vielleicht haben Sie bereits einmal probiert, eine eigene Klasse direkt mit `print()` auszugeben.

Das Resultat (hier am Beispiel der Klasse `Car`) auf der Konsole sah bestimmt irgendwie in dieser Art aus:

```
Car(brand='Toyota', model='Corolla', construction=2010)
```

Das liegt daran, das `print()` die Methode `__repr__` der Klasse `Car` aufruft, diese Methode wird für sie durch den `@dataclass` Decorator erzeugt. Der Aufbau ist dabei immer identisch, die generierte Repr-Zeichenkette enthält den Klassennamen sowie den Namen und die Repr der einzelnen Attribute in der Reihenfolge, in der sie in der Klasse definiert sind.

__repr__

Die `__repr__` Methode hat den Zweck, die „offizielle“ String-Darstellung eines Objekts zu erzeugen. Diese Repräsentation wird für Debugging- und Protokollierungszwecke verwendet und sollte eine Zeichenkette sein, die, wenn sie ausgewertet (`eval()`) wird, ein Objekt mit demselben Wert wie das ursprüngliche Objekt erzeugen würde.

Diese `__repr__` Methode sollte daher nicht leichtfertig einfach überschrieben werden. Trotzdem gibt es sicherlich den Fall, dass wir die `print()`-Ausgabe eines Objektes an unsere Bedürfnisse anpassen möchten. Dafür gibt es die `__str__` Methode.

__str__

Die Methode `__str__` in Python ähnelt der Methode `__repr__`, aber sie wird verwendet, um die „informelle“ String-Darstellung eines Objekts zu erzeugen. Diese Darstellung wird immer dann verwendet, wenn eine String-Darstellung eines Objekts angefordert wird, z.B. bei der Verwendung der Funktion `print()`, der Umwandlung eines Objektes in einen String `str(car)` oder bei der String-Verkettung. Die Methode `__str__` sollte eine Zeichenkette erzeugen, die benutzerfreundlicher und leichter zu lesen ist als die `__repr__`-Darstellung.

Zu beachten gilt: Ist die `__str__` Methode nicht implementiert, so wird beim `print()` auf die implementierte `__repr__` Methode ausgewichen.

Beispiel:

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    address: str
    city: str
    zip :str


if __name__ == '__main__':
    p = Person('Marcel Ferreira','Hinter dem Haus 3',
'Hinterhelfenschwil', '8005')
    print(p)
```

Die Ausgabe entspricht nun der Ausgabe der Methode `__repr__`, welche `@dataclass` für uns generiert hat.

```
Person(name='Marcel Ferreira', address='Hinter dem Haus 3',
city='Hinterhelfenschwil', zip='8005')
```

Ergänzen wir jetzt die `__str__`-Methode können wir die Ausgabe beinflussen:

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    address: str
    city: str
    zip :str


    def __str__(self):
        return self.name + '\n' + self.address + '\n' +
self.zip+ ' ' + self.city


if __name__ == '__main__':
    p = Person('Marcel Ferreira','Hinter dem Haus 3',
'Hinterhelfenschwil', '8005')
    print(p)
```

Die Ausgabe entspricht nun der Ausgabe der Methode `__repr__`, welche `@dataclass` für uns generiert hat.

```
Marcel Ferreira
Hinter dem Haus 3
8005 Hinterhelfenschwil
```

Sonderfall print() von Listen

```
@dataclass
class Car():
    brand: str
    model: str
    construction: int

    def __str__(self):
        return f"I'm a {self.model} from {self.brand}
constructed in {self.construction}"


if __name__ == "__main__":

    car = Car('Toyota', 'Corolla', 2010)
    car1 = Car('Tesla', 'Model 3', 2019)

    cars = [car, car1]

    print(car)
    print(cars)
```

```
I'm a Corolla from Toyota constructed in 2010
[Car(brand='Toyota', model='Corolla', construction=2010),
Car(brand='Tesla', model='Model 3', construction=2019)]
```

Obwohl die Klasse `Car` die Methode `__str__` implementiert hat, wird die `__repr__` Methode aufgerufen, wenn mehrere Objekte der Klasse `Car` in einer Liste geprintet werden. Eine mögliche Lösung für diesen Fall, wäre der Umweg über eine Listen-Abstraktionen:

```
...
print([str(item) for item in cars])
```

```
["I'm a Corolla from Toyota constructed in 2010", "I'm a
Model 3 from Tesla constructed in 2019"]
```

Objekte vergleichen

Sie haben sich vielleicht bereits gefragt, wie Sie nun eigene Objekte miteinander vergleichen können. Verleiche sind wichtig um beispielsweise sortieren zu können. Nehmen wir an, wir haben eine Liste mit 5 Autos. Und möchten diese Autos nun nach Jahrgang sortieren.

```
@dataclass
class Car():
    brand: str
    model: str
    construction: int


if __name__ == "__main__":
    cars = [Car('BMW', 'M3', 2019), Car('Audi', 'A4', 2018),
Car('Mercedes', 'C200', 2017),
            Car('Tesla', 'Model 3', 2019), Car('Toyota',
'Corolla', 2012)]
```

Standardmäßig implementiert eine `dataclass` die `__eq__` -Methode welche für den == Vergleich benötigt wird.

Um verschiedene Arten von Vergleichen wie `__lt__` ⇒ < , `__le__` ⇒ <=, `__gt__` ⇒ >, `__ge__` ⇒ >= zu erlauben, kann man das Argument `order` des `@dataclass` Dekorators auf `True` setzen:

```
from dataclasses import dataclass

@dataclass(order = True)
class Car():
    brand: str
    model: str
    construction: int


if __name__ == "__main__":
    cars = [Car('BMW', 'M3', 2019), Car('Audi', 'A4', 2018),
Car('Mercedes', 'C200', 2017),
            Car('Tesla', 'Model 3', 2019), Car('Toyota',
'Corolla', 2012)]
```

Auf diese Weise sortiert die Datenklasse die Objekte nach jedem Feld (in der Reihenfolge wie sie deklariert sind), bis sie einen Wert findet, der nicht gleich ist.

In der Praxis möchte man in der Regel eher einen oder vielleicht 2 bestimmte Werte vergleichen.

Um die Klasse `Car` nach `construction` sortierbar zu machen

Im Folgenden finden Sie ein Beispiel, wie Sie diese Vergleichsmethoden für die Klasse `Car` definieren können:

```
@dataclass
class Car():
    brand: str
```

```
    model: str
    construction: int

    def __lt__(self, other):
        if isinstance(other, Car):
            if self.construction < other.construction:
                return True
        return False

    def __le__(self, other):
        if isinstance(other, Car):
            return self.construction <= other.construction
        return False

    def __gt__(self, other):
        if isinstance(other, Car):
            return self.construction > other.construction
        return False

    def __ge__(self, other):
        if isinstance(other, Car):
            return self.construction >= other.construction
        return False

    def __eq__(self, other):
        if isinstance(other, Car):
            return self.construction == other.construction
        return False

    def __ne__(self, other):
        return not __eq__(other)
```

In diesem Beispiel prüfen die Methoden `__lt__`, `__le__`, `__gt__`, `__ge__` und `__eq__`, ob das andere Objekt eine Instanz der Klasse `Car` ist. Ist dies der Fall, vergleichen sie das Attribut `construction` der beiden Car-Objekte, um festzustellen, ob sie den angegebenen Vergleichsoperator erfüllen. Wenn das andere Objekt keine Instanz der Klasse `Car` ist, dann gibt die Vergleichsmethode False zurück.

Sobald Sie diese Vergleichsmethoden für die Klasse Car definiert haben, können Sie die Operatoren <, <=, >, >=, == und != verwenden, um Car-Objekte miteinander zu vergleichen. Automatisch werden nun die von Ihnen definierten Vergleichsmethoden verwendet, um die relative Reihenfolge der Objekte zu bestimmen. Ein Beispiel:

```
# Create two Car objects
c1 = Car('Ford', 'Fiesta', 2010)
c2 = Car('Toyota', 'Camry', 2012)

# Check if the first Car is less than the second Car
if c1 < c2:
  print('c1 is less than c2')
```

```
else:
  print('c1 is not less than c2')
```

In diesem Beispiel wird die `__lt__`-Methode der `Car`-Klasse aufgerufen, um die beiden `Car`-Objekte zu vergleichen, und sie gibt `True` zurück, wenn das Attribut `construction` des ersten `Car`-Objekts kleiner ist als das Attribut `construction` des zweiten `Car`-Objekts, und `False`, wenn dies nicht der Fall ist.

Die Logik einer Vergleichs-Methode kann auch umfangreicher sein, als nur der Vergleich eines Attributs.

```
...
    def __lt__(self, other):
        if isinstance(other, Car):
            if self.make < other.make:
                return True
            if self.brand < other.brand:
                return True
        return False
```

Hier wird zuerst der Hersteller und falls beide gleich sind, das Modell verglichen.

Sortieren

Sobald Sie die Klasse `Car` sortierbar gemacht haben, können Sie sie auch mit der Funktion `sorted()` verwenden, um eine Liste von Car-Objekten zu sortieren. Zum Beispiel:

```
@dataclass
class Car():
    brand: str
    model: str
    construction: int

    def __lt__(self, other):
        if isinstance(other, Car):
            return self.construction < other.construction
        return False


if __name__ == "__main__":
    cars = [Car('BMW', 'M3', 2019), Car('Audi', 'A4', 2018),
Car('Mercedes', 'C200', 2017),
            Car('Tesla', 'Model 3', 2019), Car('Toyota',
'Corolla', 2012)]
    sorted_cars = sorted(cars)
```

```
    print(sorted_cars)
```

```
[Car(brand='Toyota', model='Corolla', construction=2012),
Car(brand='Mercedes', model='C200', construction=2017),
Car(brand='Audi', model='A4', construction=2018),
Car(brand='BMW', model='M3', construction=2019),
Car(brand='Tesla', model='Model 3', construction=2019)]
```

Für ein einfaches Sortieren mit der `sorted(list)`-Funktion reicht es, wenn Sie die `__lt__`-Funktion implementiert haben.

getter_for_not_existing_attribute() - Methode

In manchen Klassen macht es Sinn einen Getter für einen berechneten Wert zu ergänzen, anstatt ein Attribut für diesen Wert zu haben. Betrachten wir das Beispiel der Klasse `Person`, eine `Person` hat ein `date_of_birth`, das `age` der `Person` ist aber jedes Jahr ein anderes. Es macht also keinen Sinn, das Attribut `age` in der Klasse zu speichern, denn der Wert des Attributes ist spätestens nach 365 Tagen wieder veraltet.

```
from dataclasses import dataclass
from datetime import datetime


@dataclass
class Person:
    name: str
    date_of_birth: datetime
```

Trotzdem kann es ganz praktisch sein, das Alter einer Person mit einer Methode abfragen zu können. Wir ergänzen den Code also um das `@property age` und berechnen in der Methode das Alter der Person und geben dieses zurück.

```
from dataclasses import dataclass
from datetime import datetime

@dataclass
class Person:
    name: str
    date_of_birth: datetime


    @property
    def age(self):
        return datetime.now().year - self.date_of_birth.year
```

```
if __name__ == '__main__':
    p = Person('Peter', datetime(1999,1,1))
    print(p.age)
```

Für das Speichern von Datumswerten ist der Datentyp `datetime` aus dem Modul `datetime` hervorragend geeignet. Mehr dazu finden Sie in der Learningunit LU14 - DateTime.

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/de/modul/m319/learningunits/lu13/funktionenfuerklassen

Last update: **2026/01/09 10:54**