Funktionen

Eine Funktion ist ein Block Code, der ausgeführt wird, wenn die Funktion aufgerufen wird. Wir haben bereits mehrere Funktionen kennen gelernt und verwendet, zum Beispiel die `print()` Funktion. Eine Funktion kann mehrere Parameter haben, die man beim Funktionsaufruf definieren kann, und ein Funktion kann Werte zurück geben.
Das ist ähnlich wie bei einer mathematischen Funktion: `y = f(x)` . In diesem Fall ist `f` die Funktion, `x` der Parameter und `y` der Rückgabewert. `y` ist abhängig von `x` , das heißt die Wahl des Parameters beeinflusst den Rückgabewert der Funktion.
Das ist beim Programmieren im Prinzip genauso. Nehmen wir wieder die `print()` Funktion als Beispiel: das Ergebnis der Funktion ist abhängig vom Parameter. Oder ein anderes Beispiel: `y = random.randint(a, b)` . `a` und `b` sind bei dieser Funktion ja die Grenzen, in denen die Zufallszahl liegt. `y` ist der Rückgabewert der Funktion; beim Aufruf der Funktion wird ein Wert erzeugt und in `y` gespeichert. Dieser Wert ist abhängig von den Parametern `a` und `b` .

Wir können Funktionen aber natürlich auch selber schreiben. Dafür benötigen wird das keyword `def` wie *definition*.

In [1]:

```
1 def hallo_welt():
2     print("Hallo Welt 123")
3
4 hallo_welt()
```

```
Hallo Welt 123
```

Die Funktion besteht aus:

- dem keyword `def`
- dem Namen der Funktion gefolgt von Klammern `hallo_welt()`
- einem Doppelpunkt `:`
- und einem eingerückten Codeblock

In Zeile 4 wird die Funktion aufgerufen. In dem Moment springt Python quasi hoch zur Funktion und führt den Code-Block aus. Wenn die Funktion abgearbeitet ist, geht es nach dem Funktionsaufruf weiter. Da unser Beispielprogramm keine Zeile 5 hat, ist es dann zu Ende.

Diese Beispielfunktion ist sehr einfach und macht nichts anderes, als einen bestimmten Text auszugeben. Prinzipiell kann eine Funktion aber beliebig kompliziert werden, je nach Bedarf. Oben haben wir von Parametern gesprochen. Diese werden in den Klammern hintern dem Funktionsnamen angegeben. Mithilfe dieser Parameter können wird Argumente an die Funktion liefern. Parameter und Argument bezeichnet in einer Funktion letzendlich dasselbe, allerdings ist der Parameter die Variable und das Argument der Wert, der in dieser Variable abgelegt wird. Deshalb definieren wird eine Funktion mit Parametern *(params)*, geben beim Aufruf der Funktion aber Argumente *(args)* an. Als Beispiel nehmen wir eine einfache Funktion, die die Summe von zwei Zahlen ausgibt:

In [5]:

```
def summe(a, b):  # a und b sind die Parameter
    c = a + b  # diese Parameter können wir in der Funktion verwenden
    print(c)

summe(3, 5)  # wir geben der Funktion Argumente
```

```
8
```

Probier das mal selber aus, indem du eine Funktion schreibst, die das Produkt von zwei Zahlen bildet und ausgibt:

In [ ]:

```
# deine Funktion hier
```

Funktionen können auch Werte zurück geben. Dafür brauchen wir das keyword `return`. Beim Aufruf der Funktion wird diese dann als ihr Rückgabewert evaluiert und kann so in eine Variable gespeichert werden. Nehmen wir als Beispiel wieder die Summe:

In [7]:

```
def summe(a, b):
    c = a + b
    return c  # der Wert in c wird zurück gegeben

y = summe(7, 5)
print(y)
```

```
12
```

Anmerkung: Wir können in diesem Code ein paar Zuweisungen einsparen. (Rechen)Operationen können auch direkt bei `return` oder als Funktionsargument evaluiert werden:

In [9]:

```
def summe(a, b):
    return a + b  # wir brauchen kein c

print(summe(7, 5))  # und kein y
```

```
12
```

Mehr Argumente

Bei der Summenfunktion haben wir zwei Parameter definiert. Daher müssen wir diese Funktion auch mit zwei Parametern aufrufen, nicht mehr und nicht weniger, ansonsten wird ein Fehler geworfen. Aber auch hier sagt uns die Fehlermeldung direkt, was wir falsch gemacht haben!

In [10]:

```
1 summe(7)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-45de3fc33e90> in <module>
----> 1 summe(7)

TypeError: summe() missing 1 required positional argument: 'b'
```

In [11]:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-60e034c994a6> in <module>
----> 1 summe(1, 2, 3)

TypeError: summe() takes 2 positional arguments but 3 were given
```

args

Eine Summe kann aber ja auch aus mehr als nur zwei Summanden bestehen. Eine Möglichkeit, dies in einer Funktion umzusetzen, ist eine Liste als Argument zu verwenden. Funktionsargumente können grundsätzlich jeder beliebiger Datentyp sein, dies muss aber natürlich in der Funktion entsprechend berücksichtigt werden:

In [12]:

```
1 def summe(summanden):
2     y = 0
3     for x in summanden:
4         y += x
5     return y
6 
7 print(summe([1, 2, 3, 4, 5]))
```

```
15
```

Python gibt uns aber auch die Möglichkeit, die Anzahl der Argumente variabel zu lassen. Wenn ein Sternchen vor den Parameter gesetzt wird, werden mehrere Argumente als Tuple in den einen Parameter gespeichert:

In [15]:

```
1 def summe(*summanden):
2     print(type(summanden))  # Datentyp von summanden
3     y = 0
4     for x in summanden:
5         y += x
6     return y
7 
8 print(summe(1, 2, 3, 4, 5))
```

```
<class 'tuple'>
15
```

Die Funktionen ist nur mit einem Parameter definiert, in Zeile 8 haben wir aber 5 Argumente an die Funktion gegeben. Diese Argumente wurden in einen Tuple geschrieben, der in der Variable `summanden` gespeichert wurde.

keyword arguments

Bei den Funktionen mit mehreren Argumenten ist es so, dass die Reihenfolge der Argumente darüber entscheidet, in welchen Parameter sie geschrieben werden. Es handelt sich also um positionelle Argumente.

In [17]:

```
1 def order(var1, var2, var3):
2     print(var1)
3     print(var2)
4     print(var3)
5 
6 order('Hallo', 42, [1, 2, 3])
```

```
Hallo
42
[1, 2, 3]
```

`'Hallo'` wurde als erstes Argument übergeben und wurde dementsprechend in den ersten Parameter `var1` gespeichert. In Python gibt es allerdings zusätzlich die Möglichkeit, Argumente über den Namen des Parameters zu übergeben. Man spricht in dem Fall von *keyword arguments (kwargs)*.

In [18]:

```
1 def order(var1, var2, var3):
2     print(var1)
3     print(var2)
4     print(var3)
5 
6 order(var3='Hallo', var1=42, var2=[1, 2, 3])
```

```
42
[1, 2, 3]
Hallo
```

In diesem Fall war `'Hallo'` wieder das erste Argument, wir haben aber festgelegt, dass es in `var3`

gespeichert wird. Keyword Argumente sind also unabhängig von der Reihenfolge. Keyword und positionelle Argumente können auch gemischt werden, dabei ist allerdings zu beachten, dass alle positionellen Argumente immer vor den keyword Argumenten zugewiesen werden müssen.

In [20]:

```
def order(var1, var2, var3):
    print(var1)
    print(var2)
    print(var3)

order('Hallo', var3=42, var2=[1, 2, 3])
```

```
Hallo
[1, 2, 3]
42
```

`'Hallo'` wurde als erstes positionelle Argument im ersten Parameter gespeichert, die darauf folgenden keyword Argumente wurden in den angegeben Parametern gespeichert.

****kwargs***

Auch bei keyword Argumenten kann es sein, dass wir eine variable Anzahl Argumente verarbeiten wollen. Dafür wird ein doppeltes Sternchen vor den Parameter gesetzt. Die übergeben Argumente werden dann als Dictionary gespeichert.

In [24]:

```
def new_student(**student):
    for x in student:
        print(x, '-', student[x])

new_student(fname='Max', lname='Mustermann', year=11)
```

```
fname - Max
lname - Mustermann
year - 11
```

Jetzt haben wir 4 verschiedene Argumenttypen kennen gelernt, die alle in einer Funktion gemeinsam verwendet werden können. Dabei ist folgende Reihenfolge zu beachten:
`function_name(args, *args, kwargs, **kwargs)`
In der Regel braucht man aber nicht alle vier Typen auf einmal.

Default-Werte für Parameter

Den Parametern einer Funktion können Default-Werte zugewiesen werden. Diese Werte werden verwendet, wenn beim Funktionsaufruf kein entsprechendes Argument übergeben wird. Dafür wird der Default-Wert in der Funktionsdefinition dem Parameter zugewiesen.

In [25]:

```
1 def new_student(fname, lname, year=11):
2     print(f"Der Schüler {fname} {lname} ist in Klasse {year}.")
3 
4 new_student('Max', 'Mustermann')
5 new_student('Peter', 'Müller', 9)
```

```
Der Schüler Max Mustermann ist in Klasse 11.
Der Schüler Peter Müller ist in Klasse 9.
```

Beim ersten Funktionsaufruf in Zeile 4 wurden nur zwei Argumente übergeben, für `year` wurde der Default-Wert `11` verwendet; beim zweiten Funktionsaufruf in Zeile 5 wurde als drittes Argument `9` übergeben, dieses Argument hat den Default-Wert für `year` überschrieben und wurde stattdessen verwendet. In diesem Beispiel wurden die Argumente positionell verwendet, das geht aber genauso mit keyword Argumenten.

In [26]:

```
1 new_student(lname='Mustermann', fname='Max', year=7)
```

```
Der Schüler Max Mustermann ist in Klasse 7.
```

Mehr Rückgabewerte

Wir haben bei Dictionaries die Funktion `.items()` kennen gelernt, die zwei Werte zurück gibt: den key und den value. Natürlich können wir auch bei eigenen Funktionen mehrere Rückgabewerte haben. Dafür müssen einfach alle Rückgabewerte mit Kommata getrennt hinter dem `return` keyword angegeben werden. Diese Rückgabewerte können beim Funktionsaufruf entweder als Tuple in eine einzelne Variable geschrieben werden, oder direkt in mehrere Variablen, die wiederum mit Kommata getrennt geschrieben werden. Dabei muss die Anzahl der Variablen mit der Anzahl der Rückgabewerte übereinstimmen. Es ist zum Beispiel nicht möglich, nur zwei von vier Rückgabewerten zuzuweisen.

In [3]:

```
1  def mr():  # Funktion mit mehreren return values
2      return 1, 2, 3, 4
3  
4  t = mr()  # alle return values als Tuple
5  print(t)
6  
7  a, b, c, d = mr()  # alle return values einzeln
8  print(a)
9  print(b)
10 print(c)
11 print(d)
```

```
(1, 2, 3, 4)
1
2
3
4
```

Übungen

Volumen eines Quaders

Schreibe eine Funktion, die das Volumen eines Quaders berechnet. Die Parameter sind Länge, Breite und Höhe des Quaders, der Rückgabewert ist das Volumen.

In [10]:

```
1 # hier deine Funktion
```

BMI

Der Body-Mass-Index berechnet sich aus der Körpergröße in Metern und dem Gewicht in Kilogramm nach

$$\mathrm{BMI} = \frac{\mathrm{Gewicht}}{\mathrm{Größe}^2} .$$

Schreibe eine Funktion, die Gewicht und Größe einer Person entgegen nimmt und den BMI zurück gibt.

In [ ]:

```
1 # deine Funktion hier
```

Kleiner Gauß

Schreibe eine Funktion, die alle Zahlen von `0` bis `x` addiert und das Ergebnis zurück gibt. `x` ist dabei ein Funktionsparameter.

In [ ]:

```
1 # deine Funktion hier
```

Module

Module sind Sammlungen von Funktionen (und Klassen). Es gibt diverse Basis-Module in Python, zum Beispiel `random` oder `math`, und es gibt diverse third-party Module, zum Beispiel `matplotlib`, die man separat via `pip install {modul}` installieren kann. Es gibt in Python für fast alles ein passendes Modul. So gibt es zum Beispiel Module, mit denen Philips Hue Lampen gesteuert werden können, oder mit denen LEGO Züge programmiert werden können. Um ein Modul zu verwenden, muss dieses mit dem `import` Statement geladen werden. Mit `as` kann dem Modul ein neuer Name zugwiesen werden, unter dem es dann angesprochen werden kann. Mit der Punktnotation kann dann auf die Funktionen des Moduls zugegriffen werden.

In [37]:

```
1 import random  # import Statement
2
3 print(random.random())  # Punktnotation um auf Funktion zuzugreifen
```

```
0.6197894503002163
```

In [38]:

```
import random as rnd  # import as, bei random aber unüblich

print(rnd.random())  # das random Modul wird jetzt mit dem Namen 'rnd' aufgerufen
```

```
0.41514254091930436
```

Mit dem `from` keyword ist es auch möglich, nur bestimmte Teile eines Moduls zu importieren. Die Syntax dafür ist: `from {modul} import {funktion}`

In [44]:

```
from random import randint  # importiere nur randint() aus dem Modul random

print(randint(1, 6))  # da wir randint() direkt importiert haben, können wir auch direk
```

```
2
```

Wir können Module auch selber erstellen. Module sind einfache Python-Dateien (.py), die die gewünschten Funktionen enthalten. Zur Verdeutlichung habe ich mit `bsp_module.py` ein Modul mit drei einfachen Funktionen erstellt, welches wir jetzt importieren und verwenden können. Vorraussetzung dafür ist, dass Python das Modul findet. Die einfachste Möglichkeit dafür ist, die Datei im selben Ordner abzulegen, in dem man aktuell programmiert. Alternativ gibt es unter Windows eine PATH-Variable, in der man einen oder mehrere Dateipfade angeben kann, in denen Python nach Modulen sucht.

In [1]:

```
import bsp_modul as bm

bm.hello_world()
print('Würfel:', bm.wuerfel())
print(bm.summe(1, 2, 3, 7, 8, 9))
```

```
Hello World!
Würfel: 6
30
```

Docstrings

Es ist immer sinnvoll, allen Code, den man schreibt, zu dokumentieren. Dies wird meistens mit Kommentaren gemacht. Für Funktionen und Module gibt es dafür Docstrings. Diese stehen ganz am Anfang von Modulen und Funktionen und werden als Multiline-String mit drei Anführungszeichen geschrieben. Diese Docstrings werden automatisch das `__doc__` Attribut einer Funktion bzw. eines Moduls und können zum Beispiel mit der `help()` Funktion ausgelesen werden.

In [2]:

```
def summe(a, b):
    """summe(a, b) -> a + b"""
    return a + b

help(summe)  # die help-Funktion gibt unter anderem den Docstring aus
print('---------------------')
print(summe.__doc__)  # der Docstring kann auch direkt aufgerufen werden
```

```
Help on function summe in module __main__:

summe(a, b)
    summe(a, b) -> a + b

---------------------
summe(a, b) -> a + b
```

Auch das Beispiel Modul, welches wir oben verwendet haben, liefert zusätzliche Informationen über den Docstring:

In [2]:

```
1 help(bm)
```

```
Help on module bsp_modul:

NAME
    bsp_modul

DESCRIPTION
    module example
    Author: Lukas Mendelsohn

    This module contains three functions. Its only use is to be an example o
f what a module might look like.

    Functions:
    hello_world(): prints "Hello World"
    wuerfel(): rolls a six sided dice
    summe(): calculates the sum of all args

FUNCTIONS
    hello_world()
        Print 'Hello World!'

    summe(*args)
        Calculate the sum of all arguments.

    wuerfel()
        Roll a digital dice.
        Returns a random value between 1 and 6.

FILE
    c:\users\micro\onedrive\studium\python kurs\4_funktionen_module\bsp_modu
l.py
```

Auch Python-Basis-Funktionen haben Docstrings. Docstrings sollten immer angeben, was die Funktion macht, und Informationen zu allen Parametern und Rückgabewerten enthalten. Für Docstrings gibt es auch einen Styleguide, der vorschreibt, wie sie aussehen sollten. Der Styleguide ist PEP 257 (https://www.python.org/dev/peps/pep-0257/).

In [5]:

```
1 help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```