Listen

Erstellen einer Liste

Oftmals haben wir es mit einer Sammlung von Daten zu tun und wollen diese speichern, z.B. Messdaten. Eine Datenstruktur, die sich dafür anbietet, ist die Liste. In Python lässt sich eine Liste leicht erstellen:

```
meine_liste = []
meine_zweite_liste = list()
```

Die beiden Ausdrücke haben das selbe Resultat: eine leere Liste wird kreiert.

Nun können wir diese Liste füllen. Wollen wir nur wenige Werte speichern, können wir diese direkt bei der Deklaration mit angeben:

```
meine_liste = ["a", "b", "c"]
```

Alternativ fügen wir sie nach und nach in die leere Liste ein:

```
meine_liste = []
meine_liste.append("a")
meine_liste.append("b")
meine_liste.append("c")
```

Zugreifen auf ein Element

Um auf das ite Element einer Liste l zuzugreifen, schreiben wir

```
dreier_liste = ["bla", "bli", "Blub"]
element = dreier_liste[0]
```

Achtung, wir fangen bei Null an zu zählen! element hat also den Wert “bla”.

Möchten wir auf eines der hinteren Elemente zugreifen, können wir auch rückwärts zählen, indem wir negative ganze Zahlen benutzen:

```
letztes_element = dreier_liste[-1]
vorletztes_element = dreier_liste[-2]
```

Entfernen von Elementen

Ebenso können wir Elemente entfernen, wenn wir sie nicht mehr brauchen. Dafür rufen wir die Methode *pop* auf:

```
entferntes_element = meine_liste.pop(2)
```

Obiger Befehl entfernt aus *meine_liste* das Element, das an zweiter Stelle (Achtung, wir fangen bei Null an, zu zählen!). Damit ist es zwar aus der Liste entfernt, aber falls nötig, können wir es in einer weiteren Variablen speichern.

Listen miteinander verknüpfen

Möchten wir zwei Teillisten zu einer großen Liste machen, so schreiben wir einfach

```
l1 = [1, 2, 3]
l2 = ["a", "b", "c"]
liste_gesamt = l1 + l2
print liste_gesamt
```

mit dem Resultat

```
[1, 2, 3, 'a', 'b', 'c']
```

Was kann eine Liste speichern?

Bisher haben wir als Beispiel nur Strings in unserer Liste gespeichert. Selbstverständlich lassen sich aber auch alle anderen Datentypen darin speichern. Selbst Listen von Listen, oder Listen von Listen von Listen (etc...) sind möglich. Auch ist es erlaubt, verschiedene Datentypen in einer Liste zu speichern:

```
bunt_gemischt = ["hallo", [1, 2, 3], 3.456, 5+3j, True, False]
```

Allerdings heißt das nicht, dass es unbedingt eine gute Idee sein muss, verschiedenste Datentypen in einer Liste zu speichern... Möglich ist es dennoch.

Iterieren über Listen

Möchten wir mit den Elementen einer Liste arbeiten, können wir eine for-Schleife benutzen:

```
for element in bunt_gemischt:
    print element
```

Möchten wir zusätzlich noch den Index jedes Elements benutzen, bietet sich das *enumerate* keyword an:

```
for i, element in enumerate(bunt_gemischt):
    print "Das", i, "te Element meiner Liste ist:", element
```

Anstatt nur über jedes Element zu iterieren, iterieren wir nun über Wertepaare (genauer gesagt Tupel) von Indices und Elementen.

Ein praktisches Beispiel

Ein Chemiker hat in seinem Grundpraktikum alle 20 Sekunden die Konzentration eines bestimmten Stoffes aufgeschrieben. Die resultierende Liste sieht dann so aus:

```
konzentrationen = [1.000,
                   0.880,
                   0.756,
                   0.656,
                   0.582,
                   0.495,
                   0.441,
                   0.387,
                   0.335,
                   0.294,
                   0.242,
                   0.219,
                   0.194,
                   0.166,
                   0.140,
                   0.130,
                   0.109,
                   0.099,
                   0.079,
                   0.078,
                   0.064
                  ]
```

Um nun sowohl Zeit, als auch Konzentration zu bekommen, schreiben wir folgenden Code:

```
print "Zeit in Sekunden,  Konzentration"
for index, messwert in enumerate(konzentrationen):
    print index*20, messwert
```

Listen "slicen"

Manchmal möchten wir nur auf einen Teil der Gesamtliste zugreifen. In diesem Fall ist das sogenannte "Slicing" von Vorteil.

Die Syntax ist:

```
teil_liste = liste[start:stop:schritt]
```

Mit diesem Befehl wird eine Teilliste kreiert, die beim Index *start* der Ursprungsliste beginnt, und beim Index *stop* -1 der Ursprungsliste endet. Der

schritt gibt dabei an, ob jedes Element zwischen *start* und *stop* (*schritt* = 1), jedes zweite (*schritt* = 2), etc. genommen werden soll.

Wenn wir z.B. in obigem Beispiel nur jeden zweiten Messwert haben wollten, würden wir schreiben:

```
teil_konzentrationen = konzentrationen[0:len(konzentrationen):2]
```

Dies lässt sich in diesem Fall sogar noch kürzer schreiben als

```
teil_konzentrationen = konzentrationen[::2]
```

Das liegt daran, dass *start* und *stop* automatisch auf 0 und Länge der Liste gesetzt werden, wenn man sie auslässt.

Tatsächlich hat die "Slicing" Syntax eine starke Ähnlichkeit mit der range-Funktion. Übungsaufgabe: Teilliste erstellen, *ohne* die Slicing Syntax zu benutzen.

Merkhilfe für die Index-Benutzung beim Slicen

Folgendes Schaubild (Quelle) macht anschaulich klar, wie die Indizes beim Slicen zu verstehen sind.

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Anstatt die Elemente zu nummerieren, werden vielmehr die Trennlinien zwischen den Elementen gezählt. Wollen wir zum Beispiel die Teilliste, die alle Elemente vom nullten bis zum dritten Element enthält (also P,y,t,h), geben wir als linke Begrenzung die nullte Trennlinie, und als rechte Begrenzung die vierte Trennline an.

Listen sortieren

In Python lassen sich Listen entweder mit der Methode *sort*, oder mit der Funktion *sorted* sortieren.

```
l1 = [8, 3, 12, 2]
l2 = sorted(l1)
```

sorted erstellt dabei eine neue sortierte Liste, während *sort* die Ursprungsliste sortiert. Die alte Sortierung geht dabei also verloren!

```
l1 = [8, 3, 12, 2]
l1.sort()
```

Möchten wir ab-, statt aufsteigend sortieren, fügen wir noch das zusätzliche Keyword “reverse=True” hinzu:

```
l1 = [8, 3, 12, 2]
l1.sort(reverse=True)
```

Unveränderliche Listen (Tupel)

In manchen Fällen ist es wüschenswert, eine unveränderliche Liste zu erstellen. Dies erreicht man folgendermaßen:

```
mein_tupel = (1, 2, 3)
```

Um aus einer gegebenen Liste einen Tupel zu machen, schreibt man:

```
meine_liste = [1, 2, 3]
mein_tupel = tuple(meine_liste)
```

Der Unterschied zur normalen Liste ist nur, dass wir weder Elemente zum Tupel hinzufügen noch entfernen können.

Listen/Tupel entpacken

Manchmal möchten wir mehrere Elemente einer Liste Variablen zuordnen. Als Beispiel haben wir einen Tupel, in dem Name, Vorname und Note eines Studenten steht:

```
dreier_tupel = ("Müller", "Daniel", 1.0)
```

Um nun die drei Werte in dem Tupel in die Variablen vorname, nachname, note zu speichern, schreiben wir:

```
(nachname, vorname, note) = dreier_tupel
```

Da in dreier_tupel drei Werte gespeichert sind, bekommt jede Variable einen davon zugewiesen. Das Entpacken funktioniert immer dann, wenn wir auf der linken Seite und der rechten Seite Tupel mit der selben Schachtelung haben.

Wir können das auch ausnutzen, um in einer Zeile gleich mehrere neue Variablen zu definieren:

```
name, vorname = "Müller", "Heinz"
```

Tatsächlich ist es auch gar nicht notwendig, die äußeren runden Klammern zu setzen.

Das Definieren von mehreren Variablen in einer Zeile ist immer dann empfehlenswert, wenn es sich um Variablen handelt, die einen ähnlichen Zweck erfüllen.

Die Schachtelung darf übrigens beliebig komplex werden:

```
(a, (b, (c, d, e), f), g, h) = (1, (2, (3, 4, 5), 6), 7, 8)
```

Listen zippen (der Reißverschluss)

Möchte man über mehrere Listen gleichzeitig iterieren, ist die zip Funktion hilfreich.

Beispiel: wir haben eine Liste mit Vornamen, eine mit Nachnamen, und eine mit Noten. Um jetzt jeweils Nachname, Vorname und Note auszugeben, schreiben wir:

```
nachnamen_liste = ["Müller", "Maier", "Schulz"]
vornamen_liste = ["Daniel", "Dieter", "Elise"]
noten_liste = [1.0, 2.3, 3.7]

for vn, nn, note in zip(vornamen_liste, nachnamen_liste, noten_liste):
    print vn, nn, note
```

Um zu sehen, was da genau passiert, schreiben wir

```
print zip(nachnamen_liste, vornamen_liste, noten_liste)
```

Die Ausgabe ist dann:

```
[('Müller', 'Daniel', 1.0), ('Maier', 'Dieter', 2.3), ('Schulz', 'Elise', 3.7)]
```

Hier sehen wir, was die zip-Funktion gemacht hat: Sie hat aus den ersten Elementen der drei Listen ("Müller", "Daniel" und 1.0) einen Tupel gemacht, ebenso aus den zweiten Elementen und aus den dritten Elementen, und diese in einer neuen Liste gespeichert. Jeder Dreier-Tupel wird nun beim Iterieren in die drei Variablen *vn*, *nn* und *note* entpackt.

Eine etwas ausführlichere Schreibweise wäre:

```
for dreier_tupel in zip(vornamen_liste, nachnamen_liste, noten_liste):
    vn, nn, note = dreier_tupel
    print vn, nn, note
```

List comprehensions

Manchmal möchten wir aus einer vorhandenen Liste eine neue Liste erstellen, die nur bestimmte Elemente enthält. Z.B. könnten wir folgende Liste haben

```
tiere = ["Affe", "Löwe", "Giraffe", "Schlange", "Nashorn"]
```

und wir möchten eine Liste mit den Tieren erstellen, deren Namen mindestens 5 Buchstaben lang sind.

Dann schreiben wir:

```
tiere_2 = [tier for tier in tiere if len(tier) >= 5]
```

Alternativ könnte man natürlich auch so vorgehen:

```
tiere_2 = []
for tier in tiere:
    if len(tier) >= 5:
        tiere_2.append(tier)
```

Verglichen mit der List-Comprehension ist diese Variante jedoch deutlich länger. Das Ergebnis ist aber das selbe.

Das können wir natürlich auch mit anderen Listen, oder generell mit Objekten machen, über die wir iterieren können.

Folgender Code speichert nur die ungeraden Zahlen von 1 bis 20 in einer Liste:

```
ungerade_zahlen = [zahl for zahl in range(1, 20) if zahl % 2 != 0]
```

Nochmal kurz Strings

Jetzt, wo wir gesehen haben, was man alles mit Listen anstellen kann, könnte sich einem die Frage stellen, ob das nicht auch auf Strings anwendbar ist? Tatsächlich haben Strings ganz ähnliche Eigenschaften wie Listen. Wir können sie addieren:

```
s1 = "Guten "
s2 = "Tag"
s3 = s1 + s2
print(s3)
```

Ausgabe:

```
Guten Tag
```

Wir können sie *slicen*:

```
text = "Einen schönen guten Tag"
print(text[::2]) # gibt jeden zweiten Buchstaben aus
```

Ausgabe:

```
EnnshnngtnTg
```

Ebenso können wir über die einzelnen Buchstaben eines Strings **iterieren**, und sogar sortieren lassen sich Strings.

Ein praktisches Beispiel

Hier finden Sie eine sogenannte xyz-Datei, in der die Struktur eines Systems aus Molekülen gespeichert ist. Die Datei hat folgenden Aufbau:

```
<Anzahl Atome im System>
<Kommentarzeile>
<Atomname 1> <x-Position> <y-Position> <z-Position>
<Atomname 2> <x-Position> <y-Position> <z-Position>
<Atomname 3> <x-Position> <y-Position> <z-Position>
...
```

Unser Ziel ist es nun, eine Liste aller Atomnamen, und eine Liste aller Positionen zu erhalten.

```
atom_list = [] # hier werden die Atomnamen gespeichert...
position_list = [] # und hier ihre Positionen
with open("molecule-example.xyz", "r") as f:
    for i, line in enumerate(f):
        if i >= 2: # wir ignorieren die ersten beiden Zeilen
            line_split = line.split() # jede Zeile wird gesplittet in eine Liste, die Atomna
            atom_list.append(line_split[0]) # der Name kommt in atom_list
            position_list.append([float(pos) for pos in line_split[1:]]) # die Liste der At
```

Mengen (Sets)

Mengenoperationen, wie man sie aus der Mathematik kennt, lassen sich in Python mit dem Datentyp *set* leicht ausführen.

Eine Menge kann mittels zweier Schreibweisen erstellt werden:

```
menge = set([1, 2, 3, 4, 1, 2])
print menge
```

oder

```
menge = {1, 2, 3, 4, 1, 2}
print menge
```

Da eine Menge jedes Element nur einmal enthält, ist die Ausgabe:

```
{1, 2, 3, 4}
```

Mengenoperationen

Mengenoperationen lassen sich folgendermaßen ausführen:

- Schnitt: ““python a = {1, 2, 3, 4} b = {1, 4, 6, 7, 8}

print a & b `Ausgabe:` {1, 4} `* Vereinigung:`python a = {1, 2, 3, 4} b = {1, 4, 6, 7, 8}

print a | b `Ausgabe:` {1, 2, 3, 4, 6, 7, 8} `* Differenz:`python a = {1, 2, 3, 4} b = {1, 4, 6, 7, 8}

print a - b print b - a `Ausgabe:` {2, 3} {6, 7, 8} `* Symmetrische Differenz`python a = {1, 2, 3, 4} b = {1, 4, 6, 7, 8}

print a ^ b ““

Ausgabe: `{2, 3, 6, 7, 8}`

- Wenn wir testen möchten, ob ein Element in der Menge enthalten ist, schreiben wir `python print 1 in {1, 2, 3} print "a" in {1, 2, 3}` Ausgabe: `True False`

Set Comprehensions

Ganz ähnlich wie bei Listen können wir Sets mithilfe von Set Comprehensions erstellen. Die Syntax ist fast identisch, nur statt eckiger benutzen wir hier geschweifte Klammern:

```
gerade_zahlen = {i for i in range(0, 200, 2)}
```

Assoziatives Datenfeld bzw. Wörterbücher (Dictionaries)

Ein weiterer nützlicher Datentyp ist das sogenannte assoziative Datenfeld, oder, einfacher ausgedrückt, Wörterbuch. Es speichert jeweils ein Paar aus Schlüsselwort (Key) und Wert (Value). Über das Schlüsselwort kann man sich den gespeicherten Wert ausgeben lassen.

Erstellen eines Dictionaries

Auch hier gibt es wieder mehrere Wege:

```
my_dict = dict(a=1, b=2, c=3)
my_dict_2 = {"a":1, "b":2, "c":3}
my_dict_3 = dict([("a", 1), ("b", 2), ("c", 3)])
```

In allen Fällen wird ein Wörterbuch mit den Schlüsselwörtern “a”, “b” und “c” und den dazugehörigen Werten 1, 2 und 3 erstellt.

Schreiben wir nun in eckigen Klammern eines der gespeicherten Keywords hinter den Variablennamen, können wir auf den zugehörigen Wert zugreifen.

```
print my_dict["a"]
```

Ausgabe:

```
1
```

Iterieren über alle Key, Value Paare

Wie bei Listen können wir auch bei Dictionaries eine for-Schleife benutzen, um über die enthaltenen Elemente zu iterieren.

```
my_dict = dict(a=1, b=2, c=3)
for key in my_dict:
    print key
```

Ausgabe:

```
a
c
b
```

Die Ausgabe zeigt zwei Dinge: erstens wird hier nur über die Keys unseres Dictionaries iteriert, und zweitens muss die Reihenfolge der Keys nicht erhalten bleiben. Falls wir die Keys noch sortieren wollen, können wir jedoch wieder die sorted Funktion verwenden.

```
for key in sorted(my_dict):
    print key
```

Ausgabe:

```
a
b
c
```

Möchten wir nicht nur auf die Keys, sondern auch auf die Werte zurückgreifen, können wir die Items-Methode unseres Dictionaries benutzen:

```
for key, value in my_dict.items():
    print key, ":", value
```

Ausgabe:

```
a : 1
c : 3
b : 2
```

Dictionary Comprehensions

Und auch hier gibt es wieder Comprehensions:

```
quadrat_zahlen_dict = {i: i*i for i in range(10)}
```

Die Schreibweise ist ähnlich wie bei den Sets, jedoch müssen wir nun mit einem Doppelpunkt getrennte Key-Value Paare angeben.