Komplexe Datentypen

Komplexe Datentypen

- Bisher: eine Variable – ein Wert

Komplexe Datentypen

- Bisher: eine Variable – ein Wert
- oder nicht?

```
>>> s = "abcdefg"
>>> s[2]
'c'
>>> s[4]
'e'
>>> s[2:4]
'cd'
```

- Zeichenkette ist Liste von Buchstaben

Komplexe Datentypen

- Bisher: eine Variable – ein Wert
- oder nicht?

```
>>> s = "abcdefg"
>>> s[2]
'c'
>>> s[4]
'e'
>>> s[2:4]
'cd'
```

```
>>> range(8)
```

- Zeichenkette ist Liste von Buchstaben

Komplexe Datentypen

Zeichenkette ist Liste von Buchstaben

- solche Datentypen "sequentiell"

  Wir betrachten
- Listen
- Tupel
- Strings

Komplexe Datentypen

sequentielle Datentypen

- erlauben Zugriff über Index
- Elemente haben feste Reihenfolge (Sequenz)

```
>>> s = "abcdefg"

>>> s[2]
'c'

>>> s[4]
'e'

>>> s[2:4]
'cd'
```

Komplexe Datentypen

Listen

- werden mit eckigen Klammern definiert
- Elemente mit Komma getrennt

```
>>> li = [ 6, "aus", 49]
>>> li2 = [ 1, 2, 3 ]
```

Komplexe Datentypen

Listen

- werden mit eckigen Klammern definiert
- können Elemente verschiedenen Typs enthalten
- Index startet bei 0

```
>>> li = [ 6, "aus", 49]
>>> li2 = [ 1, 2, 3 ]

>>> type(li)
<class 'list'>

>>> type(li[1])
<class 'str'>

>>> type(li[0])
<class 'int'>
```

Komplexe Datentypen

Aufgabe

Erstellen Sie eine Liste mit den Elementen "a", 27, True und "ende".

Greifen Sie auf das Element an der zweiten Position zu und geben Sie dieses aus.

Komplexe Datentypen

Listen können verändert werden

(mutable)

Strings können nicht verändert werden

(immutable)

Komplexe Datentypen

mutable / immutable

- komplexe Typen oft in beiden Versionen

  Liste / Tupel

  Menge / frozenset

Komplexe Datentypen

Listenoperationen -- Ändern eines Wertes

- Zuweisung an index
- ebenfalls eckige Klammern
- Länge bleibt gleich

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

>>> li[1] = 77

>>> li
[1, 77, 3]
```

Komplexe Datentypen

Listenoperationen -- Ändern eines Wertes

- Zuweisung an index
- ebenfalls eckige Klammern
- Länge bleibt gleich

```
>>> s = "abc"
>>> s
'abc'
>>> s[1] = "B"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```

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

>>> li[1] = 77

>>> li
[1, 77, 3]
```

Komplexe Datentypen

Listenoperationen -- Ersetzen von Teillisten

- Länge mit slicing angeben

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

>>> li[2:4] = [ 8, 8 ]

>>> li
[1, 2, 8, 8, 5]
```

- Längen müssen nicht gleich sein

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

>>> li[2:4] = [ 8, 8, 8, 8 ]

>>> li
[1, 2, 8, 8, 8, 8, 5]
```

Komplexe Datentypen

Slicing

- Angabe von Start- und Endindex
- in eckigen Klammern
- getrennt von Doppelpunkt

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

>>> li[2:4] = [ 8, 8 ]

>>> li
[1, 2, 8, 8, 5]
```

Komplexe Datentypen

Listenoperationen -- Einfügen

- Ersetzen einer leeren Teilliste

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

>>> li[2:2] = [ 8, 8, 8, 8 ]

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

Komplexe Datentypen

Slicing -- Negative Indizes

- Indizes können auch negativ angegeben werden
- Zählen vom Ende

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

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

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

Komplexe Datentypen

Slicing -- Leere Indizes

- weggelassene Indizes ersetzt durch Maximalwert
- Doppelpunkt zeigt an welcher fehlt

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

>>> li[:2]
[1, 2]

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

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

Komplexe Datentypen

Slicing -- Leere Indizes zum Kopieren

- wenn beide Indizes fehlen
- ganze Liste, jedoch

  nicht Original

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

>>> li2 = li[:]

>>> li2[3] = 88

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

>>> li2
[1, 2, 3, 88, 5]
```

Komplexe Datentypen

Zuweisung kopiert nicht

- liefert Referenz, Zeiger
- zeigt auf selbes Objekt

im Speicher

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

>>> li2 = li

>>> li2[3] = 88

>>> li
[1, 2, 3, 88, 5]

>>> li2
[1, 2, 3, 88, 5]
```

Komplexe Datentypen

Bei elementaren Datentypen anders

- Zuweisung erzeugt neue Instanz im Speicher

```
>>> a=5

>>> b=a

>>> b=7

>>> b
7

>>> a
5
```

Komplexe Datentypen

Auch bei String

- Zuweisung erzeugt neue Instanz im Speicher
- hier: weil immutable

```
>>> s="abc"

>>> t=s

>>> t="TTT"

>>> t
'TTT'

>>> s
'abc'
```

Komplexe Datentypen

Slicing mit Schrittweite

- Wird ein dritter Parameter angegeben (nach zweitem Doppelpunkt), so werden Elemente übersprungen

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

>>> li[::3]
[1, 4, 7, 10]

>>> li[2::3]
[3, 6, 9]
```

Komplexe Datentypen

Slicing mit negativer Schrittweite

- bei negativer Schrittweite

  Start von hinten
- D.h. erster Wert muss

  größer sein
- Defaults angepasst

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

>>> li[::-1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

>>> li[8:2:-1]
[9, 8, 7, 6, 5, 4]

>>> li[8:2:-2]
[9, 7, 5]
```

Komplexe Datentypen

Öffnen Sie die Kommandozeile und starten Sie Python. Weisen Sie folgende Liste einer Variable zu:

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

Finden Sie Slicing-Operationen, um folgende Listen zu erzeugen:

```
[5, 6, 7]
[11, 8, 5]
[]
[1,2,3,4,5,6,7,8,9,10,11]
[11,9,7,5,3,1]
```

Komplexe Datentypen

Öffnen Sie die Kommandozeile und starten Sie Python. Weisen Sie folgende Liste einer Variable zu:

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

Finden Sie Anweisungen, die die Liste in folgende Listen ändern:

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

Komplexe Datentypen

Listen können auch Listen als Elemente enthalten.

```
1 li = [1, 2, 3,
2       [ "a" , "b", "c"],
3       5, 6, 7]
```

Komplexe Datentypen

Listen können auch Listen als Elemente enthalten.

```
1 li = [1, 2, 3,
2       [ "a" , "b", "c"],
3       5, 6, 7]
4
5 print( li[3][1])

Ausgabe:
b
```

Zugriff über mehrere eckige Klammern

Komplexe Datentypen

Aufgabe

Definieren Sie die Funktion multimult(), die

- beliebig viele Ganzzahlen als Parameter nimmt
- alle Parameter miteinander multipliziert
- und das Ergebnis zurückliefert.