Funktionen

Funktionen sind kleine Programmstücke, die einen Namen haben und die über ihren Namen aufgerufen werden. Nach der Abarbeitung der Funktion wird wieder zum Aufrufer zurückgesprungen. Eine Python-Funktion (ebenso wie C/C++) hat immer ein Ergebnis:

1) Funktionen ohne expliziten Rückgabewert (z.B. `print(1.0)`) haben das Resultat `None` vom Typ `NonType`. Sie enden entweder am Ende ihres Programmblocks oder durch eine `return`–Anweisung **ohne** Wert.
2) Funktionen mit explizitem Rückgabewert haben diesen Wert als Ergebnis (z.B. `abs(1.0)`) . Sie enden durch einer `return`–Anweisung **mit** Rückgabewert.

Eingeleitet werden Funktionsdefinitionen mit dem Schlüsselwort `def`, es folgt der Funktionsname und in runden Klammern allfällige Funktionsargumente. Nach der schließenden runden Klammer `)` **muss ein Doppelpunkt folgen**. Gleich anschließend steht der Code der Funktion, **der eingerückt sein muss:**

```
def Funktionsname(Argumentliste):
     Anweisungen der Funktion
     ...

def hello():    # definiert die Funktion hello() ohne Argumente, Ergebnis None
     print("Hello World")

def hello_user(user):     #  Argument user, Ergebnis None
     print("Hello", user)

def get1():          #  Funktion ohne Argumente mit Ergebnis 1
     return 1

def sum(a,b):        #  Funktion mit genau 2 Argumenten a, b und Ergebnis a+b
     return a+b

hello()
hello_user("Reinhard Stix")
x = get1()           # 1
y = sum(1,2.1)       # 3.1
z = sum("Reinhard", "Stix")   # "ReinhardStix"
u = sum([1, 2, 3], [x y z])
               # [ 1, 2, 3, 1, 3.1, "ReinhardStix"]

falsch = sum(1, 2, 3)   # Fehler, da 3 Argumente
```

Optionale Argumente mit Defaultwert

Optionalen Argumenten (diese dürfen beim Aufruf weggelassen werden) „weist“ man in der Funktionsdefinition den Defaultwert zu. Fehlt das Argument beim Aufruf, so wird dafür der Defaultwert eingesetzt. Wird das Argument beim Aufruf angegeben, so wird dieser Wert verwendet:

```
def hello(user = "World"):     # user darf beim Aufruf fehlen
     print("Hello", user)

hello()              -> Hello World
hello("Reinhard")    -> Hello Reinhard

def sum(a = 0, b = 2):         # 2 Default-Argumente
     print("sum =",  a + b)

sum()                     -> sum = 2  ( 0 + 2 )
sum(1)                    -> sum = 3  ( 1 + 2 )
sum("Reinhard", "Stix") -> sum = "ReinhardStix"
```

Hat ein Argument einen Defaultwert, müssen auch **alle folgenden** Parameter einen Default-Wert besitzen, d.h. Argumente mit Defaultwert müssen am Ende stehen:

```
def f(a, b = 2, c = 3):        # ok: Defaultargumente stehen am Ende
def f(a = 0, b = 2, c):        # Fehler:  c hat keinen Defaultwert
```

Beliebig viele Argumente

Man kann auch Funktionen mit beliebig vielen Argumenten definieren, indem man einem Argumentnamen das Zeichen `*` voranstellt. Dieser Parameter ist dann ein Tupel mit allen übrigen Aufrufparametern:

```
def print_args(*args):      # args ist ein Tupel mit allen Argumenten
     print("args =", args)
     for i in range(len(args)):
          print("args[", i , "] = ", args[i], sep="")

print_args()  -> args = ()   leeres Tupel
print_args(1, 3.1, "Reinhard Stix")
     -> args = (1, 3.1 "Reinhard Stix")
```

```
def sum_all(first, *rest):    # first muss vorhanden sein
     print_args(first, rest)  # gib zuerst alle Argumente aus
     sum = first
     for i in rest:
          sum += i
     print ("Summe =", sum)


sum_all(3)               # first = 3, rest = ()
sum_all(3, 4, 7.1)       # first = 3, rest = (4, 7.1)
```

Aufruf mit benannten Argumenten

```
def f(a, b, c):          # die Funktion f hat 3 Argumente: a, b, c
     ...
```

```
f(1, 2, 3.5)
```

hier erfolgt die Zuordnung der Argumente aus der Position:

```
     1 -> a,  2 -> b, 3.5 -> c
```

```
f(c = 3.5, a = 1, b = 2)
```

hier erfolgt die Zuordnung der Argumente aus der Benennung (das geht in C++ nicht!)

```
     1 -> a,  2 -> b, 3.5 -> c
```

```
f(1, c = 3.5, b = 2)
```

hier erfolgt die Zuordnung der Argumente aus der Position (a) und der Benennung (b, c):

```
     1 -> a,  2 -> b, 3.5 -> c
```