Wiederholung: Listen, Referenzen

Symbolische Programmiersprache

Benjamin Roth and Annemarie Friedrich

Wintersemester 2016/2017

Centrum für Informations- und Sprachverarbeitung
LMU München

Wiederholung

Kommandozeilenargumente & Dateien

Listen

Variablen, Werte, Referenzen

Kommandozeilenargumente

clbeispiel.py

```
1  import sys
2
3  # Aufruf in der bash / cli etc:
4  # python3 clbeispiel.py hallo welt
5
6  print(sys.argv)
```

Gibt aus:

```
1  ['clbeispiel.py', 'hallo', 'welt']
```

- `sys.argv` ist eine Liste, die die Kommandozeilenargumente als Strings enthält
- Das erste Element ist der Name des Python-Skripts

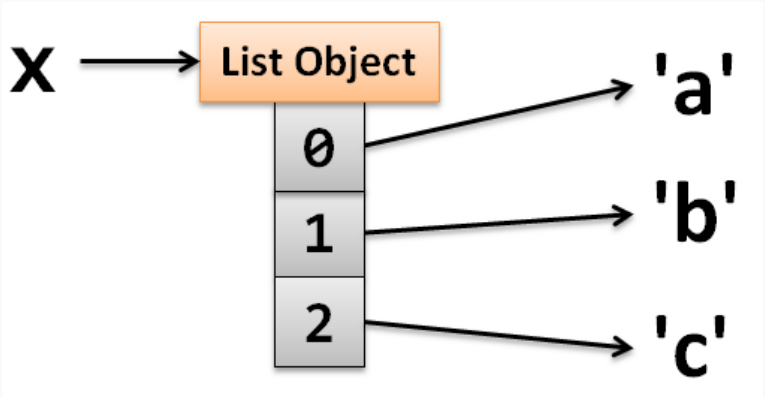
Kommandozeilenargumente & Dateien

clbeispiel.py

```
1  import sys
2  # Aufruf in der bash / cli etc:
3  # python3 clbeispiel.py text.txt
4
5  with open(sys.argv[1]) as f:
6      for line in f:
7          print(line)
```

Listen-Objekte

```
1  x = ['a', 'b', 'c']
2  print("x[0] is: ", x[0])
3  print("x[1] is: ", x[1])
4  print("x[2] is: ", x[2])
```



Listen: Indizes

```
1  myList = ["a", "b", "c", "d", "hello"]
2  myList[3] = "world"
```

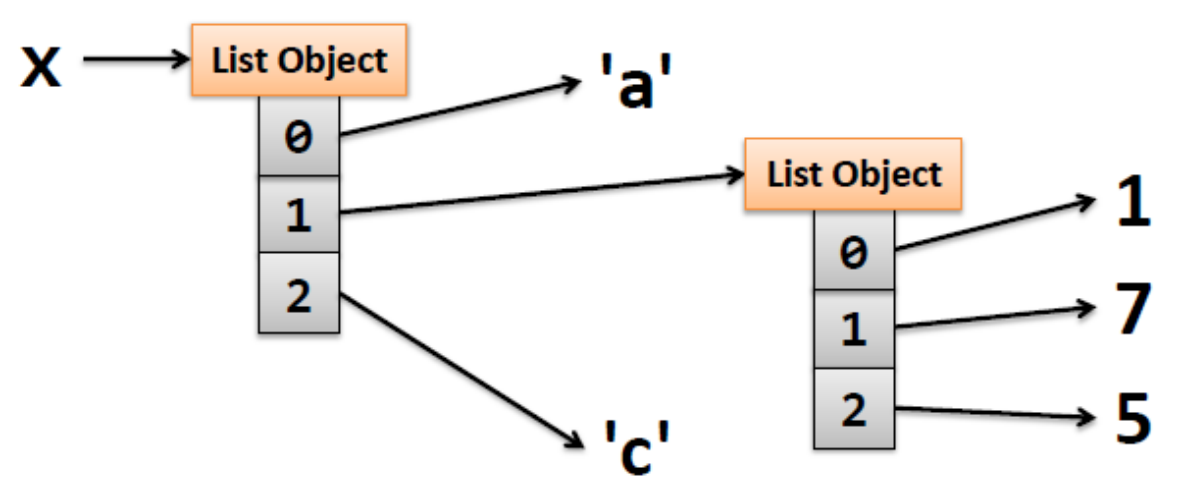
0	1	2	3	4
“a”	”b”	”c”	”d”	”hello”
-5	-4	-3	-2	-1

Was wird hier ausgegeben?

```
1  print(myList[1])
2  print(myList[4])
3  print(myList[-2])
4  print(myList[4][1])
```

Mehrdimensionale Listen

```
1  x = ['a', [1, 7, 5], 'c']
2  print("x[0] is: ", x[0])
3  print("x[1] is: ", x[1])
4  print("x[2] is: ", x[2])
5  print("x[1][0] is: ", x[1][0])
6  print("x[1][1] is: ", x[1][1])
7  print("x[1][2] is: ", x[1][2])
```



Mehrdimensionale Listen

```
1  myList = ["a", "b", [1, 2, 3], "d", "e"]
2  myList[3] = [4, 5, 6]
```

0	1	2	3	4
“a”	“b”	$[1,2,3]$	$[4,5,6]$	“e”

a) Was wird hier ausgegeben?

```
1  print(myList[2][0])
```

b) Wie greift man auf die ‘5’ zu?

c) Was passiert in den folgenden Fällen?

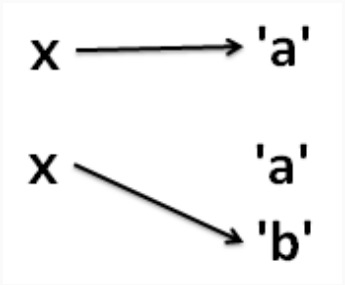
```
1  print(myList[5])
2  print(myList[2][3])
```

Mutable vs. immutable types

mutable = veränderbar; immutable = nicht veränderbar

Mutable objects	Immutable objects
list	integer, float, string, boolean, ...
can be changed (items can be added, removed, modified)	never changes, assignments result in new objects

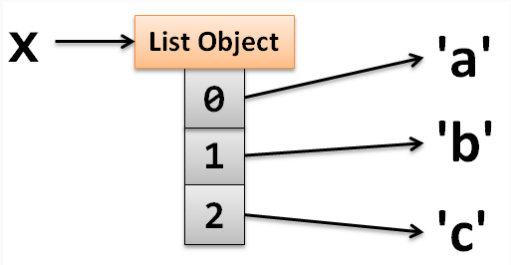
```
1  x = 'a'
2  x = 'b'
3  # the string object 'a' is NOT
4  # modified:
5  # 'b' is a new string object!
```

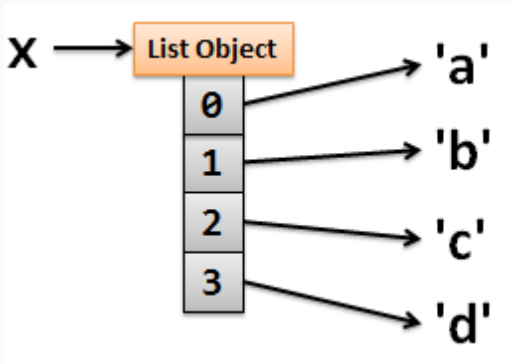


Mutable vs. immutable types

Mutable objects	Immutable objects
list	integer, float, string, boolean, ...
can be changed (items can be added, removed, modified)	never changes, assignments result in new objects

```
1  x = ['a', 'b', 'c']
2  x.append('d')
3  # x still points to the
4  # same list object,
5  # which has been modified!
```

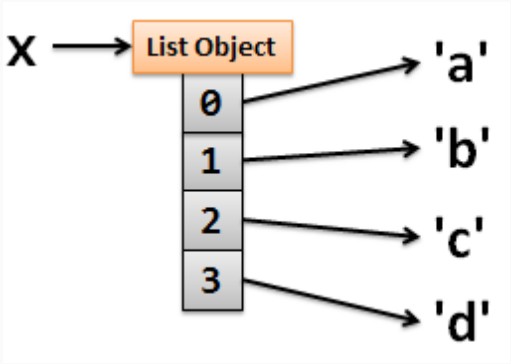


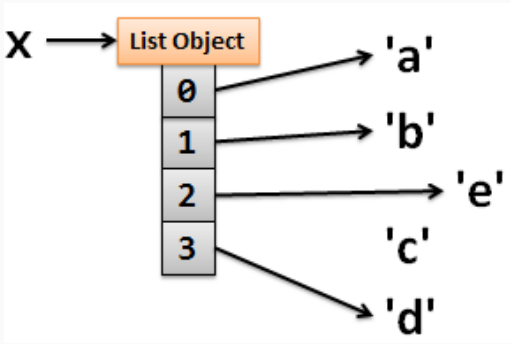


Mutable vs. immutable types

Mutable objects	Immutable objects
list	integer, float, string, boolean, ...
can be changed (items can be added, removed, modified)	never changes, assignments result in new objects

```
1  x = ['a', 'b', 'c']
2  x.append('d')
3  x[2] = 'e'
4  # x still points to the
5  # same list object,
6  # which has been modified!
7  # the string object 'c'
8  # has not been modified,
9  # 'e' is a new string
10 # object!
```





Methoden von Listen

- **Methoden** = Funktionen, die “auf ein Objekt” angewendet werden
- `someObject.methodName(parameters)`
- verändern normalerweise das Objekt, “auf dem sie aufgerufen” werden

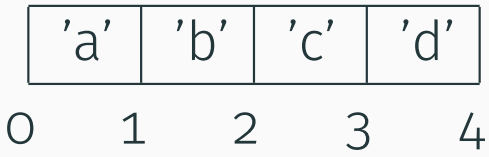
```
1  someList = [1, 2, 3]
2  someList.append(5)
3
4  # appends 5 to the list
5  >>> print(someList)
6  [1, 2, 3, 5]
```

Noch mehr Methoden von Listen

What are the effects of the following list functions?
For each line of the program, draw the current list as a table and write a short documentation explaining what each of the functions does. Freiwillig – Tutorium?

```
 1  myList = [3, 2, 6, 1, 8]
 2  myList.reverse()
 3  x = len(myList)
 4  myList.sort()
 5  myList.insert(2, 5)
 6  myList.sort(reverse=True)
 7  myList.append(3)
 8  x = myList.count(3)
 9  myList.remove(3)
10  x = myList.pop()
11  y = myList.pop()
12  z = 4 in myList
13  myList = myList + [7, 8, 9]
14  del myList[:]
```

Slicing



erstellt **Kopien** der Listen-Objekte!

```
 1  >>> myList = ['a', 'b', 'c', 'd']
 2  >>> myList[0:3]
 3  ['a', 'b', 'c']
 4  >>> myList[2:3]
 5  ['c']
 6  >>> myList = ['a', 'b', 'c', 'd']
 7  >>> myList[0:3]
 8  ['a', 'b', 'c']
 9  >>> myList[2:4]
10  ['c', 'd']
11  >>> myList[1:]
12  ['b', 'c', 'd']
13  >>> myList[:3]
14  ['a', 'b', 'c']
15  >>> myList[:]
16  ['a', 'b', 'c', 'd']
```

String Immutability

- Strings sind auch Sequenzen (von Strings: je ein Zeichen)
- Wir können auf die Zeichen eines Strings zugreifen:

```
1  >>> myString = "telephone"
2  >>> print(myString[2])
3  l
4  >>> print(myString[4:]) # copy!
5  phone
```

- Strings sind immutable: unveränderbare Sequenzen
  `myString[0] = "T"` $\Rightarrow$ DOES NOT WORK!
- Konkatenation erstellt neue String-Objekte.
  `myString = "T"+ myString[1:]`

Shared References (gemeinsame Referenzen)

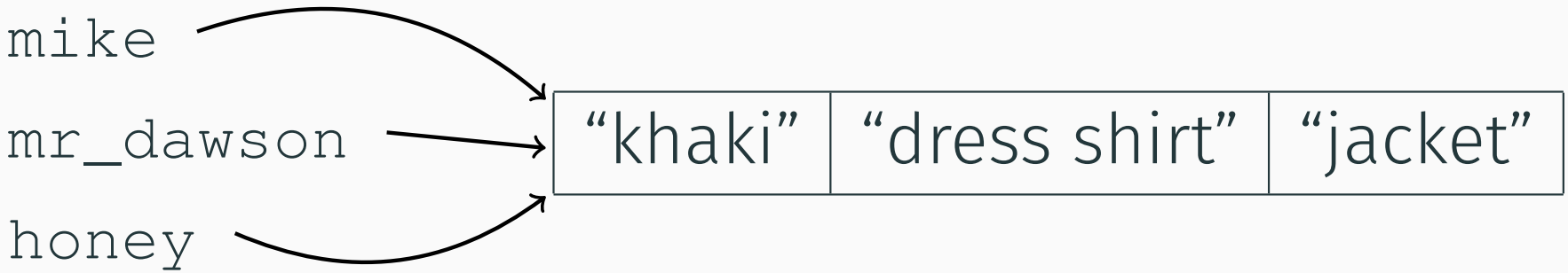
- Variablen *enthalten* nicht die zugehörigen Werte (wie eine Tupperbox Essen enthält)
- Variable *zeigen* auf Positionen im Arbeitsspeicher, so wie ein Name auf eine Person zeigt (der Name enthät die Person nicht), und die Positionen im Arbeitsspeicher enthalten die Werte.

```
1  >>> mike = ["khakis", "dress shirt", "jacket"]
2  >>> mr_dawson = mike
3  >>> honey = mike
4  >>> mike
5  ['khakis', 'dress shirt', 'jacket']
6  >>> mr_dawson
7  ['khakis', 'dress shirt', 'jacket']
8  >>> honey
9  ['khakis', 'dress shirt', 'jacket']
```

Beispiel siehe M. Dawson, Python Programming for the Absolute Beginner, 3rd edition, Course Technology, p.138

Shared References (gemeinsame Referenzen)

```
1  >>> mike = ["khakis", "dress shirt", "jacket"]
2  >>> mr_dawson = mike
3  >>> honey = mike
```

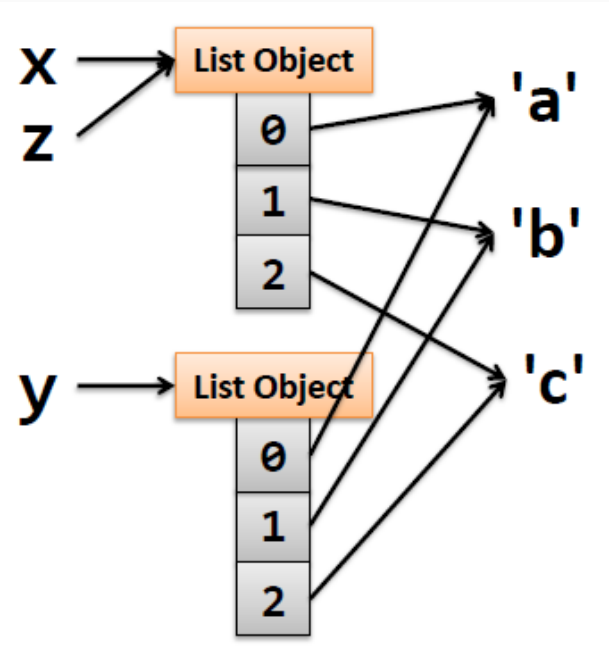



```
1  >>> honey[2] = "red sweater"
2  >>> honey
3  ['khakis', 'dress shirt', 'red sweater']
4  >>> mike
5  ['khakis', 'dress shirt', 'red sweater']
```

Kopien von Listen (Shallow Copy)

- Slicing erstellt eine *shallow copy* einer Liste.
- Eine *shallow copy* erstellt ein neues Listen-Objekt und fügt in dieses Referenzen auf die gleichen Objekte, auf die das Original verweist, ein. (`http://docs.python.org/3.2/library/copy.html`)
- Der **is**-Operator gibt an, ob zwei Variablen auf dasselbe Objekt zeigen oder nicht.

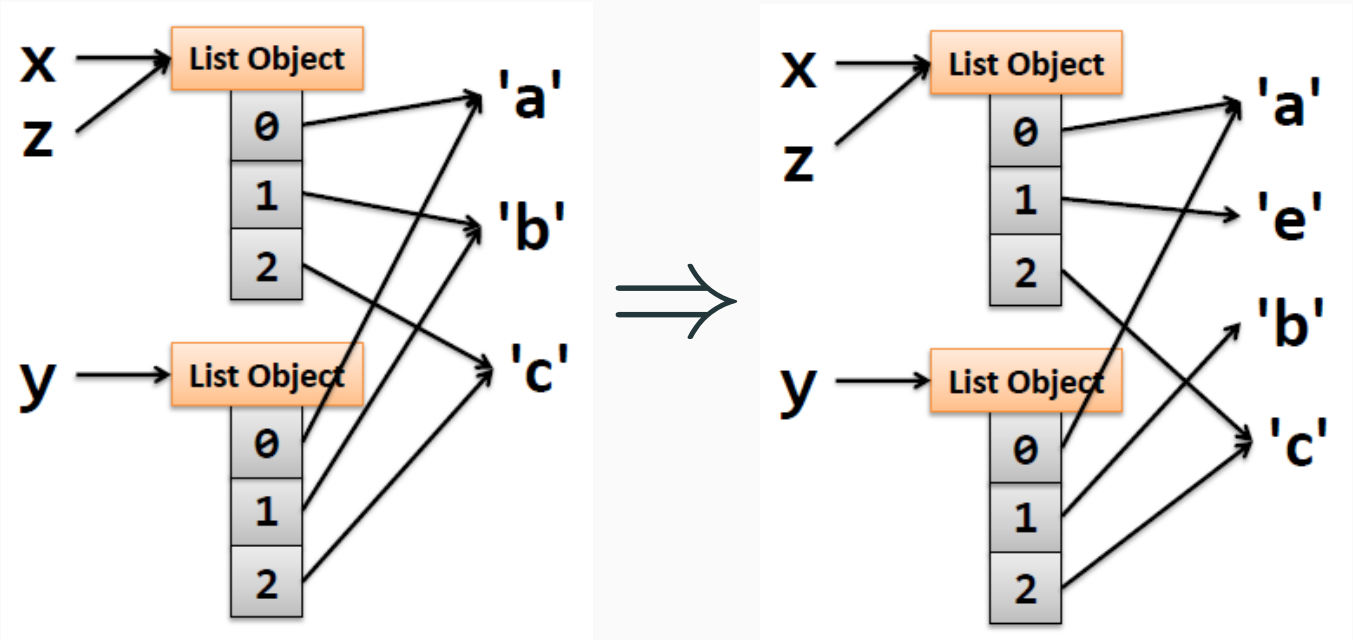
```
1  >>> x = ['a', 'b', 'c']
2  >>> z = x
3  >>> y = x[:]
4  >>> y
5  ['a', 'b', 'c']
6  >>> z is x
7  True
8  >>> y is x
9  False
```



Listen kopieren (Shallow Copy)

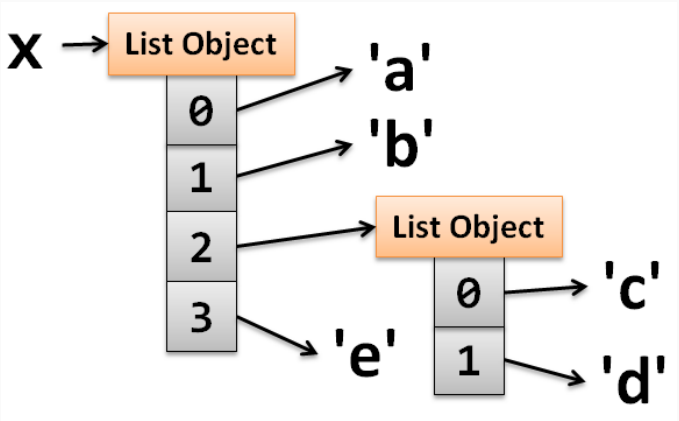
```
1  >>> x = ['a', 'b', 'c']
2  >>> z = x
3  >>> y = x[:]
4  >>> y
5  ['a', 'b', 'c']
```

```
1  >>> x[1] = 'e'
2  >>> x
3  ['a', 'e', 'c']
4  >>> y
5  ['a', 'b', 'c']
6  >>> z
7  # what is printed?
```



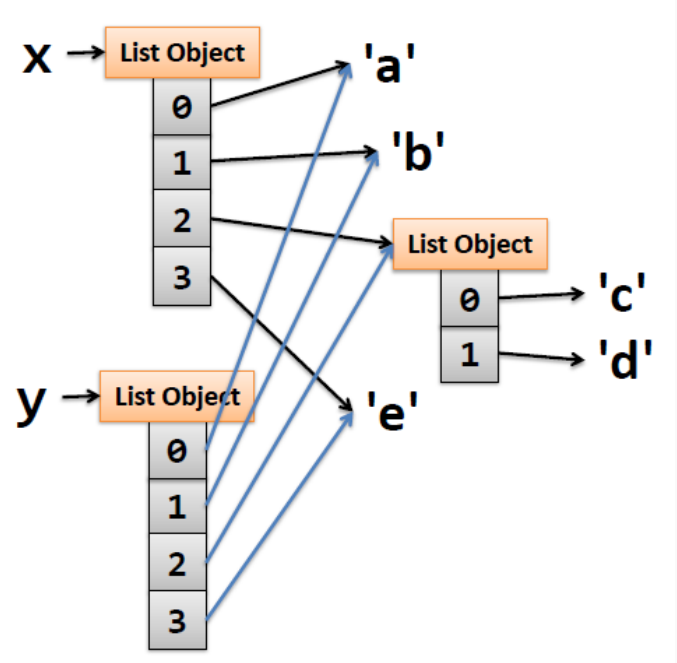
Shallow Copy einer verschachelteten Liste

```
1  >>> x = ['a', 'b', ['c', 'd'], 'e']
```



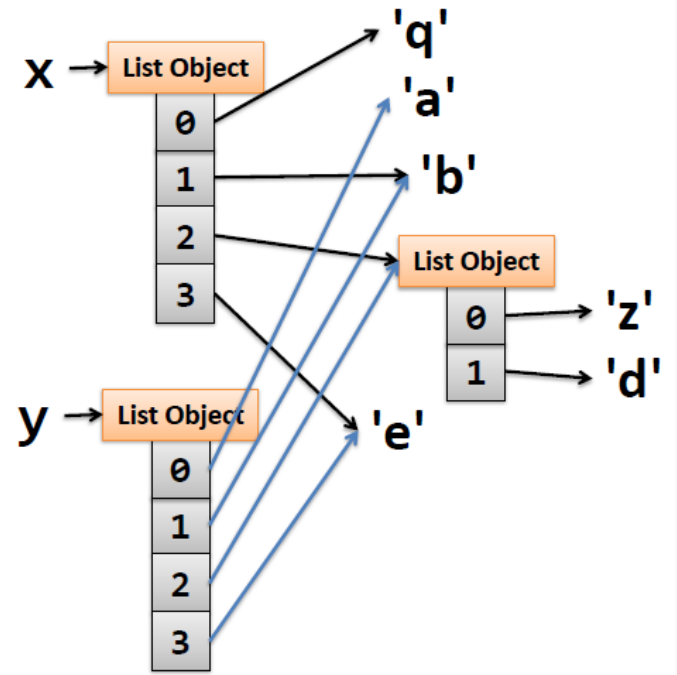
Shallow Copy einer verschachelteten Liste

```
1  >>> x = ['a', 'b', ['c', 'd'], 'e']
2  >>> y = x[:]
3  >>> y
4  ['a', 'b', ['c', 'd'], 'e']
```



Shallow Copy einer verschachelteten Liste

```
1   >>> x[2][0] = 'z'
2   >>> x
3   ['a', 'b', ['z', 'd'], 'e']
4   >>> y
5   ['a', 'b', ['z', 'd'], 'e']
6   >>> x[0] = 'q'
7   # a new string object 'q' is
8   # created, 'a' is not changed
9   >>> x
10  ['q', 'b', ['z', 'd'], 'e']
11  >>> y
12  ['a', 'b', ['z', 'd'], 'e']
```

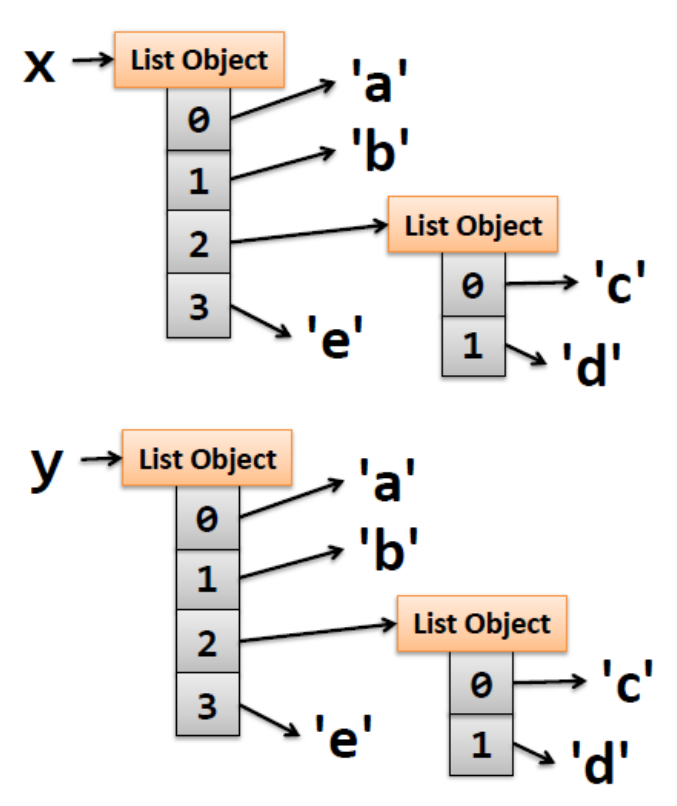


Deep Copy (“tiefe Kopie”) einer verschachtelten Liste

- Eine *deep copy* erstellt ein neues Listen-Objekt und fügt in dieses Referenzen auf (ebenfalls erstellte) *Kopien* der Objekte des Originals ein. Achtung: kann ‘teuer’ sein!

  (http://docs.python.org/3.2/library/copy.html)

```
>>> x = ['a', 'b',
['c', 'd'], 'e']
>>> from copy import deepcopy
>>> y = deepcopy(x)
>>> y
['a', 'b', ['c', 'd'], 'e']
>>> x[0] = 'q'
>>> y[2][1] = 'z'
>>> x
['q', 'b', ['c', 'd'], 'e']
>>> y
['a', 'b', ['c', 'z'], 'e']
```



Exkurs: Werte vergleichen

```
1  >>> x = [1, [2, 3], 4]
2  >>> y = x[:] # make a shallow copy of x
3  >>> # Do x and y contain the same values?
4  >>> x == y
5  True
```

- **`var1 == var2`** gibt aus, ob die Werte, auf die **`var1`** und **`var2`** gleich sind (egal ob sie genau dasselbe Objekt sind oder nicht).

Exkurs: Referenzen vergleichen

```
1  >>> x = [1, [2, 3], 4]
2  >>> y = x[:] # make a shallow copy of x
3  >>> # Do x and y point to the same memory location?
4  >>> # = Are x and y the same list object?
5  >>> x is y
6  False
7  >>> # Do x and y contain the same sublist?
8  >>> x[1] is y[1]
9  True
```

- **`var1 is var2`** sagt uns, ob **`var1`** und **`var2`** auf dieselbe Adresse im Arbeitsspeicher zeigen.
- *Achtung: Wenn man Werte von unveränderbare (immutable) Typen mit Hilfe des `is`-Operators vergleicht, kann das Verhalten unerwartet sein (auf Grund Python-interner Optimierung). Hier nutzen wir den `is`-Operator um herauszufinden, ob zwei Variablen auf dieselben veränderbaren Werte (z.B. vom Typ list, set etc.) zeigen.*

Exkurs: Referenzen vergleichen

Freiwillige Aufgabe

- Try to make a drawing showing which of the variables point to the same list objects.
- Check your assumptions using the **`is`** operator.

```
>>> x = [[1, 2], [3, [5, 6]], 7]
>>> y = x[0][:]
>>> a = x[0]
>>> b = x[1:][:]
>>> c = x[1][1]
>>> d = b[0][1]
>>> e = b[0]
>>> f = x[1]
```

Fragen?