Merkblatt - Funktionen und Methoden

September 22, 2017

1 Merkblatt: Funktionen & Methoden

1.1 Funktionen

Funktionen sind zusammengefasste Codeblöcke. Mittels Funktionen können wir es vermeiden, mehrmals verwendete Codeblöcke zu wiederholen. Wir definieren stattdessen einmal eine Funktion, die diese Codeblöcke enthält und brauchen an weiteren Stellen nur noch (kurz) die Funktion aufzurufen, ohne die in ihr enthaltenen Codezeilen zu kopieren.

1.1.1 Eine Funktion definieren und aufrufen

Wir haben schon einige Funktionen kennen gelernt, die uns Python zur Verfügung stellt. Die Funktion, die wir bislang wohl am häufigsten verwendet haben, ist die print-Funktion.

```
In [1]: print("HALLO WELT")

HALLO WELT
```

Wenn wir eine eigene Funktion verwenden wollen, müssen wir sie zuerst definieren. Eine solche Funktionsdefinition hat die allgemeine Syntax:
def Funktionname(): **Code**

```
In [4]: def multi_print():
            print("Hallo Welt!")
            print("Hallo Welt!")
```

Um eine Funktion auszuführen, die definiert wurde, schreiben wir: **Funktionname()**

```
In [5]: multi_print()

Hallo Welt!
Hallo Welt!
```

1.1.2 Funktionen mit einem Argument

Man kann Funktionen ein **Argument** übergeben, d.h. einen Wert, von dem der Code innerhalb der Funktion abhängt.

def Funktionsname(Argument): Code in dem mit dem spezifischen Argument gearbeitet wird

```
In [6]: def multi_print2(name):
            print(name)
            print(name)

        multi_print2("HALLO")
        multi_print2("WELT")
```

```
HALLO
HALLO
WELT
WELT
```

Du kannst dir einen solchen Parameter als eine zu einer Funktion gehörige Variable vorstellen. Vermeide es, einen Funktionsparameter wie eine bereits bestehende Variable zu benennen - Verwirrungsgefahr!

```
In [7]: name = "MARS"

        def multi_print2(name):
            print(name)
            print(name)

        multi_print2("HALLO")
        multi_print2("WELT")

        print(name)
```

```
HALLO
HALLO
WELT
WELT
MARS
```

Du siehst, dass der Wert der Variable *name* keinen Einfluss auf das Argument *name* der Funktion hat!

1.1.3 Weitere Funktionen in Python

Auch die len-Funktion für Listen kennst du schon. :-)

```
In [1]: print(len(["Hallo", "Welt"]))
```

```
2
```

Du kannst die len-Funktion auch auf Strings anwenden.

```
In [2]: print(len("Hallo"))
```

```
5
```

Eine Übersicht über Funktionen in Python findest du hier: https://docs.python.org/3/library/functions.html

1.1.4 Funktionen mit mehreren Argumenten

Eine Funktion darf auch mehrere Argumente enthalten.
def Funktionenname(Argument1, Argument2, ...): Code in dem mit Argument1, Argument2,... gearbeitet wird

```
In [1]: def multi_print(name, count):
            for i in range(0, count):
                print(name)

        multi_print("Hallo!", 5)
```

```
Hallo!
Hallo!
Hallo!
Hallo!
Hallo!
```

1.1.5 Funktionen in Funktionen

Funktionen können auch ineinander geschachtelt werden.

```
In [5]: def weitere_funktion():
            multi_print("Hallo!", 3)
            multi_print("Welt!", 3)

In [6]: weitere_funktion()
```

```
Hallo!
Hallo!
Hallo!
Welt!
Welt!
Welt!
```

1.1.6 Einen Wert zurückgeben

Bislang führen wir mit Funktionen einen Codeblock aus, der von Argumenten abhängen kann. Funktionen können aber auch mittels des Befehls **return** Werte zurückgeben.

```
In [2]: def return_element(name):
            return name

        print(return_element("Hi"))
```

```
Hi
```

Solche Funktionen mit return können wir dann wie Variablen behandeln.

```
In [10]: def return_with_exclamation(name):
             return name + "!"

         if return_with_exclamation("Hi") == "Hi!":
             print("Right!")
         else:
             print("Wrong.")
```

```
Right!
```

```
In [1]: def maximum(a, b):
            if a < b:
                return b
            else:
                return a

        result = maximum(4, 5)
        print(result)
```

```
5
```

2 Funktionen vs. Methoden

2.0.1 Funktionen

Bei ihrem Aufruf stehen Funktionen "für sich" und das, worauf sie sich beziehen steht ggf. als Argument in den Klammern hinter ihnen.

```
In [36]: liste = [1, 2, 3]
```

```
In [28]: print(liste)
```

```
[1, 2, 3]
```

```
In [29]: print(len(liste))
```

```
3
```

2.0.2 Methoden

Daneben kennen wir aber auch schon Befehle, die mit einem Punkt an Objekte angehängt werden. Eine Liste ist ein solches **Objekt**. Jedes Objekt hat Methoden, auf die wir zurückgreifen können. Diese Methoden können wir aber nicht auf ein Objekt eines anderen Typs anwenden (meistens zumindest).

Schauen wir uns einige nützliche Methoden des Listen-Objektes an :-) (du brauchst sie dir nicht alle merken)

```
In [37]: # ein Element anhängen
         liste.append(4)

         print(liste)
```

```
[1, 2, 3, 4]
```

```
In [39]: # ein Element an einem bestimmten Index entfernen
         liste.pop(2)
```

```
Out[39]: 2
```

```
In [ ]: # wir sehen, dass die Methode nicht die aktualisierte Liste, sondern das entfernte Eleme

In [40]: print(liste)
```

```
[1, 4, 3, 4]
```

```
In [41]:  # Ein Element an einer bestimmten Stelle einfügen
          # das erste Argument bei insert gibt an, welches Element in die Liste eingefügt wird,
          # das zweite Argument bei insert gibt an, an welcher Stelle das Element eingefügt wird;
          # beachte, dass der Index des ersten Elements in einer Liste 0 ist!
          liste.insert(1, 4)

          print(liste)

In [ ]: # ein Element entfernen
        liste.remove(4)

        print(liste)

In [21]: # den Index eines Elementes angeben (die erste Stelle, an der es vorkommt)
         print(liste.index(3))
```

```
4
```

```
In [23]: print(liste.index(4))
```

```
1
```

```
In [24]: print(liste.count(4))
```

```
3
```

```
In [19]: # mit reverse können wir die Reihenfolge einer Liste umkehren
         liste.reverse()
         print(liste)
```

```
[1, 4, 4, 4, 3]
```

```
In [ ]:
```

Merkblatt - Variable Funktionsparameter uebergeben

September 23, 2017

1 Variable Funktionsparameter

1.1 Variable Funktionsparameter entgegennehmen

Manchmal möchtest du einer Funktion erlauben, eine variable Anzahl an Parametern entgegenzunehmen.

Dafür kannst du die * - Schreibweise verwenden, die Parameter landen dann in einem Tupel. Dadurch akzeptiert diese Funktion dann eine variable Anzahl an Parametern:

```
In [1]: def calculate_max(*params):
            print(params)
            current_max = params[0]
            for item in params:
                if item > current_max:
                    current_max = item
            return current_max

        calculate_max(1, 2, 3)
```

```
(1, 2, 3)
```

```
Out[1]: 3
```

Zudem hast du die Möglichkeit, über die ** - Schreibweise mehrere, benannte Parameter entgegen zu nehmen. Diese Parameter landen dann in einem Dictionary, und du kannst aus der Funktion darauf zugreifen:

```
In [2]: def f(**args):
            print(args)

        f(key="value", key2="Value 2")
```

```
{'key': 'value', 'key2': 'Value 2'}
```

Das beides funktioniert natürlich auch kombiniert. Wichtig ist hierbei, der Parameter mit einem Sternchen (hier: `*params` muss vor dem Parameter mit zwei Sternchen stehen `**args`.

Alle normalen Parameter landen jetzt in dem Tupel `*params`, alle benannten Parameter landen im Dictionary ‘`**args`.

```
In [7]: def f(*params, **args):
            print(params)
            print(args)

        f("Ein weiterer Wert", "Noch ein Wert", key="value", key2="value2")
```

```
('Ein weiterer Wert', 'Noch ein Wert')
{'key': 'value', 'key2': 'value2'}
```

1.2 Funktion mit Variablen Funktionsparametern aufrufen

Ähnlich wie ein `*` in der Funktionsdefinition mehrere Parameter zusammengefasst hat, können wir auch mehrere Parameter quasi "entpacken".

Hier in dem Fall haben wir eine Liste `l`, und wir möchten, dass das erste Listenelement als Parameter `a` übergeben wird, und das zweite als Parameter `b`:

```
In [8]: def f(a, b):
            print(a)
            print(b)

        l = [1, 2]
        f(*l)
```

```
1
2
```

Gleiches funktioniert natürlich auch für ein Dictionary, hier brauchen wir `**`, um die Parameter zu entpacken:

```
In [10]: def f(a, b):
             print(a)
             print(b)

         l = {"a": 1, "b": 2}
         f(**l)
```

```
1
2
```

1.2.1 Warum machen wir das?

Manchmal möchten wir Parameter einfach nur "durchschleifen", also gar nicht groSS entgegen nehmen, sondern einfach an eine andere Funktion weiterleiten.

Hier im Beispiel werden also Parameter in ein Dictionary gepackt (`**plot_params`, Zeile 4), dadurch können variable Parameter übergeben werden.

Und diese Parameter werden dann in `plt.plot([1, 2, 3], [5, 6, 5], **plot_params)` wieder aus dem Dictionary entpackt, und in normale Funktionsparameter umgewandelt.

So können wir uns z.B. in einen solchen Prozess einklinken.
Beispiel:

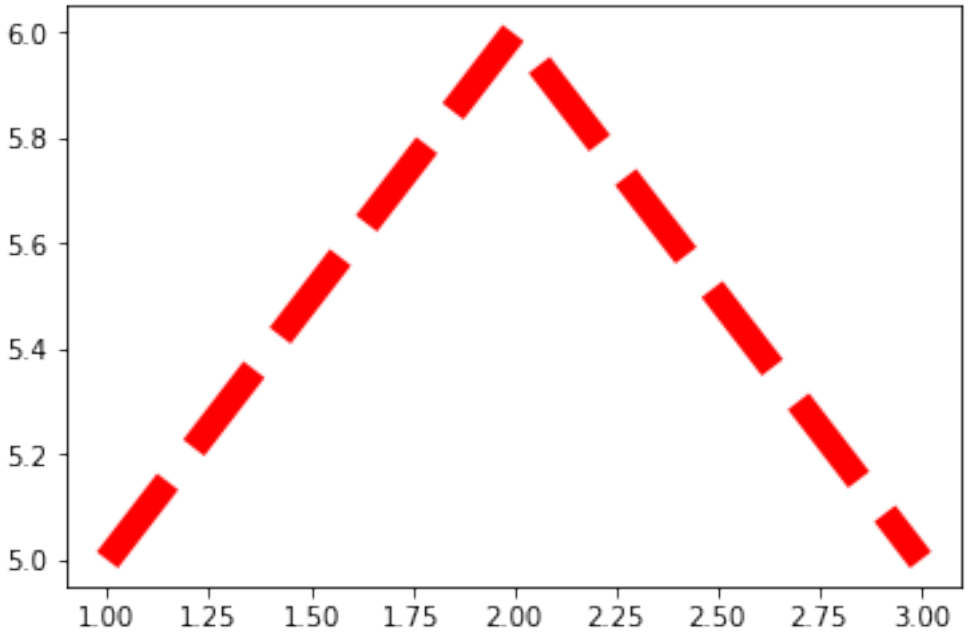
```
In [11]: %matplotlib inline
         import matplotlib.pyplot as plt

         def create_plot(**plot_params):
             print(plot_params)

             plt.plot([1, 2, 3], [5, 6, 5], **plot_params)
             plt.show()

         create_plot(color="r", linewidth=10, linestyle="dashed")
```

```
{'color': 'r', 'linewidth': 10, 'linestyle': 'dashed'}
```




```
In [ ]:
```