

# Funktionen (fortgeschritten)

---

Dominik Pataky

27. November 2017

Python-Kurs 2017

## 1. Nutzung von Funktionen

Default Parameter

Aufruf mit Namen

## 2. Aggregatoren

Positionale Aggregatoren

Benannte Aggregatoren

Benannte und Positionale  
Aggregatoren

# Nutzung von Funktionen

---

# Funktionen als Werte

Funktionen können wie alle anderen Werte zugewiesen werden

```
1 def function(params):  
2     return 4  
3  
4 my_var = function  
5 my_var(2) # ==> 4
```

Oder als Parameter mitgegeben werden

```
1 def callif(boolean, callback):  
2     if boolean:  
3         callback()  
4  
5 callif(True, lambda: print("hello world"))
```

# Methoden sind Funktionen

```
1 class MyClass(object):  
2     def function(self, param):  
3         return 4  
4  
5 my_var = MyClass.function  
6 my_var(MyClass(), 2) # ==> 4
```

# Default Parameter

- Funktionen können vordefinierte Werte für Parameter haben.
- Parameter ohne `default`-Werten werden positionale Argumente genannt
- Parameter mit `default`-Werten werden name

```
1 def greet(name, greeting='Hello'):  
2     return '{} {}'.format(greeting, name)  
3  
4 greet('Herbert') # ==> 'Hallo Herbert'  
5 greet('Herbert', 'Guess Gott') # ==> 'Guess Gott Herbert'
```

**Niemals mutable Values (änderbare Werte) als default-Parameter verwenden!**

mutable Values `list`, `dict`, `set` und eigene Klassen  
(bzw. deren Attribute)

immutable Values `string`, `function`, `int`, `type` und `None`

## Warum?

```
1 def func(param1, param2=[]):  
2     print(param2)  
3     param2.append(param1)
```

Man denkt, die Funktion hat jedes mal eine leere List, jedoch passiert folgendes:

```
1 func(1) # ==> []  
2 func(2) # ==> [1]  
3 func('j') # ==> [1,2]
```

Die Liste wird einmalig zum Start angelegt und fortgeführt.



So sollte man es machen:

```
1
2 def func(param1, param2=None):
3     param2 = [] if param2 is None else param2
4     pass
```

Den `default`-Parameter als `None` setzen und dann, falls er `None` ist als z.B. leere Liste setzen.

# Aufruf mit Namen

Funktionsparameter können direkt mit ihrem Namen aufgerufen werden, dann spielt die Aufruffreihenfolge keine Rolle mehr.

```
1 def land(house, tree, pond):
2     return 'You own land with a {} a {} and a {}'.format(house,
3         tree, pond)
4
5 land('green house', 'maple', 'fish pond')
6
7 # oder mit Aufruf durch Namen:
8 land(house='green house', pond='fish pond', tree='maple')
9
10 # or vermischt
11 land('green house', pond='fish pond', tree='maple')
12
13 # folgendes funktioniert NICHT!
14 land('maple', house='green house', tree='maple')
```

Es gelten folgende Regeln:

- Alle Parameter können an ihrer Position angesprochen werden
- Es können auch alle mit ihrem Namen angesprochen werden
- Wenn eins mit dem Namen angesprochen wurde, müssen die folgenden ebenfalls mit Namen angesprochen werden

# Aggregatoren

---

**Aggregatoren** (auch Sammler genannt), sind sehr nützlich, wenn man, zusätzlich zu bereits definierten Parametern, in einer Funktion eine unbestimmte Anzahl an Funktionsargumenten entgegen nehmen will.

# Positionale Aggregatoren

- jede Funktion kann einen Aggregator haben
- dieser muss der *letzte* positionale Parameter sein
- Positionale Aggregatoren werden durch einen `*` gekennzeichnet
- nach einem Aggregator können nur noch benannte Parameter definiert werden, diese müssen auch mit Namen aufgerufen werden

Der Inhalt des Aggregators wird in einem *Tupel* gespeichert:

```
1 def f(*args):  
2     print(type(args)) # ==> tuple
```

# Positionale Aggregatoren - Beispiel

```
1 def function(param1, *aggr, param2=0):
2     pass
3
4
5 function(1, 2, 3, 4)    # korrekt, aggr = (2,3,4)
6 function(1, 2, 4, 5, 6, 78, 9, 90, 0)
7 # auch korrekt, aggr = (2,4,5,6,78,9,90,0)
8 function()    # inkorrekt, param1 braucht mindestens ein
9     Argument
10 function(1, param2=7)    # korrekt, aggr = ()
11 function(param2=8)    # inkorrekt, param1 braucht einen Wert
12 function(param2=0, param1=6)    # korrekt
```

# Positionale Aggregatoren

- Eine Funktion kann auch nur einen Aggregator als Parameter entgegennehmen (keine anderen Parameter)
- Ohne Argumente ergibt sich für `len(args)` 0
- Werden keine anderen Parameter erwartet, nennt man den Aggregator meist `args` (kurz für *Arguments*)

In **Python 3** kann man Aggregatoren auch ohne Namen definieren:

```
1 def funktion(param1, param2, *, param3=6):  
2     pass
```

Auf diesen Aggregator kann nicht zugegriffen werden. Er erzwingt lediglich, dass alle folgenden Parameter mit Namen aufgerufen werden.



Analog zu Parametern gibt es auch benannte Aggregatoren. Diese werden mit `**` vor dem Parameternamen definiert. Diese Aggregatoren akzeptieren lediglich benannte Parameter und sind vom Typ `dict`.

# Benannte Aggregatoren - Beispiel

```
1 def function(param1, **aggr):  
2     pass  
3  
4  
5 function(1)    # korrekt, aggr = {}  
6 function(1, some=9)    # korrekt, agar = {'some': 9}  
7 function(some=6)    # inkorrekt, param1 braucht einen Wert  
8 function(some=0, param1=8, param2=4)  
9 # korrekt, agar = {'some': 0, 'param2': 4}
```

Wenn eine Funktion keine anderen Parameter erwartet, nennt man den Aggregator meist **\*\*kwargs** (kurz für *Keyword Arguments*)

# Benannte und Positionale Aggregatoren

Beide Aggregatoren können gleichzeitig in einer Funktion verwendet werden. Die Regel dabei ist: Von jeder Sorte nur *ein* Aggregator.

Ein Beispiel:

```
1 def name(  
2     [params, ...]  
3     [, *[aggregator]]  
4     [, kwparams=kwvalue, ...]  
5     [**kwaggreg]  
6     ):  
7     pass
```

*Eckige Klammern stehen für optionale Parameter/Namen.*

# Generelle Funktionsstruktur

Wenn beide Aggregatoren zum Einsatz kommen sollen, ergibt sich folgende Funktionsstruktur:

```
1 def function(  
2     param1,  
3     param2,  
4     *args,  
5     kwparam1=0,  
6     kwparam2=None,  
7     **kwargs  
8     ):  
9     pass
```