Programmierkurs Python

Michaela Regneri & Stefan Thater
Universität des Saarlandes
FR 4.7 Allgemeine Linguistik (Computerlinguistik)

Winter 2010/11

Iteratoren

```
for item in [1,2,3]:
    print(item)

for item in (1,2,3):
    print(item)

for item in {1,2,3}:
    print(item)

for key in {1:'eins', 2:'zwei', 3:'drei'}:
    print(key)

for ch in 'eins':
    print(ch)
```

Iteratoren

```
>>> it = iter([1,2,3])
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```


Iteratoren

- Mit Iteratoren kann man über die Elemente beliebiger Sammeltypen (Listen, Mengen, usw.) iterieren
- Iteratoren kann man sich als Zeiger vorstellen, die auf die Elemente eines Sammeltyps verweisen
- Primäre Operation: `next(iterator)`
  - Zugriff auf das aktuelle Element
  - Seiteneffekt: Iterator zeigt auf das nächste Element

for item in collection

- Erzeuge einen Iterator:
  - `it = iter(collection)`
- In jedem Schleifendurchlauf:
  - Aufruf von `next(it)`
    - das Ergebnis wird in `item` gespeichert
  - Abbruch, wenn `StopIteration` geworfen wird
  - Sonst: führe den Code im Schleifenkörper aus

```
for item in collection:
   <Schleifenkörper>
```

=

```
it = iter(collection)
while True:
   try:
      item = next(it)
   except StopIteration:
      break
   <Schleifenkörper>
```


Iteratoren in Python

- Keine spezielle Klasse für Iteratoren
- **Iteratoren** sind Objekte, die die folgenden Methoden implementieren:
  - __next__() liefert das nächste Element
    - wird von next(iterator) aufgerufen
  - __iter__() gibt das Iterator-Objekt zurück
    - wird von iter(x) aufgerufen
- Ein Objekt o ist **iterierbar**, wenn es iter(o) unterstützt:
  - Die Methode __iter__ ist implementiert und liefert einen Iterator

Listen-Iterator

```
class ListIterator:
  def __init__(self, lis):
    self.lis = lis
    self.index = -1
  def __iter__(self):
    return self
  def __next__(self):
    self.index += 1
    if self.index >= len(self.lis):
      raise StopIteration
    return self.lis[self.index]
```


Achtung!

- Vorsicht beim Iterieren über und gleichzeitigem Ändern einer Liste:
  - for x in lis: lis.append(x)
- Stattdessen: Iterieren über eine Kopie der Liste
  - for x in lis[:]: lis.append(x)

Iteratoren Verwenden

- Iteratoren können verwendet werden …
  - for-Schleifen
  - in-Operator (if x in it)
  - um Sammeltypen zu erzeugen
    - list(it), tuple(it), dict(it), set(it)
  - usw.
- Sprich: sie können fast überall verwendet werden, wo man auch Listen verwenden kann
  - Beachte: len(iterator) wird nicht unterstützt
- Iteratoren sind häufig effizienter (Speicherbedarf)
  - Viele eingebaute Funktionen liefern Iteratoren

Iteratoren Verwenden

```
# Variante (a)
xs = [1,2,3]
# Variante (b)
xs = iter([1,2,3])

for x1 in xs:
   for x2 in xs:
      print(x1 * x2)
```

Achtung: hier macht es einen Unterschied, ob wir (a) Listen oder (b) Itertoren verwenden

Iteratoren & Wörterbücher

- Die Dictionary-Methoden liefern sogenannte „views“ zurück
  - `dict.keys()`
  - `dict.values()`
  - `dict.items()`
- Views können wir als Iteratoren betrachten

Iteratoren & Dateien

- Man kann über Dateiobjekte (zeilenweise) iterieren:
  - for line in f.readlines(): ...
  - for line in f: ...
- Variante 2 ist effizienter
  - Die Datei braucht nicht vollständig in den Hauptspeicher geladen zu werden (weniger Speicherbedarf)
  - Man kann mit der Iteration beginnen, wenn die Datei noch gar nicht vollständig geladen worden ist.

Eigene Iteratoren definieren

```
class Circ:
  def __init__(self, seq):
    self.seq = seq
    self.idx = -1
  def __iter__(self):
    return self
  def __next__(self):
    self.idx = (self.idx + 1) % len(self.seq)
    return self.seq[self.idx]
```

```
>>> for x in Circ([1,2,3]):
>>>    print(x, end=',')
1,2,3,1,2,3,...
```


`itertools`

- Das `itertools`-Modul implementiert eine Reihe nützlicher Funktionen über Iteratoren
- Zum Beispiel:
  - itertools.islice(it, [start], stop, [step])
    - slice Operator für iteratoren
  - itertools.count(0, 2) ⇒ `0, 2, 4, 6,...`
  - `etc.`

Iteratoren für eigene Datenstrukturen

- Iteratoren kann man auch für eigene Datenstrukturen definieren.
- Beispiel: Stack mit verketteten Listen

Iteratoren für eigene Datenstrukturen

```
class Stack:
  class Iter:
    ...
  def __init__(self):
    self.data = None
  def push(self, elt):
    self.data = (elt, self.data)
  def pop(self):
    ...
  def __iter__(self):
    return self.Iter(self)
```

interne Hilfsklasse

Iteratoren für eigene Datenstrukturen

```
class Stack:
  class Iter:
    def __init__(self, stack):
      self.data = stack.data
    def __iter__(self):
      return self
    def __next__(self):
      if self.data == None: raise StopIteration
      elt, self.data = self.data
      return elt
  ...
```