Informatik I

16. Verlinkte Listen

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

18. Januar 2011

Verlinkte Listen

- Wir werden später noch ein weiteres Beispiel zum Thema "Veränderlichkeit von Objekten" sehen. Deshalb hätten wir eigentlich das vorige Kapitel fortsetzen können.
- Für dieses Beispiel benötigen wir allerdings Listen. Mit diesen würde das vorige Kapitel sehr groß werden.
- Überhaupt wird uns die objektorientierte Programmierung weiter begleiten. Trotzdem muss man irgendwo einen Schnitt machen, und hier ist eine gute Stelle.

Verlinkte Listen

- Verlinkte Listen sind im Wesentlichen das, was wir in Scheme Listen nennen. In C++ (z.B.) gibt es ebenfalls Listen, als eingebaute Klasse.

Verlinkte Listen

- Verlinkte Listen sind im Wesentlichen das, was wir in Scheme Listen nennen. In C++ (z.B.) gibt es ebenfalls Listen, als eingebaute Klasse.
- Leider hat ein in Python eingebauter Typ (Klasse) den Namen `list`, und hierbei handelt es sich um eine andere Datenstruktur als das, was wir bisher als Listen bezeichnet haben.

Verlinkte Listen

- Verlinkte Listen sind im Wesentlichen das, was wir in Scheme Listen nennen. In C++ (z.B.) gibt es ebenfalls Listen, als eingebaute Klasse.
- Leider hat ein in Python eingebauter Typ (Klasse) den Namen `list`, und hierbei handelt es sich um eine andere Datenstruktur als das, was wir bisher als Listen bezeichnet haben.
- Deshalb sagen wir statt "Listen" fortan "verlinkte Listen" (linked lists).

Erinnerung: (Verlinkte) Listen in Scheme

Listen in Scheme

Um uns zu erinnern, worum es bei (verlinkten) Listen ging, wiederholen wir Listen in Scheme.

Die leere Liste

In Scheme

Was die Null für die Zahlen, das ist die leere Liste für die Listen; es ist also die Liste mit 0 Elementen.

Wir brauchen ein Symbol für die leere Liste. Saubere Lösung: Wir definieren einen Record.

Die leere Liste

In Scheme

Was die Null für die Zahlen, das ist die leere Liste für die Listen; es ist also die Liste mit 0 Elementen.

Wir brauchen ein Symbol für die leere Liste. Saubere Lösung: Wir definieren einen Record.

Das werden wir in Python nicht so machen: es wird keine eigene Klasse für die leere verlinkte Liste geben.

Die leere Liste
Verwendung

In Scheme

```
    (make-leere-liste)
=> #<record:leere-liste>
```

Um eine leere Liste zu konstruieren, müssen wir `make-leere-liste` auf 0 Argumente anwenden.

Die leere Liste
Verwendung

In Scheme

```
    (make-leere-liste)
=> #<record:leere-liste>
```

Um eine leere Liste zu konstruieren, müssen wir `make-leere-liste` auf 0 Argumente anwenden.

In Python wird es nur einen Listenkonstruktor geben; dieser wird die leere (verlinkte) Liste konstruieren.

Die leere Liste
Sortenprädikat

In Scheme

Das Sortenprädikat `leer?`:

```
    (leer? "Banane")
=> #f
```

Die leere Liste
Sortenprädikat

In Scheme

Das Sortenprädikat `leer?`:

```
   (leer? "Banane")
=> #f
```

Das Pendant dazu wird `is_empty` heißen, allerdings nennen wir es nicht "Typprädikat" o.ä.

Allgemeine Listen

In Scheme

Wir werden gleich eine Sorte für nichtleere (Zahlen-)Listen definieren. Diese Sorte wird `nll` heißen.

Die Sorte für die leere Liste heißt also `leere-liste`, und für nichtleere Listen `nll`. Die Sorte für allgemeine (leere oder nichtleere) (Zahlen-)Listen ist demnach eine gemischte Sorte.

```
(define zahlenliste
  (signature
   (mixed leere-liste nll)))
```

Allgemeine Listen

In Scheme

Wir werden gleich eine Sorte für nichtleere (Zahlen-)Listen definieren. Diese Sorte wird `nll` heißen.

Die Sorte für die leere Liste heißt also `leere-liste`, und für nichtleere Listen `nll`. Die Sorte für allgemeine (leere oder nichtleere) (Zahlen-)Listen ist demnach eine gemischte Sorte.

```
(define zahlenliste
  (signature
   (mixed leere-liste nll)))
```

Gemischte Sorten betrachten wir in Python nicht, und wir werden nur eine Klasse haben, die sich an nichtleeren (verlinkten) Listen orientiert und die leere (verlinkte) Liste auf spezielle Weise darstellt.

Nichtleere Listen: `nll`

In Scheme

Erinnerung: eine (Zahlen-)Liste ist eine Aneinanderreihung von beliebig vielen Zahlen. Man kann aber nicht beliebig viele Zahlen "auf einen Schlag" aneinanderreihen. Deshalb sind nichtleere Listen folgendermaßen definiert:

- Verknüpfe eine Zahl mit der leeren Liste, um eine Liste der Länge 1 zu erhalten.
- Verknüpfe eine Zahl mit einer Liste der Länge 1, um eine Liste der Länge 2 zu erhalten.
- ...

Allgemein setzt sich eine nichtleere Liste der Länge $n+1$ (wobei $n \geq 0$) also zusammen aus einer Zahl und einer Liste der Länge n.

Nichtleere Listen: `nll`

In Scheme

Erinnerung: eine (Zahlen-)Liste ist eine Aneinanderreihung von beliebig vielen Zahlen. Man kann aber nicht beliebig viele Zahlen "auf einen Schlag" aneinanderreihen. Deshalb sind nichtleere Listen folgendermaßen definiert:

- Verknüpfe eine Zahl mit der leeren Liste, um eine Liste der Länge 1 zu erhalten.
- Verknüpfe eine Zahl mit einer Liste der Länge 1, um eine Liste der Länge 2 zu erhalten.
- ...

Allgemein setzt sich eine nichtleere Liste der Länge $n+1$ (wobei $n \geq 0$) also zusammen aus einer Zahl und einer Liste der Länge n.

Dies gilt auch in Python.

Der Record `nll`
Wahl der Namen

In Scheme

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
```

Der Record `nll`
Wahl der Namen

In Scheme

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
```

Wir haben hier zwei Komponenten `kopf` und `rumpf`. Hieran werden wir uns auch beim Entwurf der Klasse `LinkedList` orientieren.

Der Record `nll`
Wahl der Namen

In Scheme

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
```

Wir haben hier zwei Komponenten `kopf` und `rumpf`. Hieran werden wir uns auch beim Entwurf der Klasse `LinkedList` orientieren.

Übrigens nennen manche Autoren den Rumpf tatsächlich Schwanz [OW02].

Klasse und Konstruktion

Die Klasse LinkedList
Attribute und Konstruktor

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
```

- Wir definieren einen Record mit zwei Attributen `data` und `next`. Das erste enthält ein Listenelement, das zweite den Rest der Liste, genauer: eine Referenz (Zeiger) auf das nächste Element der Liste (denken Sie an die Fische und ihre Transponder-Chips).
- Die leere Liste wird so modelliert, dass beide Attribute auf `None` gesetzt werden.
- Der Konstruktor `LinkedList` konstruiert eine leere Liste.

Die leere Liste: Konstruktion

Python-Interpreter

```
>>> from linkedlist import *
>>> x = LinkedList()
>>> print(x.data)
None
>>> print(x.next)
None
```

Test auf Leerheit

Um zu testen, ob eine Liste leer ist, definieren wir eine Methode (da die leere Liste keine eigene Klasse darstellt, nennen wir das nicht “Klassenprädikat” o.ä.)

Test auf Leerheit II

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
    ...

    def is_empty(self):
        return self.next is None
```

- Wir nennen dies `is_empty`, weil das so üblich ist und weil `empty?` nicht erlaubt wäre.
- Man hätte auch

  ```
  return self.next == None
  ```

  schreiben können, aber ersteres ist geringfügig effizienter und "pythonischer".

Verwendung von `is_empty`

Python-Interpreter

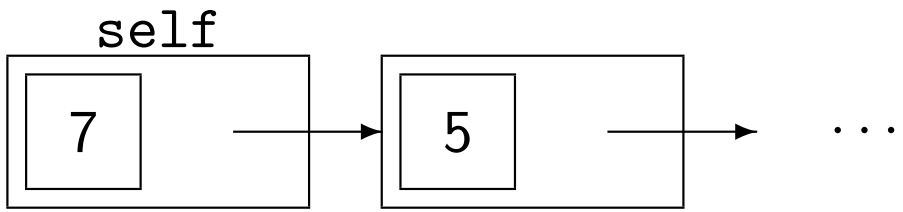
```
>>> x = LinkedList()
>>> x.is_empty()
True
```

Konstruktion einer nichtleeren verlinkten Liste

Eine nichtleere Liste konstruiert man, in dem man vor eine gegebene Liste ein weiteres Element hängt.

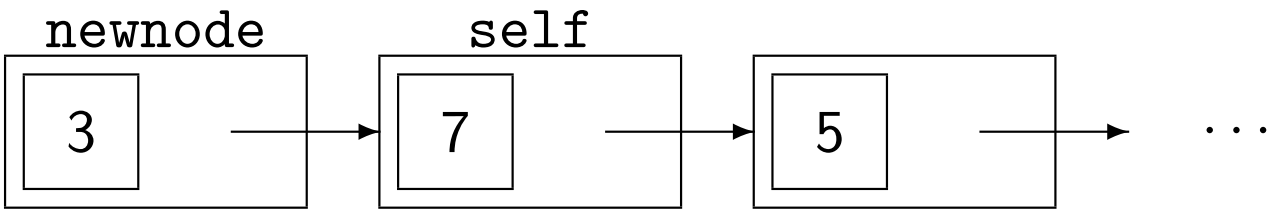
Konstruktion einer nichtleeren verlinkten Liste

Eine nichtleere Liste konstruiert man, in dem man vor eine gegebene Liste ein weiteres Element hängt.



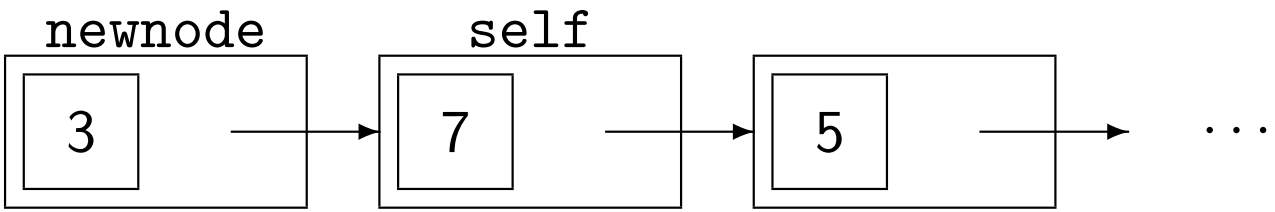
Konstruktion einer nichtleeren verlinkten Liste

Eine nichtleere Liste konstruiert man, in dem man vor eine gegebene Liste ein weiteres Element hängt.



Konstruktion einer nichtleeren verlinkten Liste

Eine nichtleere Liste konstruiert man, in dem man vor eine gegebene Liste ein weiteres Element hängt.



Die Methode wird `newnode` zurückgeben.

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
    ...

    def cons(self, element):
        newnode = LinkedList()
        newnode.data = element
        newnode.next = self
        return newnode
```

Die Methode heißt `cons`, da sie gleichsam eine nichtleere Liste konstruiert.

Verwendung von cons

Python-Interpreter

```
>>> z = LinkedList()
>>> y = z.cons(5)
>>> x = y.cons(7)
>>> w = x.cons(3)
>>> print(w.data)
3
>>> print(w.next.data)
7
>>> print(w.next.next.data)
5
>>> print(w.next.next.next.data)
None
```

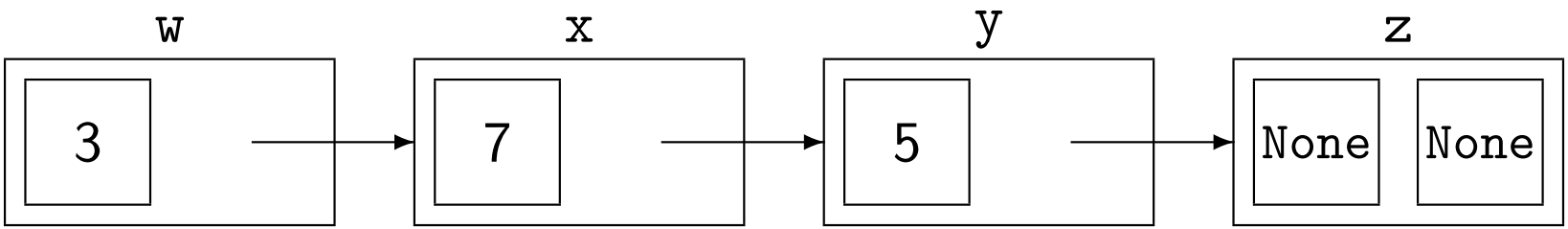
Verwendung von cons

Python-Interpreter

```
>>> z = LinkedList()
>>> y = z.cons(5)
>>> x = y.cons(7)
>>> w = x.cons(3)
>>> print(w.data)
3
>>> print(w.next.data)
7
>>> print(w.next.next.data)
5
>>> print(w.next.next.next.data)
None
```

Wir sehen hier einen iterierten Attributzugriff.

Illustration von cons



Veränderung von `x`

Wir haben Obiges eingetippt und machen jetzt weiter:

Python-Interpreter

```
>>> x.data = 8
```

- Was ist jetzt der Wert von `w.next.data`?

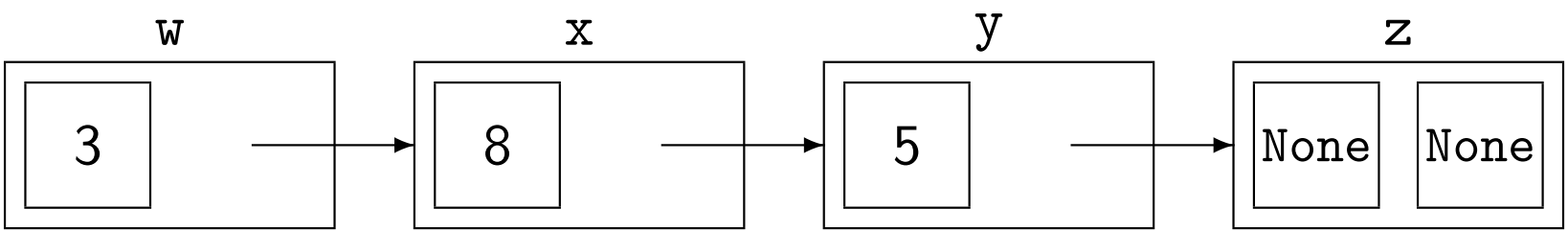
Veränderung von `x`

Wir haben Obiges eingetippt und machen jetzt weiter:

Python-Interpreter

```
>>> x.data = 8
```

- Was ist jetzt der Wert von `w.next.data`?
  8, nicht etwa 7!
- Wenn wir also vor eine Liste `x` ein neues Element mittels `cons` hängen, und das Ergebnis sei `w`, so wird `x` nicht kopiert! Verändern wir also `x`, wirkt sich das indirekt auch auf `w` aus.



Vermeidung von Zwischenvariablen

Da `cons` die neue Liste zurückgibt, kann man es iterieren.

Python-Interpreter

```
>>> x = LinkedList().cons(5).cons(7)
>>> w = x.cons(3)
>>> print(w.data)
3
>>> print(w.next.data)
7
>>> print(w.next.next.data)
5
>>> print(w.next.next.next.data)
None
```

`x` habe ich als Zwischenvariable gelassen, damit ich immer noch die eben erwähnte Problematik erläutern könnte.

Attributzugriff

first und rest

- In der objektorientierten Programmierung ist es etwas verpönt, direkt auf Attribute zuzugreifen.
- Außerdem mögen wir vielleicht die Scheme-Namen `first` und `rest`. Also schreiben wir entsprechende Methoden.

linkedlist.py

```
class LinkedList:
    ...

    def first(self):
        return self.data

    def rest(self):
        return self.next
```

Verwendung von `first` und `rest`

Python-Interpreter

```
>>> w = LinkedList().cons(5).cons(7).cons(3)
>>> x = w.rest()
>>> x.data = 8
>>> w.rest().first()
8
```

- Wir sehen: die Verwendung von `first` und `rest` erlaubt uns lesenden, aber keinen schreibenden Zugriff auf die Attribute.
- Auch für schreibenden Zugriff könnte man Methoden schreiben, aber wir tun es einstweilen nicht.
- `rest()` kopiert den Rumpf der Liste nicht. Verändern wir also `x`, wirkt sich das auch auf `w` aus.

prettyprint

prettyprint

- Dass wir zur Darstellung einer Liste mehrere Attributzugriffe der Form `print(w.data)`, `print(w.next.data)` bzw. Aufrufe der Form `print(w.first())`, `print(w.rest().first())` etc. benötigen, ist umständlich.
- Wir schreiben jetzt eine Methode zur Bildschirmausgabe einer verlinkten Liste.

prettyprint I

linkedlist.py

```
class LinkedList:
    ...

    def prettyprint(self):
        if self.next is None:
            print("empty")
        else:
            print("cons", end=" ")
            self._prettyprint()
```

- Es gibt eine Fallunterscheidung, je nachdem, ob die Liste leer ist oder nicht.

prettyprint II

linkedlist.py

```
class LinkedList:
    ...

    def prettyprint(self):
        if self.next is None:
            print("empty")
        else:
            print("cons", end=" ")
            self._prettyprint()
```

- Das zusätzliche, sogenannte benannte Argument `" "` von `print` hat den Effekt, dass nach der Ausgabe von `"cons"` lediglich ein Leerzeichen, aber kein Zeilenumbruch erfolgt.

prettyprint III

linkedlist.py

```
class LinkedList:
    ...

    def prettyprint(self):
        if self.next is None:
            print("empty")
        else:
            print("cons", end=" ")
            self._prettyprint()
```

- prettyprint ruft eine Hilfsmethode _prettyprint auf, die die Listenelemente ausgibt.

Hilfsmethode _prettyprint

linkedlist.py

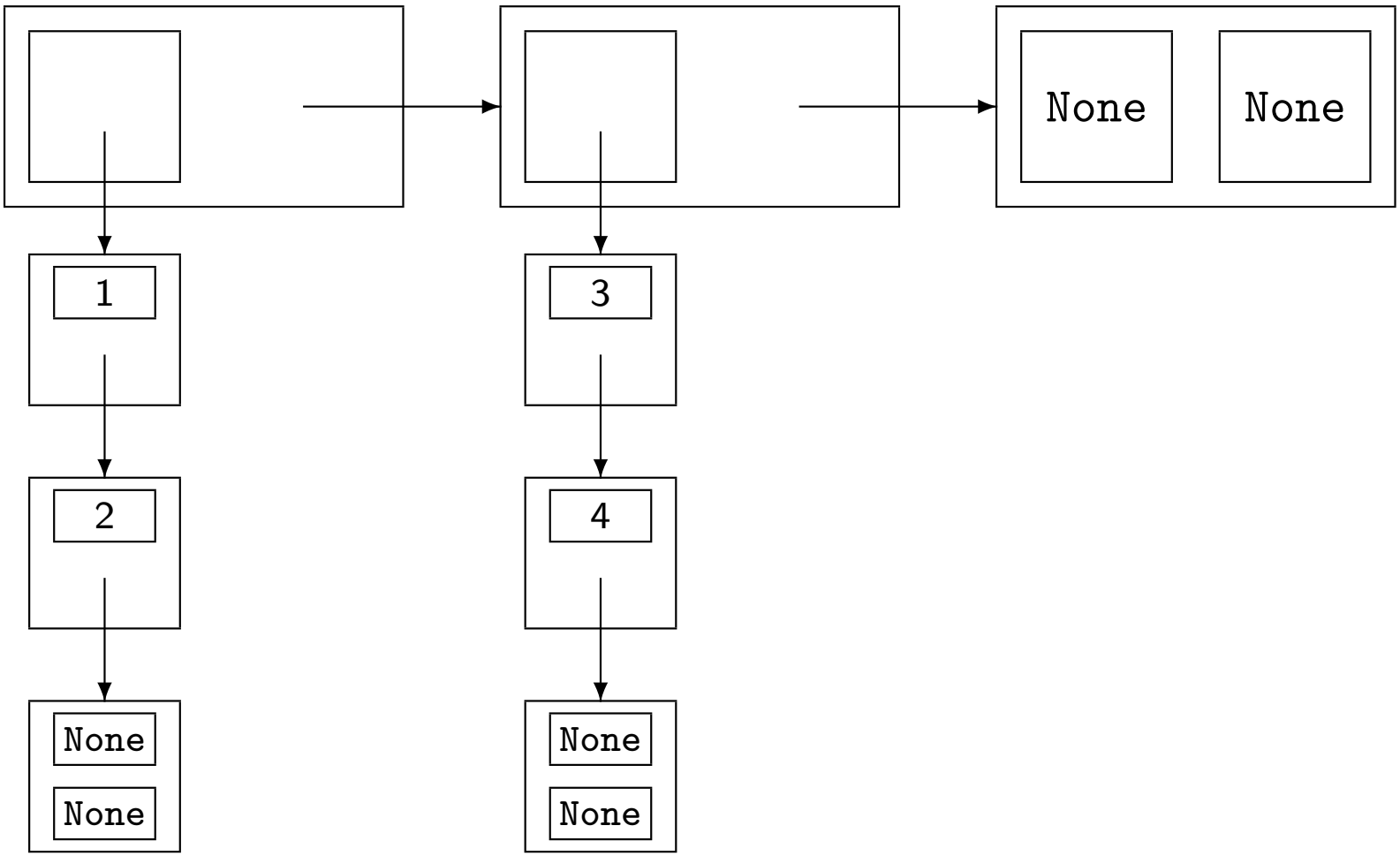
```
class LinkedList:
    ...

    def _prettyprint(self):
        if self.next is not None:
            print(self.data, end=" ")
            self.next._prettyprint()
        else:
            print()
```

- Die Hilfsmethode `_prettyprint` ist rekursiv definiert.
- Der Aufruf `print()` bewirkt einen Zeilenumbruch.
- Der Unterstrich im Namen der Methode besagt per Konvention, dass die Methode von außen nicht verwendet werden soll.

Geschachtelte verlinkte Listen

Wie auch in Scheme kann man Listen von Listen etc. konstruieren.



Geschachtelte verlinkte Listen ausgeben

Allerdings kommt `prettyprint` damit nicht gut zurecht:

Python-Interpreter

```
>>> a = LinkedList().cons(2).cons(1)
>>> b = LinkedList().cons(4).cons(3)
>>> A = LinkedList().cons(b).cons(a)
>>> A.prettyprint()
cons <linkedlist.LinkedList object at 0xb75527cc>
<linkedlist.LinkedList object at 0xb755276c>
```

Wirklich hübsch wird die Ausgabe nur, wenn die `print`-Funktion für die Listenelemente eine hübsche Ausgabe erzeugt.
Man könnte das lösen, aber einstweilen verzichten wir darauf.

Eindeutigkeit der leeren Liste

Eindeutigkeit der leeren Liste

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
    ...

empty =  LinkedList()
```

- Außerhalb der Klassendefinition definieren wir die globale Variable `empty` analog zu Scheme:
  `(define leer (make-leere-liste))`
- Idee: es gibt nur eine leere Liste!
- Wir müssen bei zukünftigen Methoden höllisch aufpassen, dass die leere Liste tatsächlich nicht verändert wird.

Verwendung von `empty`

Python-Interpreter

```
>>> a = empty.cons(2).cons(1)
>>> b = empty.cons(4).cons(3)
```

Kopieren

Kopieren einer verlinkten Liste

linkedlist.py

```
class LinkedList:
    ...

    def copy(self):
        newnode = LinkedList()
        newnode.data = self.data
        if self.next is not None:
            newnode.next = self.next.copy()
        return newnode
```

- Die Methode copy ist rekursiv.
- Jeder Aufruf ruft den Konstruktor auf, es werden also neue Objekte ("Knoten") angelegt.

Verwendung von copy

Python-Interpreter

```
>>> a = empty.cons(2).cons(1)
>>> a.prettyprint()
cons 1 2
>>> b = a.copy()
>>> b.data = 7
>>> a.prettyprint()
cons 1 2
>>> b.prettyprint()
cons 7 2
```

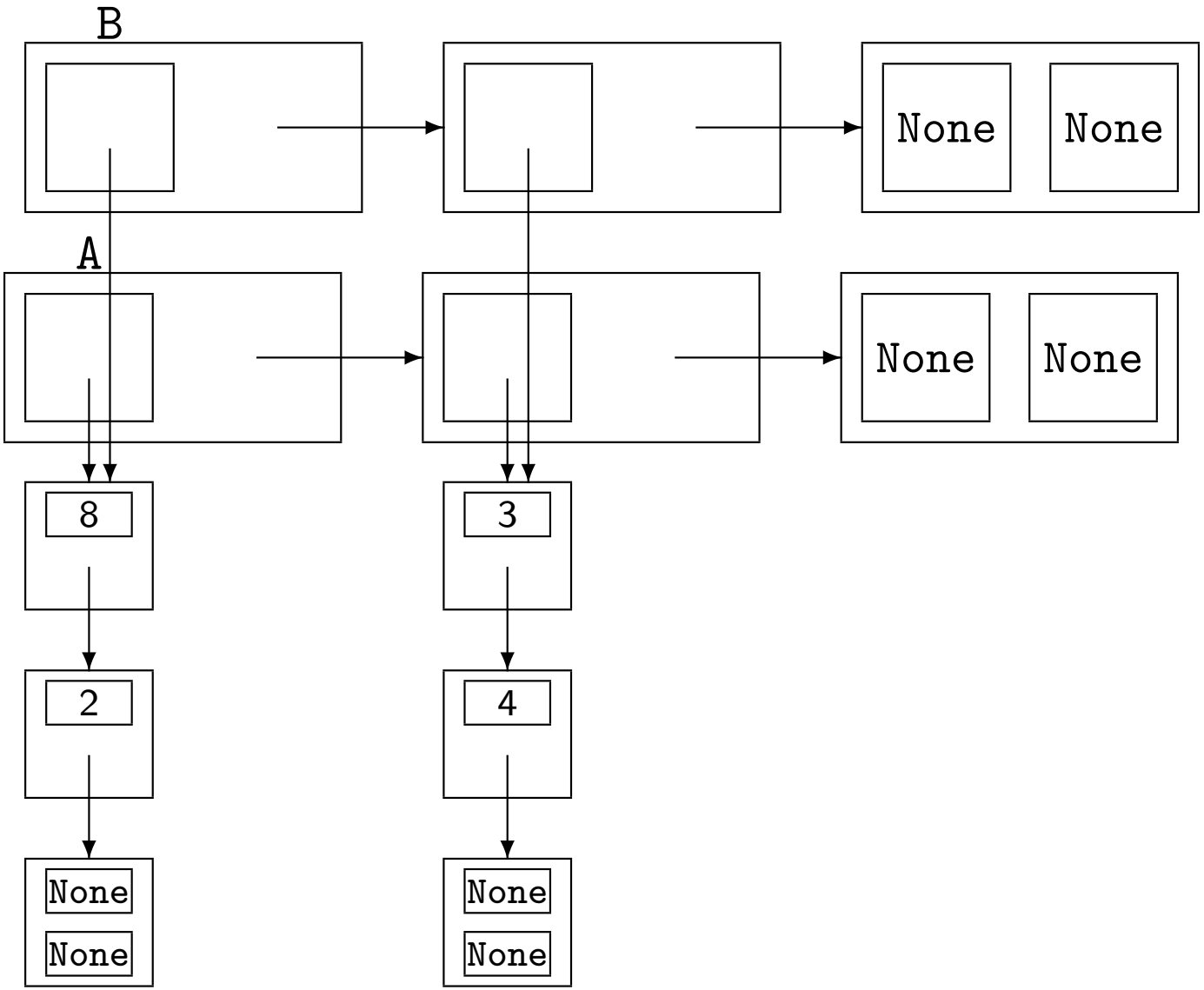
Flache Kopie

Python-Interpreter

```
>>> a = empty.cons(2).cons(1)
>>> b = empty.cons(3).cons(4)
>>> A = empty.cons(b).cons(a)
>>> B = A.copy()
>>> a.data = 8
>>> B.data.data
8
```

copy kopiert flach (shallow), d.h., die Listen der unteren Ebene werden nicht kopiert.
Man könnte das ändern, aber wir tun es einstweilen nicht.

Flache Kopie: Illustration



append

append

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
    ...

    def append(self, other):
        if self.is_empty():
            return other
        else:
            self.next = self.next.append(other)
            return self
```

Illustration von append

Vorher:

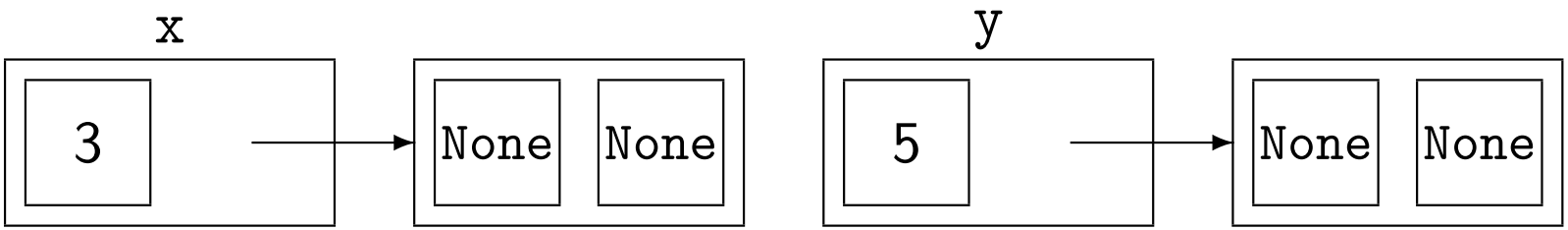
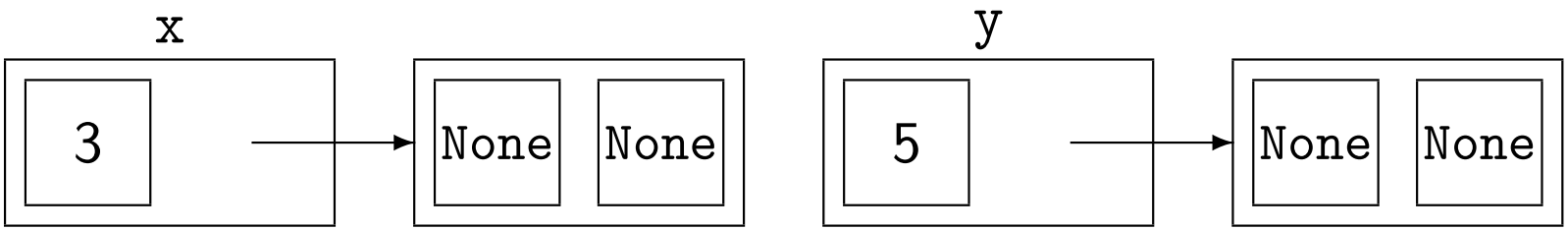
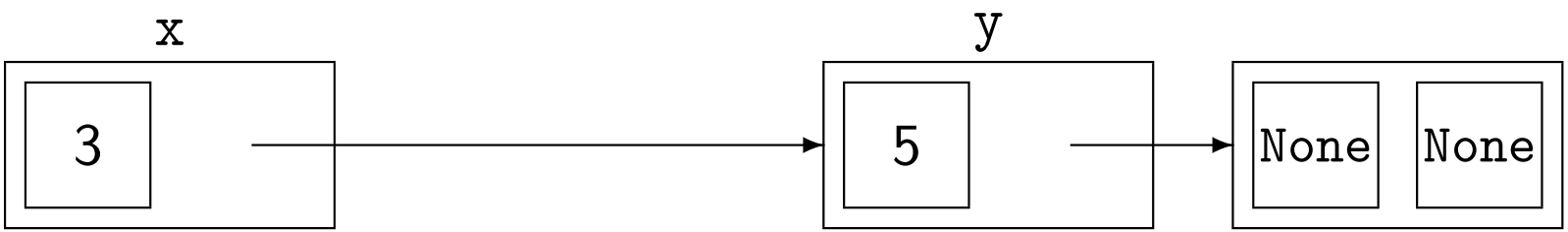


Illustration von append

Vorher:



Nach `x.append(y)`:



Der Aufruf von `x.append(y)` hat den Effekt, dass die Referenz (`next`-Attribut) im letzten echten Element von `x`, welche auf die leere Liste zeigt, umgebogen wird, so dass sie auf `y` zeigt.

Verwendung von append

Python-Interpreter

```
>>> x = empty.cons(2).cons(1)
>>> y = empty.cons(4).cons(3)
>>> x = x.append(y)
<linkedlist.LinkedList object at 0xb7595ecc>
>>> x.prettyprint()
cons 1 2 3 4
```

Nochmalige Anwendung von append

Wir haben Obiges eingetippt und machen weiter:

Python-Interpreter

```
>>> x.prettyprint()
cons 1 2 3 4
>>> x = x.append(y)
<linkedlist.LinkedList object at 0xb7595ecc>
>>> x.prettyprint()
```

Nochmalige Anwendung von append

Wir haben Obiges eingetippt und machen weiter:

Python-Interpreter

```
>>> x.prettyprint()
cons 1 2 3 4
>>> x = x.append(y)
<linkedlist.LinkedList object at 0xb7595ecc>
>>> x.prettyprint()
cons 1 2 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4
3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4
3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 ...
```

Nochmalige Anwendung von append

Wir haben Obiges eingetippt und machen weiter:

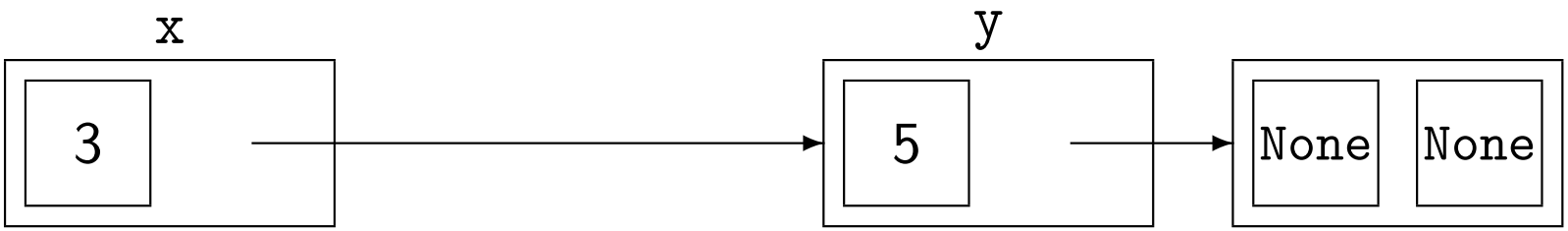
Python-Interpreter

```
>>> x.prettyprint()
cons 1 2 3 4
>>> x = x.append(y)
<linkedlist.LinkedList object at 0xb7595ecc>
>>> x.prettyprint()
cons 1 2 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4
3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4
3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 ...
```

Was ist hier passiert?

Illustration der nochmaligen Anwendung

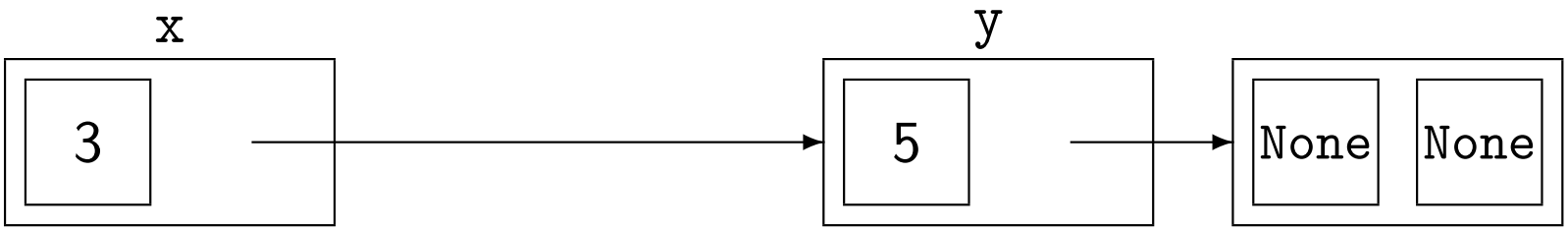
Vorher:



Der Aufruf von `x.append(y)` hat den Effekt, dass die Referenz (`next`-Attribut) im letzten echten Element von `x`, welche auf die leere Liste zeigt, umgebogen wird, so dass sie auf `y` zeigt.

Illustration der nochmaligen Anwendung

Vorher:



Der Aufruf von `x.append(y)` hat den Effekt, dass die Referenz (`next`-Attribut) im letzten echten Element von `x`, welche auf die leere Liste zeigt, umgebogen wird, so dass sie auf `y` zeigt.

Wir haben jetzt keine richtige verlinkte Liste mehr. Die Liste ist "zirkulär" geworden.

append kopiert nichts!

Weder `x` noch `y` werden bei `x.append(y)` kopiert. Wir fangen nochmals frisch an:

Python-Interpreter

```
>>> x = empty.cons(2).cons(1)
>>> y = empty.cons(4).cons(3)
>>> x = x.append(y)
<linkedlist.LinkedList object at 0xb767e9ec>
>>> y.data = 7
>>> x.prettyprint()
cons 1 2 7 4
```

Alternative Definition von append?

Man könnte denken, dass `x.append(y)` sowieso `x` verändern muss (da ja keine Liste kopiert wird), und append folgendermaßen schreiben:

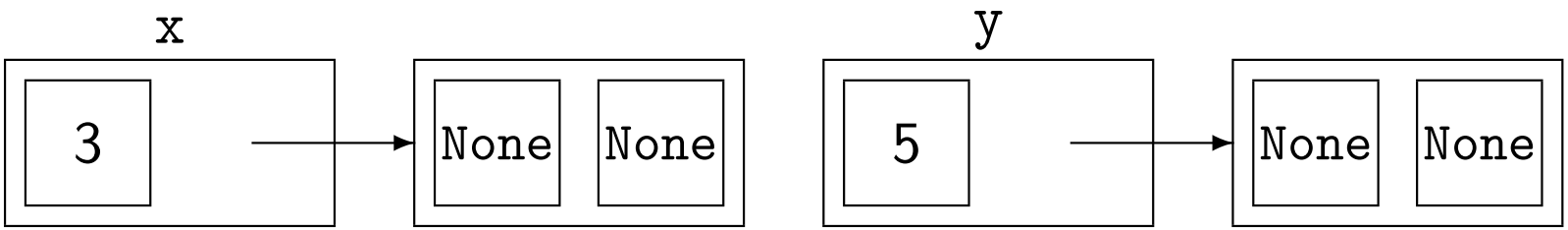
linkedlist.py

```
class LinkedList:
    ...

    def append2(self, other):
        if self.is_empty():
            self.data = other.data
            self.next = other.next
            return self
        else:
            self.next = self.next.append2(other)
            return self
```

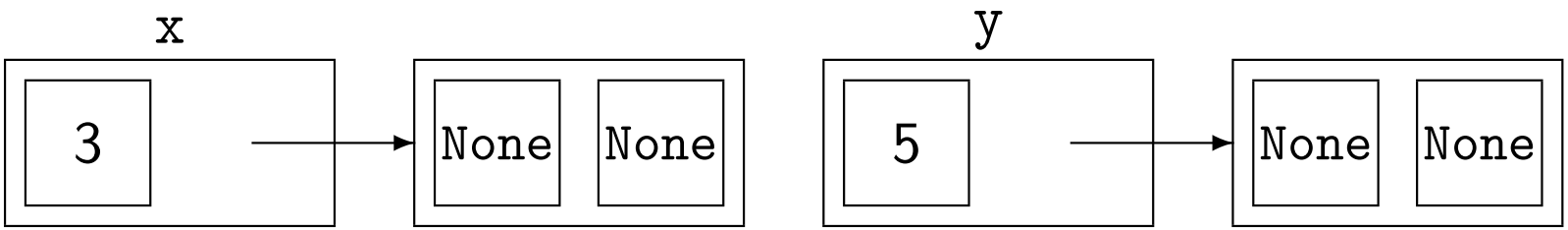
Idee von append2

Vorher:

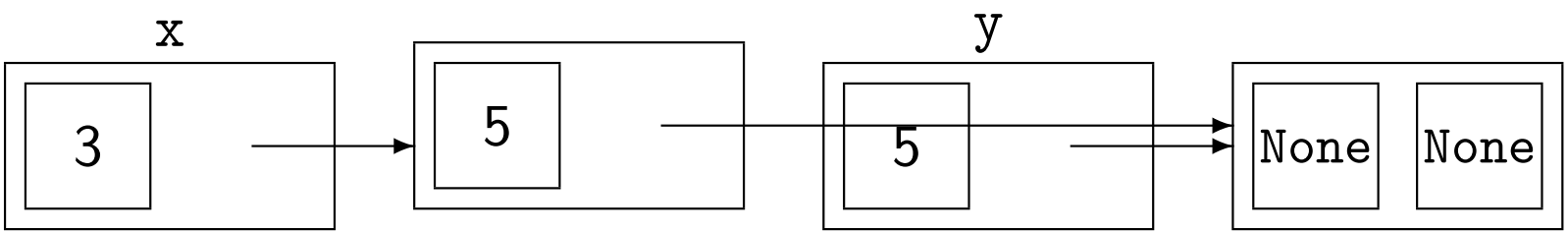


Idee von append2

Vorher:



Nach `x.append2(y)` (vermeintlich):



Der Aufruf von `x.append2(y)` hat den Effekt, dass das letzte, unechte Element (also die leere Liste) von `x` eine Kopie des ersten Elements von `y` wird.

Verwendung von append2

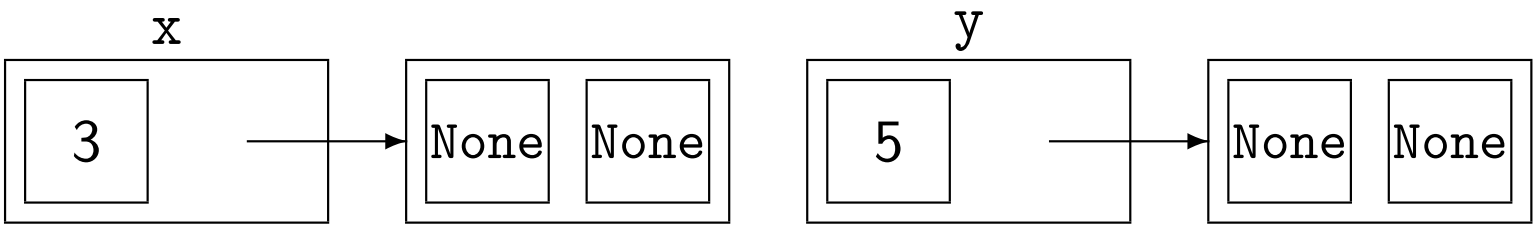
Python-Interpreter

```
>>> x = empty.cons(2).cons(1)
>>> y = empty.cons(4).cons(3)
>>> x = x.append2(y)
<linkedlist.LinkedList object at 0xb769b9ec>
>>> x.prettyprint()
cons 1 2 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4
3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3
4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 4 3 ...
```

Irgendetwas ist schiefgegangen . . .

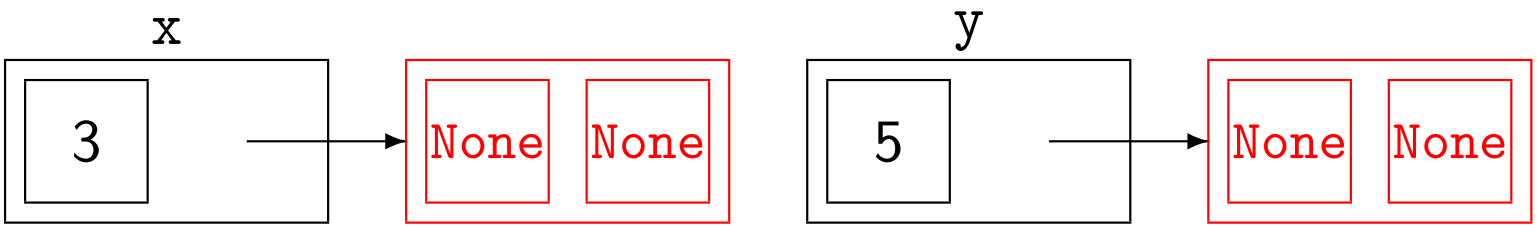
Realität von append2

Vorher:

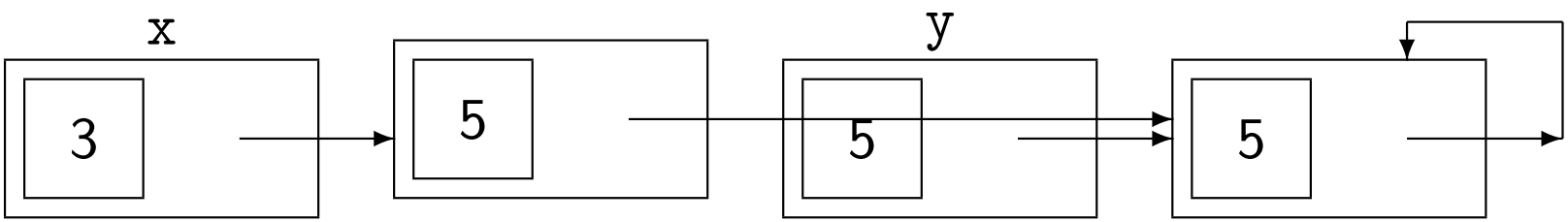


Realität von append2

Vorher:



Nach `x.append2(y)`:



Wir haben nicht höllisch aufgepasst! Es soll nur eine leere Liste geben, und wir haben sie verändert!
Die Aussage, dass `x.append(y)` sowieso `x` verändern muss, stimmt nicht ganz: wenn `x` leer ist, gilt das nicht.

Mit append das Objekt verändern

Da `x.append(y)` nicht unbedingt `x` verändert, wäre es falsch, wenn wir Veränderung wollen, einfach nur `x.append(y)` ohne Zuweisung aufzurufen.
Hier geht das gut:

Python-Interpreter

```
>>> x = empty.cons(2).cons(1)
>>> y = empty.cons(4).cons(3)
>>> x.append(y)
<linkedlist.LinkedList object at 0xb75da1ac>
>>> x.prettyprint()
cons 1 2 3 4
```

x.append(y) ohne Zuweisung

Aber hier nicht:

Python-Interpreter

```
>>> x = empty
>>> y = empty.cons(4).cons(3)
>>> x.append(y)
<linkedlist.LinkedList object at 0xb75d8fac>
>>> x.prettyprint()
empty
```

So geht's richtig

Python-Interpreter

```
>>> x = empty.cons(2).cons(1)
>>> y = empty.cons(4).cons(3)
>>> x = x.append(y)
<linkedlist.LinkedList object at 0xb75da1ac>
>>> x.prettyprint()
cons 1 2 3 4
```

Python-Interpreter

```
>>> x = empty
>>> y = empty.cons(4).cons(3)
>>> x = x.append(y)
<linkedlist.LinkedList object at 0xb75d8fac>
>>> x.prettyprint()
cons 3 4
```

Argument der Methode kopieren

Man kann auch eine Version von append definieren, die die anzuhängende Liste kopiert:

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
    ...

    def append_copy(self, other):
        new = other.copy()
        return self.append(new)
```

Verwendung von append_copy

Wir fangen frisch an:

Python-Interpreter

```
>>> x = empty.cons(2).cons(1)
>>> y = empty.cons(4).cons(3)
>>> x = x.append_copy(y)
<linkedlist.LinkedList object at 0xb769faec>
>>> x = x.append_copy(y)
<linkedlist.LinkedList object at 0xb769faec>
>>> x.prettyprint()
cons 1 2 3 4 3 4
```

Nochmals copy

Übrigens: obwohl wir eigentlich davon ausgehen, dass es nur eine leere Liste gibt, kopiert copy diese:

Python-Interpreter

```
>>> x = empty.copy()
>>> x = x.cons(6)
>>> x.prettyprint()
cons 6
>>> empty.prettyprint()
empty
```

Man könnte dies ändern oder auch nicht ...

Listen als Input

Listen als Input

- Wir betrachten jetzt zwei Methoden, die wir schon von Scheme kennen: `sum` und `length`.
- Bei diesen Methoden ist das Listenobjekt klar als Input aufzufassen, und der Output ist eine Zahl.
- Hier gibt es zum Glück keinerlei Problematik der Veränderung von Objekten und der Frage "Kopieren oder nicht?".

Listenelemente aufsummieren

In Scheme hatten wir eine Prozedur `sum`.
Die entsprechende Methode in Python sieht so aus:

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
    ...

    def sum(self):
        if self.is_empty():
            return 0
        else:
            return self.data + self.next.sum()
```

Verwendung von `sum`

Python-Interpreter

```
>>> x = empty.cons(4).cons(3).cons(1)
>>> x.sum()
8
```

Länge einer verlinkten Liste

Dies kennen wir ebenfalls schon von Scheme:

linkedlist.py

```
class LinkedList:
    def __init__(self):
        self.data = None
        self.next = None
    ...

    def length(self):
        if self.is_empty():
            return 0
        else:
            return 1 + self.rest().length()
```

Verwendung von `length`

Python-Interpreter

```
>>> x = empty.cons(4).cons(3).cons(1)
>>> x.length()
3
```

Entfernen und Ersetzen

Entfernen und Ersetzen

- Wir werden jetzt zwei Methoden schreiben, die Standard sind und die auch im nächsten Kapitel benötigt werden. Es geht um das Entfernen und das Ersetzen eines Elements.
- Leider haben wir hier wieder die Problematik der Veränderung von Objekten und der Frage “Kopieren oder nicht?”.

Elemente entfernen

linkedlist.py

```
class LinkedList:
    ...

    def without(self, el):
        if self.is_empty():
            return self
        else:
            if self.data == el:
                return self.next
            else:
                self.next = self.next.without(el)
                return self
```

Verwendung von `without`

Python-Interpreter

```
>>> x = empty.cons(7).cons(6).cons(5).cons(4)
>>> y = x.without(5)
>>> y.prettyprint()
cons 4 6 7
>>> z = y.without(7)
>>> z.prettyprint()
cons 4 6
```

`without` verändert das Objekt nicht immer

Wie bei append gilt: `without` verändert das Objekt nicht immer:

Python-Interpreter

```
>>> x = empty.cons(7).cons(6).cons(5).cons(4)
>>> x.without(5)
<linkedlist.LinkedList object at 0xb75da18c>
>>> x.prettyprint()
cons 4 6 7
>>> x.without(4)
<linkedlist.LinkedList object at 0xb75d69cc>
>>> x.prettyprint()
cons 4 6 7
```

Nochmals `without`

Im Falle, dass das zu entfernende Element das erste Listenelement ist, wird das Objekt nicht verändert:

linkedlist.py

```
class LinkedList:
    ...
    def without(self, el):
        if self.is_empty():
            return self
        else:
            if self.data == el:
                return self.next
            else:
                self.next = self.next.without(el)
                return self
```

So geht's richtig

Python-Interpreter

```
>>> x = empty.cons(7).cons(6).cons(5).cons(4)
>>> x.prettyprint()
cons 4 5 6 7
>>> x = x.without(5)
>>> x = x.without(4)
>>> x.prettyprint()
cons 6 7
```

replace

linkedlist.py

```
class LinkedList:
    ...

    def replace(self, old, new):
        if self.is_empty():
            return False
        else:
            if self.data == old:
                self.data = new
                return True
            else:
                return self.next.replace(old, new)
```

Funktionsweise von `replace`

`replace` wurde anders konzipiert als die anderen Methoden der Listenveränderung:

- Der Rückgabewert ist keine verlinkte Liste, sondern ein Wahrheitswert (war das Ersetzen erfolgreich?). D.h., die ggf. veränderte Liste wird nicht zurückgegeben, sondern das Objekt wird ggf. verändert.
- Es werden nirgends Referenzen umgebogen, sondern es wird ein `data`-Attribut verändert.
- Ich habe das so gemacht, weil es gut zum nächsten Kapitel passt.
- Sowohl dieses `replace` als auch eine Variante mit Rückgabe der veränderten Liste hätten in einer Bibliothek zu verlinkten Listen ihre Berechtigung.

Verwendung von replace

Python-Interpreter

```
>>> x = empty.cons(7).cons(6).cons(5).cons(4)
>>> x.replace(7, 3)
True
>>> x.prettyprint()
cons 4 5 6 3
>>> x.replace(4, 1)
True
>>> x.prettyprint()
cons 1 5 6 3
>>> x.replace(4, 1)
False
```

Gleichheit vs. Identität

Sowohl `without` als auch `replace` verwenden die Gleichheit, nicht die Identität.

D.h., ein Element der Liste wird entfernt bzw. ersetzt, wenn es das gleiche ist wie das entsprechende Argument der Methode. Es braucht nicht das selbe zu sein.

Zusammenfassung

Die Datenstruktur “Verlinkte Liste”

- Verlinkte Listen sind eine wichtige und prominente Datenstruktur, aber es gibt natürlich noch andere Datenstrukturen.
- Entscheidender Vorteil: Um ein Element zu einer verlinkten Liste zu entfernen oder hinzuzufügen, braucht man lediglich ein bis zwei Referenzen umzubiegen. Man braucht keineswegs alle Elemente vor oder hinter der betroffenen Stelle umzukopieren.
- Entscheidender Nachteil: Man kann nicht in konstanter Zeit auf das n-te Element zugreifen, sondern man muss hierzu n Elemente “entlanglaufen”.

Zusammenfassung

- Wir haben eine Datenstruktur in Python implementiert, die wir schon von Scheme kennen: verlinkte Listen.
- Wir haben folgende Methoden implementiert: `__init__`, `is_empty`, `cons`, `first`, `rest`, `prettyprint`, `_prettyprint`, `copy`, `append`, (`append2`), `append_copy`, `sum`, `length`, `without`, `replace`.
- Man könnte sich noch einige andere Methoden vorstellen.

Zusammenfassung II

- Für die meisten Methoden, die eine veränderte Liste "berechnen", gilt: die veränderte Liste wird zurückgegeben. Man kann sich nicht darauf verlassen, dass das veränderte Objekt das Ergebnis der Berechnung darstellt.
- Trotzdem können einige Methoden das Objekt verändern, und müssen das auch, wollen sie nicht das Objekt kopieren.
- Bei der `replace`-Methode ist die Veränderung des Objekts die eigentliche Berechnung (plus die Rückgabe, ob die Ersetzung Erfolg hatte).
- Am Beispiel vom wiederholten Anwendung von `append` haben wir in verschärfter Form gesehen, welche Probleme entstehen können, wenn zwei Variablen (direkt oder indirekt) auf das selbe Objekt zeigen.

Zusammenfassung III

Kurzum: wir haben

- eine wichtige Datenstruktur noch besser kennengelernt, und
- viel darüber gelernt, wie man im Allgemeinen Datenstrukturen in Python (und anderen imperativen/objektorientierten Sprachen) implementiert.

Literatur


Thomas Ottmann and Peter Widmayer.
Algorithmen und Datenstrukturen.
Spektrum der Wissenschaft, 4 edition, 2002.