Mutation und Klassen

Richard Müller, Tom Felber

18. November 2021

Python-Kurs

Gliederung

1. Wiederholung
2. Gesamtübersicht
3. Referenzen und Mutation
4. Klassen und Objekte

Wiederholung

Wiederholung

Beim letzten Mal

- Tupel

```
1 tupel_eins = (1, True, 4.5, "hallo")
```

- Dictionary

```
1 lexikon = {"Haus": "Substantiv", "stehlen": "Verb", "
      Geld": "Substantiv"}
```

Gesamtübersicht

Gesamtübersicht

Themen der nächsten Stunden

- Referenzen Erklärung
- Klassen
- Imports
- Nützlich funktionen zur Iteration
- Lambda
- File handeling
- Listcomprehension
- Unpacking
- Dekoratoren

Referenzen und Mutation

Referenzen

Python benutzt ein Konzept namens "Call by Object-Reference".

Alle Objekte, die in einem Programm auftauchen, liegen im Arbeitsspeicher. Variablen dienen dann dazu, um diesen Objekten einen Namen, eine sogenannte Referenz zu geben.

Variablen sind also nicht ihre Objekte selber, sondern bloß Bezeichner für diese.

Referenzen

```
1 hallo = "Hallo Welt"
2 print(hallo)
```

Führt man diesen Code aus, so legt Python den String 'Hallo Welt' irgendwo in den Arbeitsspeicher.

Die Variable `hallo` ist nun die Referenz. Greift man auf sie zu, so folgt Python der Referenz in den Speicher und liest dort das eigentliche Objekt aus.

Referenzen

Ein Objekt kann auch mehrere Bezeichner haben.

```
1 a = [1]
2 b = a
```

Hier ist `a` der Name für eine Liste. In Zeile 2 wird nun nicht die Liste kopiert, sondern nur die Referenz darauf. `b` ist also nur ein weiterer Bezeichner für *dieselbe* Liste.

Dieser Sachverhalt wird deutlich, wenn wir versuchen, `b` zu verändern:

```
1 b.append(2) # a und b sind [1, 2]
```

Python folgt `b` bis zu der Liste und fügt eine 2 an. Da `a` immer noch eine Referenz auf dieselbe Liste hat, wird die Veränderung auch ersichtlich, wenn wir uns `a` statt `b` anschauen.

Mutable

Das Beispiel aus der letzten Folie funktioniert so, weil Listen veränderlich (mutable) sind. Das heißt, die Liste selbst wird beim appenden verändert, sodass die Änderung bei allen Referenzen sichtbar wird. Die Referenzen selber ändern sich allerdings nicht. Dieser Sachverhalt gilt für *alle* veränderlichen Objekte.

Immutable

Unveränderliche (immutable) Typen, wie Integer oder Strings, verhalten sich anders.

```
1 a = 1
2 b = a
3
4 b += 5
```

Hier hat ein Integer zwei Referenzen (`a` und `b`). Nutzt man nun `b` , um auf den Integer etwas drauf zu addieren, so kann Python nicht einfach das bestehende Zahlenobjekt verändern, sondern ist gezwungen, ein neues zu erstellen. Anschließend wird die Referenz von `b` auf das neue Objekt gesetzt, es wird also die Referenz verändert und nicht das Objekt. Daher sind `a` und `b` nun nicht mehr das selbe Objekt.

Listen und Referenzen

Listen enthalten keine Objekte, sondern nur Referenzen auf diese. Die Objekte selber liegen irgendwo anders.

```
1 liste = [[1, 2], 20]
2 innere_liste = liste[0]
3 innere_liste.append(5)
4
5 print(liste) # liste = [[1, 2, 5], 20]
```

In diesem Beispiel verändert man die innere Liste unabhängig von der äußeren. Trotzdem wird die Veränderung auch in der großen Liste sichtbar, da man wieder nur mehrere Referenzen auf das selbe Objekt hat.

Klassen und Objekte

Klassen und Objekte

Menschen denken in Objekten, denen Eigenschaften und Funktionen zugeordet werden.

Ein Rennwagen ist schnell und die Kuh macht "muh".

Deswegen eignet sich dieses Konzept gut, um Code intuitiv zu strukturieren.

bekannte Beispiele

Listen: `liste.append('element')`
Dictionaries: `dictionary.keys()`

Die `keys()` -Funktion ist Teil des Dictionary Objekts. Eine Liste z.B. hat keine `keys()` -Funktion.

`liste.keys()` wird fehlschlagen.

Mit dem Punkt `.` kann auf die Funktionen des Objekts und Attribute zugegriffen werden.

```
1 rennwagen1.speed
```

Mit den Klammern wird angezeigt, dass man eine Funktion des Objekts ausführen will.

```
1 kuh1.muh()
```

Funktion eines Objekts $\rightarrow$ Methode

Klassen / Objekte selbst definieren

Um ein Objekt zu erhalten, muss zunächst eine Klasse definiert werden. Anschließend muss diese Klasse ausgeführt (instanziiert) werden. Die `__init__` Funktion wird jedesmal aufgerufen, wenn eine neue Instanz der Klasse erzeugt wird (Konstruktor).

```
1 class Rennwagen:
2     def __init__(self):
3         self.speed = "sehr hoch"
4
5 rennwagen1 = Rennwagen()
```

```
1 rennwagen1.speed
```

self

In jeder Methode einer Klasse wird `self` als Eingabe mitgegeben. `self` repräsentiert eine Referenz zu dem jeweiligen Objekt, dass die Methode aufgerufen hat.
In der `__init__` Methode kann `self` benutzt werden, um die Attribute jeder Instanz der Klasse zu setzen. (Zeile 3)

```
1 class Rennwagen:
2     def __init__(self):
3         self.speed = "sehr hoch"
4
5 rennwagen1 = Rennwagen()
```

```
1 rennwagen1.speed
```

Methoden

Methoden können nach dem gleichen Schema angelegt werden, wie Funktionen, mit zwei Unterschieden.

- methoden enthalten mindestens `self` als Eingabe-Argument

```
1     def muh(self):
```

- methoden werden unter Klassen definiert

```
1 class Kuh:
2     def muh(self):
```

Magische Methoden

Es gibt einige Methodennamen, die in Klassen reserviert sind. Alle starten und enden mit doppeltem Unterstrich. Diese Methoden heißen auch "magic methods", weil sie spezifische Dinge, scheinbar magisch, automatisch passieren lassen:

`__*f_name*__`

Methodenname	Zweck
`__init__`	Konstruktor: wird bei Objekt() aufgerufen
`__str__`	bestimmt Rückgabewert von str(Objekt)
`__add__`	bestimmt Rückgamewert Objekt + Objekt

Beispiel


GIRATINA
Lv50
KP
PANFERNO
♂Lv50
KP
150/150
GIRATINA (Gegner) übt seine
Fähigkeit Erzwinger aus!

Beispiel

Problemmodellierung: Pokemon

- Eigenschaften
  - Name
  - maximale HP (Health Points)
  - aktuelle HP
  - Angriffs Kraft
- Aktionen / Funktionen
  - angegriffen werden
  - anderes Pokemon angreifen
  - geheilt werden

zu beachten:

- HP sollen nicht unter 0 sinken
- Pokemon mit 0 HP können nicht angreifen