Informatik I

19. Schleifen und Iteration für verlinkte Listen

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

27. Januar 2011

Informatik I

27. Januar 2011 — 19. Schleifen und Iteration für verlinkte Listen

19.1 Die Klasse `LinkedList`

Sequenztypen

- Wir werden uns jetzt weiter mit Schleifen und Iteration beschäftigen, und zwar für verlinkte Listen.
- Verlinkte Listen sind ein Sequenztyp, den wir selbst gebaut haben.
- Es gibt auch einige eingebaute Sequenztypen in Python, die wir später betrachten werden.
- Damit verlinkte Listen ein vollwertiger Sequenztyp im Sinne von Python wären, müssten wir noch einige weitere Methoden zur Verfügung stellen ...
- Entscheidend aber ist: Objekte eines Sequenztyps enthalten andere Dinge in einer bestimmten Reihenfolge. Es ist also für jedes Ding wohldefiniert, was das nächste Ding ist.
  Äquivalent: es ist wohldefiniert, was das erste, zweite, ... Ding ist.

Iterieren über Sequenzen

- Über Sequenzen kann man iterieren, d.h., man kann eine Schleife schreiben, die nacheinander die Elemente einer Sequenz "besucht" und ggf. irgendetwas damit tut.
- Alle rekursiven Methoden der `LinkedList`-Klasse kann man auch so umschreiben, dass sie über das Objekt iterieren, d.h., man kann iterative anstatt rekursive Methoden schreiben.
  Das werden wir jetzt tun.
- Für die eingebauten Sequenztypen gibt es eine besonders mächtige Unterstützung für das Iterieren, aber dazu im nächsten Kapitel ...

19.1 Die Klasse LinkedList

- Ausdrucken
- Kopieren
- Anhängen
- Entfernen
- Ersetzen
- Summe und Länge
- Zusammenfassung

Erinnerung: die Attribute

Hier noch einmal die Definition von `__init__` der Klasse `LinkedList`, damit wir die Attribute sehen:

linkedlist_iterative

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

prettyprint

Die erste rekursive Prozedur war _prettyprint. Zunächst prettyprint:

linkedlist_iterative

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

_prettyprint

Jetzt _prettyprint:

linkedlist_iterative

```
class LinkedList:
    ...
    def _prettyprint(self):
        curr = self
        while curr.next is not None:
            print(curr.data, end=" ")
            curr = curr.next
        print()
```

Erklärung zu _prettyprint

- Anfangs ist curr (das erste Element von) self.
- Die Bedingung curr.next is not None bedeutet: curr ist ein echtes Listenelement von self, also nicht das letzte, unechte Listenelement von self (die leere Liste).
- In jedem Schleifendurchlauf wird das data-Attribut von curr ausgedruckt und curr auf seinen Nachfolger gesetzt.
- Somit nimmt curr nacheinander die Werte aller echten Listenelemente an.
- Danach folgt der Aufruf print() für einen Zeilenumbruch.

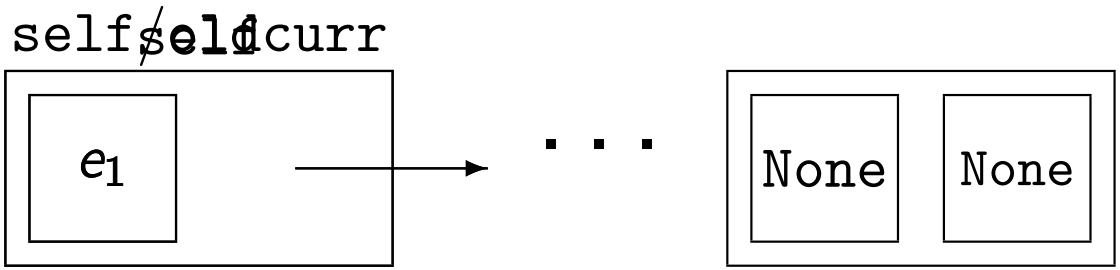
copy

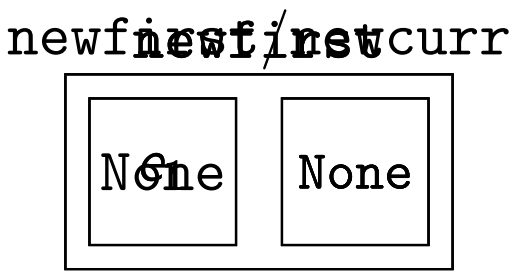
linkedlist_iterative

```
class LinkedList:
    ...
    def copy(self):
        newfirst = LinkedList()
        newfirst.data = self.data
        oldcurr = self
        newcurr = newfirst
        while oldcurr.next is not None:
            newcurr.next = LinkedList()
            newcurr.next.data = oldcurr.next.data
            newcurr = newcurr.next
            oldcurr = oldcurr.next
        return newfirst
```

Vor der Schleife

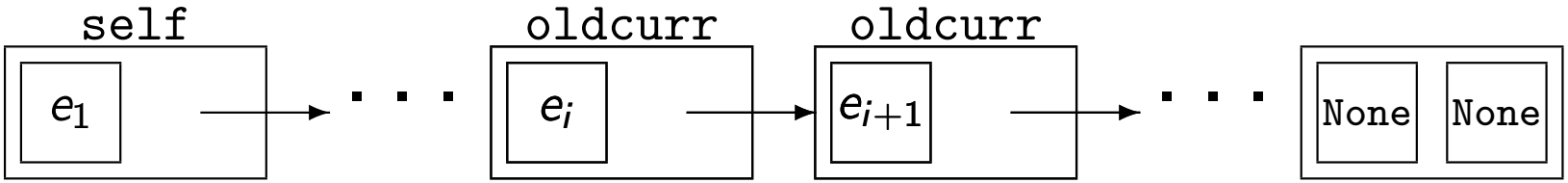
Die vier Zeilen vor der Schleife haben folgenden Effekt:

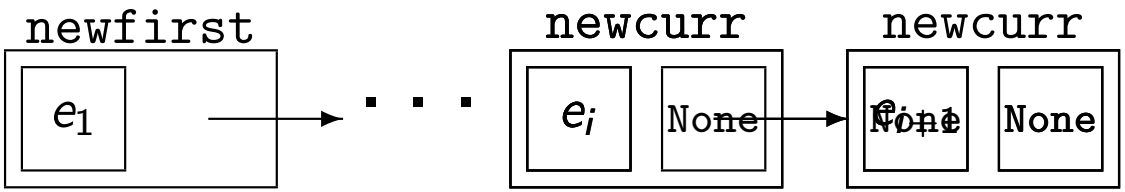




`newfirst` ist momentan keine korrekte verlinkte Liste!

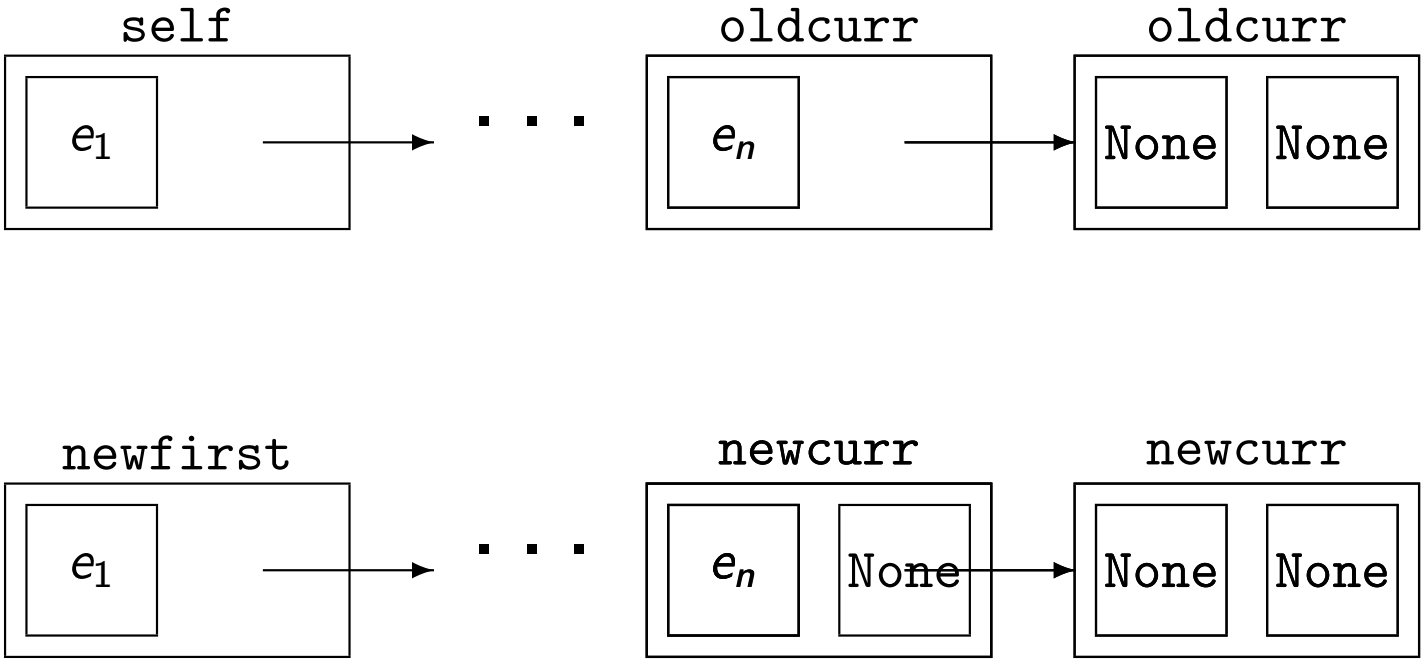
Allgemeiner Schleifendurchlauf





- Die ersten i Einträge sind schon kopiert. `newfirst` ist monentan keine korrekte verlinkte Liste.
- Jetzt ist `newfirst` eine korrekte verlinkte Liste.
- Und jetzt schon wieder nicht.
- Die ersten $i+1$ Einträge sind kopiert. Im Grunde ist die Situation exakt die gleiche wie vor dem Durchlauf, nur um ein Element verschoben.

Letzter Schleifendurchlauf



- Alle n Einträge sind schon kopiert. `newfirst` ist monentan keine korrekte verlinkte Liste.
- Jetzt ist `newfirst` eine korrekte verlinkte Liste.
- `None` wird "kopiert". `newfirst` bleibt eine korrekte verlinkte Liste.

Korrektheit von Programmen

- Wir haben uns soeben recht gründlich davon überzeugt, dass copy korrekt ist.
- Es gibt einen Formalismus, der eine solche Argumentation noch viel präziser vollzieht: Hoare-Kalkül.
- Der Satz

  Im Grunde ist die Situation exakt die gleiche wie vor dem Durchlauf, nur um ein Element verschoben.

  hat in diesem Formalismus eine Bezeichnung: Schleifeninvariante.

append

linkedlist_iterative

```
class LinkedList:
    ...
    def append(self, other):
        if self.is_empty():
            return other
        else:
            curr = self
            while curr.next.next is not None:
                curr = curr.next
            curr.next = other
            return self
```

Erklärungen zu append

- Die Bedingung `curr.next.next is not None` bedeutet: `curr` ist ein echtes Listenelement von `self`, und zwar spätestens das vorletzte.
- Sobald die Bedingung `curr.next.next is not None` erstmals verletzt ist, bedeutet dies: `curr` ist das letzte echte Listenelement. Die Zuweisung `curr.next = other` sorgt dann dafür, dass das `next`-Attribut des letzten Listenelements auf `other` umgebogen wird.

without

linkedlist_iterative

```
class LinkedList:
    ...
    def without(self, entry):
        if self.is_empty():
            return self
        else:
            if self.data == entry:
                return self.next
            else:
                curr = self
                while curr.next.next is not None:
                    if curr.next.data == entry:
                        curr.next = curr.next.next
                        break
                    else:
                        curr = curr.next
                return self
```

Erklärungen zu `without`

- Bis zum zweiten `else` sieht es auch wie die rekursive Variante.
- Wie oben auch bedeutet die Bedingung `curr.next.next is not None`: `curr` ist ein echtes Listenelement von `self`, und zwar spätestens das vorletzte.
- In jedem Durchlauf vergleichen wir nicht etwa `curr.data` mit `entry`, sondern `curr.next.data`.
- Warum? In dem Moment, in dem wir `entry` in der Liste finden, müssen wir Zugriff auf den Vorgänger dieses Elements haben, denn dessen `next`-Attribut muss umgebogen werden.

replace

linkedlist_iterative

```
class LinkedList:
    ...
    def replace(self, old, new):
        curr = self
        while (curr.data != old and
               curr.next is not None):
            curr = curr.next
        if curr.next is not None:
            curr.data = new
            return True
        return False
```

Erklärungen zu replace

- Die Schleifenbedingung könnte aus zwei Gründen verletzt sein:
  - `curr.data` enthält den zu ersetzenden Eintrag. In diesem Fall ersetzen wir und geben `True` zurück.
  - `curr` ist das letzte, unechte Element von `self<`. Dann verlassen wir die Schleife ohne etwas zu tun. Außerhalb der Schleife geben wir dann `False` zurück.

sum und length

- Wir hatten dann noch zwei rekursive Methoden `sum` und `length`.
- Diese dürfen Sie als Übung im Betreuten Programmieren machen!

Zusammenfassung

- Wir haben gesehen, wie man über eine verlinkte Liste iteriert. Es gibt eine Variable, die nacheinander die Elemente der Liste als Wert annimmt.
- Im Detail kann es trickreich sein: eine solche Methode kann unter Umständen:
  - erst beim zweiten Element beginnen (kein Beispiel);
  - nur bis zum vorletzten Element gehen;
  - verfrüht abbrechen.
- Beliebter Fehler: Zugriff auf Attribute eines `None`-Objekts.
- Die rekursiven Formulierungen sind mitunter viel eleganter als die iterativen.
- Die iterativen sind allerdings effizienter (selbst im Vergleich zu Endrekursion, da diese von Python nicht auf besondere Weise unterstützt wird).