lektion2

October 21, 2026

Inhalt

1 Kopieren (Wiederholung)

2 Kopieren II (Wiederholung)

3 Operationen auf Listen

4 Boolsche Variablen und Operationen

5 Vergleiche

6 Verzweigungen, If Anweisungen

7 Formatierte Strings

8 Funktionen

9 Rekursionen

9.1 erstes Beispiel

9.2 zweites Beispiel

1 Lektion 2

1.1 Kopieren (Wiederholung)

Wir weisen a den Wert 1 zu

[25]:
```
a = 1
a
```

[25]: 1

Zuweisung von b

[2]:
```
b = a
b
```

[2]: 1

id gibt die Identität aus

[3]:
```
id(a), id(b)  # a und b haben dieselbe Identität
```

[3]: (93978680354952, 93978680354952)

hier wird a ein neuer Wert zugewiesen

[6]:
```
a = 2
a
```

[6]: 2

b bleibt

[7]:
```
b
```

[7]: 1

a ist jetzt neu

[9]:
```
id(a), id(b)
```

[9]: (93978680354984, 93978680354952)

1.2 Kopieren II (Wiederholung)

Erzeuge Liste al

[16]:
```
al = [1, 2, [3, 4]]
```

bl wird al zugewiesen

[17]:
```
bl = al
```

Jetzt verändern wir in bl die 3 in der Unterliste [3, 4] und die 2.

[24]:
```
bl[2][0] = "Hallo"
bl[1] = 22
bl
```

[24]: [1, 22, ['Hallo', 4]]

Dadurch wurde auch al verändert, weil bl **keine** Kopie von al ist, sondern ein Name für das gleiche Listenobjekt.

[19]:
```
al
```

[19]: [1, 22, ['Hallo', 4]]

Hier wird eine Kopie von al erstellt

[26]:
```
cl = al[:]
cl
```

[26]: [1, 22, ['Hallo', 4]]

Jetzt ersetzen wir die 1 in al durch 42

[27]:
```
cl[0] = 42
cl
```

[27]:
```
[42, 22, ['Hallo', 4]]
```

Ist das ok?

[28]:
```
al
```

[28]:
```
[1, 22, ['Hallo', 4]]
```

Klappt das?

[29]:
```
cl[2][0] = 2018
cl
```

[29]:
```
[42, 22, [2018, 4]]
```

Warum?

[24]:
```
al
```

[24]:
```
[1, 22, [2018, 4]]
```

Hier brauchen wir deepcopy, eine tiefe Kopie, siehe Übungsaufgabe 5.

1.3 Operationen auf Listen

max, min, len, sorted, ...

[38]:
```
zahlen = [1, 2, 3, 1, 6, 5, 6]
```

[39]:
```
max(zahlen)
```

[39]:
```
6
```

[40]:
```
min(zahlen)
```

[40]:
```
1
```

[41]:
```
len(zahlen)
```

[41]:
```
7
```

[42]:
```
sorted(zahlen)
```

[42]:
```
[1, 1, 2, 3, 5, 6, 6]
```

Listen bringen ihre eigenen Funktionen mit, z.B. append, remove und pop.

[ ]:
```
zahlen.
```

[30]:
```
zahlen.append(4)  # anfügen
zahlen
```

[30]:
```
[1, 2, 3, 1, 6, 5, 6, 4]
```

[31]:
```
zahlen.remove(6)  # entfernt den ersten Eintrag mit Wert 6
zahlen
```

[31]:
```
[1, 2, 3, 1, 5, 6, 4]
```

[32]:
```
zahlen.pop(1)    # entfernt das Element mit Index 1 und gibt es zurück
```

[32]:
```
2
```

[33]:
```
zahlen.pop(-1)   # entfernt das letzte Element und gibt es zurück
```

[33]:
```
4
```

[34]:
```
zahlen
```

[34]:
```
[1, 3, 1, 5, 6]
```

1.4 Boolsche Variablen und Operationen

Eine boolesche Variable kann zwei Werte annehmen:

`True` oder `False`

Boolesche Variablen werden verwendet, um Bedingungen zu überprüfen und innerhalb eines Programms zu entscheiden, wie es weitergeht.

[43]:
```
True
```

[43]:
```
True
```

[44]:
```
False
```

[44]:
```
False
```

Die drei wichtigsten Operatoren sind

`and` (und) `or` (oder) und `not` (nicht)

“und” Verknüpfung

[62]:
```
True and True
```

[62]:
```
True
```

[63]:
```
True and False, False and True, False and False
```

[63]:
```
(False, False, False)
```

[65]:
```
True & True  # bitweises und
```

[65]:
```
True
```

"oder" Verknüpfung

[66]:
```
True or False  # logisches oder
```

[66]:
```
True
```

[67]:
```
True | False  # bitweises oder
```

[67]:
```
True
```

Verneinung (Negation)

[51]:
```
not True  # Negation
```

[51]:
```
False
```

1.5 Vergleiche

[56]:
```
2 <= 2
```

[56]:
```
True
```

[59]:
```
2 < 2
```

[59]:
```
False
```

Gleichheit

[60]:
```
1 == 2
```

[60]:
```
False
```

[61]:
```
1 + 1 == 2
```

[61]:
```
True
```

das geht nicht nur zwischen Zahlen

[ ]:
```
'a' == 'a'
```

[68]:
```
[1, 2] == [2, 3]
```

[68]:
```
False
```

ungleich

[69]:
```
1 != 2
```

[69]:
```
True
```

1.6 Verzweigungen, If Anweisungen

Bedingte Anweisungen (If-Anweisungen) dienen dazu, den Programmfluss zu verzweigen. Damit kann man, während das Programm ausgeführt wird, entscheiden, ob bestimmte Programmteile ausgeführt werden oder nicht.

```
if bedingung1:
    anweisung1
```

Ist bedingung1 wahr wird anweisung1 ausgeführt.

```
if bedingung1:
    anweisung1
else:
    anweisung2
```

Ist bedingung1 wahr wird anweisung1 ausgeführt, sonst anweisung2.

```
if bedingung1:
    anweisung1
elif bedingung2:
    anweisung2
```

Ist bedingung1 wahr wird anweisung1 ausgeführt. Ist bedingung1 falsch aber bedingung2 wahr wird anweisung2 ausgeführt. elifs dürfen beliebig viele vorkommen und am Ende auch ein else.

[85]:
```
import numpy as np
rng = np.random.default_rng() # Zufallszahlengenerator (random number generator)
c = rng.integers(1, 7) # integers(1,7) "würfelt" zufällig Zahl zwische 1 und 7␣
 ↪(exklusive)
c
```

[85]:
```
np.int64(1)
```

[122]:
```
c = rng.integers(1, 7)
print(f'c = {c} zu Anfang')

if c > 2:
    c = -c

print(f'c = {c} zu Ende')
```

```
c = 1 zu Anfang
c = 1 zu Ende
```

[123]:
```
a = []
if not a:
    print('a ist eine leere Liste')
```

```
a ist eine leere Liste
```

Eine leere Liste hat den Wahrheitswert `False`

1.7 Formatierte Strings

Formatierte Strings (f-Strings) sind die einfachste und schnellste Methode, um Variablen und Ausdrücke direkt in einen Text einzubauen.

Um einen f-String zu erstellen, setzt schreibt man den Buchstaben f oder F direkt vor die Anführungszeichen des Strings. Variablen oder Ausdrücke, die in den Text eingefügen werden sollen, schreibt man in geschweifte Klammern {}.

[184]:
```
name = "Anna"
alter = 25

# Ein normaler f-String
text = f"Hallo, ich heiße {name} und bin {alter} Jahre alt."
print(text)
```

```
Hallo, ich heiße Anna und bin 25 Jahre alt.
```

[190]:
```
x = 10
y = 25

print(f" x + y = {x+y}")
```

```
 x + y = 35
```

1.8 Funktionen

Funktionen fassen Programmteile zusammen, um Code übersichtlicher und weniger fehleranfällig zu machen. Ein Kernmerkmal ist die Rückgabe von Werten an das aufrufende Programm über die return-Anweisung. Sobald diese erreicht wird, verlässt das Programm die Funktion und liefert das nachfolgende Objekt zurück. Fehlt eine return-Anweisung, wird automatisch None zurückgegeben.

```
def funktionsname(parameter):
    anweisungen
```

```
def funktionsname(parameter):
    etwas = wird berechnet
    return etwas
```

In Python beginnen Funktionen mit dem Schlüsselwort def, gefolgt vom Funktionsnamen und optionalen Parametern in Klammern. Ein Doppelpunkt schließt die Zeile ab; alle folgenden Anweisungen werden eingerückt.

[153]:
```
def mypar(x):
    return x**2 + x + 1
```

[154]:
```
mypar(2)
```

[154]: 7

[155]:
```
def myparn(x):
    erg = x**2 + x + n
    return erg
```

[156]:
```
n = 1/2
myparn(2)
```

[156]: 6.5

besser

[167]:
```
def myparn(x, n=1/2):
    """ Berechnet x**2 + x + n. Falls n nicht gegeben ist, wird x**2 + x + 1/2␣
 ↪berechnet """
    erg = x**2 + x + n
    return erg
```

[166]:
```
?myparn
```

[170]:
```
ydefault, y = myparn(2), myparn(2, 3)
ydefault, y
```

[170]: (6.5, 9)

Achtung !

[175]:
```
def f(a, L=[]):
    print(f'Liste in der Funktion {L}')
    L.append(a)
    return L
```

[176]:
```
L = 23
f(1)
```

```
Liste in der Funktion []
```

[176]: [1]

[ ]:
```
print(f'L ausserhalb {L}')
```

[ ]:
```
f([1, 2])
```

[ ]:
```
f([1, 2, 3])
```

[182]:
```
def f(a, L=[]):
    print(f'Liste in der Funktion {L}')
    L.append(a)
    return L
```

[183]:
```
# Der default Wert wird nur einmal ausgewertet und L ist hier veränderlich
print(f'Rückgabe f(1) = {f(1)}')
print(f'Rückgabe f(2) = {f(2)}')
print(f'Rückgabe f(3) = {f(3)}')
print(f'Rückgabe f(4,[1,2,3]) = {f(4,[1,2,3])}')
```

```
Liste in der Funktion []
Rückgabe f(1) = [1]
Liste in der Funktion [1]
Rückgabe f(2) = [1, 2]
Liste in der Funktion [1, 2]
Rückgabe f(3) = [1, 2, 3]
Liste in der Funktion [1, 2, 3]
Rückgabe f(4,[1,2,3]) = [1, 2, 3, 4]
```

Besser initialisiert man mit `None` statt mit einem veränderlichen Objekt wie einer leeren Liste.

[ ]:
```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

[ ]:
```
print(f(1))
print(f(2))
print(f(3))
print(f(4, [2, 3, 1]))
```

1.9 Rekursionen

1.9.1 erstes Beispiel

Rekursive Funktionen sind Funktionen, die sich selbst aufrufen. Ein einfaches Beispiel ist die Fakultätsfunktion

$$n! = n(n-1)! \quad \text{und} \quad 0! = 1$$

[206]:
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

[207]:
```
factorial(4)
```

[207]:
```
24
```

Rekursive Funktionen bestehen aus einer Rekursionsbasis, d.h. einer Beschreibung des kleinsten Falls ($0! = 1$), und einem rekursivem Aufruf

$$n! = n \cdot (n-1)!$$

[208]:
```
def factorial(n):
    print(n)
    if n == 0:
        print(f'Basis {n}')
        return 1
    print(f'berechne {n} * {n-1}!')
    return n * factorial(n - 1)
```

[205]:
```
factorial(4)
```

```
4
berechne 4 * 3!
3
berechne 3 * 2!
2
berechne 2 * 1!
1
berechne 1 * 0!
0
Basis 0
```

[205]:
```
24
```

[210]:
```
def fakultaet(n):
    return 1 if n==0 else n*factorial(n-1)
fakultaet(5)
```

[210]:
```
120
```

1.9.2 zweites Beispiel

Berechne rekursiv die Summe der Einträge einer Liste. (Das kann sum eigentlich schon.)

- Basisfall: $\sum_{j=0}^{-1} a_j = 0$
- Für nichtnegative ganze Zahlen n ist

$$\sum_{j=0}^{n} a_j = a_n + \sum_{j=0}^{n-1} a_j$$

[220]:
```
def rek_sum(seq):
    if not seq:
        return 0
    elem = seq.pop()
    assert isinstance(
        elem, (int, float, complex)
    ), f'Die Liste darf nur Zahlen enthalten aber {elem} ist {type(elem)}'
    return elem + rek_sum(seq)
```

[221]:
```
seq1 = [1, 2., 1j]
rek_sum(seq1), sum(seq1)
```

[221]:
```
((3+1j), 0)
```

[222]:
```
seq1 = [1, 2., 1j]
seq2 = seq1.copy()
rek_sum(seq1), sum(seq2)
```

[222]:
```
((3+1j), (3+1j))
```

Eingabeüberprüfung

```
assert Bedingung, Fehlermeldung
```

Wirft einen Fehler, einen sogenannten AssertionError, falls Bedingung nicht erfüllt ist.

[223]:
```
seq2 = ['a', 1, 2]
rek_sum(seq2)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[223], line 2
      1 seq2 = ['a',1,2]
----> 2 rek_sum(seq2),

Cell In[220], line 8, in rek_sum(seq)
      4 elem = seq.pop()
      5 assert isinstance(
      6     elem, (int, float, complex)
      7 ), f'Die Liste darf nur Zahlen enthalten aber {elem} ist {type(elem)}'
----> 8 return elem + rek_sum(seq)

Cell In[220], line 8, in rek_sum(seq)
      4 elem = seq.pop()
      5 assert isinstance(
      6     elem, (int, float, complex)
      7 ), f'Die Liste darf nur Zahlen enthalten aber {elem} ist {type(elem)}'
----> 8 return elem + rek_sum(seq)
```

```
Cell In[220], line 5, in rek_sum(seq)
      3     return 0
      4 elem = seq.pop()
----> 5 assert isinstance(
      6     elem, (int, float, complex)
      7 ), f'Die Liste darf nur Zahlen enthalten aber {elem} ist {type(elem)}'
      8 return elem + rek_sum(seq)

AssertionError: Die Liste darf nur Zahlen enthalten aber a ist <class 'str'>
```

[227]:
```
seq2 = ['a', 1, 2]
sum(seq2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[227], line 2
      1 seq2 = ['a', 1, 2]
----> 2 sum(seq2)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```