Python und Listen

Manchmal bekommt man erst dann einen tieferen Einblick und Verständnis, wenn man bei der Entwicklung auf ein Problem stößt. Anlass für dies Problem ist das Konvertieren eines Scheme-(Racket-)Programms zur Anwendung des A*-Verfahrens für das Acht-Puzzle (oft auch Neunerpuzzle genannt) in eine Pythonversion.
Nachdem das Programm fertig gewesen ist, zeigten Tests, dass es auf inakzeptable Weise langsamer als die Schemeversion gearbeitet hat. Nach vielen Versuchen, die notwendigen Optimierungen (rechtzeitiges Entfernen von schlechteren Alternativen usw) zu überprüfen und ggf zu verbessern, bin ich schließlich auf eine einfache Lösung gestoßen.

Was macht Scheme?

Listen sind bei Scheme die Standarddatenstruktur für Sammlungen. Sie sind als einfach verkettete Liste realisiert. Zugriffe erfolgen grundsätzlich vom Kopf her,

- `(car liste)` zum Lesen des ersten Elements (→ `first`)
- `(cdr liste)` zum Zugriff auf die Restliste (→ `rest`)
- `(cons element liste)` zum Erweitern der Liste am Kopf

Eine Bestimmung der Länge einer Liste hat dann zur Folge, dass man sich durch die ganze Liste „hindurchhangeln“ muss beim Zählen.
Dasselbe gilt für jeden anderen Zugriff auf weiter hinten in der Liste stehende Elemente. Ein Anhängen an die Liste erfolgt daher auch durch ein rekursives Durchlaufen der Liste bis zum Ende, um dann dort das neue letzte Listenelement anzuhängen.

Und Python?

Listen sind in Python zwar nicht der einzige Sammlungstyp, dennoch aber sehr wichtig, da sie der einzige Sammlungstyp sind, der modifizierbar ist. Während man Tupel, ein Beispiel dafür ist (1,2,3), nicht modifizieren kann, sondern gegebenenfalls ein neues Objekt erzeugen muss, wie bei (1,2,3)+(4,5) → (1,2,3,4,5), gibt es zu Listen auch Zugriffs-methoden, die das Objekt verändern. Listen stellen die folgenden Methoden bereit, von denen die ersten beiden nur lesend wirken

- index gibt den Index des ersten Auftretens des Objekts in der Liste an
- count die Anzahl des Auftretens des Objekts in der Liste

während die folgenden modifizierend wirken

- append erweitert eine Liste am Ende um ein Objekt
- extend erweitert eine Liste am Ende um eine Sammlungen
- insert erweitert eine Liste an der übergebenen Position um ein Objekt
- pop holt und entfernt das erste Element der Liste
- remove entfernt das erste Auftreten des Objekts in der Liste
- reverse kehrt die Liste um *(„reverse in place“)*
- sort sortiert die Liste, wenn die Objekte eine Ordnungsrelation haben

Ein kurzer Vergleich

Das Beispiel der Methode **`reverse`** ermöglicht leicht ein Verständnis für einen wesentlichen Unterschied bei der Entwicklung von Programmen mit Scheme und Python. Die folgenden Zeilen der Auswertungsumgebung (Shell) von Python zeigen, dass die ursprünglich definierte **`liste`** sich nach Anwendung der Methode verändert hat.

```
>>> liste=[1,2,3]
>>> liste
[1, 2, 3]
>>> liste.reverse()
>>> liste
[3, 2, 1]
```

Das sieht bei Scheme nicht nur anders aus

```
(define meine-liste '(1 2 3))
(define
  (umkehren liste akku)
  (cond
    ((null? liste) akku)
    (else (umkehren (rest liste) (cons (first liste) akku)))
    ))

(writeln (umkehren meine-liste '()))
(writeln meine-liste)
```

sondern liefert mit

```
(3 2 1)
(1 2 3)
```

ein Ergebnis, das zeigt, dass die in der Umgebung (für Scheme unüblich) vordefinierte Liste nicht modifiziert worden ist.

call-by-value gegenüber call-by-reference

Das ist hier auch kein Wunder, da die Liste rekursiv im akku neu aufgebaut worden ist. Entscheidend ist aber, das selbst Änderungen von Liste sich nicht bemerkbar machen würden, da Scheme mit call-by-value arbeitet (beim Aufruf einer Funktion wird für deren Auswertungsteil ein eigenständiges Objekt erzeugt), während Python (ebenfalls Java!) mit call-by-reference (es wir der Funktion nur ein Zeiger auf das Objekt übergeben) arbeitet, was zur Folge hat, dass Modifizierungen im Inneren einer Funktion auch nach außen wirken.
Man kann den Unterschied zeigen, wenn man in die oben angegebene Schemefunktion vor die Rückgabe des Ergebnisses eine set! - Anweisung einfügt:

```
    ((null? liste)
     (set! liste '(A B C))
     (writeln liste)
     akku)
```

→

```
(A B C)
(3 2 1)
(1 2 3)
```

Zum Vergleich ein Python - „Programm“ (ausführlich im Anhang)

```
def umkehren(liste):
    liste.reverse()
    liste.append('A')

liste = [3,1,4,2]
print(liste)
umkehren(liste)
print(liste)
```

liefert

```
[3, 1, 4, 2]
[2, 4, 1, 3, 'A']
```

Was soll‘s?

Für beide Konzeptionen gibt es gute Gründe. Das Scheme-Konzept ist typisch für funktionale Programmiersprachen. Es erfüllt den Zweck, jeder Funktion ihre eigene, gegen außen abgeschirmte Auswertungsumgebung bereit zu stellen und andererseits zu verhindern, dass sie auf die „Außenwelt“ verändernd zugreift.
Das was anderenfalls gemacht wird, reduziert den Verwaltungsaufwand beim Aufruf von Funktionen / Prozeduren / Methoden, da nur eine Zeiger übergeben wird, also die Mitteilung, wo dies Objekt im Speicher gehalten wird.
Das bedeutet dann aber, dass man bei der Programmentwicklung sehr aufpassen muss, dass man nicht ungewollt in einer aufgerufenen Funktion / Prozedur / Methode verändernd auf das übergebene Objekt zugreift. Das zu vermeiden, ist manchmal nicht ganz einfach, man muss es aber im Blick haben.

Zurück zum Laufzeitproblem

Die gemessenen Unterschiede zum Laufzeitverhalten sind für mich verblüffend gewesen. Man sollte erwarten, wenn eine Programmiersprache für Listen vorrangig die Methode **append** anbietet, dass die Datenstruktur optimiert ist für Zugriffe am Ende der Liste.
Dazu gibt es beispielsweise die Möglichkeit, direkte Zugriffsmöglichkeiten auf beliebige Stellen in der Liste anzubieten, in jedem Fall aber speziell auf das Ende der Liste. Ein Objekt der Klasse Liste müsste sich dafür beispielsweise die Adresse des aktuell letzten Elements merken. Das ist bei Python offensichtlich nicht der Fall.
Nachdem ich alle Zugriffe auf Zugriffe am Kopf umgestellt habe, also Elemente immer am Kopf hinzugefügt, gelesen oder entfernt habe, hat sich das Laufzeitverhalten an das der Schemeversion angepasst. Also gilt die Regel:

In Programmen mit aufwändigen Zugriffen auf Listen möglichst immer am Kopf arbeiten.

```
liste=[1,2,3]
temp = liste[0]
liste=liste[1:]
print( liste)
[2, 3]
```

```
l=[1,2,3]
l=['neu']+l
print( l )                    # liefert
['neu', 1, 2, 3]
```

Anhang

Ein ausführlicheres Programm zu call-by-reference, ergänzt um eine Beispiel mit dem Einsatz von **`global`** in der Funktion (Prozedur).

```
# Parameter liste wird uebergeben
print('liste als Parameter')
liste = [3,1,5,6]
print('extern vorher: ',liste)

def umkehren(liste):
    print('intern vorher: ',liste)
    liste.reverse()
    print('intern hinterher: ',liste)
umkehren(liste)

print('extern hinterher: ',liste)

# Parameter liste wird beim Aufruf neu erstellt
print('neu erstellter Parameter')
liste = [3,1,5,6]
print('extern vorher: ',liste)

def umkehren(liste):
    print('intern vorher: ',liste)
    liste.reverse()
    print('intern hinterher: ',liste)
umkehren(liste[:1]+liste[1:])

print('extern hinterher: ',liste)

# global verwenden
print('global verwenden')
liste = [3,1,5,6]
print('extern vorher: ',liste)

def umkehren():
    global liste
    print('intern vorher: ',liste)
    liste.reverse()
    print('intern hinterher: ',liste)
umkehren()
```

Ausgaben:

```
liste als Parameter
extern vorher:  [3, 1, 5, 6]
intern vorher:  [3, 1, 5, 6]
intern hinterher:  [6, 5, 1, 3]
extern hinterher:  [6, 5, 1, 3]

neu erstellter Parameter
extern vorher:  [3, 1, 5, 6]
intern vorher:  [3, 1, 5, 6]
intern hinterher:  [6, 5, 1, 3]
extern hinterher:  [3, 1, 5, 6]

global verwenden
extern vorher:  [3, 1, 5, 6]
intern vorher:  [3, 1, 5, 6]
intern hinterher:  [6, 5, 1, 3]
extern hinterher:  [6, 5, 1, 3]
```