Programmierkurs Python I

Michaela Regneri

2010-01-14

(Folien basieren auf dem gemeinsamen Kurs mit Stefan Thater)

Iteratoren

- Mit Iteratoren kann man über die Elemente eines Sammeltyps (Liste, Menge, …) iterieren
- Iteratoren kann man sich als Zeiger vorstellen, die auf die Elemente eines Sammeltyps verweisen
- Primäre Operationen:
  - Zugriff auf das aktuelle Element
  - Selbstmodifikation, um auf nächstes Element zu zeigen

Iteratoren in Python

- Keine spezielle Klasse für Iteratoren
- Ein Objekt `it` ist ein Iterator, wenn `it` die Methoden `__next__()` und `__iter__()` implementiert
  - `__next__()` liefert das nächste Element
  - `__iter__()` gibt das Iterator-Objekt zurück; es wird von der builtin-Methode `iter(x)` aufgerufen
- Ein Objekt **`x`** ist iterierbar, wenn es `iter(x)` unterstützt:
  - Die Methode `__iter__` ist implementiert und liefert einen Iterator

`for x in collection: body(x)`

- Hole einen Iterator:
  - `it = iter(collection)`
- In jedem Schleifendurchlauf:
  - rufe `it.next()` auf und binde das Ergebnis an x
  - breche die Schleife ab, wenn `StopIteration` geworfen wird
  - Sonst: führe `body(x)` aus (entspricht dem Block der Schleife)

```
it = iter(collection)
while True:
   try:
      x = it.next()
   except StopIteration:
      break
   body(x)
```

`iter(x)`

- wird die Methode `__iter__` vom Objekt `x` implementiert, wird sie aufgerufen
- Ansonsten wird die „Prä-2.2“ Semantik simuliert: Zugriff auf Elemente über Indices 0, 1, 2, ...
  - Objekt muss `x[i]` unterstützen
- Auch möglich: `iter(callable, sentinel)`

Iteration über Listen

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

Iteration über Listen – Vorsicht

- Die Liste wird nicht kopiert!
- Vorsicht beim gleichzeitigem Iterieren über und Ändern der Liste:
  - `for x in lis: lis.append(x)`
- Stattdessen: Iterieren über eine Kopie der Liste
  - `for x in lis[:]: lis.append(x)`

iter(callable, sentinel)

```
class SentinelIterator:
   def __init__(self, callable, sentinel):
      self.callable = callable
      self.sentinel = sentinel
   def __iter__(self):
      return self
   def __next__(self):
      result = self.callable()
      if result == self.sentinel:
         raise StopIteration
      return result
```

Iteratoren Verwenden

- Iteratoren können verwendet werden …
  - in `for`-Schleifen
  - im `in`-Operator: `if x in it`
  - `list(it), tuple(it), dict(it), set(it)`
  - usw.
- Sprich: sie können fast überall verwendet werden, wo man auch Sequenzen verwenden kann
- Iteratoren sind häufig effizienter (Speicherbedarf)

Iteration über Wörterbücher

- Die Dictionary-Methoden liefern *views* zurück (`dict.keys(), dict.values(), dict.items()`)
- Views können wir als Iteratoren betrachten
- Listen werden nicht explizit berechnet:

  ⇒ effizienter

Iteratoren & Dateien

- Man kann über Dateiobjekte (zeilenweise) iterieren:

```
(1)for line in f.readlines(): ...
(2)for line in f: ...
```

- Variante 2 ist Speichereffizienter: Die Datei braucht nicht vollständig in den Hauptspeicher geladen zu werden
- Variante 2 ist außerdem flexibler: Man kann mit der Iteration beginnen, wenn die Datei noch gar nicht vollständig geladen worden ist

Eigene Datenstrukturen

```
class Stack:
  class Iter:
    ...
```

interne Hilfsklasse

```
  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)
```

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
   ...
```


„Unendliche“ Iteratoren

```
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: Iteratoren als Sequencen
  - `itertools.islice(it, [start], stop, [step])`
  - `itertools.map(callable, it1, it2, ...)`
  - usw.

```
for x in itertools.islice(Circ([1,2,3]),0,20,3):
  print(x, end=“,“)
⇒ 1,1,1,1,1,1,1,
```


Generatoren

- Generatoren sind Funktionen, die das Schlüsselwort `yield` enthalten.

```
def plus2(it):
  for x in it:
    yield x + 2
```

- Wenn ein Generator aufgerufen wird, wird der Funktionskörper nicht komplett ausgeführt
- Stattdessen wird ein Iterator geliefert, der die Funktion kapselt und den aktuellen Berechnungszustand speichert

Generatoren

- Generator-Funktionen liefern einen Iterator `it`
  - Der erste Aufruf von `it.next()` beginnt mit dem Auswertung des Funktionskörpers
  - `yield <wert>` friert den Berechnungszustand ein und liefert `<wert>`
    - weitere Aufrufe von `it.next()` tauen den Berechnungszustand wieder auf und machen weiter
  - `return` Anweisungen beenden die Iteration: `StopIteration` wird geworfen
    - ein Generator darf nur `return`-Anweisungen ohne Wert enthalten

Iteratoren vs. Generatoren

```
class StackIter:
   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
```

```
def stackiter(s):
   data = s.data
   while data != None:
      elt, data = data
      yield elt
```

Generatoren „degenerieren“

- Generatoren können in äquivalente „normale“ Funktion übersetzt werden:
  - Am Anfang der Funktion: `_list = []`
  - `yield expr` ersetzen durch `_list.append(expr)`
  - `return` (und `raise StopIteration`)ersetzen durch `return iter(_list)`
- Liefert das gleiche Resultat, benötigt aber mehr Speicher
- Einschränkung: Das geht natürlich nur mit „endlichen“ Generatoren

Generatoren „degenerieren“

```
def stackiter(s):
    data = s.data
    while data != None:
        (elt, data) = data
        yield elt
```

```
def stackiter(s):
    _list = []
    data = s.data
    while data != None:
        (elt, data) = data
        _list.append(elt)
    return iter(_list)
```

Mehrere `yield` Anweisungen

- Ein Generator kann mehrere yield Anweisungen enthalten:

```
def double(it):
  for item in it:
    yield item
    yield item

>>> list(double([1,2,3]))
[1,1,2,2,3,3]
```


Weitere Beispiele

```
import re

word = re.compile('\w+')

def byWord(f):
   for line in f:
      for wrd in word.finditer(line):
        yield wrd.group()
```

Weitere Beispiele

```
import re, string

word = re.compile('\w+')

def byParagraphs(f):
   p = []
   for line in f:
      if line.isspace():
         if p: yield string.join(p)
         p = []
      else: p.append(line)
   if p:
      yield string.join(p)
```


Generatoren & Rekursion

- Ein Generator g darf (kann) keinen anderen Generator direkt aufrufen.
  - Genauer: das ganze hat nicht den gewünschten Effekt
- Insbesondere kann man keine rekursiven Generatoren implementieren.
- Man kann aber die Rekursion „einkapseln“, indem man den entsprechenden Iterator in einer for-Schleife verwendet.

```
def g():
   # ...
   yield something
   # ...
   for bla in g(): ...
```

Generatoren & Rekursion

- Ein konkretes Beispiel: Binäre Bäume (Mengen)
- Knoten haben drei Datenfelder:
  - value: Ein Wert (eine Zahl)
  - left: Das linke Kind (ein Knoten)
  - right: Das rechte Kind (ein Knoten)
- Beim Einfügen von Werten:
  - Wenn der einzufügende Wert kleiner ist als der Wert des Knotens: Füge den Wert dem linken Kind hinzu.
  - Wenn der einzufügende Wert größer ist: Füge den Wert dem rechten Kind hinzu.

Binäre Bäume

```
class Node:
   def __init__(self, value, left, right):
      self.value = value
      self.left = left
      self.right = right
   def add(self, value):
      if self.value < value:
         self.left = self.left.add(value)
      elif self.value > value:
         self.right = self.right.add(value)
      return self

class Empty:
   def add(self, value):
      return Node(value, Empty(), Empty())
```

Binäre Bäume

```
class Tree:
   def __init__(self):
      self.data = Empty()
   def add(self, value)
      self.data = self.data.add(value)
```


Rekursion & Generatoren

```
class Node:
   def __init__(self, value, left, right): ...
   def add(self, value): ...
   def __iter__(self):
      for val in self.left:
         yield val
      yield self.value
      for val in self.right:
         yield val
class Empty:
   def add(self, value): ...
   def __iter__(self): return self
   def __next__(self): raise StopIteration
```

Rekursion & Generatoren

```
class Tree():
   def __init__(self):
      self.data = Empty()
   def add(self, value)
      self.data = self.data.add(value)
   def __iter__(self):
      return iter(self.data)
```