Komplexe Datentypen II

Komplexe Datentypen

Mehrere eingebaute Funktionen

für sequentielle Datentypen

Funktionieren meist

- wenn Elemente sequentiell geordnet

Komplexe Datentypen

Länge einer Liste

- eingebaute Funktion len()

```
>>> len(['a', 3, 4, "Baum"])
4

>>> len("Baum")
4
```

Komplexe Datentypen

Kleinstes und größtes Element

- eingebaute Funktionen min() und max()

```
>>> min(li)
1

>>> max(li)
12

>>> min("abcde")
'a'

>>> min("aaabcde")
'a'
```

Komplexe Datentypen

Kleinstes und größtes Element

- eingebaute Funktionen min() und max()
- nur wenn

  Ordnungsrelation

```
>>> min([3, 4 ,"ab"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances
of 'str' and 'int'

>>> min([3, 4 ,7.0])
3

>>> max([3, 4 ,7.0])
7.0
```

Komplexe Datentypen

Aufgabe

Erstellen Sie eine Funktion mit zwei Parametern.

- Der erste Parameter ist eine Liste von Ganzzahlen.
- Der zweite Parameter ist optional.
- Fehlt der zweite Parameter oder ist er nicht False, dann liefern Sie vom Maximum der Listenelemente und der Länge der Liste den größeren Wert.
- Ist der zweite Parameter False, dann liefern Sie vom Minimum der Listenelemente und der Länge der Liste den kleineren Wert.

Komplexe Datentypen

Element suchen

- index() liefert die Position (nicht Wert)
- zwei optionale Parameter für Suche in Teilliste
- auch hier negative Werte möglich

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

>>> li.index(4)
3
>>> li.index(7)
6

>>> li.index(7, 3, 10)
6

>>> li.index(7, 3, -3)
6
```

Komplexe Datentypen

Element suchen

- index() liefert Fehler, wenn Wert nicht vorhanden

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

>>> li.index(44)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 44 is not in list
```

Komplexe Datentypen

Elemente zählen

- Die Funktion count() zählt die Vorkommen eines Elementes

```
>>> li = [1,2,2,3,3,3,4,4,4,4]

>>> li.count(2)
2

>>> li.count(4)
4

>>> s = "abbcccddd"

>>> s.count("c")
3
```

Komplexe Datentypen

Aufgabe

Schreiben Sie eine Funktion min_max_diff(), die

- ein Argument hat (eine Liste)
- den Abstand zwischen den Positionen ihres größten und kleinsten Elements als positive, ganze Zahl zurückliefert

Komplexe Datentypen

Wir betrachten eine Reihe von Methoden nur für Listen

- verändern die Liste
- daher nicht für immutable Strings und Tupel

Komplexe Datentypen

append()

- hängt ein Element

  am Ende der Liste an

```
>>> li = [1,2,3]

>>> li.append(4)

>>> li.append(5)

>>> li
[1, 2, 3, 4, 5]
```

Komplexe Datentypen

extend()

- hängt mehrere Elemente am Ende der Liste an
- Parameter iterierbares Objekt

```
>>> li = [1,2,3]

>>> li.extend([4,5])

>>> li
[1, 2, 3, 4, 5]

>>> li.extend("abc")

>>> li
[1, 2, 3, 4, 5, 'a', 'b', 'c']
```

Komplexe Datentypen

insert()

- fügt ein Element an Index in Liste ein
- Parameter erst Index, dann Element

```
>>> li = [1,2,3]

>>> li.insert(1,"A")

>>> li
[1, 'A', 2, 3]

>>> li.insert(1,"B")

>>> li
[1, 'B', 'A', 2, 3]
```

Komplexe Datentypen

insert()

- ist Index zu gross

  oder zu klein

  wird an letzter bzw.

  erster Stelle eingefügt

```
>>> li
[1, 'B', 'A', 2, 3]

>>> li.insert(77,"C")

>>> li
[1, 'B', 'A', 2, 3, 'C']

>>> li.insert(-77,"D")

>>> li
['D', 1, 'B', 'A', 2, 3, 'B']
```

Komplexe Datentypen

pop()

- enfternt Element an gegebenem Index
- ohne Index: letztes Element
- liefert Element zurück

```
>>> li = [1,2,3,4,5]

>>> li.pop()
5

>>> li.pop(1)
2

>>> li
[1, 3, 4]
```

Komplexe Datentypen

pop()

- enfternt Element an gegebenem Index
- ohne Index: letztes Element
- liefert Element zurück
- ungültiger Index: Fehler

```
>>> li = [1,2,3,4,5]

>>> li.pop()
5

>>> li.pop(1)
2

>>> li
[1, 3, 4]

>>> li.pop(77)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

Komplexe Datentypen

remove()

- entfernt erstes Vorkommen eines Elements
- Fehler wenn nicht vorhanden

```
>>> li = [1,2,1,4,1]

>>> li.remove(1)

>>> li
[2, 1, 4, 1]

>>> li.remove(1)

>>> li
[2, 4, 1]
```

Komplexe Datentypen

reverse()

- kehrt Reihenfolge um
- kein Argument

```
>>> li = [1,2,3,4,5]

>>> li.reverse()

>>> li
[5, 4, 3, 2, 1]
```

Komplexe Datentypen

reverse()

- kehrt Reihenfolge um
- kein Argument
- verändert Originalliste,

  nicht Kopie wie [::-1]

```
>>> li = [1,2,3,4,5]

>>> li.reverse()

>>> li
[5, 4, 3, 2, 1]

>>> li[::-1]
[1, 2, 3, 4, 5]
```

Komplexe Datentypen

Aufgabe

- Schreiben Sie eine Funktion remove_all(), die
- als Argument eine Liste und

  ein Element x nimmt

```
remove_all( [1,2,3,2,5], 2)

Ausgabe:
[1, 3, 5]
```

- Rückgabewert ist die Liste, aus der alle Vorkommen von x gelöscht wurden
- wir gehen davon aus, dass x vorkommt

Komplexe Datentypen

sort([key, reverse])

- sortiert Liste
- "Standard"-Ordnung für geg. Datentyp
- falls keine Ordnung: Fehler

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

>>> li.sort()

>>> li
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Komplexe Datentypen

sort([key, reverse])

- Parameter key ist

  Funktion,

  die getrennt für beide

  Elemente aufgerufen

  Vergleichswert liefert

```
>>> l = ["Katharina", "Peter", "Jan",
"Florian", "Paula", "Ben"]

>>> l.sort(key=len)

>>> l
['Jan', 'Ben', 'Peter', 'Paula',
'Florian', 'Katharina']
```

Komplexe Datentypen

sort([key, reverse])

- Im Bsp. werden nicht Strings verglichen sondern Werte von len()

```
>>> l = ["Katharina", "Peter", "Jan",
"Florian", "Paula", "Ben"]

>>> l.sort(key=len)

>>> l
['Jan', 'Ben', 'Peter', 'Paula',
'Florian', 'Katharina']
```

Komplexe Datentypen

sort([key, reverse])

- Parameter "reverse"

  kehrt Reihenfolge um

- Beide sind
- Schlüsselwortparameter

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

>>> li.sort(reverse=True)

>>> li
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Komplexe Datentypen

Aufgabe

Schreiben Sie eine Funktion zweiter_bst(li), die

- eine Liste von Strings als Argument nimmt,
- diese Liste nach dem zweiten Buchstaben sortiert und zurückliefert.

Beispiel: ["Bimm", "Bamm", "Bumm"] -> ["Bamm", "Bimm", "Bumm"]

Komplexe Datentypen

Listen sind iterable

- können mit for-Schleifen durchlaufen werden

```
1 li = [ "a", "b", "c", "d"]
2
3 for bst in li:
4     print( bst , end = "-" )

Ausgabe:
a-b-c-d-
```

Komplexe Datentypen

Listen sind iterable

- können mit for-Schleifen durchlaufen werden

- kein Index nötig
- Werte in Laufvariable

```
1 li = [ "a", "b", "c", "d"]
2
3 for bst in li:
4     print( bst , end = "-" )

Ausgabe:
a-b-c-d-
```

Komplexe Datentypen

Listen sind iterable

- können mit for-Schleifen durchlaufen werden

- Laufvariable kann

  Typ ändern

```
1 li = [ "a", 1, 2.3, True]
2
3 for bst in li:
4     print( bst , end = "-" )
```

Komplexe Datentypen

Listen sind iterable

- wenn Index benötigt:
- enumerate liefert Tupel aus Index und Wert

```
1  li = [ 'a', 'b', 'c', 'd' ]
2
3  for paar in enumerate( li ):
4      print( paar )

(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
```

Komplexe Datentypen

Listen sind iterable

- Tupel von enumerate können direkt in zwei Laufvariablen entpackt werden

```
1  li = [ 'a', 'b', 'c', 'd' ]
2
3  for ind, wert in enumerate( li ):
4      print( ind, " mit Wert ", wert)
```

```
0  mit Wert  a
1  mit Wert  b
2  mit Wert  c
3  mit Wert  d
```

Komplexe Datentypen

Aufgabe

Schreiben Sie eine Funktion mit einer Liste ganzer Zahlen als Argument, die

- jedes Element um den Wert seines Index erhöht