PYTHON

04 – Funktionen II, Module

Funktionen

Funktionen

- enthalten Programm-Teile
- können später im Programm aufgerufen werden
- können beliebig oft aufgerufen werden (und man muss nur die Funktion aufrufen, nicht den ganzen Code neu schreiben!)

Vorteile von Funktionen

- Wiederverwendung
- Fehlerquellen minimieren
- Lesbarkeit / Verständlichkeit
- einfache Wartung

Funktionen:
Syntax

- Funktionen werden mit dem Schlüsselwort `def` gekennzeichnet, dann der Name der Funktion gefolgt von runden Klammern und Doppelpunkt
- der Programm-Code wird mit Einrückung geschrieben

```
def meineFunktion():
    … Anweisung …
```

- in den runden Klammern können Parameter übergeben werden

Funktionen:
Parameter

- Funktionen müssen nicht immer das gleiche zurückgeben
- man kann sie auch steuern, indem man ihnen Variablen (=Parameter) mitgibt
- diese übergibt man in der runden Klammer

Funktionen:
Parameter

- beim Definieren der Funktion gibt man den Parametern Namen, mit denen man die Parameter in der Funktion nutzen kann

```
def meineFunktion(zahl):
    print(zahl)
```

Funktionen:
Parameter

- beim Aufrufen der Funktion wird der entsprechende Parameter mitgegeben

```
meineFunktion(2)
```

- obiger Befehl ruft die Funktion `meineFunktion` auf und übergibt ihr den Wert 2

</> Aufgabe

`trennstrich-parameter.py`

- erweitere die Funktion `trennstrich` so, dass die Anzahl der Trennstriche als Parameter mit übergeben werden können

- `trennstrich(10)` macht 10 Striche (`----------`)
  `trennstrich(2)` macht 2 Striche (`--`)

</> Aufgabe

`func-addieren.py`

- schreibe eine Funktion `addieren`, die 2 Parameter annimmt und diese miteinander addiert
- die Funktion printed das Ergebnis direkt
- z.B. `addieren(10,5)` gibt 15 aus

Funktionen:
Rückgabewert (return)

- mit `return` können Funktionen Werte zurück geben
- anstatt also direkt zu entscheiden, was mit dem berechneten Wert passieren soll, kann die Funktion das Ergebnis auch einfach nur zurück geben
- der Rückgabewert kann dann direkt in eine Variable gespeichert werden:

```
ergebnis = addieren(10,5)
print(ergebnis)
```

</> Aufgabe

`func-addieren.py`

- schreibe die Funktion `addieren` so um, dass das Ergebnis mittels `return` zurück gegeben wird und der print-Befehl erst nach Aufrufen der Funktion erfolgt

Funktionen:
optionale Parameter

- man kann für Parameter auch einen default-Wert angeben
- diese Parameter müssen dann nicht mit übergeben werden, sind also optional

```
def meineFunktion(zahl1, zahl2=5)
```

- wird an `meineFunktion` nur <u>eine</u> Zahl mit übergeben, wird für die zweite Zahl der default-Wert genommen (in dem Fall 5)

Funktionen:
optionale Parameter

- wir kennen bereits eine (System-) Funktion mit optionalen Parametern: `range()` hat optionalen Step-Parameter
- per default ist der `step`-Wert 1 (zählt immer um 1 rauf)

```
range(start, stop)
range(start, stop, step)
```

Funktionen:
optionale Parameter

- die optionalen Parameter müssen immer am Ende der Parameter-Liste stehen!

</> Aufgabe

`func-addieren-optional.py`

- schreibe eine Funktion `addieren` mit 2 unbedingten und 2 optionalen Parametern (default-Wert 0)
- addiere die Parameter in der Funktion und gib das Ergebnis zurück

Funktionen:
Positionsparameter

- die Parameter werden der Reihe nach vergeben:

```
def meineFunktion(z1, z2)
```

```
meineFunktion(2, 4)
```

Funktionen:
Schlüsselwortparameter

- oft ist es nicht einfach, sich zu merken, an welcher Stelle welcher Parameter steht
- besonders problematisch, wenn man verschiedene Datentypen übergeben möchte

```
def schreiben(wort,anzahl,groesse,schriftart)
```

Funktionen:
Schlüsselwortparameter

- man kann beim Aufrufen angeben, welchem Parameter welcher Wert zugewiesen werden soll

```
schreiben(wort="hallo",schriftart="Arial",
groesse=12, anzahl=5)
```

</> Aufgabe

`func-schreiben.py`

- schreibe eine Funktion `schreiben` mit 2 Parametern: `wort` und `anzahl` (wie oft das Wort geschrieben werden soll)
- `schreiben("hello", 5)` gibt 5 Mal hello aus
- tausche beim Aufrufen die beiden Parameter

Funktionen:
beliebig viele Parameter

- mit einem Stern * vor dem Parameternamen wird gekennzeichnet, dass an seiner Stelle beliebig viele Parameter übergeben werden können

```
def addieren(*zahlen)
```

- `zahlen` ist dabei ein **Tupel** aus den übergebenen Parametern

Funktionen:
beliebig viele Parameter

- Beispiel mit mehreren Parametern:

```
def meineFunktion(a,b,*z)
```

- ruft man diese Funktion auf, werden die ersten beiden Werte als `a` und `b` übergeben und alle weiteren im Tupel `z`

</> Aufgabe

`func-addieren.py`

- schreibe die Funktion `addieren` so um, dass sie zwei unbedingte und beliebig viel weitere Parameter annehmen und miteinander addieren kann

- Hinweis:
    - Tupel! -> for-Schleife
    - nutze eine Variable am Anfang der Funktion, zu der du jedes Mal in der Schleife den jeweiligen Wert dazu zählst (`+=`)

Übungen

uebungen.py

</> Aufgabe 1

- nutze die Funktion `addieren` aus dem letzten Beispiel
- definiere eine Liste `liste` mit mehreren Zahlen
- addiere nun bestimmte Stellen aus der Liste miteinander (indem du sie als Parameter an die Funktion `addieren` übergibst)

</> Aufgabe 2

- definiere ein Dictionary `auto` mit bestimmten Eigenschaften (Farbe, Türen, Kraftstoff, ...)

- gib die Eigenschaften des Autos aus:

```
Türen: 5
Farbe: blau
Kraftstoff: Diesel
Gänge: 6
```

- ändere nach der Ausgabe eine Eigenschaft aus dem Dictionary! z.B. Farbe auf silber

Module

Module

- man kann Python-Code aufteilen in verschiedene Dateien (= Module)
  - z.B. Auslagerung von Funktionen
  - die Module können dann eingebunden werden, wenn sie benötigt werden

Globale Module

- stehen jedem Python-Programm zur Verfügung
- kommen von Python direkt (“Standard-Bibliothek")
- oder z.B. von Drittanbietern
- z.B. `math`

Lokale Module

- selbst geschriebene Module, auf die nur man selbst zugreifen kann
- Aufteilung des eigenen Codes in mehrere Dateien
- macht Code übersichtlicher
- werden wie globale Module eingebunden

Module

- werden eingebunden mit dem Befehl `import`
- mehrere Module können untereinander oder hintereinander mit Beistrich eingebunden werden

```
import math
import random

import math, random
```

Module

- die jeweiligen Funktionen und Variablen des Moduls können dann verwendet werden, indem sie mit Punkt an den Modulnamen angehängt werden

```
import math

math.pi
math.sin
```

Module

- um nicht jedes Mal den gesamten Modul-Namen schreiben zu müssen, kann man das Modul auch unter einem neuen Namen importieren:

```
import math as m

m.pi
m.sin
```

Module

- man kann auch nur Teile eines Moduls importieren
- ist besser, als eine Menge Funktionen usw. zu importieren, die man gar nicht braucht!

```
from math import pi
```

- importiert nur die Konstante pi
- diese wird aber direkt importiert, man kann sie also direkt als `pi` aufrufen anstatt `math.pi`!

</> Aufgabe

`math-pi.py`

- gib die Zahl pi aus, indem du einmal die Standard-Bibliothek `math` importierst, und einmal nur die Variable `pi` daraus!

</> Aufgabe

`math_functions.py & math-programm.py`

- lege ein Modul `math_functions.py` an, das die Funktionen `addieren` und `multiplizieren` enthält
- verwende die Module in der Datei `math-programm.py`

- Modul-Namen dürfen nur Buchstaben, Zahlen und Underscore (_) enthalten!