lektion3

October 24, 2024

1 Schleifen (loops)

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

In der letzten Woche hatten wir gesehen, dass auch durch Verwendung eines rekursiven Funktionsaufrufs ein wiederholtes Abarbeiten von Anweisungen möglich ist. Für viele Anwendungen sind Schleifen aber der offensichtlichere Ansatz um Anweisungen wiederholt auszuführen.

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. Damit erhält man alle ganzen Zahlen in einem Bereich.

`range([anfang],ende)`

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

[3]:
```
list(range(1, 4))
```

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

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

```
1
2
3
```

Auch Wörterbücher, Mengen und Zeichenketten sind iterierbar

[5]:
```
wb = {1: 'a', 2: 'b', (1, 2): 'a+b', 0: 'Null'}
for k in wb:
    print(k, wb[k])
```

```
1 a
2 b
(1, 2) a+b
0 Null
```

[6]:
```
menge = {1, 3, 4, 1, 2}
for element in menge:
    print(element)
```

```
1
2
3
4
```

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

```
+
-
+
-
```

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

[8]:
```
for zeichen in '+-+-':
    print(zeichen, end=',')
```

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

2 Einschub

Mit

```
n = 10
f'abc {n}'
```

erzeugt man den String ‘abc 10’, wobei der Ausdruck in { } ausgewertet wird

[1]:
```
jahr = 2024
f'Willkommen in {jahr}/{jahr - 1999}'
```

[1]:
```
'Willkommen in 2024/25'
```

2.0.1 break und continue

Mit **break** kann man eine Schleife vorzeitig verlassen und mit **continue** springt man zur nächsten Iterierten und überspringt den Rest.

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

```
2 = 2*1
4 = 4*1
```

[11]:
```
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}')
```

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

[12]:
```
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** "verschieben"

[13]:
```
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, alsoe 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
```

oder weglassen

[14]:
```
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
        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
```

2.0.2 Nützliche built-in Funktionen

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

zip() verbindet mehrere Sequenzen und erlaubt ein gemeinsames Iterieren.

[2]:
```
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.

[3]:
```
for i in reversed(lst1):
    print(i)
```

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

enumerate() zählt die durchiterierten Elemente ab

[4]:
```
for i, j in enumerate(lst1):
    print(i, j)
```

```
0 Affe
1 Esel
2 Igel
3 Otter
```

```
4 Uhu
5 Zebra
```

[5]:
```
for i, j in enumerate(lst1,
                      start=1):  # Man kann auch bei start zu zählen beginnen
    print(i, j)
```

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

[6]:
```
list(enumerate(zip(str1, lst1)))
```

[6]:
```
[(0, ('A', 'Affe')),
 (1, ('E', 'Esel')),
 (2, ('I', 'Igel')),
 (3, ('O', 'Otter')),
 (4, ('U', 'Uhu'))]
```

2.1 while Schleife

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

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

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

[20]:
```
a = 1
print(a)
while a <= 10:
    a = a + (a + 1)
    print(a)
```

```
1
3
7
15
```

[21]:
```
a = 1
print(a)
while True:
```

```
    if a > 10:
        break
    a = a + (a + 1)
    print(a)
```

```
1
3
7
15
```

[22]:
```
def arek(a):
    print(a)
    if a > 10:
        return a
    return arek(a + (a + 1))


a = arek(1)
```

```
1
3
7
15
```

[23]:
```
from numpy.random import randint
```

[24]:
```
zaehler = 0
summe = 0
while summe < 100:
    summe += 1 + randint(6)  # += inplace plus (summe = summe + 1 + randint(6))
    zaehler += 1
print(f'Nach {zaehler} mal würfeln ist die Augensumme {summe}')
```

```
Nach 23 mal würfeln ist die Augensumme 103
```

2.2 Listen-Abstraktion

2.2.1 für Listen (list comprehension)

Statt

[25]:
```
a = [[1]]
for _ in range(5):
    a.append([1])
a
```

[25]:
```
[[1], [1], [1], [1], [1], [1]]
```

verwendet man auch die kompaktere Form

[26]:
```
aa = [[1] for _ in range(5)]
aa
```

[26]:
```
[[1], [1], [1], [1], [1]]
```

2.2.2 für Wörterbücher (dictionary comprehension)

[27]:
```
{i: i**2 for i in range(5)}
```

[27]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

2.2.3 für Mengen (set comprehension)

[28]:
```
{i**2 for i in range(5)}
```

[28]:
```
{0, 1, 4, 9, 16}
```

man kann dabei auch Bedingungen abfragen

[29]:
```
{i**2
 for i in range(5) if (i**2) % 2 > 0}  # alle ungeraden Quadratzahlen bis 16
```

[29]:
```
{1, 9}
```

2.2.4 Generator

[30]:
```
g = (i**2 for i in range(5))
```

[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.

Primzahlen bis N

[47]:
```
N = 20
[n for n in range(2, N) if all((n % m > 0 for m in range(2, n)))]
```

[47]:
```
[2, 3, 5, 7, 11, 13, 17, 19]
```

2.3 Ein Beispiel: Minimum

[33]:
```
def mymin(a, b):
    """ Berechnet das Minimum der Eingabeparameter"""
    if a < b:
```

```
        return a
    else:
        return b
```

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

[34]:
```
2.1
```

2.3.1 Packing und Unpacking

Mit * kann man beliebig viele Eingabeparameter in eine Liste zusammenfassen

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

[35]:
```
[1, 2, 3]
```

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

[36]:
```
([2, 3, 4], 0, 1)
```

[37]:
```
def mymin(*eingabewerte):
    #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
```

[39]:
```
lst
```

[39]:
```
[0, 1, 2, 3, 4]
```

Der * dient auch dazu eine Liste zu entpacken

[40]:
```
print(*lst)
```

```
0 1 2 3 4
```

[41]:
```
liste = [2, 3, 4, 1]
```

[42]:
```
mymin(*liste)
```

[42]:
```
1
```

[43]:
```
mymin(liste)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[43], line 1
----> 1 mymin(liste)

Cell In[37], line 3, in mymin(*eingabewerte)
      1 def mymin(*eingabewerte):
      2     #print(eingabewerte, type(eingabewerte))
----> 3     minimum = abs(eingabewerte[0])
      4     for a in eingabewerte:
      6         assert isinstance(
      7             a, (float, int, complex)), 'Eingabewerte müssen Zahlen sein

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

[44]:
```
min(liste)  # eingebaute Minimumfunktion in Python
```

[44]:
```
1
```

[45]:
```
?min
```

[ ]:
```

```