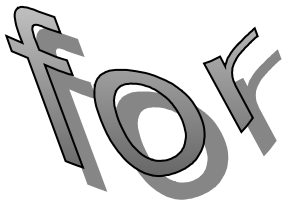
Wiederholungsstrukturen

Bei Wiederholungsstrukturen geht es um Anweisungen, die benutzt werden, um eine oder mehrere andere Anweisungen wiederholt auszuführen. Eine solche Anweisung besteht aus zwei Teilen, einmal dem Schleifenkopf, also dem Teil, der die Steuerung beschreibt und der Beschreibung dessen, was wiederholt wird, dem Schleifenkörper.

Die for - Anweisung



Syntax:

```
for <Iterationsvariable> in <Sammlungsobjekt>:
    <Schleifenkörper>
```

Python kennt keine *"normalen Zählschleifen"*. Die for-Schleife arbeitet nicht mit einer Zählvorschrift, sondern grundsätzlich mit einer Iteration über die Elemente einer Sammlung. Beispiele für Sammlungsklassen sind unten im **Anhang** angegeben.
Beginnen wir hier einmal mit dem Beispiel, bei dem zwei solche Sammlungen auftreten, nämlich ein String und eine Liste. Das gewählte Beispiel dient hier nur dem Vergleich der beiden Wiederholungsanweisungen und zur Demonstration, wie man in Python Zählschleifen erstellt.
Die folgende Funktion gibt zu einem übergebenen String eine Liste aller ASCII-Codes seiner Zeichen aus.

```
def gibCodes(wort):
    rueckgabe = []
    for buchstabe in wort:
        rueckgabe.append(ord(buchstabe))
    return rueckgabe
```

```
>>> gibCodes('Hund und Katze')
[72, 117, 110, 100, 32, 117, 110, 100, 32, 75, 97, 116, 122, 101]
```

In der Definition einer **`for`**-Schleife ist also die Definition eines Variablennamens notwendig, mit dem auf die Elemente der Sammlung zugegriffen wird und nach **`in`** folgt dann der Name der Sammlung.
Selbstverständlich gibt es auch die Möglichkeit, über einen Index auf die Zeichen in einem String zuzugreifen. Das folgende Programm zeigt aber, wie umständlich das im Vergleich zur oberen Lösung aussieht.

```
def gibCodes(wort):
    rueckgabe = []
    for index in range(len(wort)):
        rueckgabe.append(ord(wort[index]))
    return rueckgabe
```

Für den Index wird zunächst eine Sammlung erzeugt. Das geschieht bei den 2-er-Versionen von Python auch wirklich real: Es wird die Länge des Wortes bestimmt und die Funktion **`range`** erzeugt eine Liste der Zahlen zwischen 0 und dem ihr übergebenen Wert. Dabei ist der Startwert 0 nicht zwingend, übergibt man zwei Parameter, wird der erste als Startwert interpretiert, bei dreien ergibt der dritte Parameter dann eine Schrittweite.

```
>>> range(len('Hund und Katze'))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

In diesem Beispiel wird also die Liste der natürlichen Zahlen von 0 bis 13 wirklich real erzeugt und dann über sie iteriert. Das kann bei umfangreichen Iterationen zum Aufbau umfangreicherer Sammlungen führen, bevor auch nur irgendeine Berechnung gemacht wurde. Um das zu vermeiden, hat man die Iterator-Funktion xrange[1] eingeführt, die dafür sorgt, dass die zu iterierenden Elemente immer erst dann erzeugt werden, wenn sie benötigt werden. Daher führt die entsprechende Programmzeile zu einem anderen Ergebnis.

```
>>> xrange(len('Hund und Katze'))
xrange(14)
```

Durch **`xrange`** wird also nur ein Iteratorobjekt zurückgegeben, das der Schleifenvariablen nacheinander die Werte 0 bis 13 zuweist.

Schleifen mit while

Syntax:

```
while <Bedingung>:
    <Schleifenkörper>
```

Das oben betrachtete Beispiel kann man zwar auch mit while realisieren:

```
def gibCodes(wort):
    rueckgabe = []
    while len(wort) != 0:
        rueckgabe.append(ord(wort[0]))
        wort = wort[1:]
    return rueckgabe
```

for oder while

Das Beispiel ist hier aber nur als Vergleichsbeispiel gezeigt. Bei realen Anwendungen sollte man darauf achten, dass beide Schleifentypen nach ihren Anforderungen eingesetzt werden:

- **`for`** sollte dann verwendet werden, wenn die zu iterierenden Objekte vorher bekannt sind,
- **`while`** sollte eingesetzt werden, wenn das Ausführen der Wiederholung an eine Bedingung geknüpft ist.

Diese Bedingung in der while-Anweisung wird in manchen Texten fälschlicherweise als Abbruchbedingung bezeichnet; es ist aber eine Laufbedingung. Nur wenn diese Bedingung nicht mehr wahr ist, wird die Schleife abgebrochen.

Eine weitere Anmerkung: Die while-Anweisung ist eine kopfgesteuerte Schleife, also eine Schleife mit Einstiegsbedingung, eine Anweisung der Form **`repeat <Anweisung> until <Abbruchbedingung>`** gibt es nicht.

1 Text aus der Dokumentation: *"This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously."* Ist das nicht eigentlich ein Stream?

Das Aussteigen aus Schleifen

Man kann Schleifen auf mehrere Arten verlassen. Einmal gibt es die reguläre Art der Beendigung, indem bei einer for-Schleife alle Elemente der Sammlung bearbeitet wurden oder indem bei einer while-Schleife die Einstiegsbedingung in den Zustand falsch gegangen ist.
In einem Schleifenkörper können aber auch die folgenden Anweisungen auftreten:

- **`continue`** beendet den aktuellen Schleifendurchlauf und es wird der nächste Schritt bearbeitet [wenn es denn noch einen gibt].
- **`break`** beendet die gesamte Schleife, als wäre bei einem **`for`** alles bearbeitet bzw. bei **`while`** die Laufbedingung nicht mehr erfüllt.
- **`return`** hat bei einer Funktion oder Methode dieselbe Wirkung. Trifft eine Funktion bei einer Auswirkung auf ein return, wird sie in jedem Fall mit dem Wert[1] hinter return abgebrochen, also auch innerhalb von Schleifen.

Endlos – Schleifen

Man wird daher in einigen Programmen den Schleifenkopf

```
while True:
    <Anweisungsfolge>
```

vorfinden, die prinzipiell unzulässig wäre, da er zu einer Endlos-Schleife führen würde. Die oben beschriebenen Anweisungen ermöglichen das. Grundsätzlich gilt:

> In einer while-Schleife muss sich der Zustand des Systems in jedem Fall so verändern können, dass sie abgebrochen werden kann.

Rekursion

Auch Python stellt die Möglichkeit bereit, durch Rekursion[2] Wiederholungen umzusetzen. Die Funktionsdefinition

```
def rekursiveFunktion(Parameter):
    if <Abbruchbedingung>:
         return <Rückgabe für den Abbruchfall>
    else:
        <Anweisungsfolge enthält Aufruf
           von rekursiveFunktion(veränderte Parameter)>
```

ermöglicht die wiederholte Ausführung der Anweisungsfolge mit jeweils geänderten Parametern. Wichtig ist, dass sich dabei irgendwann der Zustand so verändert, dass die Abbruchbedingung erfüllt ist.

Kampf gegen ~~„if-Schleifen"~~

Immer wieder findet man in Schülertexten die Bezeichnung *„if-Schleifen"*. **`if`** leitet zwar auch eine **Kontrollstruktur** ein, allerdings keine Wiederholung, sondern eine Verzweigung.

1 Es ist auch zulässig, dass hinter return nichts steht. Dann wird None zurückgegeben.
2 Leider erkennt Python keine Endrekursion → Gefahr von zu großer Rekursionstiefe.

Beispiele für Iterationen mit for auf Sammlungen

Listen

Dieser Zugriff taucht im Raumplaner-Projekt beispielsweise bei Schrankwand auf. Die Klasse ist in diesem Beispiel auf reine Ausgaben verändert.

```
class Schrankwand:
    def __init__(self, anzahl=3):
        self.schrankliste = []
        for i in range(anzahl):
            self.schrankliste.append("Schrank Nr. "+str(i))

    def GibFigur(self):
        print "Man erhält die Gesamtfigur durch"
        for schrank in self.schrankliste:
            print "Hinzufügen von " + schrank

schrankwand = Schrankwand()
schrankwand.GibFigur()
```

Das sonst sinnlose Programm ergibt die Ausgaben:

```
>>>
Man erhält die Gesamtfigur durch
Hinzufügen von Schrank Nr. 0
Hinzufügen von Schrank Nr. 1
Hinzufügen von Schrank Nr. 2
```

Tupel

In diesem Beispiel wird die Berechnung der Länge eines Vektors betrachtet.

```
import math
def laenge(vektor):
    summe = 0
    for komponente in vektor:
        summe += komponente**2
    return math.sqrt(summe)

vektor = (3,4,6)
print laenge(vektor)
```

Strings

Auch Strings, also Zeichenketten, sind Sammlungen, eben von Zeichen[1].

```
def allesGross(wort):
    rueckgabe = ''
    for buchstabe in wort:
        rueckgabe += buchstabe.upper()
    return rueckgabe
```

[2]

Der Aufruf:

1 Achtung: Python kennt keinen eigenen Typ für Zeichen. Das sind einfach Strings mit der Länge 1.

2 Zur Erinnerung: Strings sind ein statischer Datentyp. Das Anhängen eines Buchstaben an einen bestehenden String geht nicht. Man muss den String neu zuweisen.

```
print allesGross('Hund und Katze')
```

liefert die Ausgabe:

```
>>>
'HUND UND KATZE'
```

Dictionary

Bei Dictionaries [Assoziationstyp] geht die Iteration zwar prinzipiell, allerdings bekommt man dabei nur den Zugriff auf die vorhandenen Schlüsselworte [key]. Ein Zugriff auf die Paare der Zuordnung gelingt nur mit dem Umweg über die Liste der items

```
dic = {'Hund':'dog', 'und':'and', 'Katze':'cat'}
for inhalt in dic.items():
    print inhalt
>>>
('Katze', 'cat')
('Hund', 'dog')
('und', 'and')
```

Die Ausgabe zeigt, dass die Reihenfolge in einem Dictionary nicht der Eingabe entspricht. Es ist bei einer Iteration also nur sicher, dass über alle Elemente iteriert wird.

```
for element in dic:
    print element
```

gibt nur die keys aus. Ausgaben:

```
>>>
Katze
Hund
und
```

Darüber kann man auf die zugeordneten Werte [value] zugreifen:

```
for element in dic:
    print dic.get(element)
>>>
cat
dog
and
```

Sets

Sets, also Mengen, enthalten ein Element genau einmal.

```
menge = set([2,4,6,5,4,3,2,1])
print menge
for element in menge:
    print element
```

Ausgaben:

```
>>>
set([1, 2, 3, 4, 5, 6])
1
2
3
4
5
6
```