Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-09 Dozent: Karsten Flügge

Themen des Tages

- Funktionen
  - Vertiefung named args : Schlüsselwortargumente

  - First-Class Functions
  - funktional 101

Syntactic Sugar for Functions

nützlich:

- **Default Argumente**
- **pass Keyword**
- **docstrings # nützlich!**

komplex:

variadic arguments

- **k**ey**w**ord **arg**uments
- **global**

Beliebige Parameterzahl

“**var**iadic **arg**uments”. **var**iierende Anzahl von **Arg**umenten

- ***vararg / list deconstruction**

Statt einer festen Anzahl von Parameter kann eine Funktion auch eine beliebige Anzahl von Parametern erhalten z.B. print(1,2,3)

Dazu schreibt man ein Sternchen vor das Argument:

```
def polyparams(*varargs):
  for arg in varargs:
    print("Wie in Liste: ", arg)


polyparams(1, 2, 3, 4, 5) # einzelne Argumente werden wie Liste übergeben
polyparams([1, 2, 3, 4, 5]) # Achtung, Liste zählt als ein Argument
polyparams(*[1, 2, 3, 4, 5]) # "unpacking deconstruction" von Liste zählt als viele Argumente

def polyparams(*varargs:int) # kann auch Typ festgelegt werden
```

Beliebige Parameterzahl

Achtung, Listen werden bei **var**iadic **arg**uments nicht in Elemente aufgelöst, sondern als ganzes Objekt betrachtet:

```
>>> print([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
>>> print(1, 2, 3, 4, 5)
1 2 3 4 5
```

```
polyparams(1, 2, 3, 4, 5) # einzelne Argumente werden wie Liste übergeben
polyparams([1, 2, 3, 4, 5]) # Achtung, Liste zählt als ein Argument
polyparams(*[1, 2, 3, 4, 5]) # "unpacking deconstruction" von Liste zählt als viele Argumente

def polyparams(*varargs:int) # kann auch Typ festgelegt werden
```

Beliebige Parameterzahl

Paradebeispiel **print** nimmt alles entgegen (auch unverpackt)

```
def mixed_polyparams(x,y,*rest):
  print(x,y,rest)

mixed_polyparams("x","y","r","e","st")
# x y ('r', 'e', 'st')

mixed_polyparams("x","y")
rest ist optional und ergibt leeres Tupel

Geht auch hinten!
print(1,2,3,sep='-',end='\n!\n')
```

Schlüsselwort Argumente

Erinnerung **keyword** Argumente sind solche die mit Namen übergeben werden z.B. print(1,2,3,**end='!'**)

* in Argumente Liste heißt: alles hiernach darf NUR als Schlüsselwort Argument übergeben werden

def fun(x,y,*,z): pass # z *muss* als z= übergeben werden

fun(1,2, 3) # Fehler. warum? irgendwann wird es unübersichtlich
fun(1,2, z=3) # OK: z NUR als Schlüsselwort Argument, muss!

Anwendungen Beliebige Parameterzahl

Generell: **Verschönern** der Funktionsaufrufe

Paradebeispiel **print** nimmt alles entgegen (auch unverpackt)
Hauptsächlich für debug Ausgaben

```
print(1,2,3,sep='-',end='\n!\n')
```

```
def summiere(*zahlen):
  sum = 0
  for zahl in zahlen:
    sum += zahl
  return sum

print(summiere(1,2,3,4,5))
```

Beliebige Parameterzahl

Paradebeispiel **print** nimmt alles entgegen (auch unverpackt)

Geht auch hinten (im Gegensatz zu default Parametern)!

```
def mixed_polyparams_vorne(*rest,x,y):
  for arg in rest:
    print("Wie in Liste: ", arg)
mixed_polyparams_vorne(1,2,3,x="x",y="y") # x=, y= muss man übergeben (oder default)
```

```
print(1, 2, 3, sep='-',end='')  # nützlich!
# 1-2-3
```

Beliebige Parameterzahl

Benamte Argumenteübergabe ist **nicht kompatibel** mit **varargs** hinten

```
mixed_polyparams(x="x",y="y","r","e","st")
mixed_polyparams(rest=*("r","e","st"), x="x") #OK
mixed_polyparams(x="x",y="y",rest=*("r","e","st"))
mixed_polyparams(x="x",y="y",*rest=("r","e","st"))
mixed_polyparams(x="x",y="y",rest=("r","e","st"))
mixed_polyparams(x="x",y="y",rest=("r","e","st"))
```

Beliebige Schlüsselwortargumente

Beliebige Parameterzahl mit Namen

- **kwargs (keyword arguments) dict deconstruction

Statt einer festen Anzahl von Parametern kann eine Funktion auch eine beliebige Anzahl von Parametern mit Namen erhalten

Dazu schreibt man zwei Sternchen vor das Argument:

```
def polyKeywords(**kwargs): # kwargs ist jetzt ein dict!
  for key, value in kwargs.items():
    print(f"{key}: {value}")

polyKeywords(a=1, b=2, c=3, d=4) # pseudo dict
polyKeywords(**{"a":1, "b":2, "c":3}) # decomposing dict
```

Diese spezielle Syntax wandelt eine Liste von key-value Argumenten in ein echtes dict um: 'kwargs'

Beliebige Schlüsselwortargumente

Beliebige Parameterzahl mit Namen
Syntactic suggar

- ****kwargs (keyword arguments) dict deconstruction**

Statt einer festen Anzahl von Parametern kann eine Funktion auch eine beliebige Anzahl von Parametern mit Namen erhalten

Dazu schreibt man zwei Sternchen vor das Argument:

```
def polyKeywords(**kwargs): # kwargs ist jetzt ein dict!
  for key, value in kwargs.items():
    print(f"{key}: {value}")

def polyKeywords2(**kwargs):
  for key in kwargs:
    value = kwargs[key]
    print(f"{key}: {value}")

polyKeywords(a=1, b=2, c=3) # pseudo dict
polyKeywords(**{"a":1, "b":2, "c":3}) # decomposing dict
```

Diese spezielle Syntax wandelt eine Liste von key-value Argumenten in ein echtes dict um: 'kwargs'

Anwendungen beliebige benamte Parameter

Generell: **Verschönern** der Funktionsaufrufe

Paradebeispiel config **optionen** nimmt alles entgegen
Damit hält man sich offen, *später neue Optionen hinzuzufügen*.

```
def config(**optionen):
  if "font" in optionen:
    print("Schrift:", optionen["font"])
  else:
    print("Nicht implementierte optionen", optionen)

config(font="Arial", size=12)
```

Anwendungen beliebige benamte Parameter

Kombinieren von beiden:

```
def config(*varags, **kwargs):
  if "font" in kwargs:
    print("Schrift:", kwargs["font"])
  else:
    print("Nicht implementierte optionen", optionen)

config(1, 2, 3, font="Arial", size=12)
# ohne Name landet in varags
# mit Name landet in kwargs
```

Durchreichen von unbekannten Parametern.
Universalfunktion kann mit beliebigen Parametern aufgerufen werden!
Dekoratoren (z.B. code beschleunigen/optimieren, Woche 3)

- break

Themen des Tages

- First-Class Functions (Verben, Aufgaben)
- funktional 101

First-Class Functions

In Python werden Funktionen als sogenannte "First-Class Citizens" behandelt. Bürger erste Klasse

Funktionen sind Objekte

- **Zuweisung zu Variablen**

```
def quadrat(x):
  return x * x
```

quadrat OHNE KLAMMERN: kein Aufruf, sondern Referenz auf das Objekt quadrat

```
meine_funktion = quadrat # hoch_drei
print(meine_funktion(5))  # Gibt 25 aus
```

First-Class Functions

In Python werden Funktionen als sogenannte "First-Class Citizens" behandelt. Bürger erste Klasse

Funktionen sind Objekte

- **Zuweisung zu Variablen**

```
def quadrat(x):
   return x * x
quadrat OHNE KLAMMERN: kein Aufruf, sondern Referenz auf das Objekt
quadrat
meine_funktion = quadrat # hoch_drei
print(meine_funktion(5))  # Gibt 25 aus
```

- ***Funktionen als Argumente***

```
def ausfuehren(funktion, argument):
   return funktion(argument)

print(ausfuehren(quadrat, 5))  # Gibt 25 aus
```

- ***Rückgabe von Funktion*** *(als Wert)*

```
def erzeuge_addierer(x):
   def addierer(y):
      return x + y
   return addierer

add_fuenf = erzeuge_addierer(5)
print(add_fuenf(3))  # Gibt 8 aus
```

First-Class Functions

Aktionen "Verben" Funktionen als Objekte

def singen(): print("lalala")

```
>>> singen
<function singen at 0x106b8c880>

>>> type(singen)
<class 'function'>

>>> singen()  # klammern zum Ausführen der Aktion!
lalala
```

First-Class Functions

Aktionen "Verben" Funktionen als Objekte lassen sich **in Variablen schreiben**

def singen(): print("lalala")

```
>>> lieblings_taetigkeit=singen   # Variable : Zeiger, Referenz, Name, Alias, Shortcut für singen
>>> lieblings_taetigkeit
<function singen at 0x106b8c880>
>>> lieblings_taetigkeit()
lalala
```

```
Der Name vom Ursprünglichen Verb bleibt erhalten.
```

First-Class Functions

Aktionen "Verben" Funktionen als Objekte lassen sich als Argumente / Parameter an Funktionen übergeben.

Man kann den Aktionsvariablen wieder Argumente mitgeben

```
def singen(lied): print("lalala", lied)

lieblings_taetigkeit=singen
lieblings_taetigkeit("Fiderallala")
```

```
lalala Fiderallala
```

First-Class Functions

Aktionen "Verben" Funktionen als Objekte lassen sich als Argumente / Parameter an **Funktionen übergeben.**

```
def singen(): print("lalala")
def tanzen(): print("hops")
def lachen(): print("haha")

def feier(aktionen):
  for act in aktionen:
    act()

nachmittags_programm = [singen, lachen, tanzen]
feier(nachmittags_programm)
```

First-Class Functions

Aktionen "**Verben**" Funktionen als Objekte lassen sich als Argumente / Parameter an Funktionen übergeben.

Man kann den Aktionsvariablen wieder Argumente mitgeben

```
def singen(name): print("lalala", name)
def tanzen(name): print("hops", name)
def lachen(name): print("haha", name)
```

```
def feier(aktionen, gaeste):
  for gast in gaeste:
    for act in aktionen:
      act(gast) # << Argument wie immer wird zu 'name' Parameter

gaeste = ["Jens", "Peter","Maria"]
nachmittags_programm = [singen, lachen, tanzen]
feier(nachmittags_programm, gaeste)
```

First-Class Functions

Aktionen "Verben" Funktionen als Objekte

```
def singen(): print("lalala")

singen() # klammern zum Ausführen der Aktion

nachmittags_programm=[singen, tanzen, lachen]

def feier(aktionen):
  for act in aktion:
     act()

feier(nachmittags_programm)

def verzögerte_aktion(aktion):
   sleep(10)
   aktion()
```

Beispiele

- **callbacks:** eine Funktion wird verzögert von einer anderen aufgerufen '**call**'
- Eine Funktion informiert die Umgebung über eine Statusänderung

```
def eigene_aufgabe():
  print("WORK!")

def wenn_ich_fertig_bin(fertig:function): # nimmt beliebige Funktion entgegen
  eigene_aufgabe()
  fertig() # juhu wird aufgerufen

def juhu(): print("Juhu")

wenn_ich_fertig_bin(juhu) # wird zum Parameter 'fertig'
"WORK!"
"Juhu"


# andere callbacks: ready, error, progress, done ... feedback pattern Muster
```

Beispiele Funktionen in Listen

- **Dynamische Abfolgen, Pläne, Todos, ...** imperativ
- deklarativ/dict {"aufgabe":[speichern, loeschen], "Wichtigkeit":10} "als Text / als Daten" html

queue: was soll als nächstes abgearbeitet werden

priority queue: was soll als nächstes abgearbeitet werden, je nach *Wichtigkeit*

topologic sorting: sortiere Aufgaben nach *Abhängigkeiten* (Milch kaufen vor Kaffee kochen vor servieren)

(Nachmittagsprogramm war nicht *komplett* ausgedacht)

- **deklarativ vs imperativ**

  - **<u>deklarativ</u> (was)** abstrakt**:**

  <**button** onclick="send()">Senden! <**button**/> # html **Strukturell** (als Text / Daten)

  {"botton":{"text":"Senden!", "action":"send()"}} # json/dict rein **Strukturell** (als **Text**)

  • Mittel:

  view={"botton":{"text":"Senden!", "action":send}} # json/dict **Strukturell** (als Daten + **Code**)

  - **imperativ (wie)** konkret

  Button button = new Button(); # Layout als **Code**, teils überspezifisch

  button.text="Senden!"

  button.action=send

Beispiele

- **Erzeugen von Funktionen**

```
def mache_aktion()→funktion:
  zufall = random.randint(1,10)
  def quadrat(x):return zufall*x*x
  return quadrat
```

Anwendungen:

Flexibilität:
Konfigurieren
Experimente
Factories

Woche 3 : dekoratoren

```
@optimize # schnappt sich Funktion und liefert die optimierte Funktion zurück GPU?
def berechne_stoemung()
  v[i]=i*i# ...
```

Beispiele

- **callbacks**
- Eine Funktion informiert die Umgebung über eine Statusänderung

```
def eigene_aufgabe():
  print("WORK!")

def wenn_ich_fertig_bin(fertig:function, progress:function, error:function):
  # bereit() # jetzt kann es losgehen
  eigene_aufgabe()
  progress("50% Fertig") # typischerweise: Anzeigen für Benutzer
  eigene_aufgabe()
  fertig() # juhu wird aufgerufen

def juhu(): print("Juhu")

wenn_ich_fertig_bin(juhu) # wird zum Parameter 'fertig'
"WORK!"
"Juhu"

# andere callbacks: ready, error, progress ...  pattern Muster
```

duck typing

duck typing

In python kann man beliebige Parameter angeben, der Typ wird erst bei der Ausführung geprüft

```
def wenn_ich_fertig_bin(fertig):
  eigene_aufgaben()
  fertig()

wenn_ich_fertig_bin(123)  # erst jetzt gibt es den Fehler:
TypeError: 'int' object is not callable  123() macht keinen Sinn


DAFÜR sind Typen gut:
def wenn_ich_fertig_bin(fertig:function)
besonders bei komplexen / lang laufenden Programmen!
```

duck typing (Nachtrag)

“If it walks like a duck and quacks like a duck, treat it as a duck.”

„Wenn es watschelt wie eine Ente und quakt wie eine Ente, behandle es wie eine Ente.“

Objekte werden durch ihr **Verhalten** oder durch ihre **Fähigkeiten** bestimmt (Aspekte je nach Situation)

In python kann man beliebige Parameter angeben, der Typ wird erst bei der Ausführung geprüft

```
def addiere(a, b:Addable):
  a+b  # a und b unterstützen '+'? dann addiere halt. sonst Fehler bei Ausführung.

addiere(2,3) # 5
addiere("2","3") # "23"

>>> addiere({1:'a',2:'b'},{3:'c'})
    a+b  # a und b unterstützen '+'? dann addiere halt. sonst Fehler.
    ~^~
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'


Eigene Klassen müssen einfach __add__ und damit den + Operator implementieren
```

Modern: Traits ≈ Interface ≈ Protocol

WAS kann mein Typ / Objekt (egal *wie*)

Objekte werden durch ihr **Verhalten** oder durch ihre **Fähigkeiten** bestimmt (Aspekte je nach Situation)

Modern: man kann diese **Fähigkeiten als Typ** definieren

```
def addiere(a:Addierbar, b):
  a+b  # a (und b) unterstützen '+' dann addiere halt. sonst Fehler bei Ausführung.

addiere(2,3) # 5
addiere("2","3") # "23"
```

```
from typing import Protocol, TypeVar

R = TypeVar("R", covariant=True) # Result type

class Addable(Protocol[R]):
    def __add__(self, other: object, /) → R: ...

def addiere(a: Addable[R], b: object) → R:
    return a + b
```

Aufgaben 'Verben'

a) speichere eine Funktion in Variable und rufe sie damit auf

```
lieblings_aktion=print;lieblings_aktion('yay')
```

b) implementiere das Nachmittagsprogramm Beispiel [singen, tanzen, lachen]

```
def singen():print("lala")   # ...
```

c)

- Schreibe zwei Funktionen die ein int bekommen und liefern (z.B. x^3)
- Stecke diese Funktionen in eine Liste
- Rufe jede Funktion in der Liste auf

```
def quad(x):return x*x
def cube(x):return x**3
actions=[quad,cube]
actions[0]()
actions[1]()
```

Auch in dictionaries:

```
>>> funcs={"quadriere":quad,"hoch drei":cube}
>>> funcs["quadriere"](3)  # 27
```

x

```
def apply(liste, func):
  return list(map(liste, func))


def apply(liste, func):
    result = []
    for item in liste:
        result.append(func(item))
    return result
```

Funktional 101

Da man in python **Funktionen als Parameter** übergeben kann, haben sich einige sehr mächtige standard Funktionen etabliert, um diese auf Listen anzuwenden:

- **map ≈ apply** wendet eine Funktion auf alle Elemente einer Liste einzeln an um diese **umzuwandeln**

```
def square(x):
  return x * x

# Using map() to square numbers
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))
assert squared_numbers == [1, 4, 9, 16, 25] # überprüfe, sonst Fehler und Abbruch
```

Funktional 101

Da man in python **Funktionen als Parameter** übergeben kann, haben sich einige sehr mächtige standard Funktionen etabliert, um diese auf Listen anzuwenden:

- **map ≈ apply** wendet eine Funktion auf alle Elemente einer Liste einzeln an um diese **umzuwandeln**
- **filter ≈ suche** wendet eine Funktion als **Kriterium** auf alle Elemente einer Liste einzeln an um Elemente zu **sammeln (filtern)**

```
def square(x):
  return x * x

def is_even(x):
  return x % 2 == 0

# Using map() to square numbers
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))
assert squared_numbers == [1, 4, 9, 16, 25] # überprüfe, sonst Fehler und Abbruch
```

Funktional 101

Da man in python **Funktionen als Parameter** übergeben kann, haben sich einige sehr mächtige standard Funktionen etabliert, um diese auf Listen anzuwenden:

.filter ≈ suche ≈ select wendet eine Funktion als **Kriterium** auf alle Elemente einer Liste einzeln an um Elemente zu **sammeln (filtern)**

```
def is_even(x):
  return x % 2 == 0
```

```
# Using filter() to filter even numbers
numbers = [1, 2, 3, 4, 5]
numbers.select(even) # <> SQL: select * from numbers where is_even()
numbers.which(is_even)
even_numbers = list(filter(is_even, numbers))
assert even_numbers == [2, 4]
```

Leider muss man das Resultat in Python noch mal extra mit list() in eine Liste umwandeln 😡

Funktional 101

Da man in python **Funktionen als Parameter** übergeben kann, haben sich einige sehr mächtige standard Funktionen etabliert, um diese auf Listen anzuwenden:

- **map ≈ apply** wendet eine Funktion auf alle Elemente einer Liste einzeln an um diese **umzuwandeln**
- **filter ≈ suche** wendet eine Funktion als **Kriterium** auf alle Elemente einer Liste einzeln an um Elemente zu **sammeln (filtern)**

```python
def square(x):
   return x * x

def is_even(x):
   return x % 2 == 0

# Using map() to square numbers
numbers = [0, 1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))
assert squared_numbers == [1, 4, 9, 16, 25]

# equivalent, machen wir später:
neue_liste = []
for n in numbers:
 neue_liste.append(f(n))

for n in numbers:
  yield f(n)


# Using filter() to filter even numbers
even_numbers = list(filter(is_even, numbers))
assert even_numbers == [2, 4]
```

Leider muss man das Resultat in Python noch mal extra mit list() in eine Liste umwandeln 😡

Funktional 101

map liefert eine optimierten 'Iterator' den man Schritt für Schritt ausführt

```
for name in map(großschreibung, names):
  print(name)
```

```
>>> map(is_even,[1,2,3,4,5])
<map object at 0x106c42e40>
```

Um alle Berechnungen auf einmal zu Sehen mit **list()** holen:

```
>>>>>> list(map(is_even,[1,2,3,4,5]))
[False, True, False, True, False]
```

ähnlich:

```
>>> reversed([1,2,3,4,5])
<list_reverseiterator object at 0x106b6cc70>
>>> list(reversed([1,2,3,4,5]))
[5, 4, 3, 2, 1]
```

Funktional 101

Da man in python **Funktionen als Parameter** übergeben kann, haben sich einige sehr mächtige standard Funktionen etabliert, um diese auf Listen anzuwenden:

- **reduce (fold, combine) "faltet" eine Liste zu einem Ergebnis zusammen (z.B. Summe, Produkt)**

```
from functools import reduce # Teil von Python aber muss importiert werden im Gegensatz zu map, filter

def multiply(x, y):
  return x * y

# Using reduce() to compute the product of elements in a list
product = reduce(multiply, [1, 2, 3, 4, 5]) # (((1 * 2) * 3) * 4) * 5
```

Bei reduce muss in der Rückgabewert der applizierten Funktion mit dem Typ in der Liste überein stimmen

Experten können mit diesen Handgriffen sehr eleganten und sauberen Code schreiben,
ohne irgendwelche imperativen for i … schleifen!

Python versus Java

was kann python besser als zB Java?

- Python ist viel ergonomischer, also **mit weniger Zeichen kann man mehr erreichen**
- module kann man einfach mit pip + import hinzuladen, viele sind von haus aus dabei (ohne pip)
- module sind (viel) besser abgekapselt: klare Funktionalität (siehe help())
- Funktionen sind 1st class citizen (argument teils veraltet)
- Data ist 1st class citizen: dict!
- Operator overloading: man kann auf eigenen Objekten '+' '*' … definieren. Gut für Matrizen!

```
class MyScript{
public static void main(String[] arg){
 // java ist sehr verbose (braucht viele Zeichen)
}
}

a = [1,2,3]
List myList = new List(1,2,3,4); # verbose
```

map

```
map ≈ apply

Daten Transformieren sehr allgemeiner, häufiger use-case

map(fun, liste)
map(square,[1,2,3,4])

# square all numbers from one to four.

# List comprehension
[square(n) for n in [1,2,3,4]]   # {n^2: n in [1,2,3,4]}

(square(n) for n in [1,2,3,4] if n%0)
```

Kürzer ist besser?

Too Verbose, **wortreich:**
Assembly ≈ Maschinensprache, Java, C#, English (teil)

Goldener Mittelweg:
Python

Teils (viel) besser und **kürzer:**
English, vor allem auf abstrakter Ebene, als Anweisung "Baue Strömungssimulation"

Zu kurz?
Mathe? Man investiert in Notation, **lesbarer** wenn man es verstanden hat, wird alles einfacher.

APL "write only"

```
 life ← {⊃1 ⍵ ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ⌽¨ ⊂⍵}
```

ein Genie hat es geschrieben, aber keine kann es lesen

Kürzer ist besser? nur bis zu einem gewissen (Genie) grad

lambda

```
def square(x): return x*x

map(fun, liste)
map(square, [1,2,3,4])

[square(n) for n in [1,2,3,4] ]   # { n^2 : n ∈ {1,2,3,4} }  für das gilt
```

Um **kleine Funtionen** nicht vorher definieren zu müssen,
kann man sie auch **ad-hoc** an der Stelle der Verwendung definieren.

Spezielles keyword **lambda "sofort Funktion ohne Namen"** ≈
Und spezielle Syntax **lambda variable: action** erzeugt namenlose Funktion. ≈ def

```
[n*n for n in [1,2,3,4]] # geht bei map nicht, weil kein n bekannt:
map(square,[1,2,3,4])  # oder? ...
map(lambda n: n*n , [1,2,3,4])   # (n) Parameter Klammern, return kann man sich sparen

gibt auch mehr-parameter lambdas "anonyme Funktionen"  # PS  n => n*n
```

lambda

Anwendungen:

Immer wenn das Schreiben einer ganzen Funktion Overkill wäre, z.B.:

```
sort(list,key=criterion)
```

```
sorted(words, key=lambda x: -len(x))
```

```
# Wert von lambda muss nur wieder sortierbar sein: int, str, date, ...    <  __lt__
```

duck typing: sorted akzeptiert alles, was < unterstützt

broadcasting ≈ übertragen

```
Automatisches Anwenden einer Funktion auf alle Elemente in einem Vektor
def square(x): return x*x

square([1,2,3,4]) # geht erst mal nicht ABER:
```

Mann muss die Funktion extra so gestallten, dass sie **Elemente** und **Listen** akzeptiert

In Numerischen / Machine Learning Bibliotheken ist das eingebaut.

Dazu erweitert man dann das List-Objekt zum eigenen e.g. **numpy, pytorch *tensoren***

```
numpy.sin(3)  # Element OK
numpy.sin([3,4,5]) # Liste auch OK
```

Aufgaben Funktional

a) Erzeugt eine Liste mit fünf Zahlen und erzeugt daraus eine neue Liste,
   in der alle Elemente **um eins erhöht** sind (*ohne* *for* Schleife)
b) map, lambda, list-comprehension: Nutze die beiden Ansätze, die in a noch nicht verwendet wurden.
c) suche alle Elemente in Liste die durch 2 teilbar sind (filter oder list-comprehension mit if)
d)

Funktionen Aufgaben 1

- a) Implementiere eigene Summen Funktion für Liste

- b) Implementiere eigene Maximum Funktion für Liste von Argumenten
  - (entweder funktional oder über *vararg)

- c) Erstelle **Volumen** Funktion mit default Werten 1(m)
- Rufe Funktion mit verschiedenen Reihenfolgen von länge breite höhe auf.

- d) Muss die Anzahl von Argumenten und Parametern immer die selbe sein?

Funktionen Aufgaben 2

```
# 1. Schreibe eine Funktion, die ein default Argument hat
# und dieses Argument in der Funktion ausgibt.

# 2. Schreibe eine Funktion, die prüft, ob ein default Argument
verändert übergeben wurde.

# 3. Schreibe eine Funktion, die versucht ein übergebenes Argument
(string) zu verändern.

Extra (nur für vararg Experten)
# 4. Schreibe eine Funktion, die beliebig viele Argumente
entgegennimmt und das grösste zurückgibt.

# 5. Schreibe eine Funktion, die beliebig viele benamte Argumente
entgegennimmt (nur für kwarg Experten)
#  und 42 zurückgibt, wenn ein Argument "sinn" heisst, sonst 0
```

Operatoren

Vorrang von Operatoren, Gegebenenfalls Operationen Klammern oder in mehrere Zeilen teilen.

'Präzedenz' / Priorisierung / Bindung / Reinfolge (Gruppierung)

```
(...)                # Klammern
x[i], x(...), x.y  # Zugriff, Aufruf  z.B. .real
**                   # Potenz
+x, -x, ~x         # Vorzeichen, bitweise Negation
*, /, //, %        # ÷ Punktrechnung   3 * True == 3
+, -                 # Strichrechnung
<<, >>               # Bitverschiebung
&                     # bitweises UND
^                     # bitweises XOR
|                    # bitweises ODER
== != < <= > >=          # Vergleiche
not                    # logisches NICHT  True, False
and                    # logisches UND
or                    # logisches ODER
=  :=  += ...       # Zuweisung; kein Ausdruck
```

Aufgaben Operator Priorisierung

Wiederholung: IM KOPF: Was ist das Ergebnis der folgenden Berechnungen in Python und in der realen Welt?

17 % 5 + 2 * 3

25 // 4 + 3 ** 2

30 // 4 % 3

2 + 18 // 4 * 3

7 + 20 % 6 // 2

50 // 6 % 4 + 1

3 * 7 % 5 + 2

100 // 9 // 2

5 + 2 ** 3 % 3

8 % 3 * 4 // 2

9 + 24 // 5 % 2

15 % 4 + 6 // 2 ** 2

Aufgaben Operator Priorisierung

10 // 3 > 3 and 8 % 3 == 2

5 + 2 * 3 == 11 or 8 // 3 == 3

not 20 % 4 == 1

7 % 4 + 1 >= 4 and 9 // 2 < 5

3 ** 2 == 9 and 12 % 5 != 1

18 // 4 == 4 or 3 * 3 == 8

2 + 3 * 4 > 10 and not 5 % 2 == 0

8 // 2 ** 2 == 2 and 7 % 4 < 3

15 % 6 == 3 or 20 // 5 > 10

Lösungen Operator Priorisierung

```
10 // 3 > 3 and 8 % 3 == 2         # False
10 // (3 > 3) and 8 % 3 == 2       # ZeroDivisionError

5 + 2 * 3 == 11 or 8 // 3 == 3     # True
5 + 2 * (3 == 11) or 8 // 3 == 3   # 5

not 20 % 4 == 1                    # True
(not 20) % 4 == 1                  # False

7 % 4 + 1 >= 4 and 9 // 2 < 5      # True
7 % (4 + 1) >= 4 and 9 // 2 < 5    # False

3 ** 2 == 9 and 12 % 5 != 1        # True
3 ** (2 == 9) and 12 % 5 != 1      # False

18 // 4 == 4 or 3 * 3 == 8         # True
18 // (4 == 4) or 3 * 3 == 8       # 18

2 + 3 * 4 > 10 and not 5 % 2 == 0  # True
(2 + 3) * 4 > 10 and not 5 % 2 == 0 # True

8 // 2 ** 2 == 2 and 7 % 4 < 3     # False
(8 // 2) ** 2 == 2 and 7 % 4 < 3   # False

15 % 6 == 3 or 20 // 5 > 10        # True
15 % (6 == 3 or 20) // 5 > 10      # False
```

Vokabeln

Vokabeln

duck typing: prüfe Typen später:
bei **Ausführung** "runtime" Laufzeit
Fehler bei Ausführung "**runtime error**" versus:
Syntax Fehler vor der Ausführung

functionale Programmierung : Funktionen als Objekte

Extra:
First-Class Citizens" behandelt. Bürger erste Klasse 'fest in Sprache eingebaut'

varargs variable
keyword args
map, reduce, lambda