Komplexe Datentypen III

Komplexe Datentypen

Test auf Vorhandensein in Liste:

- Möglichkeiten:
  - count()
  - index()

Komplexe Datentypen

Test auf Vorhandensein in Liste:

- am besten: in

```
1  liste = [ 2, 4, 5, 7, 2, 3, 2]
2
3  print( 34 in liste)
4  print( 7 in liste)

False
True
```

Komplexe Datentypen

Test auf Vorhandensein in Liste:

- in und not in

```
Wert  in  Collection

Wert not in Collection
```

Komplexe Datentypen

Test auf Vorhandensein in Liste:

- in und not in

- für viele Datentypen

```
Wert  in  Collection

Wert not in Collection
```

Komplexe Datentypen

List Comprehensions

- kompakte Anwendung von Anweisungen

  auf sämtliche Elemente

ähnlich den bedingten Ausdrücken

```
1  var = (20 if x == 1 else 30)
```

Komplexe Datentypen

List Comprehensions

- erzeugen neue Liste
- sind in eckige Klammer gefasst
- bestehen aus einem oder mehr for/in-Bereichen
- legen iterierten Bezeichner fest

```
1 li_neu = [ x+1 for x in li ]
```

Komplexe Datentypen

List Comprehensions

```
1 li_neu = []
2 for x in li:
3     li_neu.append( x+1 )
```

```
1 li_neu = [ x+1 for x in li ]
```

Komplexe Datentypen

List Comprehensions

- nach for/in-Bereich Bedingung möglich
- für Anwendung nur auf einen Teil der Liste

```
1 li_neu = [ x+1 for x in li if x!=0 ]
```

Komplexe Datentypen

List Comprehensions

- möglich für mehrere Listen

```
>>> li1 = [1, 2, 3]
>>> li2 = [-11, 0, 12]

>>> [li1[i] + li2[i] for i in range(len(li1))]
[-10, 2, 15]
```

Komplexe Datentypen

List Comprehensions

Bsp. mit zwei for/in

```
>>> lst1 = ["A", "B", "C"]
>>> lst2 = ["D", "E", "F"]

>>> [(a,b) for a in lst1 for b in lst2]
[('A', 'D'), ('A', 'E'), ('A', 'F'),
('B', 'D'), ('B', 'E'), ('B', 'F'),
('C', 'D'), ('C', 'E'), ('C', 'F')]
```

Komplexe Datentypen

Aufgabe

Schreiben Sie eine list comprehension, die

- aus einer Liste *wortliste* von Strings eine neue Liste erzeugt, die
- nur diejenigen Wörter enthält, die mindestens ein Vorkommen des Buchstaben 'x' haben.

Komplexe Datentypen

Nachtrag: del

Funktion zum Löschen von Einträgen und

Teillisten über den Index

```
>>> li = [0,1,2,3,4,5,6,7,8,9,10,11]

>>> del li[7]
>>> li
[0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11]

>>> del li[7:10]
>>> li
[0, 1, 2, 3, 4, 5, 6, 11]
```

Komplexe Datentypen

Nachtrag: del

mit allen Varianten des Slicing

```
>>> li = [0,1,2,3,4,5,6,7,8,9,10,11]

>>> del li[:-4:3]

>>> li
[1, 2, 4, 5, 7, 8, 9, 10, 11]
```

Komplexe Datentypen

Tupel: die unveränderliche Schwester der Liste

nur die Operationen für immutable

Erzeugung mit runden Klammern

```
>>> tup = ( 0, 1, 2, 3, 4, 5 )

>>> tup[3]
3
```

Komplexe Datentypen

leeres Tupel wie bei Listen

```
>>> tup = ()
>>> tup
()
```

Komplexe Datentypen

Achtung bei nur einem Element

```
>>> tup = (1)

>>> tup
1

>>> type(tup)
<class 'int'>

>>> tup = (1,)

>>> tup
(1,)

>>> type(tup)
<class 'tuple'>
```

Komplexe Datentypen

Tuple Packing: Definition ohne Klammern

```
>>> tup = 1,

>>> tup
(1,)

>>> tup = 0, 1, 2, 3, 4, 5

>>> tup
(0, 1, 2, 3, 4, 5)
```

Komplexe Datentypen

Unpacking: Verteilung der Werte auf Variablen

```
>>> tup = 0, 1, 2, 3

>>> x1, x2, x3, x4 = tup

>>> x1
0

>>> x2
1

>>> (x1, x2, x3, x4) = tup
```

Komplexe Datentypen

Unpacking: Kann mehrere Zuweisungen zusammenfassen

```
>>> a, b = 10, 20

>>> a, b = b, a

>>> a
20

>>> b
10
```

Komplexe Datentypen

Unpacking: auch für Strings und Listen

```
>>> x1, x2, x3, x4 = "Baum"

>>> x4
'm'

>>> x1, x2, x3, x4 = [1,2,3,4]

>>> x3
3
```

Komplexe Datentypen

Unpacking: Fehler bei unpassender Länge

```
>>> a,b = [2,3,4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)

>>> a,b,c = [2,3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 3, got 2)
```

Komplexe Datentypen

Unpacking: für Tupel unbekannter Länge

```
>>> tup = ( 0, 1, 2, 3, 4, 5)

>>> x1, *x, x2 = tup

>>> x1
0

>>> x2
5

>>> x
[1, 2, 3, 4]
```

Komplexe Datentypen

Gegeben: ein Tupel tup der Länge mindestens 7

- Weisen Sie die ersten beiden Werte in tup den Variablen x und y zu, den Rest einem Tupel rest
- Weisen Sie den ersten und letzten Wert in tup den Variablen x und y zu, den Rest einem Tupel rest

Komplexe Datentypen

Unpacking: Auch erlaubt bei nicht sequentiellen Typen --- allerdings nicht klar definiertes Ergebnis

```
>>> a,b,c = { 'c', 'b', 'a' }

>>> a    oder    >>> a     oder  ....
'a'               'b'

>>> b
'b'

>>> c
'c'
```

Komplexe Datentypen

Veränderliche Unveränderlichkeit

```
>>> tup = ( [], (), [] )

>>> tup[0].append(3)

>>> tup
([3], (), [])

>>> tup[2].append(234)

>>> tup
([3], (), [234])
```

Komplexe Datentypen

Veränderliche Unveränderlichkeit

```
>>> tup = ( [], (), [] )

>>> tup[0].append(3)

>>> tup
([3], (), [])

>>> tup[2].append(234)

>>> tup
([3], (), [234])
```

indirekte Veränderung der Werte möglich

Komplexe Datentypen

Tupel und Listen lassen sich ineinander umwandeln

```
>>> tup = 2, 3, 4

>>> tup
(2, 3, 4)

>>> list(tup)
[2, 3, 4]



>>> li = [2, 3, 4]

>>> tuple(li)
(2, 3, 4)
```

Komplexe Datentypen

Welche der Funktionen, die wir für Listen kennen, funktionieren auch für Tupel?

join

join() erzeugt String aus Liste von Strings

- Funktion eines Strings, der als Trennstring fungiert
- Liste als Argument

```
1  liste = [ "String1", "String2", "String3"  ]
2
3  teilsymbol = " - "
4
5  z = teilsymbol.join( liste )
6
7  print( z )

String1 - String2 - String3
```

map / filter

map() und filter(): Funktionen für Iterables

- map() wendet Funktion auf alle Elemente an
- filter() wählt Werte aus

map / filter

map()

- Funktion und iterable als Argumente
- iterable zurück
- Funktion angewendet

  auf alle Elemente

```
1  def double(n):
2      return n * 2

4  numbers = [5, 6, 7, 8]
5  result = map(double, numbers)
6  print(list(result))

[10, 12, 14, 16]
```

map / filter

filter()

- Funktion und iterable als Argumente
- Selektion von Elementen

```
1  def is_even(n):
2      return n % 2 == 0

4  numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

6  even_numbers = filter(is_even, numbers)
7  print("Even numbers:", list(even_numbers))

Even numbers: [2, 4, 6, 8, 10]
```

map / filter

filter()

- Selektion von Elementen

```
1 def ab_k( z ):
2     return z < "k"
3
4 s = "axbycz"
5 print( "".join( filter( ab_k, s ) ) )
6

Ausgabe:
abc
```

map / filter

Aufgabe

Schreiben Sie eine Funktion laengen(), die

- eine Liste von Strings als Argument nimmt
- ein Tupel zurückgibt, das besteht aus
  - einer Liste mit den Längen aller Strings, die länger als vier sind
  - einem String, der die Längen aller Strings, die länger als vier sind, enthält, getrennt von drei Bindestrichen
- Bsp: ['abcde', 'ggf', '236788'] → ([5,6], '6---5')