List Comprehension

Einleitung

Bei der *List Comprehension* handelt es sich um eine Verfahren zur Listenbildung. Die Liste gehört in Python zu den sog. **Kollektionen**, also Datentypen, die, wie der Name es sagt, 'Kollektionen', d.h. Sequenzen von Elementen und Abbildungen von Elementen aufeinander umfassen. Zu den uns bekannten Kollektionen gehören Zeichenketten, Dictionaries, Tupel und Listen. Diese Kollektionstypen unterscheiden sich in einer Reihe von Faktoren, z.B. hinsichtlich

- ihrer Notation und Interpretation
- den ihnen jeweils zugeordneten Methoden,
- der Frage, ob sie mutierbar sind oder nicht.

Nehmen wir als Beispiel die Zeichen 'a' und 'b'. Diese können wir wie folgt als Kollektionen darstellen:

1. Liste: ['a', 'b']
2. Zeichenkette: 'ab'
3. Tupel: ('a','b')
4. Dictionary {'a':'b'}

Hier sehen wir die verschiedenen Notationsformen für die unterschiedlichen Datentypen. Dabei ist zu berücksichtigen, dass deren Interpretation (und also die auf den Typ bezogenen Methoden) durchaus unterschiedlich ist. Liste, Zeichenkette und Tupel können beispielsweise problemlos durch ein weiteres Zeichen ergänzt werden:

a) Liste: ['a', 'b','c']
b) Zeichenkette: 'abc'
c) Tupel: ('a','b','c')

Im Falle des Dictionaries geht das aber nicht, denn im Dictionary besteht ein Element aus einem Paar von Objekten. Das erste dieser Objekte wird *Schlüssel* genannt (engl. *key*), das zweite *Wert* (engl. *value*). Während die Liste in 1, die Zeichenkette in 2 und das Tupel in 3 also aus jeweils zwei Elementen bestehen, den Zeichen 'a' und 'b', besteht das Dictionary in 4 nur aus einem Element, dem Schlüssel-Wert-Paar ('a','b'). Entsprechend anders sehen auch die Methoden aus, die uns zur Verfügung stehen, um die einzelnen Elemente einer Kollektion zu referenzieren. Dazu ein Beispiel. Sehen Sie den folgenden Ausschnitt aus der Python-Shell:

```
>>> Liste = ['a','b']
>>> ZK = 'ab'
>>> Tupel = ('a','b')
>>> WB = {'a':'b'}
>>> for element in Liste:
 print element,
a   b
>>> for element in ZK:
 print element,
a   b
>>> for element in Tupel:
 print element,
a   b
>>> for element in WB:
 print element,
a
```

Die **`for-in`** Anweisung iteriert prinzipiell über die Elemente einer Kollektion, d.h. wir können sie nutzen, um die einzelnen Elemente von Listen, Tupeln usw. zu referenzieren. Im Falle des Dictionaries liefert sie uns per Default aber nur die jeweiligen Schlüssel der Paare, vgl.

```
>>> WB = {'a':'b', 'c':'d', 'e':'f'}
>>> for element in WB:
 print element,
a   c   e
```

Um nun die Schlüssel-Wert-Paare zu referenzieren, können wir die Methode **`.items()`** verwenden, die speziell für den Datentyp Dictionary definiert ist:

```
>>> for element in WB.items():
 print element,
('a','b')  ('c','d') ('e','f')
```

Abgesehen von den unterschiedlichen Methoden, die mit den verschiedenen Kollektionstypen verbunden sind, unterscheiden sich diese auch hinsichtlich der Frage, ob sie mutierbar (veränderbar) sind, oder nicht. Der folgende Ausschnitt zeigt den Unterschied am Beispiel von Listen und Zeichenketten:

```
1.  >>> ZK = 'ab'
2.  >>> ZK
3.  'ab'
4.  >>> ZK + 'c'
5.  'abc'
6.  >>> ZK
7.  'ab'
```

```
1.  >>> Liste = ['a','b']
2.  >>> id(Liste)
3.  18273464
4.  >>> Liste
5.  ['a', 'b']
6.  >>> Liste.append('c')
7.  >>> Liste
8.  ['a', 'b', 'c']
9.  >>> id(Liste)
10. 18273464
```

Das der Variablen *ZK* zugewiesene Objekt, die Zeichenkette 'ab', wird in Zeile 4 mit einem weiteren Zeichen verkettet. Diese Anweisung wird sofort in Zeile 5 ausgewertet; dabei bleibt das eigentliche Objekt aber unverändert, wie Zeilen 6 und 7 zeigen. Bei dem der Variablen *Liste* zugewiesenen Objekt ['a','b'] sieht das anders aus. Zur Verdeutlichung wurde die Methode **`id()`** verwendet, die die Speicheradresse des fraglichen Objektes zurückgibt. In Zeile 6 wird an das Objekt namens *Liste* per **`.append()`** ein weiteres Zeichen angehängt. Diese Prozedur ändert das Objekt, wie man in Zeilen 7 – 10 sehen kann: das an gleicher Speicheradresse geführte Objekt ist nun die Liste ['a','b','c'].

An diesem Beispiel haben wir schon eine Methode gesehen, die man für die Veränderung einer Liste einsetzten kann: **`.append()`**. Diese Methode kann man auch für die Bildung einer Liste verwenden, vorausgesetzt, man gibt als 'Starter' eine leere Liste vor:

```
>>> Liste = []
>>> Liste.append('a')
>>> Liste.append('b')
>>> Liste.append('c')
>>> Liste.append('d')
>>> Liste
['a', 'b', 'c','d']
```

Das hätte man aber auch einfacher haben können, und dazu kommen wir zum eigentlichen Gegenstand dieses Textes.

List Comprehension

Bleiben wir für den Einstieg beim **konstruierten** Beispiel. Was wir wollen, ist die Bildung und sukzessive Erweiterung eine Liste durch die Elemente 'a', 'b', 'c' und 'd'. Dafür bietet es sich an, eben diese Elemente zunächst ihrerseits in einer Kollektion, z.B. einer Zeichenkette zu führen:

```
ZK = 'abcd'
```

Basierend auf dieser Zuweisung können wir nun wie folgt eine Liste bilden:

```
>>> Liste = []
>>> for element in ZK:
        Liste.append(element)
>>> Liste
['a', 'b', 'c', 'd']
```

Hier wird explizit eine leere Liste vorgegeben, die dann sukzessive um die Elemente aus ZK erweitert wird. Dasselbe Resultat erzielen wir durch die folgende Anweisung in Form der *List Comprehension* (fortan *LC*):

```
>>> Liste = [element for element in ZK]
>>> Liste
['a', 'b', 'c', 'd']
```

Wie zu sehen ist, ist es bei der LC nicht nötig, eine leere Liste vorzugeben, und auch die **`.append()`** Methode kommt nicht zum Einsatz. Stattdessen wird die Liste sozuagen *from scratch* neu gebildet.

Funktionieren kann das Ganze allerdings nur, wenn es eine 'Inputkollektion' gibt (im Beispiel ZK), auf deren Elemente sich bei der LC bezogen wird. Was wir hier machen, ist letztendlich ja die Abbildung zweier Elementmengen aufeinander. Im konstruierten Beispiel ist diese Abbildung recht trivial, anders sieht das im folgenden Fall aus.

Inputkollektion: die Zeichenkette 'ABCD'

Outputkollektion die Liste: ['a', 'b', 'c', 'd']

Hier soll die Outputkollektion alle Elemente aus der Inputkollektion in Form von Kleinbuchstaben enthalten. Hier 'passiert' m.a.W. etwas mit den Elementen der Inputkollektion. Per LC sähe das so aus:

```
>>> ZK = 'ABCD'
>>> Liste = [element.lower() for element in ZK]
>>> Liste
['a', 'b', 'c', 'd']
```

In etwas schräger umgangssprachlicher Formulierung könnte man das so beschreiben:

> *Bilde eine Outputkollektion in Form einer Liste basierend auf den Elementen aus der Inputkollektion, die du elementweise durchgehst und dabei ggf. bestimmte Methoden oder Operationen auf die Elemente anwendest.*

In den bis jetzt verwendeten Beispielen gab es jeweils eine Input- und eine Outputkollektion. Diese Beispiele waren insofern konstruiert, als es in Python natürlich eine Reihe von Methoden zur Typkonversion gibt, von denen wir einige auch schon kennengelernt haben. Um die Zeichenkette 'abcd' auf die Liste ['a', 'b', 'c', 'd'] abzubilden, würde man 'in echt' natürlich eher wie folgt vorgehen:

```
>>> list('abcd')
['a', 'b', 'c', 'd']
```

Auch die Geschichte mit 'ABCD' vs ['a', 'b', 'c', 'd'] würde man anders machen:

```
>>> ZK = 'ABCD'
>>> list(ZK.lower())
['a', 'b', 'c', 'd']
```

Etwas anders sieht es in der folgenden LC aus, in der die Inputkollektion eine Liste ist, deren Elemente nur unter einer bestimmten Bedinung für die Outputkollektion quadriert werden :

```
>>> Liste1 = [1,2,3,4,5,6,7,8,9]
>>> Liste2 = [element * element for element in Liste1 if element <= 5]
>>> Liste2
[1, 4, 9, 16,25]
```

Inputkollektion ist die Liste der Ziffern von 1...9. Outputkollektion ist eine Liste von Zahlen, die jeweils das Quadrat der Ziffern aus der Inputkollektion darstellen – aber nur, wenn letzere kleiner oder gleich 5 sind. Wie man hier sieht, können auch Bedingungen problemlos in LCs formuliert werden.

Noch ein Beispiel:

```
>>> wb = {'a':2, 'b':3, 'c':4, 'd':5}
>>> Liste = [str(element*2)+'mal' for element in wb.values()]
>>> Liste
['4mal', '8mal', '6mal', '10mal']
```

Hier wurden die Elemente aus aus der Inputkollektion, konkret die Werte aus **`wb`**, für die Outputkollektion mit 2 multipliziert und als **`string`**, an den die Zeichenkette 'mal' gehängt wurde, zurückgegeben

Um die Geschichte mit der LC abzuschließen noch ein kurzer Hinweis auf die Definition von Mengen in der Mengenlehre, da die LC aus der Mengennotation abgeleitet ist.

Der Ausdruck M = {1 + x | x ∈ ℕ, x < 5} beschreibt die Menge {2,3,4,5}

Er ist wie folgt zu lesen: die Menge aller 1 + x, wobei x eine Variable ist, die für ein Element aus der (Input)Menge der natürlichen Zahlen ℕ steht, falls dieses kleiner als 5 ist. Der senkrechte Strich kann als 'wobei' oder 'unter der Voraussetzung, dass' paraphrasiert werden; das Komma als 'falls'.

Der Teilausdruck " x ∈ ℕ, x < 5" definiert, für sich genommen, also auch eine Menge:

M = {x | x ∈ ℕ, x < 5} beschreibt die Menge {1,2,3,4}.

Um auf den ersten Ausdruck zurückzukommen, können wir diesen wie folgt annotieren:

$$M = \{ \underbrace{1 + x}_{\text{Outputfunktion}} \mid \underbrace{x}_{\text{Variable}} \in \underbrace{\mathbb{N}}_{\text{Inputmenge}} , \underbrace{x < 5}_{\text{Bedingung}} \}$$

Wollten wir nun in Python eine Liste aus den in dieser Menge beschriebenen Elementen per LC bilden, so müssten wir zunächst durch **`range()`**[1] die Inputmenge begrenzen: **`range(1,10)`** z.B. gibt alle natürlichen Zahlen von 1 bis 10 zurück. Dann setzen wir die Mengennotation direkt in Form einer LC um:

```
>>> [x+1 for x in range(1-10) if x < 5]
[2, 3, 4, 5]
```

[1] Mehr zu **`range()`** erfahren Sie über Pythons **`help()`**-Funktion