Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-09 Dozent: Karsten Flügge

Themen des Tages

- **Funktionen**
  - Argumente ≈ Parameter
  - named args : Schlüsselwortargumente
  - default Standardwerte
  - call by value / call by reference

Themen des Tages

- Funktionen
  - Argumente ≈ Parameter
  - named args : Schlüsselwortargumente
  - default Standardwerte
  - call by value / call by reference
  - Beliebige Anzahl von Parametern (*vararg / list deconstruction)
  - Beliebige Anzahl von Parametern mit Namen **kwargs keyword arguments
  - kombiniert
  - """Docstrings"""
  - functional

Funktionen in Python

Funktionen haben die Funktion, Aktionen auszuführen
oder das Ergebnis von Berechnungen zurück zu liefern.

Beispiele die wir schon kennen:
print("hi") # gibt Text aus
len("hi") # gibt Länge **aus**
input("frage?") # gibt Benutzereingabe **zurück**

Funktionen die zu Objekten gehören nennt man auch **Methoden!**
"max muster"**.**title()

- 💡 **Methoden** werden immer mit Punkt *hinter dem Objekt* aufgerufen
- 💡 **Funktionen** werden immer vor dem Objekt *ohne Punkt* aufgerufen

Funktionen Beispiele

Beispiele die wir schon kennen:

print("hi") # gibt Text aus Aktion
len("hi")   # gibt Länge **aus** Information
input("frage?") # gibt Benutzereingabe **zurück Daten Sammeln**
open(file)
range(100) # Zahlen Intervall
type(x) #
float("2") => 2.0 # Typumwandlung "constructor" int("7") => 7   str(7)=>"7
sort**ed**(x) # sortierte Liste versus liste**.sort()** !   siehe reverse(d)
min() max()
exit() # Aktionen

liste**.**append(1) # Spezielle Funktion: **Methode**

help() ? eingebaute Sonderfunktion?
def, continue, **break keyword ohne () keine Funktion!**

Syntactic Suggar

- **Multi return a,b oder als Liste return [a, b]**
- **a, b = b, a**

```
def teile_name_in_vor_und_nachname(name):
  vorname =  name.split(" ")[0]
  nachname =  name.split(" ")[1]
  return vorname, nachname # multiple return values

beides = teile_name_in_vor_und_nachname("Max Mustermann")
print("Beide zusammen: ",beides) # ("Max", "Mustermann")
vorname, nachname = teile_name_in_vor_und_nachname("Max Mustermann")
print("Vorname:",vorname)
```

Unpacking oder **Entpacken** von iterierbaren Objekten.

```
>>> liste = [1,2,3]
>>> a,b,c = liste  #
```

Funktionen in Python

Funktionen haben die Funktion, Aktionen auszuführen oder das Ergebnis von Berechnungen zurück zu liefern.

Beispiele die wir schon kennen:
print("hi") # gibt Text aus
help("hi") # gibt Hilfe aus
dir("hi") # gibt Methoden aus
len("hi") # gibt Länge **aus** <built-in function len>
input("frage?") # gibt Benutzereingabe **zurück**

Funktionen die zu Objekten gehören nennt man auch **Methoden!**
"max muster"**.**capitalize()
[1,2,3]**.**append(4)
[1,2,3]**.**pop() # letztes element loeschen UND

`in` **operator** bildet auf __contains__() Methode ab

- 💡 **Methoden** werden immer mit Punkt *hinter dem Objekt* aufgerufen, 7.add(9)
- 💡 **Funktionen** werden immer vor dem Objekt *ohne Punkt* aufgerufen add(7,9) procedure, routine synonym
- 💡 **Prozedur** Funktion die eine Aktion ausführt aber nichts zurück liefert (verändernd)
- 💡 **Operatoren** sind wie Funktionen die man ohne Klammer aufrufen kann und die in der Mitte stehen 7 **+** 9 => (7).__add__(9)

`attribute  "aufrufbar" "callable" oder nicht`

Eigene Funktionen

```
def f(x):   # was wird definiert? 'f'
      print(x)
      return x*x  # quadriere x
```

Unterschiede zu Mathematischen Funktionen **$f(x)=x^2$**

- **def :** "**def**inition" hieran erkennt man Funktions-Definitionen
- **return** *notwendig wenn* etwas zurück geliefert werden soll
- python Funktionen **müssen nichts erhalten oder zurück liefern**:
- def sayHallo(): print("Hi") =>
- Funktionen in python können **Nebeneffekte** haben, also das Programm oder das System ausserhalb der Funktion beeinflussen.

- Funktionen in python müssen **nicht deterministisch** sein, dass heisst sie können zufällig verlaufen und zufällige Werte zurück liefern!

- : statt =

Eigene Funktionen

Funktion von Funktionen:

- **Berechnen**
- **Gruppieren** (bei mehrfachen Aufrufen)
- **Abkapselung** (Sichtbarkeit: verstecke bestimmte Aspekte/Variablen vom Rest des Programmes)

```
def hallo_hallo():
    print("hallo")
    print("hallo")
```

Eigene Funktionen

- Oft führt eine Funktion entweder eine **Aktion** aus oder eine **Berechnung**
    - aber idealer weise nicht beides:

```
def drucke_titel(x):
  print(f"Titel ist {x.title()}")  # liefert nichts (None) zurück
drucke_titel("max muster")

def formatiere_titel(x):
  return f"Titel ist {x.title()}"
formatiert=formatiere_titel("max muster")
print(formatiert)
print(formatiere_titel("max muster"))
```

💡 der Aufruf von Funktionen welche etwas berechnen aber keine Aktion ausführen macht man dann Sinn wenn man die berechneten Wert **weiter verwendet** also zum Beispiel in einer **Variablen gespeichert** und **oder ausdruckt:**

⚠️ es kann sein dass man einen Fall vergisst und die Funktion am Ende implizit **None** zurück liefert! Daher am Ende stets den/einen **default return** Wert liefern!

Eigene Funktionen Aufrufen

Nach dem Definieren einer Funktion muss man sie **Aufrufen**

```
def hallo():
  print "hallo"

hallo()  # Aufruf mit Namen und runder Klammer

def drucke_titel(x):
  print(f"Titel ist {x.title()}")  # liefert nichts (None) zurück

drucke_titel("max muster")

def formatiere_titel(x):
  return f"Titel ist {x.title()}"
print(formatiere_titel("max muster"))
```

Funktionen Körpern

Alles was nach dem Doppelpunkt kommt nennt man

body / block

Da darf alle rein, was wir schon als legalen python code kennen

```
def positiv_oder_negativ(x):
  if x>0:
    return "positiv"
   elif x<0:
     return "negativ"

Oft Einrückungs'hierarchie'
```

Funktionen Vertiefung

- Eine Funktion kann **mehrere** alternative **return** statements haben, es kann aber **nur eines erreicht werden** (ähnlich break)

```
def positiv_oder_negativ(x): # body / block
  if x>0:
    return "positiv"
  elif x<0:
    return "negativ"
  # else ... ?
  # return "huch, Fall vergessen"
  # nichts also None
```

⚠️ es kann sein dass man einen Fall vergisst und die Funktion am Ende implizit **None** zurück liefert! Daher am Ende stets den/einen **default return** Wert liefern!

```
positiv_oder_negativ(0) !?
positiv_oder_negativ("0") !?
```

Funktionen Vertiefung

- Eine python Funktion kann theoretisch mehrere **return Typen** haben, dies ist aber in der Regel weder sinnvoll noch guter Stil:

```
def chaos(x):
  if x>0:
    return rand() #int
  elif x<0:
    return "string"
   return {"anderer Typ":"erlaubt!"}

type(chaos(1)) # <class 'float'>
type(chaos(-1)) # <class 'str'>
type(chaos(0)) # <class 'dict'>
```

Funktionen Vertiefung

Funktionen in python müssen nicht deterministisch sein und können bei jedem Aufruf etwas anderes zurückliefern oder machen.

```
>>> from random import randint
>>> randint(0,10)  # Liefert bei jedem Aufruf eine zufällige Zahl zwischen null und zehn.

>>> from random import random
>>> random()
0.07064828561109959  # Zufälliger float Wert zwischen null und eins.
```

deterministisch = immer das selbe Ergebnis

PS: Pseudozufall dank NSA

Kleine Aufgaben

Schreibe kleine Funktion
a) zum Ausgeben von drei mal "Hallo"
b) zum Berechnen von 3+3 (ausgeben oder liefern)
c) zum Berechnen von "kubus" x3 (mit return)**
d) zum Berechnen von Kehrwert 1/x (gegebenenfalls Sonderfälle behandeln)
e) zum Einlesen von Benutzer input und Ausgabe mit Begrüßung

```
def addiere_dreien():
  return 3+3

print(addiere_dreien())  # oder
ergebnis = addiere_dreien()
print(ergebnis)
```

Parameter und Argumente

f(x) / f(2)

Parameter und Argument sind eng verwandte Begriffe und **oft synonym**

Parameter ist der erhaltene Wert / Variable (hier: p)

Argument ist der übergebene Wert / Variable (hier: a)

```
def beispiel(p):  # Funktionsvariable ≈ Parameter
  print("Parameter x wurde dieser Wert übergeben:")
  print(p*p)

beispiel(2) # 2 ist das Argument welches für die Funktion zum Parameter wird

a = 42
beispiel(a) # a ist das Argument welches für die Funktion zum Parameter x=42 wird
```

Parameter und Argumente

f(x) / f(2)
Argument und Parameter sind eng verwandte Begriffe und **oft synonym**

Argument ist der übergebene Wert / Variable
Parameter ist der erhaltene Wert / Variable

```
def beispiel(p):  # Funktionsvariable
  print("Beispiel Parameter x wurde dieser Wert als Argument übergeben:")
  print(p)

beispiel(2) # 2 ist das Argument welches für die Funktion zum Parameter wird

a = 42
beispiel(a) # a ist das Argument welches für die Funktion zum Parameter x=42 wird
```

⚠️ beim Aufruf werden **nur Werte (Objekte oder Referenzen)** übergeben,
also weiss beispiel Funktion nicht, dass der Name der **Argument-Variable 'a'** lautete.
⚠️ Ebenso weiss a nicht, dass dessen Wert nun als Parameter in beispiel Funktion unter dem Namen 'x' weiter verwendet wird

Mehrere Parameter

Man kann problemlos Funktionen mit mehreren Parametern definieren:

f(x,y)=x*y

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

```
def begruessung(vorname, nachname):
  print(f"Hallo {vorname} {nachname}")

begruessung("Max","Mustermann")
```

Parameter verschiedener Methoden

```
x = 7
def f(x): return x*x
def g(x): return x*x*x
```

Mehrere Funktionen können die selben Parameter Namen verwenden *ohne sich zu beeinflussen*!

'*Scope*'
Die Funktionsvariable x ist nur innerhalb von der Funktion *sichtbar*

'Überblendung' shadowing / hiding:
Eine Variable ausserhalb einer Funktion ist nicht sichtbar wenn eine Funktionsvariable(Parameter) den selben Namen hat.

Parameter Sichtbarkeit

```
def f(x, y): return x*y
print(f(2, 3)) # 6
print(x)
# Ausserhalb der Funktion steht der Parameter x NICHT zur Verfügung
```

'Überblendung' shadowing / hiding:
Eine Variable ausserhalb einer Funktion ist nicht sichtbar wenn eine Funktionsvariable(Parameter) den selben Namen hat.
x = 8
f(2,3) x Parameter (x=2) blendet altes x=8 aus!

Variablen, Parameter und Argumente

Kleine Verwirrung: Funktionen können globale Variablen lesen.

x = 7 # Variable ausserhalb der Funktion!

def beispiel(**parameter**): # Funktionsvariable
  parameter = 8 # x bleibt unverändert! Funktionen kapseln (fast) alles ab
  print(**x**) # kann ich trotzdem *Lesen* ⚠️ schlechter Stil!
  print("Beispiel parameter wurde dieser Wert als Argument übergeben:")
  print(parameter)

beispiel(x) # x ist das **Argument** welches für die Funktion zum **Parameter 42** wird

Faustregel: Verwende nur die Funktions-**Parameter** und nicht andere Variablen
Wenn man etwas von außen braucht, dann bitte als **Parameter**

Variablen, Parameter und Argumente

Funktionen können globale Variablen NICHT* schreiben

```
x = 7
def beispiel(parameter):  # Funktionsvariable
  x=42   # Hat KEINEN Einfluss auf x ausserhalb ( eigenes x )
  print(x) # 42
  print("Beispiel parameter wurde dieser Wert als Argument übergeben:")
  print(parameter)

beispiel(x) # x ist das Argument welches für die Funktion zum Parameter 42 wird
print(x) # x ist immer noch 7
```

*💡 nur mit extra keyword '**global**'

Functions Rekursion

Funktionen können auch andere Funktionen aufrufen, inklusive sich selbst!

```
def go_deeper(x: int = 0):
  print(x)
  go_deeper(x + 1)

go_deeper()
```

Rekursion ≈ Selbstaufruf

Anwendungen:
  Mathematisches Zeugs: **Fibonacci**, Turm von Babel, **divide and conquer** Algorithmen
  Sortieren
  Tree-Walking z.B. Directory Festplatten Verzeichnisse durchsuchen:

```
    suche_file("C:\\","wo_ist.exe") => alle Unterverzeichnisse
    suche_file("C:\\Programme","wo_ist.exe")
    for folder in current_folder: # alle Unterordner durchgehen
      suche_file(folder, "wo_ist.exe")
```

Zu Rekursionsschleifen gehört eigentlich immer eine **Ausstiegs-Prüfung**

Ent-Rekursivierung:
Theoretisch lässt sich jede Rekursion in eine (sauberere) **Schleife** umschreiben

Functions Rekursion

```
def Fibonacci(n):
  if n == 0: # Abbruchbedingung
    return 0
  elif n == 1:
    return 1
  else:
    return Fibonacci(n-1) + Fibonacci(n-2)
# 0 1 1 2 3 5 8 13 21 34 55 89 144

def geometrie(n):    # hier x=2
  if n <= 0: # Abbruchbedingung
    return 1
  else:
    return geometrie(n-1) * 2

x^y = x*(x^(y-1))  x^0 = 1   oder x^1 = x  zum 'terminieren'
print(geometrie(64))
print(2**64)
```

Functions Kaskaden

Funktionen können auch andere Funktionen aufrufen "Delegation"

```
def get_temperature():
  return 42 # Placeholder # TODO!

def show_temp(): # erstmal nur definiert, kein automatischer Aufruf
  temperature = get_temperature()
  print("Temperatur ist ",temperature)

show_temp() # Aufruf nicht vergessen!
```

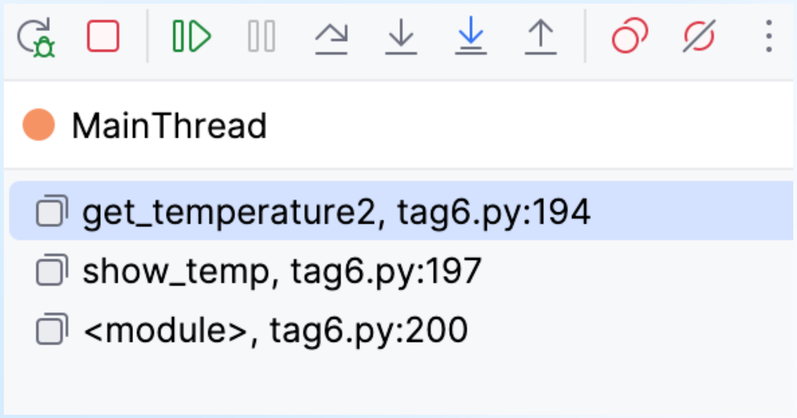
Functionen pausieren / debuggen

 <<< mit dem Käfer kann man ein Programm 'de**bug**gen'

```
def get_temperature():
  return 42 # Placeholder # TODO!

def show_temp():
  temperature = get_temperature()  # 🔴 breakpoint: mit Maus auf Zeilennummer + click
  print("Temperatur ist ",temperature)
```

Aufgabe: breakpoint in show_temp setzen und in get_temperature rein und raus steppen



Call by value / Call by reference

In python muss man sich meist keine Gedanken darüber machen, wie die Objekte im Speicher behandelt werden. Am Beispiel von Listen haben wir aber schon gesehen, dass man manchmal aufpassen muss, ob das **Original**, eine **Kopie** oder ein identisches Objekt vor sich hat:

a = [1, 2]
b = a # sogenannte **Referenz** auf selbes Objekt, zwei **Namen** fürs selbe Objekt
c **=** [1, 2] # neue Liste, ähnlich, equivalent, aber nicht identisch
a **is** b # True, geprüft: identisches Objekt
a **==** b # True, geprüft: selber **Inhalt / Wert**
a == c # True, geprüft: selber **Inhalt / Wert**
a is c # False NICHT IDENTISCH

Primitive Typen werden immer **kopiert**: int, float, bool alle **'identisch'**
Unveränderliche Typen: string, tuple (nicht veränderbar) alle **'identisch'**
Komplexe Typen werden immer **referenziert**: list, dict, **eigene**
Dazu später mehr wenn wir **Klassen** haben: normalerweise Veränderlich: Instanzen **referenziert**
Tradeoff: Kopie vs Effizienz

Call by value / Call by reference

Variablen können generell nicht verändert werden, aber manchmal deren Inhalt

Funktionen können veränderliche Objekte verändern

Funktionen können unveränderliche Objekte nicht verändern

```
def change_list(liste):
   liste.pop()

liste = [1,2,3]
change_list(liste)
liste == [1,2]
```

Call by value / Call by reference

Variablen können generell nicht verändert werden, aber manchmal deren Inhalt

Funktionen können veränderliche Objekte verändern
Funktionen können auch veränderliche Objekte NICHT per *Zuweisung* ändern!

```
def change_list(liste):
    liste.pop()
    liste = [1,2,3,4] # KEIN EFFEKT NACH AUSSEN!

liste = [1,2,3]
change_list(liste)
liste == [1,2]
```

Call by value / Call by reference

Variablen können generell nicht verändert werden, aber manchmal deren Inhalt

Funktionen können veränderliche Objekte verändern

Funktionen können unveränderliche Objekte nicht verändern

wir können aber natürlich von einem unveränderliche Objekt eine veränderte KOPIE zurückliefern

Call by value / Call by reference

Funktionen können veränderliche Objekte verändern

Funktionen können unveränderliche Objekte nicht verändern

Was sind unveränderliche Objekte?

- Literale: Zahlen, Bools (True, False)
- Strings! (man kann natürlich neue Strings erzeugen die so ähnlich sind)
- Tupel
- … Tag 10

Was sind veränderliche Objekte?

- Dictionaries
- Listen!
- ALLES ANDERE andere Objekte (Klassen Tag 8)

Call by value / Call by reference

Den selben Unterschied muss man sich bei Funktionsaufrufen vor Augen führen:

```
a = [1,2]
def modify_list(liste): liste.pop() # impure!
modify_list(a)
a == [1]
```

Dazu sagt man, dass modify_list mit einer **Referenz** auf die Liste in a aufgerufen wurde.

💡 wenn man nicht möchte, dass eine Funktion ein Objekt modifiziert, sollte man **sicherheitshalber manuell eine Kopie übergeben** (z.B. a**.copy()** oder a**[:]**)

Pure Funktionen

"pure" bedeutet im wesentlichen mathematisch, **berechnend**, **deterministisch**, oder **ohne *Nebenwirkungen***

f(x)=x^2

def f(x): return **x*x** # pur! => nicht modifizierend

"**impure**" heisst mit **Nebenwirkungen**, die das Programm oder das System ausserhalb der Funktion beeinflussen

Nebenwirkungen heisst: **System modifizierend,**
z.B. print open file globale variablen schreiben!

Spezialfall **"modifizierend"** heisst, dass das **Argument verändert** wird

Leider gibt es keine universell akzeptierte Art, pure (oder unreine) Funktionen als solche zu kennzeichnen.

Idealerweise: **kein return** = modifizierend reverse vs revers**ed** sort vs sort**ed**

Pure Funktionen

"pure" bedeutet im wesentlichen mathematisch, **berechnend**, **deterministisch**, oder ohne **Nebenwirkungen**

f(x)=x^2

def f(x): return x*x # pur! => nicht modifizierend

"**impure**" heisst mit **Nebenwirkungen**, die das Programm oder das System ausserhalb der Funktion beeinflussen

GUTER STIL: Schreibe möglichst nur pure Funktionen.
Nach Beendigung des Buches: KEIN PRINT/INPUT mehr in Funktionen!
print innerhalb einer Funktion nur für 'debuggen' oder für Übersicht

Pure Funktionen

"pure" bedeutet im wesentlichen mathematisch, **berechnend**, **deterministisch**, oder ohne **Nebenwirkungen**

f(x)=x^2 "mathematische Notation wie in der Schule, KEIN code"

def f(x): **return** x*x # pur! => nicht modifizierend ... oder?

Beispiele:

```
def addiere(zahl1: int, zahl2: int) -> int:
  return zahl1 + zahl2
# hier ist es PUR, weil wir wissen dass __add__ für int fest definiert ist

def addiere_irgendwas(zahl1, zahl2):
  return zahl1 + zahl2 # ggf impure, weil wir später __add__ überschreiben könnten
```

Function body

- Jede Funktion braucht einen sogenannten "Körper" / body

Das ist genau was die Funktion tun soll. Abfolge von Aktionen (Ausdrücke / Statements)

```
def f():
    body
```

Ein body/block *darf nicht leer sein* (auch bei Kontrollstrukturen)

- pass # Keyword: tue nichts

```
pass # nop no operation KEIN EFFEKT einfach Platzhalter  EINZIGER ZWECK:  leere Blöcke  else: pass
wann nutzt man das? Wenn beim Auskommentieren nichts übrig bleibt
```

Blöcke body

- **Jeder Block braucht einen sogenannten "Körper" / body**

Blöcke erkennt man daran dass sie einem **Doppelpunkt** folgen.

block ist eine Abfolge (mit newline) von Anweisungen / Statements

for i in liste:

block

```
if True:
  pass  # nützlich wenn man die Aktion auskommentiert hat:
  # print("OK")
```

block darf nicht leer sein, kann aber 'pass' sein.

pass "passe" = tue nichts.

Empty block : pass

Ein body/block *MUSS etwas enthalten* (auch bei Kontrollstrukturen)

wenn man nicht weiss was, dann

- **pass # Keyword: tue nichts**

```
pass # nop no operation KEIN EFFEKT einfach Platzhalter
EINZIGER ZWECK:  leere Blöcke  else: pass

wann nutzt man das?
• Wenn beim Auskommentieren nichts übrig bleibt
• Wenn man noch nicht weiss was rein soll / andere Sachen zuerst macht
    • oft zusammen mit TODO Kommentaren (in vielen IDEs unterstützt)
```

Function body

return statt break
Schleifen kann man mit return keyword direkt verlassen

```
def leave_x_y():
  for x in range(5):
    # if break_outer:
    #  break
    for y in range(5):
      print(x, y)
      if x == 2 and y == 2:
        # break_outer = True
        # break
        return # verlasse beide Schleifen und die Funktion

leave_x_y()
```

Blöcke Sichtbarkeit von Variablen

Schleifen Variablen sind nur innerhalb der Schleife sichtbar

```
for i in [1,2,3]:
  i += 1
print(i) # bleibt sichtbar
```

Variablen innerhalb eines Funktionskörpers

sind nur innerhalb der Funktion sichtbar:

```
def t():
  zzz = 7
t()
print(zzz) # NameError: name 'zzz' is not defined
```

Blöcke Sichtbarkeit von Variablen

Achtung,

python Sichtbarkeit ausserhalb von Blöcken ist anders als in anderen Sprachen:

```
def prüfe_sichtbarkeit():
  if True:
    x = 42
  print(x) # 42 ist sichtbar!

prüfe_sichtbarkeit()
print(x) # 42 ist NICHT sichtbar!
```

Blöcke Sichtbarkeit von Variablen

Schlechte Praxis: globale Variablen

```
global zzz
def t():
  global zzz
  zzz = 7   # NEBENEFFEKT!

t()
print(zzz)
```

Besser: nutze **return** um interne Werte einer Funktion nach aussen 'sichtbar' zu machen.

Aufgaben

Einfache Aufgaben

1. Schreibe eine Funktion `add(a, b)`, die zwei Zahlen addiert.
2. Schreibe eine Funktion `maximum(a, b)`, die die größere Zahl zurückgibt.
3. Schreibe eine Funktion `rechteck_flaeche(breite, hoehe)`.
4. Schreibe eine Funktion `rechteck_umfang(breite, hoehe)`.
5. Schreibe eine Funktion `mittelwert(a, b, c)`.
6. Schreibe eine Funktion `sekunden(stunden, minuten, sekunden)`, die alles in Sekunden umrechnet.
7. Schreibe eine Funktion `tausche(a, b)`, die die Werte vertauscht zurückgibt.
8. Schreibe eine Funktion `ist_teilbar(a, b)`, die prüft, ob `a` durch `b` teilbar ist.
9. Schreibe eine Funktion film(titel, jahr), die z.B. "Matrix (1999)" erzeugt.

Aufgaben

Mittlere Aufgaben

1. Schreibe eine Funktion `max3(a, b, c)`, die die größte von drei Zahlen zurückgibt.
2. Schreibe eine Funktion `sortiere2(a, b)`, die zwei Zahlen aufsteigend zurückgibt.
3. Schreibe eine Funktion `zwischen(x, a, b)`, die prüft, ob `x` zwischen `a` und `b` liegt.
4. Schreibe eine Funktion `tempo(strecke, zeit)`, die die Durchschnittsgeschwindigkeit berechnet.
5. Schreibe eine Funktion `rabatt(preis, prozent)`.
6. Schreibe eine Funktion `note(punkte, maximum)`, die den Prozentwert berechnet.
7. Schreibe eine Funktion `kreis(radius, pi)`, die die Kreisfläche berechnet.
8. Schreibe eine Funktion `potenz(basis, exponent)` ohne `**`.
9. Schreibe eine Funktion `tage(wochen, tage)` und rechne alles in Tage um.
10. Schreibe eine Funktion `uhrzeit(stunden, minuten)` und gib die Gesamtminuten zurück.
11. Schreibe eine Funktion `groesste_differenz(a, b, c)`.
12. Schreibe eine Funktion `preis_mit_steuer(preis, steuersatz)`.
13. Schreibe eine Funktion schach(feld_x, feld_y), die prüft, ob ein Feld schwarz oder weiß ist.

Aufgaben

Schwerere Aufgaben

1. Schreibe eine Funktion `gueltiges_dreieck(a, b, c)`, die prüft, ob drei Seiten ein Dreieck bilden können.
2. Schreibe eine Funktion `dreieck_typ(a, b, c)`:
   - gleichseitig
   - gleichschenklig
   - ungleichseitig
3. Schreibe eine Funktion `ggt(a, b)` für den größten gemeinsamen Teiler.
4. Schreibe eine Funktion `kgv(a, b)` für das kleinste gemeinsame Vielfache.
5. Schreibe eine Funktion `zeit_diff(h1, m1, h2, m2)`.
6. Schreibe eine Funktion `punkte(x1, y1, x2, y2)` für die Entfernung zweier Punkte.
7. Schreibe eine Funktion `lineare_funktion(m, b, x)` für y = mx + b.
8. Schreibe eine Funktion `zins(kapital, zinssatz, jahre)` für einfachen Zins.

Vokabeln

Vokabeln

Funktion
Methode
Prozedur
pur
Block
Funktionskörper / **body**
Funktionsvariable(n)
Parameter(liste)
Argument
Rekursion
Breakpoint
Thread (gibt erst man nur das Hauptprogramm)
Stack **Trace** / Back Trace / Traceback