Strukturierte Datentypen

Python kennt mehrere ähnliche strukturierte Datentypen.

Tupel

Tupel sind nicht veränderbare Reihungen von beliebigen Objekten, die untereinander auch unterschiedlichen Typ haben können. Um ein Tupel zu erzeugen, muss man die Liste der Objekte durch Komma trennen und man kann die eigentlich notwendigen Klammern weglassen.

```
x = 1,'und',2
```

geht zwar, besser lesbar ist aber die Schreibweise

```
x = (1,'und',2)
```

Bei x=() - leeres Tupel, wozu auch immer – sind die Klammern notwendig und für ein Tupel mit einem Element muss man nach dem Element ein Komma einsetzen:

```
x=('eines',)
```

Andernfalls wird die Klammerung einfach ignoriert.

unpacking

Durch Mustervorgaben kann man bei Tupeln unpacking durchführen. Wenn links vom Zuweisungszeichen = mehrere Variablen und rechts ein Tupel stehen, erfolgt die Zuweisung elementweise, daher kann man folgendes machen:

```
>>> adresse='hans','meier','hamburg'
>>> vorname,nachname,ort=adresse
>>> vorname
'hans'
>>> nachname
'meier'
>>> ort
'hamburg'
```

indizierter Zugriff

Man kann auf die einzelnen Positionen im Tupel über den Index zugreifen, allerdings nur lesend, da Tupel nicht veränderbar sind. Es geht also:

```
>>> adresse[0]
'hans'
```

Dagegen ist

```
>>> adresse[2]='München'
```

unzulässig.

Listen

In Listen sind die einzelnen Objekte veränderbar. Eine Liste erzeugt man, indem man ihre Elemente in eckige Klammern schreibt und die Objekte durch Komma trennt. Die eckigen Klammern sind hier zwingend notwendig.

```
>>> x=[]                          # eine leere Liste
>>> y=['eines']                   # Liste mit einem Element
>>> z=[1,'zweites',3.0]           # Liste mit drei Elementen
```

Auch auf die Objekte in den Listen kann man über den Index zugreifen.

Werte zuweisen

Im Gegensatz zu den Tupeln kann man hier einzelnen Objekten oder ganzen Abschnitten (slices) Werte zuweisen. Es geht also:

```
>>> z[1]="neu"
>>> z                          # →   [1, 'neu', 3.0]
```
und
```
>>> z=[1,2,3,4,5,6]
>>> z[2:5]=['drei','vier','fuenf']
>>> z                          # →   [1, 2, 'drei', 'vier', 'fuenf', 6]
```
Dabei kann die Liste sogar in der Länge verändert werden:
```
>>> z=[1,2,3,4,5,6]
>>> z[2:2]=['drei','vier','fuenf']
>>> z                    # →   [1, 2, 'drei', 'vier', 'fuenf', 3, 4, 5, 6]
```
Bei gleichem Start- und Ende-Index wird also die gesamte neue Liste an der Stelle eingefügt.

append

Listen sind Instanzen von Klassen und verfügen über eine Reihe von Methoden.
Mit append kann die Liste verlängert werden.
```
>>> z.append('noch ein objekt')
>>> z  # →  [1, 2, 'drei', 'vier', 'fuenf', 3, 4, 5, 6, 'noch ein objekt']
```
ACHTUNG!!! Die Liste selbst wird dabei verändert! Will man die alte Liste unverändert behalten, reicht es nicht einmal, sie einer neuen Variablen zuzuweisen und diese zu bearbeiten:
```
>>> z=[1,2,3,4,5,6]
>>> neu=z
>>> neu.append('noch ein objekt')
>>> z       # →   [1, 2, 3, 4, 5, 6, 'noch ein objekt']
>>> z.append('noch ein objekt')
>>> neu     # →   [1, 2, 3, 4, 5, 6, 'noch ein objekt', 'noch ein objekt']
```
Das kann zu üblen Nebeneffekten führen. Man kann das verhindern, wenn man die neue Liste beispielsweise aus ihren einzelnen Objekten neu aufbaut:
```
>>> z=[1,2,3,4,5,6]
>>> neu=[]
>>> for index in range(len(z)): neu.append(z[index])
>>> neu
[1, 2, 3, 4, 5, 6]
>>> z.append('noch ein objekt')
>>> neu
[1, 2, 3, 4, 5, 6]
>>> z
[1, 2, 3, 4, 5, 6, 'noch ein objekt']
```
Einfacher mit:
```
>>>  neu = [] + z
```

count, index, insert, remove

x.count(y) gibt die Zahl der Elemente in der Liste x zurück, die gleich dem Objekt y sind.
```
>>> z=[1,2,1,2,3,1,2,3,4]
>>> z.count(1)          # →   3
```
Die Methoden, die das Objekt nicht verändern, werden auch von strings bereitgestellt, also z.B. z.index(3) zum Bestimmen der Position, aber nicht z.insert(2,'neu') zum Einfügen
Um ein Objekt aus der Liste wieder zu entfernen, verwendet man remove. Beachtet, dass bei remove nicht die Position in der Liste angegeben wird, sondern das Objekt.
z.reverse() dreht die Liste um, z.sort() sortiert die Liste mit <=, sodass die kleinsten Elemente vorne stehen.
Weitere Methoden bitte bei Bedarf in der Hilfe nachlesen.
Dictionary bitte in der Hilfe selbst nachlesen und ggf. nachfragen.

Dictionary

Ein Lexikon – englisch Dictionary – ist eine von der Sprache Python standardmäßig bereit gestellte Datenstruktur.

```
>>> lexikon = {'cat': 'katze', 'dog': 'hund', 'eats': 'frisst', 'jumps':
'springt', 'the': 'die', 'the': 'der'}
```

definiert ein Lexikon mit den Einträgen cat, dog, eats... [keys], denen die Übersetzungen Katze, Hund, frisst, … [values] zugeordnet sind.

Methoden

Die Klasse dict stellt mehrere Methoden bereit. Während einige, wie beispielsweise len(d) in der Regel selbsterklärend sind, sind die Zugriffe etwas ungewohnt, da sie sich an der Art orientieren, wie allgemein in Python auf Sequenztypen – und dazu zählt auch das Dictionary – zugegriffen wird:

```
>>> lexikon['dog']
'hund'
```

liefert also zum key-Wert den zugehörigen value-Wert.
Dasselbe macht die get-Methode:

```
>>> lexikon.get('dog')
'hund'
```

Wichtig ist noch zu wissen, wie ein Lexikon erweitert werden kann:

```
>>> lexikon.update({'frog': 'frosch'})
```