Unpacking und Lambdas

Richard Müller, Tom Felber

9. Dezember 2021

Python-Kurs

Gliederung

1. Wiederholung

2. Unpacking

3. Lamdba

Gesamtübersicht

Gesamtübersicht

Themen der nächsten Stunden

- Referenzen Erklärung
- Klassen
- Imports
- Nützliche Funktionen zur Iteration
- Lambda
- Unpacking
- File handeling
- Listcomprehension
- Dekoratoren

Wiederholung

Wiederholung

Beim letzten Mal:

- Imports

```
1 import time
2 # hält das Programm für 5s an
3 time.sleep(5)
```

```
1 for t in itertools.product(numbers, repeat=2):
2     print(t)
```

- Keyword Arguments

```
1 aktion("X", argument_3="Hallo")
```

Unpacking

Was ist Unpacking?

Unpacking, bzw. Packing, bezeichnet eine Operation, mit der man Elemente eines `iterables` direkt an Variablen binden kann. Wir kennen das bereits von Tupeln:

```
1 (a, b, c) = (1, 2, 3)
2 # oder
3 a, b, c = (1, 2, 3)
4 # oder
5 a, b, c = 1, 2, 3
```

Das funktioniert mit jedem `iterable`, also auch mit Listen zum Beispiel.

Der *-Operator

Der *-Operator gestattet das Zusammenfassen mehrerer Werte in einer einzigen Variable:

```
1 *a, = 1, 2, 3
2 # a = [1, 2, 3]
```

Auch wenn nach der Variable (hier `a`) nichts mehr folgt, muss trotzdem das Komma gesetzt werden.

Der *-Operator

Man kann den *-Operator auch zusammen mit dem herkömmlichen Packing/Unpacking verbinden. Python ermittelt selbst, welche Variable welche Werte bekommen muss:

```
a*, b = 1, 2, 3
# a = [1, 2], b = 3
a, *b = 1, 2, 3
# a = 1, b = [2, 3]
a, *b, c = 1, 2, 3, 4, 5
# a = 1, b = [2, 3, 4], c = 5
*a, b, c, d = 1, 2, 3, 4, 5
# a = [1, 2], b = 3, c = 4, d = 5
```

Achtung In so einem Ausdruck darf der *-Operator höchstens einmal vorkommen.

Der *-Operator

Man kann Unpacking auch in Verbindung mit Funktionen benutzen:

```
1 def func(*args):
2     print(args)
3
4 func(42, 1337, True, None, "Hello")
5 # gibt (42, 1337, True, None, "Hello") aus
```

oder anders herum:

```
1 def func(a, b, c):
2     print(f"{a}, {b}, {c}")
3
4 func(*(False, "Hi", 8))
5 # gibt "False Hi 8" aus
```

Der *-Operator

oder zusammen:

```
1 def func(*args):
2     print(args)
3
4 func(*(False, "Hi", 8))
5 # gibt (False, "Hi", 8) aus
```

Hier wird das Tupel ausgepackt, sodass alle Elemente einzeln in die Funktion gegeben werden. Effektiv verhält es sich also wie in Beispiel 1.

Der **-Operator

Der **-Operator ist das Unpacking-Equivalent für Dictionarys. Dieser ist vor allem für Funktionen interessant, denn durch ihn kann man das Unpacking mit Keyword-Argumenten verbinden:

```
1 def func(**kwargs):
2     print(kwargs)
3
4 func(var_1=42, var_2="Hello World", var_3=True)
5 # gibt {var_1: 42, var_2: "Hello World", var_3=True aus}
```

oder zusammen mit dem *-Operator:

```
1 def func(*args, **kwargs):
2     print(args)
3     print(kwargs)
4
5 func(1, 2, wert_1=None, wert_2=1337)
6 # args ist (1, 2)
7 # kwargs ist {wert_1: None, wert_2: 1337}
```

Lamdba

Funktionen als Objekte

Funktionen können als Objekte behandelt werden, also z.B. als Argument in einer weiteren Funktion angenommen werden.

```
1 def hallo():
2     print("Hallo Welt")
3     return 100
```

```
1 print(hallo()) # 100
2 print(hallo)   # <function hallo at 0x7fa0353a30d0>
```

Die Klammern machen dabei den Unterschied, ob das Funktions Objekt gemeint ist, oder der Wert, den die Funktion zurück gibt.

Funktionen als Objekte

Beispiel einer Funktion, die eine weitere Funktion `action` als Argument entgegen nimmt und ausführt. (und die Zeit der Ausführung ausgibt)

```
1  import time
2
3  def hallo():
4      print("Hallo Welt")
5      return 100
6
7  def zeit_messen(action):
8      t1 = time.time()
9
10     action()
11
12     t2 = time.time()
13     print(t2 - t1)
14
15 zeit_messen(hallo)
```

Funktionen als Objekte

Anwendungsfall: Eine Funktion wird nur einmal benötigt, und zwar als Argument.

```
def wrapper(funktion):
    print(funktion())
```

Kann man die klassische Funktionsdefinition direkt als Argument reingeben ? NEIN! (und es sieht komisch aus)

```
wrapper(
    def meine_funktion():
        return "Hallo"
)
```

Stattdessen:

```
wrapper(lambda: "Hallo")
```

Lambda Funktionen

Eine Lambda Funktion startet mit dem keyword `lambda` . Es folgen mit Komma getrennte Argumente, dann ein Doppelpunkt. Danach folgt, was ausgegeben werden soll.

```
1 def funktion(a1, a2):
2     return a1 + a2
3
4 lambda a1, a2: a1 + a2
```

`lambda` Funktionen verhalten sich wie gewöhnliche Funktionen, nur das sie in einer Zeile definiert werden können und damit keinen Namen brauchen. Deshalb nennt man sie auch anonyme Funktionen.

Lambda Funktionen

Beispiel `map`

```
1  def map(funktion, liste):
2      out = []
3      for element in liste:
4          neues_element = funktion(element)
5          out.append(neues_element)
6      return out
7
8  numbers = [1, 2, 3, 4, 5, 6]
9  numbers = map(lambda x: x**2, numbers)
10 print(numbers) # [1, 4, 9, 16, 25, 36]
```