Listenverarbeitung in Python

Datentypen für Sequenzen von Objekten:
Tupel, Listen und Strings

Tupel und Listen sind analog zu Haskells Tupel und Listen:

$(1, 2, 3)$	3-Tupel aus den Zahlen 1,2,3,
$[1, 2, 3]$	Liste der Zahlen 1,2,3

Tupel haben feste Länge.
Listen haben variable Länge

Listenfunktionen in Python

Hauptfunktionalität der Listenfunktionen ist analog zu Haskell.

Präfix-Schreibweise:	I.a. Keine destruktiven Änderungen
Attribut-Schreibweise:	destruktive Änderungen möglich

Hier ein Auszug aus einer Dokumentation zu Python.

Operationen auf allen Arten von Folgen (Listen, Tupel, Strings)

Aus Quick-Reference `http://rgruet.free.fr/#QuickRef`

Operation	Resultat	Bem.
x `in` s	`True` wenn ein Eintrag von $s = x$, sonst `False`	
x `not in` s	`False` wenn ein Eintrag von s gleich x, sonst `True`	
$s + t$	Konkatenation von s und t	
$s * n$	n Kopien von s aneinander gehängt	
$s[i]$	i-es Element von s, Start mit 0	(1)
$s[i : j]$	Teilstück s ab i (incl.) bis j (excl.)	(1), (2)
`len`(s)	Länge von s	
`min`(s)	Kleinstes Element von s	
`max`(s)	Größtes Element von s	

Dokumentation: Bemerkungen

(1) Wenn i oder j negativ ist, dann ist der Index relativ zum Ende des String, d.h. len(s)+i oder len(s)+j wird eingesetzt. Beachte, dass -0 = 0.

(2) Das Teilstück von s von i bis j wird definiert als die Folge von Elementen mit Index k mit `i <= k < j`. Wenn i oder j größer als len(s) sind, nehme len(s). Wenn i weggelassen wurde, nehme len(s). Wenn i $\geq$ j, dann ist das Teilstück leer.

Operation auf Listen (in Attributschreibweise)

Operation	Resultat
$s[i] = x$	item i von s ersetzt durch x
$s[i : j] = t$	Unterliste s von i bis j ersetzt durch t
`del` $s[i : j]$	dasselbe wie $s[i : j] = []$
s.`append`(x)	dasselbe wie s[`len`(s) : `len`$(s)] = [x]$
s.`extend`(x)	dasselbe wie s[`len`(s) : `len`$(s)] = x$
s.`count`(x)	Return: Anzahl der i mit $s[i] == x$
s.`index`(x)	Return: kleinstes i mit $s[i] == x$
s.`insert`(i, x)	dasselbe wie $s[i : i] = [x]$ wenn $i >= 0$
s.`remove`(x)	dasselbe wie $del\ s$[s.`index`(x)]
s.`pop`$([i])$	dasselbe wie $x = s[i]; del\ s[i];$ `return` x
s.`reverse`$()$	Umdrehen von s in place
s.`sort`$([cmpFct])$	Sortiere s in place cmpFct: optional bei eigener nicht Standard cmpFct: return $-1, 0, 1$, wenn $x < y, x = y, x > y$ sein soll

Seiteneffekte der Listen-Funktionen

Python-Listen-Funktionen haben meist Seiteneffekte

`s.count(x)` und `s.index(x)`

haben keine Seiteneffekte.

Die anderen Funktionen modfizieren die Liste direkt.

Weitere Listen-Funktionen

In Präfix-Schreibweise:

Operation	Resultat
`map`(f,s)	Liste [f s[0], f s[1], . . .]
`filter`(p,s)	Liste der Elemente mit $p(s[i]) =$ `True`
s + t	Liste der Elemente aus s und t.
`reduce`(op,s)	$s[0]$ op $s[1]$ op . . .
`reduce`(op,s,init)	Dasselbe wie init op (`reduce`(op,s))
`zip`(s,t)	Liste der Paare der Elemente von s,t.

Rückgabewert i.a. analog zu dem der entsprechenden Haskellfunktion

Beispiele

Wir geben zur Listenverarbeitung in Python einige Beispiele an:

```
>>> range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(5,10)
[5, 6, 7, 8, 9]
>>> len(range(10))
10
>>> len(range(1000000))
Traceback (most recent call last):
  File "<input>", line 1, in ?
MemoryError
>>>len(xrange(1000000))
1000000
>>> a = ['a', 'b','c']
```

```
>>> a
['a', 'b', 'c']
>>> b = [3,4,5]
>>> a.extend(b)
>>> a
['a', 'b', 'c', 3, 4, 5]   ## a ist modifiziert
>>> b
[3, 4, 5]
>>> a.append(b)
>>> a
['a', 'b', 'c', 3, 4, 5, [3, 4, 5]]
>>> a.reverse()
>>> a
[[3, 4, 5], 5, 4, 3, 'c', 'b', 'a']
>>> b
[3, 4, 5]              ## b bleibt
>>> b.reverse()
>>> a
[[5, 4, 3], 5, 4, 3, 'c', 'b', 'a']        ##  a ist modifiziert!
```

`a.extend(b)` verwendet eine Kopie der Liste `b`

Schleifen in Python

mittels `while` oder `for`:

Beispiel für `for`:

```
>>> for i in range(1,11):
...   print(i)
...
1
2
3
....
8
9
10
>>>
```

Beispiele zu vordefinierten Listenfunktionen

Listenfunktionen	`map`, `filter`, und `reduce`
analog zu Haskells	`map`, `filter`, und `foldl`

Folgende Funktion `listcopy` erzeugt eine Kopie einer Liste:

```
def id(x): return x
def listcopy(x):
  return map(id, x)

def geradeq(n):
   return n%2 == 0
>>> filter(geradeq,range(20))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> map(quadrat,range(1,10))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [1,2,3]+[11,12,13]
[1, 2, 3, 11, 12, 13]
```

Beispiele zu vordefinierten Listenfunktionen (2)

```
def add(x,y): return x+y

>>> reduce(add,range(100))
4950
def mal(x,y): return x*y
def fakultaetred(n): return reduce(mal,range(1,n+1))
```

Alias-Namens-Problematik

Folgendes ist möglich

Variablen, die in einem Aufruf $f(t_1, \ldots, t_n)$ nicht erwähnt werden, können nach der Auswertung dieses Aufrufs andere Werte haben.

Alias-Problematik (Aliasing):
tritt auf, wenn der gleiche Speicherbereich
unter verschiedenen Namen referenziert wird (s.u.)

Effekt: Der Wert einer Variablen x kann sich im Laufe der Programmbearbeitung verändern,
ohne dass diese Variable in einem Befehl, der den Speicher verändert, vorkommt.

Vorsicht bei

xxy wurde im Aufruf nicht erwähnt, also ist es nach der Auswertung des Aufrufs `f(z)` nicht verändert ??

Alias-Namens-Problematik; Beispiel

```
>>> a = [3,2,1]
>>> b = a
>>> a
[3, 2, 1]
>>> b
[3, 2, 1]
>>> a.sort()
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
```

Alias-Namens-Problematik; Beispiel

```
>>> a = [3,4,5]
>>> b = ['a','b','c']
>>> a.append(b)
>>> a
[3, 4, 5, ['a', 'b', 'c']]
>>> b
['a', 'b', 'c']
>>> b.reverse()      ## hier passierts
>>> a
[3, 4, 5, ['c', 'b', 'a']]
>>> b
['c', 'b', 'a']
```

Modell der internen Darstellung von Listen

Notation und Vereinbarungen

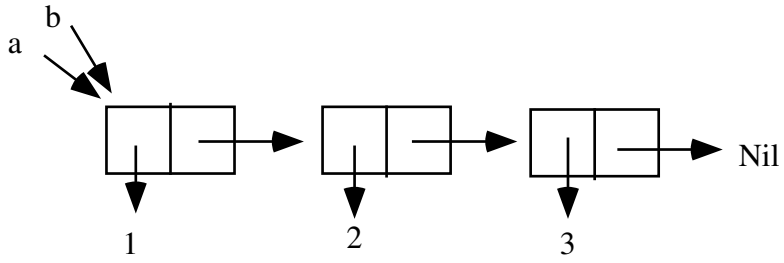
- Pfeile bedeuten Bindungen
-  repräsentiert ein Paar
  (von zwei impliziten Namen)

  Pfeil aus linker Zelle: Bindung des Listenelementes.
  Pfeil aus rechter Zelle: Bindung an die Restliste

Interne Darstellung von Listen

Darstellung der Liste $[1, 2, 3]$:

```
>>> a = [1,2,3]
>>> b = a
```

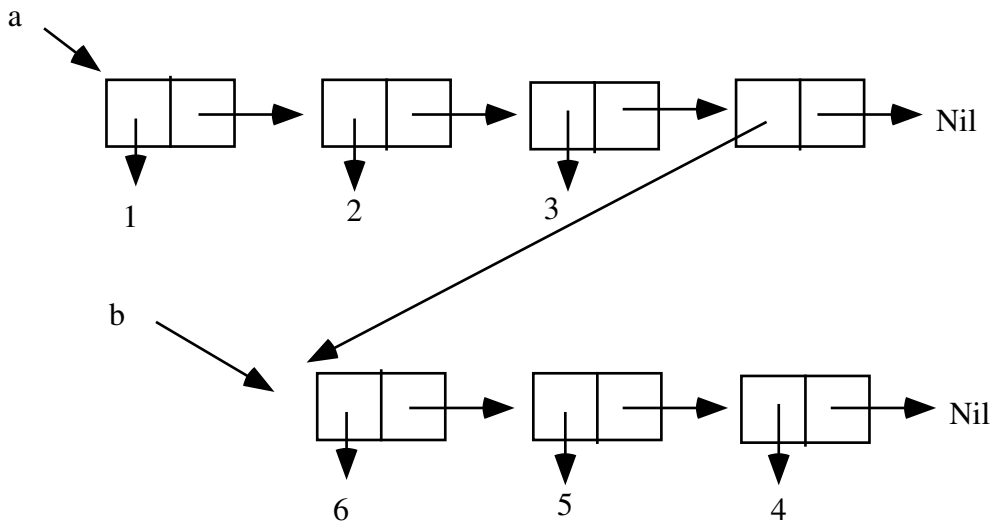


Repräsentation von Listen

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> a.append(b)
>>> a
[1, 2, 3, [4, 5, 6]]
>>> b.reverse()
>>> a
[1, 2, 3, [6, 5, 4]]
>>>
```

Repräsentation von Listen (2)

Bild des Speichers nach den obigen Befehlen:

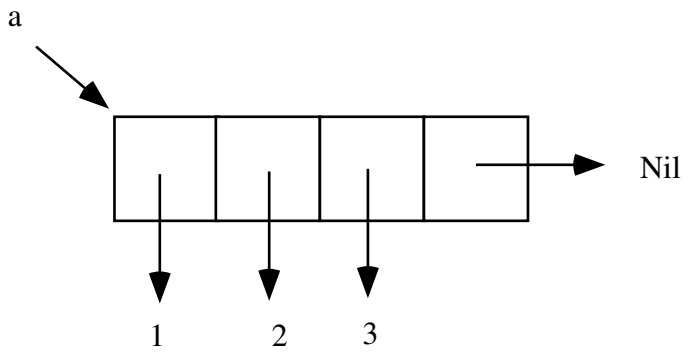


Pythons Listendarstellung

Ist intern optimiert:

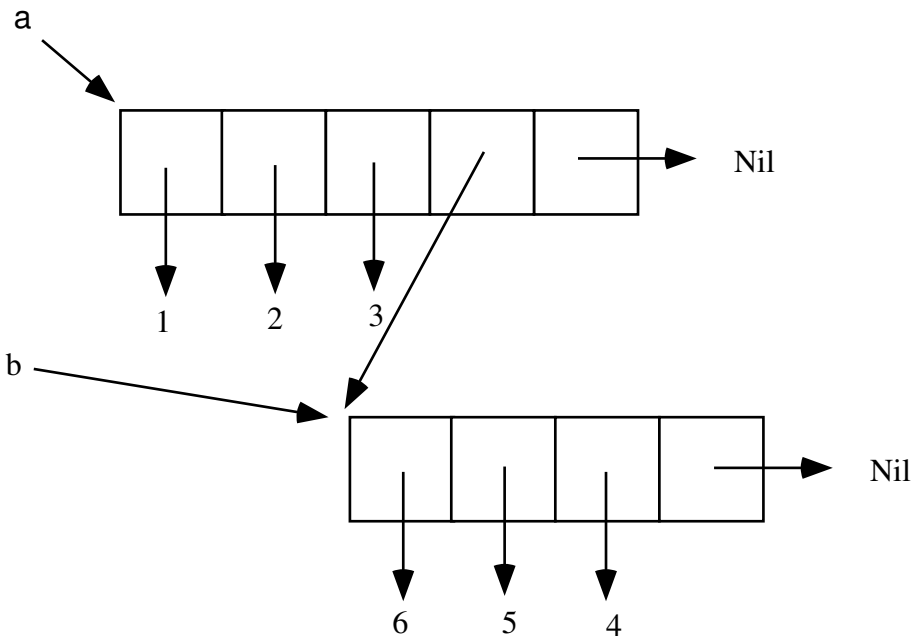
D.h. es gibt eigentlich keine Teillisten.
Das entspricht folgendem Diagramm:

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



Pythons Listendarstellung (2)

```
>>> b = [4,5,6]
>>> a.append(b)
>>> a
[1, 2, 3, [4, 5, 6]]
>>> b.reverse()
>>> a
[1, 2, 3, [6, 5, 4]]
```



Stack in Python

In Python leicht und effizient implementierbar:
`push` b auf Stack a: `a.insert(0,b)`

Benutzung der Listen von links:

```
a.insert(0,b)          ##push(b)
a.pop(0)
```

Benutzung der Listen von rechts:

```
a.append(b)
a.pop()
```

Fehler bei `pop()` auf den leeren Stack

Beispiele: Stack in Python

Verwendung eines Umgebungsstacks zur Auswertung von (Python-)Ausdrücken.

Umdrehen einer Liste (siehe Haskell-Kapitel)

Schlange, Queue

Reihenfolge: first-in; first out (fifo).

Effiziente Implementierung von links oder von rechts:

`a.insert(0,b)`	Links `b` in die Liste `a` einfügen
`a.pop()`	Letztes Element entfernen
`a.append(b)`	Rechts `b` in die Liste `a` einfügen
`a.pop(0)`	Erstes Element entfernen