Vorkurs Informatik

Dirk Frettlöh

dfrettloeh@techfak.de

19. September 2023

Variablen

Daten, die wir

- im Vorfeld nicht kennen (Usereingabe, veränderte Werte)
- öfters benötigen
- ...

können wir in einer Variable speichern.

Variablentypen

Beispiele für Typen

boolean: True oder False

int: 42

float: 3.1415926

string: "Hallo Welt"

Strings ("Zeichenketten") müssen in Anführungszeichen stehen. In python ist es egal, ob ' oder ". (Aber am Besten konsistent)

Wie benutzt man Variablen?

Python erkennt (oft) den Datentyp, daher ist das einfach.

```
1 a = 1
2 b = True
3 c = 2.7182818
```

Wichtig !!!: Zum Trennen von Vor- und Nachkommastelle wird **kein** Komma, sondern ein Punkt verwendet.

```
a = 5
b = 10
print(a + b)
```

```
print(a - b)
```

```
print(a + b)
```

```
print(a % b)
```

```
c = True
print(c)
```

Numerische Operationen

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
**	Potenz
%	Modulo (Division mit Rest)
+=	Addition mit Zuweisung a += b heißt a = a + b
-=	Subtraktion mit Zuweisung a -= b heißt a = a - b
	(usw.: *=, /=, **=,...)

```
a = 10
b = 20
```

```
a += b
print(a)
```

```
b -= a
print(b)
```

Noch ein paar Infos am Rande

Python kann auch die Rechenregeln

5+5*3 ist nicht das gleiche wie (5+5)*3

...und es versucht den Typ zu raten

```
1 23.0/3.0 = 7.666666666666667
```

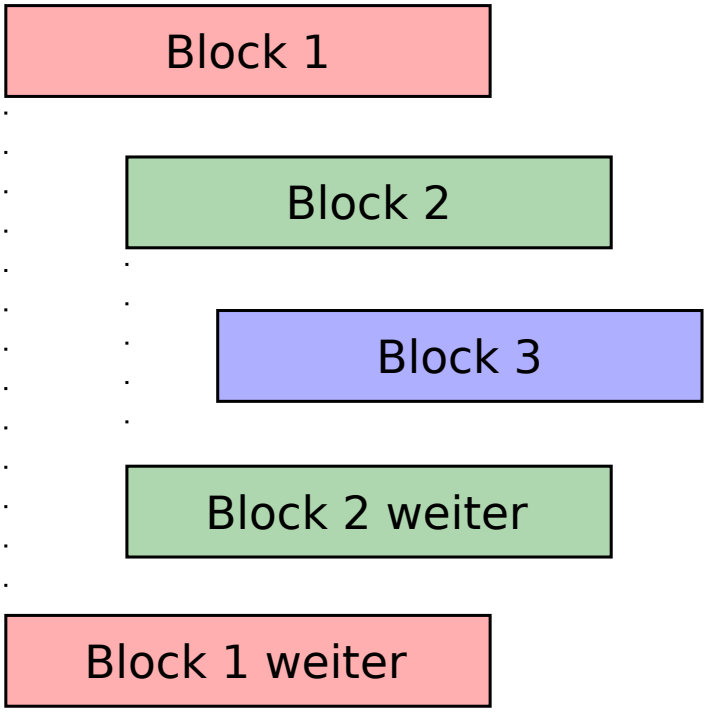
und

```
1 23/3 = 7.666666666666667
```

aber

```
1 21/7 = 3.0
```

Blöcke



Damit der Code übersichtlicher ist, kann und muss man ihn in Blöcke unterteilen.

Die Einrückung besteht aus 4 Die Leerzeichen **oder** einem Tab.

WICHTIG: Innerhalb einer Datei darf nicht gewechselt werden.

meisten Editoren übersetzen einen TAB automatisch in vier Leerzeichen.

Es geht auch mit 2 oder 3 oder 5 Leerzeichen, aber 4 ist etablierter Standard.

Python Keywords

```
1 pass
```

Mit `pass` kann man sagen, dass nichts passiert. Dies ist wichtig, wenn eine Einrückung erforderlich ist, aber dort nichts gemacht werden soll.

```
1 # Ich bin ein kommentar
```

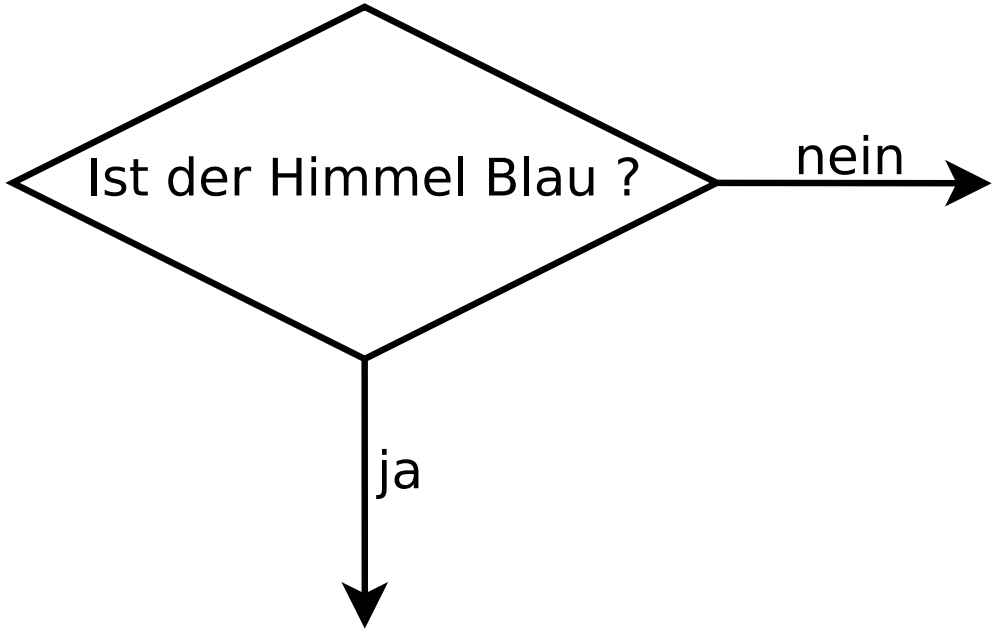
Alle Zeilen, die mit einer # anfangen, werden von Python ignoriert.

If-Abfragen

Abfragen

Bis jetzt ist es so, dass jede Zeile, die ihr in die Datei schreibt, auch ausgeführt wird.
Das ist aber nicht immer gewollt und dafür gibt es bedingte Befehlsblöcke (if).

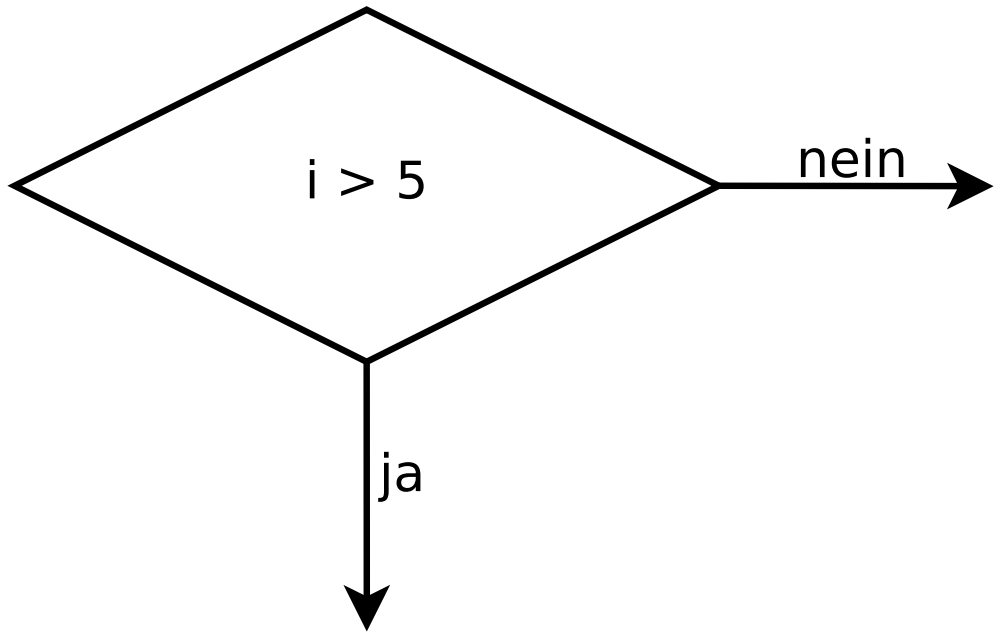
if-Abfrage



Frage?

Wie sähe die Abfrage aus, wenn man überprüfen möchte, ob eine Zahl größer 5 ist?

Antwort



if-Anweisung in Python

```
1 if (5 < i):
2     print("5 ist kleiner i")
```

Wenn 5 kleiner als i ist, wird der print Befehl ausgeführt, sonst nicht.

Blockanweisungen in Python

```
1 if (5 < i):
2     print("5 ist kleiner i")
3
4     if(4 < i):
5         print("4 ist kleiner i")
6
7         if (3 < i):
8             print("3 ist kleiner i")
```

Nur wenn die `if`-Anweisung in Zeile 1 wahr ist, wird die in Zeile 4 überprüft. Und nur wenn diese wahr ist, die in Zeile 7.

if-Anweisung in Python

```
1 if (5 < i):
2     print("5 ist kleiner i")
```

In den runden Klammern in Zeile 1 können unterschiedliche Dinge stehen. Sie müssen aber logische Aussagen sein, sodass der Computer bestimmen kann, ob sie wahr oder falsch sind.

Abstrakter Aufbau

```
1 if( <logische Aussage> ):
2 ____
```

- strikt kleiner

```
if (5 < i):
```

- strikt größer

```
if (5 > i):
```

- kleiner gleich

```
if (5 <= i):
```

- größer gleich

```
if (5 >= i):
```

- ungleich

```
if (5 != i):
```

- gleich (Vergleich! keine Zuweisung)

```
if (5 == i):
```

Anmerkungen zur if-Abfrage

Aussagen negieren

Mit `not` kann eine Aussage negiert werden.

```
1 if ( not 5 > i ):
2     print( "5␣ist␣nicht␣groesser␣i")
```

Verbinden von Aussdrücken

Mehrere logische Ausdrücke können in einer `if`-Abfrage überprüft werden.

```
1 if ( <Aussage1> and (<Aussage2> or <Aussage3>)):
```

Vergleiche Kapitel “Formale Logik” im Matheteil:

Bei `and` müssen beide Aussagen wahr sein.
Bei `or` reicht es, wenn eine der beiden Aussagen wahr ist.

Anmerkungen zur if-Abfrage

if und else

Stellen wir uns vor, wir wollen wissen, ob `i` größer als 6 ist, oder kleiner gleich 6.

Das könnte so aussehen:

```
1 if (i > 6 ):
2     print( "i␣ist␣groesser␣6")
3 if (i <= 6 ):
4     print("i␣ist␣kleiner␣gleich␣6")
```

Jedoch würden wir so zweimal prüfen. Mit einem `else` geht das leichter und lesbarer.

```
1 if (  i > 6 ):
2     print( "i␣ist␣groesser␣6")
3 else:
4     print("i␣ist␣kleiner␣gleich␣6")
```

Verschachtelung von if

Manchmal möchte man sicherstellen, dass nur, wenn das eine nicht zutrifft, etwas anderes überprüft wird.
Das könnte man so realisieren:

```
1 if (i % 2 == 0):
2     print("i␣ist␣gerade")
3 else :
4     if (i < 10):
5         print("i␣ist␣ungerade␣und␣kleiner␣10␣")
```

Oder so:

```
1 if (i % 2 == 0):
2     print("i␣ist␣gerade")
3
4 if ((i < 10) and not (i % 2 == 0) ):
5     print("i␣ist␣ungerade␣und␣kleiner␣10␣")
```

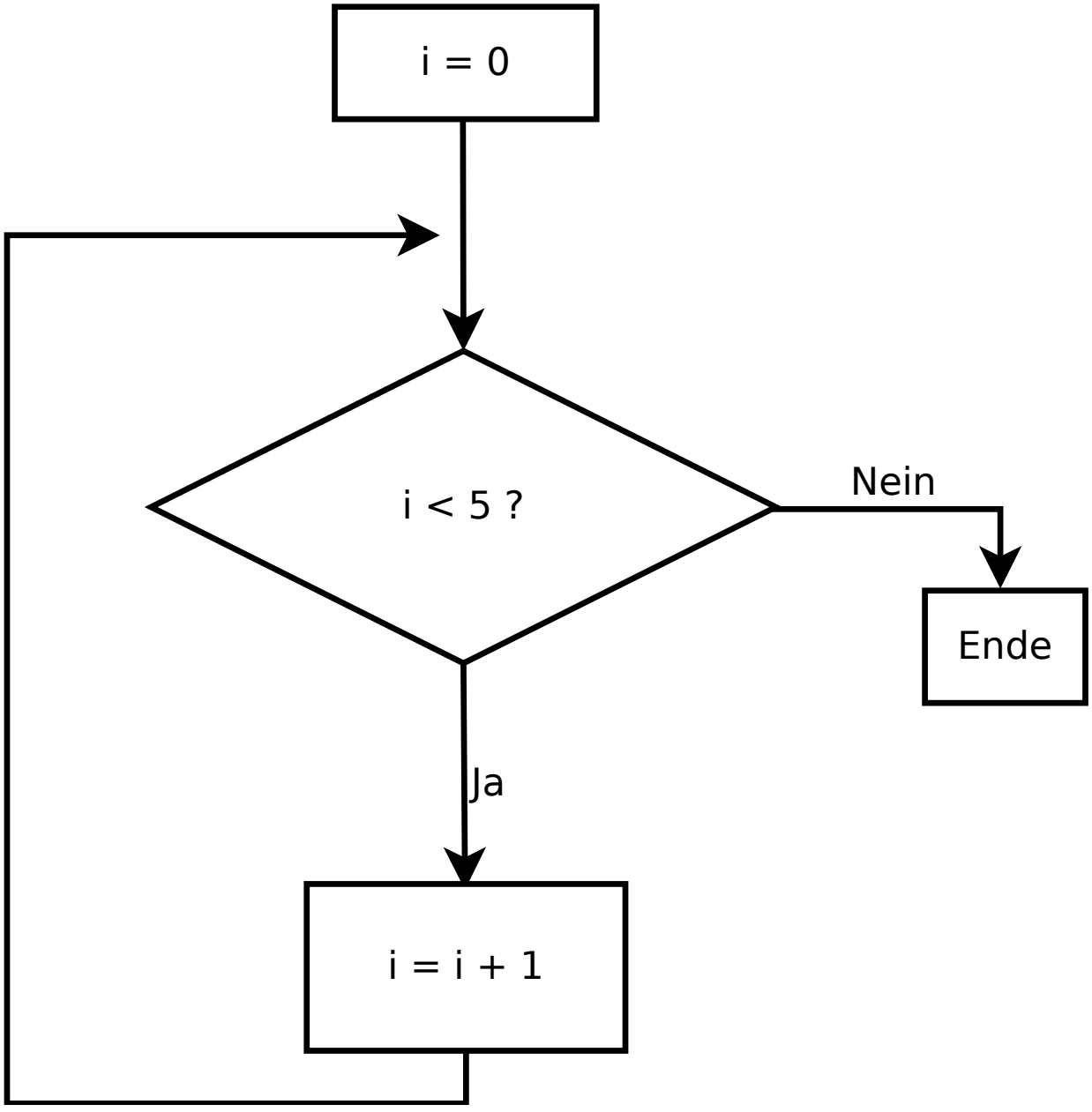
Verschachtelung von if

Da man so etwas aber sehr häufig braucht, gibt es in Python eine Kurzschreibweise.

```
1 if (i % 2 == 0):
2     print("i␣ist␣gerade")
3 elif (i < 10):
4     print("i␣ist␣ungerade␣und␣kleiner␣10␣")
5 else:
6     print("i␣ist␣nicht␣gerade␣und␣auch␣nicht␣kleiner␣10")
```

Bis jetzt können wir mit Python nicht viel mehr als mit einem handelsüblichen Taschenrechner. Denn eine wichtige Sache fehlt uns noch.

Schleifen


i = 0
i < 5 ?
Nein
Ende
Ja
i = i + 1

while-Schleife in Python

```
1 i = 0
2 while (i < 5):
3     i += 1
```

Der eingerückte Teil `i += i` nach dem `while` wird so lange ausgeführt, bis die Aussage in den runden Klammern `i < 5` falsch wird.

Bis 10 Zählen

```
1 i = 0
2 while (i < 10):
3     print(i)
4     i+=1
```

Abstrakter Aufbau

```
1 while (<auswertbare Aussage>):
```

Natürlich kann man (wie beim `if`) mehrere `while`'s verschachteln.

```
1 i = 0
2 while(i < 10):
3     j = 0
4     while(j < 10):
5         print(j)
6         j += 1
7     i += 1
```

Verbindung von while und if

Natürlich kann man `if` und `while` beliebig verschachteln.

```
1 i = 0
2 while (i < 100):
3     if (i % 2 == 0):
4         print(i, "ist gerade")
5     else:
6         print(i, "ist ungerade")
7     i += 1
```

Listen

Wenn man viele Werte speichern möchte, ist es sehr umständlich für jeden Wert eine Variable anzulegen. Und wenn man nicht weiß, wie viele Werte der Benutzer angibt, ist es sogar unmöglich.

Stellt euch vor, ihr möchtet für eine Formel wie

$$f(x) = x^2 + x * 100$$

die Werte im Bereich zwischen -100 und +100 berechnen und speichern. Das wären 201 Variablen.
Als Lösung haben wir Listen (`list`).

Listen in Python

Eine Liste in Python zu erzeugen geht so:

```
1 a = [4,6,12,4,76,8,12]
```

oder so

```
1 a = ["Hund","Katze","Maus"]
```

Die Kommata trennen die einzelnen Einträge.

WICHTIG: Benutzt die eckigen Klammern [ ] und nicht die runden ( ) oder die geschwungenen { }.

Besonderheiten von Listen

Die erste Zahl in der Informatik ist immer die 0 (bis auf wenige Ausnahmen). Also hat das erste Element einer Liste den Index 0.

Umgang mit Listen

```
1 liste = [12,16,18,20,22]
2 #Zugriff auf ein Element
3 print(liste[0])
4 print(liste[1])
5 print(liste[2])
6 print(liste[3])
```

```
1 liste = [12,16,18,20,22]
2 print(liste[0])
3 #ueberschreiben von einem Element
4 liste[0] = 11
5 print(liste[0])
```

Umgang mit Listen (Fehlerzustand)

Was passiert, wenn man als Index ein Element angibt, das es nicht gibt?

```
1 liste = [12,16,18]
2 #Zugriff auf ein Element, das noch nicht existiert
3 print(liste[3])
```

```
1 Traceback (most recent call last):
2 File "<stdin>", line 1, in <module>
3 IndexError: list index out of range
```

Python sagt: "`list index out of range`"

Umgang mit Listen (Element anhängen)

Jetzt könnte man sich fragen, was es nützt, wenn alle Elemente einer Liste vorab mit einem Wert belegen werden müssen, damit die Liste lang genug ist. Man kann auch zur Laufzeit eine Liste verlängern.

```
1 liste = [12,16,18]
2 print(liste)
3 liste.append(20)
4 #jetzt ist das element mit dem Index 3 da.
5 print(liste)
```

Ausgabe:

```
1 [12, 16, 18]
2 [12,16,18,20]
```

Umgang mit Listen (Element entfernen)

Und natürlich kann man auch ein Element löschen.

```
1 liste = [12,14,16,18]
2 del(liste[2])
```

Ausgabe:

```
1 [12,14,18]
```

Für die, die Arrays (Felder) kennen: Listen verhalten sich offenbar leicht anders.

Umgang mit Listen (Länge der Liste)

Um zu wissen, wie lang eine Liste ist, gibt es in Python den `len()` Befehl.

```
1 liste = [12,16,18]
2 print(len(liste))
3 liste.append(20)
4 print(len(liste))
```

Ausgabe:

```
1 3
2 4
```

Umgang mit Listen (Liste enthält?)

Um zu testen, ob eine Element in einer Liste ist, gibt es den `in()` Befehl.

```
1 liste = [12,16,18]
2 print(12 in a)
3 print(13 in a)
```

Ausgabe:

```
1 True
2 False
```

Listen von Listen

Manchmal benötigt man eine Liste von Listen; auch dies ist natürlich möglich.

```
1 temp = [[1,2,3],[4,5,6]]
2 print(temp[0][0])
3 print(temp[1][0])
```

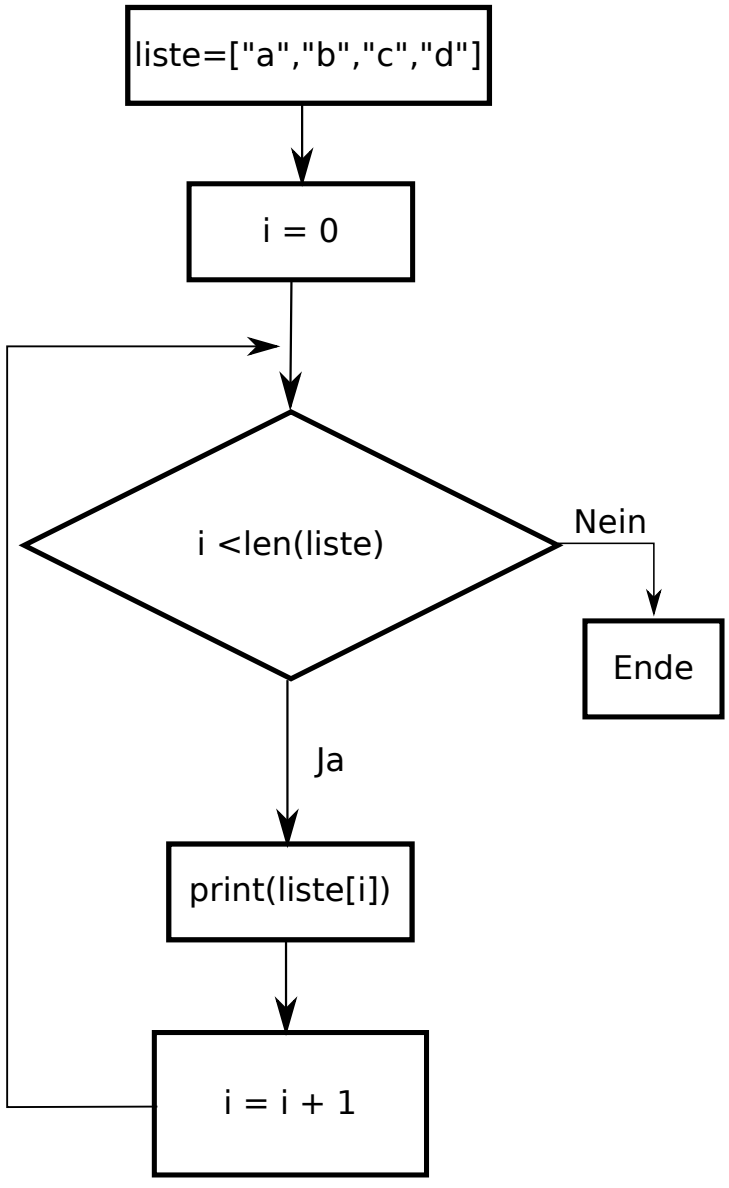
Wie man sich vorstellen kann, ist es möglich Listen weiter zu verschachteln.

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

Jedoch wird das dann sehr schnell unübersichtlich.

Eine kleine Aufgabe

Wie sähe es aus, wenn man mit einer `while`-Schleife alle Elemente einer Liste ausgeben möchte?




```
1 liste = ["a","b","c","d"]
2
3 i=0
4
5 while(i < len(liste)):
6     print(liste[i])
7     i = i + 1
```

for

Da man sehr häufig über Listen iteriert, gibt es dafür eine Kurzschreibweise.

while

```
1 liste = ["a","b","c","d"]
2
3 i=0
4
5 while(i < len(liste)):
6     print(liste[i])
7     i = i + 1
```

for

```
1 liste = ["a","b","c","d"]
2
3
4
5 for i in liste:
6     print(i)
```

Der Befehl range

Wie wir gesehen haben, kann man mit `for` sehr gut über eine Liste iterieren. Aber `for` kann noch mehr.

Allgemein wird die '`for`-Schleife' auch als Zählschleife bezeichnet. Da es aber sehr mühsam ist für eine Schleife, die z. B. 10 Iterationen durchlaufen soll, die Liste

```
1 lauf = [0,1,2,3,4,5,6,7,8,9]
2 for i in lauf:
3     print(i)
```

von Hand zu tippen, gibt es in Python den Befehl `range`.

```
1 for i in range(0,10):
2     print(i)
```

range

```
1 range(n,m)
```

`range` erstellt eine Liste, die von n bis $m-1$ (!) geht. Ein Beispiel:

```
1 range(10,20)
```

`range` kann aber noch mehr als nur $+1$ zu rechnen.
Mit einer dritten Zahl kann die Schrittweite angegeben werden.

```
1 for i in range(0,10,2):
2     print(i)
```

Auch ist es möglich mit Negativen Zahlen zu arbeiten.

```
1 range(5, 0, -1)       entspricht [5, 4, 3, 2, 1]
2 range(0, -5, -1)      entspricht [0, -1, -2, -3, -4]
```

Fragen?

Geschafft

Viel Spaß im Tutorium