lektion3

October 29, 2026

Table of Contents

1 for-Schleifen

1.1 break und continue

1.2 Nützliche built-in Funktionen

2 while Schleife

3 Comprehensions

3.1 für Listen (list comprehension)

3.2 für Wörterbücher (dictionary comprehension)

3.3 für Mengen (set comprehension)

3.4 für Generatoren

3.5 geschachtelter Aufruf

4 Funktionen mit beliebig vielen Argumenten

4.1 Packing und Unpacking mit *

1 Schleifen (loops)

Schleifen sind Strukturelemente, die ein wiederholtes Durchlaufen von Programmblöcken ermöglichen. Man unterscheidet zwischen while-Schleifen und for-Schleifen.

In der letzten Woche hatten wir gesehen, dass auch mit rekursiven Funktionsaufrufen ein wiederholtes Abarbeiten von Anweisungen möglich ist. Oft sind Schleifen aber naheliegender.

1.1 for-Schleifen

Der Aufbau einer for Schleife ist wie folgt, wobei das else optional ist

```
for element in iterierbarers_objekt:
    block von anweisungen
[else:]
    block von anweisungen
```

[1]:
```
for i in [1, 2, 4]:
    print(i)
```

```
1
2
4
```

[2]:
```
for i in (1, 2, 3):
    print(i)
```

```
1
2
3
```

Das geht einfacher mit dem Range Objekt (siehe Blatt 1). Damit erhält man alle ganzen Zahlen in einem Bereich.

`range([anfang],ende)`

Wird der optionale Anfangsindex anfang nicht angegeben, beginnt der Bereich bei 0. Der Endeindex ende ist wieder exklusiv.

[3]:
```
for i in range(1, 4):
    print(i)
```

```
1
2
3
```

Auch Wörterbücher, Mengen und Zeichenketten sind iterierbar

[4]:
```
wb = {1: 'a', 2: 'b', (1, 2): 'a+b', 0: 'Null'}
# Iteration über die Schlüssel (keys)
for k in wb:
    print(f'Key {k=} \t Eintrag: {wb[k]}')
```

```
Key k=1          Eintrag: a
Key k=2          Eintrag: b
Key k=(1, 2)     Eintrag: a+b
Key k=0          Eintrag: Null
```

[5]:
```
menge =  {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
for element in menge:
    print(element)
```

```
pear
apple
orange
banana
```

[6]:
```
for zeichen in '+-+-':
    print(zeichen)
```

```
+
-
```

```
+
-
```

print beendet jede Ausgabe mit einer neuer Zeile. Mit dem end Paramater kann man das Ende der Ausgabe festlegen.

[7]:
```
for zeichen in '+-+-':
    print(zeichen, end=',')
```

```
+,-,+,-,
```

1.1.1 break und continue

Mit break verlässt man eine Schleife vorzeitig, mit continue springt man zum Anfang und überspringt den Rest.

[8]:
```
for n in range(1, 10):
    if n % 4 == 0:
        print(f'{n=} teilbar durch 4:{n} = 4*{n//4}')
        break
    if n % 2 == 0:
        print(f'{n=} teilbar durch 2:{n} = 2*{n//2}')
```

```
n=2 teilbar durch 2:2 = 2*1
n=4 teilbar durch 4:4 = 4*1
```

[9]:
```
for n in range(1, 10):
    if n % 4 == 0:
        print(f'{n=} teilbar durch 4: {n} = 4*{n//4}')
        continue
    if n % 2 == 0:
        print(f'{n=} teilbar durch 2: {n} = 2*{n//2}')
```

```
n=2 teilbar durch 2: 2 = 2*1
n=4 teilbar durch 4: 4 = 4*1
n=6 teilbar durch 2: 6 = 2*3
n=8 teilbar durch 4: 8 = 4*2
```

Das optionale else in einer for-Schleife wird selten verwendet. Es ist vermutlich nur nützlich, wenn man mit einem break eine Schleife abbricht.

Beispiel aus https://docs.python.org/3/tutorial/controlflow.html Abschnitt 4.5

[10]:
```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:  # x teilt n
            print(f'{n} ist {x} * {int(n/x)}')
            break
```

```
4 ist 2 * 2
6 ist 2 * 3
```

```
8 ist 2 * 4
9 ist 3 * 3
```

[11]:
```python
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:  # x teilt n
            print(f'{n} ist {x} * {int(n/x)}')
            break
    else:  # die Schleife  x in 2..n-1 wurde nicht abgebrochen oder die␣
 ↪Schleife ist leer
        print(f'{n} ist prim')
```

```
2 ist prim
3 ist prim
4 ist 2 * 2
5 ist prim
6 ist 2 * 3
7 ist prim
8 ist 2 * 4
9 ist 3 * 3
```

Es kommt etwas anderes raus, wenn wir das else ein- oder ausrücken

[12]:
```python
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:  # x teilt n
            print(f'{n} ist {x} * {int(n/x)}')
            break
        else:  # hier gehört das else zum if, also falls x kein Teiler von n ist
            print(f'{n} ist prim')
```

```
3 ist prim
4 ist 2 * 2
5 ist prim
5 ist prim
5 ist prim
6 ist 2 * 3
7 ist prim
7 ist prim
7 ist prim
7 ist prim
7 ist prim
8 ist 2 * 4
9 ist prim
9 ist 3 * 3
```

1.1.2 Nützliche built-in Funktionen

Built-in Funktionen, die das Programmieren mit for-Schleifen vereinfachen, sind zip(), enumerate(), reversed(), sorted().

zip() Reißverschluss: Verbindet Elemente aus mehreren Listen oder Iterables

[13]:
```
str1 = 'AEIOU'
lst1 = ['Affe', 'Esel', 'Igel', 'Otter', 'Uhu', 'Zebra']

for i, j in zip(str1, lst1):
    print(f'{i} wie {j}')
```

```
A wie Affe
E wie Esel
I wie Igel
O wie Otter
U wie Uhu
```

reversed() dreht die Reihenfolge um.

[14]:
```
for i in reversed(lst1):
    print(i)
```

```
Zebra
Uhu
Otter
Igel
Esel
Affe
```

enumerate() zählt zusätzlich die durchiterierten Elemente ab

[15]:
```
for index, element in enumerate(lst1):
    print(index, element)
```

```
0 Affe
1 Esel
2 Igel
3 Otter
4 Uhu
5 Zebra
```

[16]:
```
for i, element in enumerate(lst1, start=1):  # fängt bei start zu zählen an
    print(i, element)
```

```
1 Affe
2 Esel
3 Igel
4 Otter
5 Uhu
6 Zebra
```

[17]:
```
list(enumerate(zip(str1, lst1)))
```

[17]:
```
[(0, ('A', 'Affe')),
 (1, ('E', 'Esel')),
 (2, ('I', 'Igel')),
 (3, ('O', 'Otter')),
 (4, ('U', 'Uhu'))]
```

1.2 while Schleife

Der Aufbau einer while-Schleife ist wie folgt, wobei das else optional ist. Der Block anweisung1 wird ausgeführt solange bedingung ein Ausdruck ist, der **True** ist.

```
while bedingung:
    anweisung1
[else]
    anweisung2
```

Wieder kann man mit break die Schleife abbrechen, oder mit continue zum Anfang der Schleife springen.

[18]:
```
a = 1
print(a)
while a <= 10:
    a = a + (a + 1)
    print(a)
```

```
1
3
7
15
```

[19]:
```
a = 1
print(a)
while True:
    if a > 10:
        break
    a = a + (a + 1)
    print(a)
```

```
1
3
7
15
```

[20]:
```
def arek(a):
    print(a)
    if a > 10:
        return a
    return arek(a + (a + 1))
```

```
a = arek(1)
```

```
1
3
7
15
```

[21]:
```
import numpy as np
rng = np.random.default_rng() # Zufallszahlengenerator (random number generator)
c = rng.integers(1, 7) # integers(1,7) "würfelt" zufällig Zahl zwischen 1 und 7␣
 ↪(exklusive)
```

[22]:
```
zaehler = 0
summe = 0
while summe < 100:
    summe += 1 + rng.integers(1, 7)  # += inplace plus (summe = summe + 1 +␣
 ↪random integer in [1,6]
    zaehler += 1 # zaehler = zaehler + 1
print(f'Nach {zaehler} mal würfeln ist die Augensumme {summe}')
```

```
Nach 24 mal würfeln ist die Augensumme 101
```

1.3 Comprehensions

1.3.1 für Listen (list comprehension)

Statt

[23]:
```
a = []
for j in range(5):
    a.append(2 * j + 1)
a
```

[23]:
```
[1, 3, 5, 7, 9]
```

verwendet man auch die kompaktere Form

[24]:
```
aa = [2*j+1 for j in range(5)]
aa
```

[24]:
```
[1, 3, 5, 7, 9]
```

1.3.2 für Wörterbücher (dictionary comprehension)

[25]:
```
{i: i**2 for i in range(5)}
```

[25]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

1.3.3 für Mengen (set comprehension)

Die Menge der Quadratzahlen bis 4^2

$\{i^2 : i \in \mathbb{N}_0 \wedge i < 5\}$

[26]:
```
{i**2 for i in range(5)}
```

[26]:
```
{0, 1, 4, 9, 16}
```

man kann dabei auch Bedingungen abfragen $\{i^2 : i \in \mathbb{N}_0 \wedge i < 5 \wedge ggT(i^2, 2) = 1\}$

[27]:
```
{i**2 for i in range(5) if (i**2) % 2 != 0}  # alle ungeraden Quadratzahlen bis␣
↪16
```

[27]:
```
{1, 9}
```

[28]:
```
from math import gcd #gcd greatest common divisor (deutsch ggT)
{i**2 for i in range(5) if gcd(i**2,2)== 1}  # alle ungeraden Quadratzahlen bis␣
↪16
```

[28]:
```
{1, 9}
```

1.3.4 für Generatoren

[29]:
```
g = (i**2 for i in range(5)) # !!keine tuple comprehension!!
```

[30]:
```
g
```

[30]:
```
<generator object <genexpr> at 0x770264354790>
```

[31]:
```
list(g)
```

[31]:
```
[0, 1, 4, 9, 16]
```

Generatoren brauchen weniger Speicher, da Sie faul (lazy) sind und das jeweils nächste Element erst berechnen, wenn es benötigt wird.

1.3.5 geschachtelter Aufruf

Primzahlen bis N

[32]:
```
N = 20
[n for n in range(2, N) if all((n % m > 0 for m in range(2, n)))]
```

[32]:
```
[2, 3, 5, 7, 11, 13, 17, 19]
```

1.4 Funktionen mit beliebig vielen Argumenten

Ein Beispiel: Minimum

[33]:
```
def mymin(a, b):
    """ Berechnet das Minimum von zwei Argumenten"""
    if a < b:
        return a
    else:
        return b
```

[34]:
```
mymin(3.2, 2.1)
```

[34]: 2.1

1.4.1 Packing und Unpacking mit *

[35]:
```
*b, = 1, 2, 3
b
```

[35]: [1, 2, 3]

[36]:
```
lst = list(range(5))
a, b, *c, = lst
c
```

[36]: [2, 3, 4]

Der * hier wird als unpacking operator bezeichnet. Die rechte Seite (der Tupel (1,2,3) bzw. die Liste [0,1,2,3,4,5]) wird entpackt und in b bzw c gespeichert.

[37]:
```
def mymin(*eingabewerte):
    ''' Berechnet das betragsweise Minimum von beliebig vielen Argumenten
        mymin(1, -1, 1+j)
        Output: 1
    '''
    #print(eingabewerte, type(eingabewerte))
    minimum = abs(eingabewerte[0])
    for a in eingabewerte:

        assert isinstance(
            a, (float, int, complex)), 'Eingabewerte müssen Zahlen sein'

        if abs(a) < minimum:
            minimum = abs(a)

    return minimum
```

[38]:
```
mymin(2, 2.1, 3, 1 + 2j)
```

[38]: 2

Der Stern * hier wird als packing operator bezeichnet, weil er die Argumente 2, 2.1, 3, 1 + 2j in das Tupel eingabewerte packt.

[39]:
```
liste = [2, 3, 4, 1]
```

[40]:
```
mymin(*liste) #unpacking
```

[40]: 1

[41]:
```
mymin(liste)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[41], line 1
----> 1 mymin(liste)

Cell In[37], line 7, in mymin(*eingabewerte)
      2 ''' Berechnet das betragsweise Minimum von beliebig vielen Argumenten
      3     mymin(1, -1, 1+j)
      4     Output: 1
      5 '''
      6 #print(eingabewerte, type(eingabewerte))
----> 7 minimum = abs(eingabewerte[0])
      8 for a in eingabewerte:
     10     assert isinstance(
     11         a, (float, int, complex)), 'Eingabewerte müssen Zahlen sein'

TypeError: bad operand type for abs(): 'list'
```

[42]:
```
min(liste)
```

[42]: 1

[43]:
```
?min
```

[44]:
```
min(*liste)
```

[44]: 1