Programmierkurs Python I

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

Winter 2010/11

Übersicht

- Kurze Wiederholung: Iteratoren
- Generatoren
- Listen & Iteration
  - `map`, filter
  - lambda
  - List Comprehensions

Iteratoren (Wdh.)

```
>>> 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 (Wdh.)

- 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 (Wdh.)

```
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]
```


Generatoren

- Generatoren sind Funktionen, die das Schlüsselwort **yield** enthalten.
- Wenn ein Generator aufgerufen wird, wird der Funktionskörper nicht ausgeführt …
- stattdessen wird ein Iterator geliefert

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

Generatoren

- Ein Aufruf von `plus2(...)` liefert einen Iterator `it`
- Der erste Aufruf von `next(it)` beginnt mit dem Auswertung des Funktionskörpers

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

- Die `yield` Anweisung friert den Berechnungszustand ein und liefert den angegebenen Wert
- Weitere Aufrufe von `next(it)` tauen den Berechnungszustand wieder auf und machen weiter

Generatoren

- Die `return` Anweisung beendet die Iteration
- Einschränkung: `return` Anweisungen sind in Generatoren nur zuässig, wenn sie keine Werte liefern
  - `return` – ok
  - `return` ⟨irgendwas⟩ – verboten

Iteratoren vs. Generatoren

```
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]
```


Iteratoren vs. Generatoren

```
def listiterator(lis):
   i = 0
   while i < len(lis):
      yield lis[i]
      i += 1
```

Mehrere `yield` Anweisungen

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

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

Beachte: Ein Generator kann mehrere yield Anweisungen enthalten

Weitere Beispiele

```
def byWord(f):
   for line in f:
      for word in line.split():
         yield word

with open(filename) as f:
   for word in byWord(f):
      ...
```

Weitere Beispiele

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


Generatoren & Rekursion

- Ein Generator kann anderen Generatoren nicht direkt aufrufen.
  - Genauer: das hätte 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
- 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
```

Binäre Bäume

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

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


Generatoren & Rekursion

```
class Node:
   ...
   def __iter__(self):
      for value in self.left:
         yield value
      yield self.value
      for value in self.right:
         yield value
```

Generatoren & Rekursion

```
class Empty:
   ...
   def __iter__(self):
      return self
   def __next__(self):
      raise StopIteration

class Tree:
   ...
   def __iter__(self):
      return iter(self.data)
```

```
>>> t = Tree()
>>> t.add(27)
>>> t.add(13)
>>> t.add(54)
>>> t.add(-1)
>>> t.add(99)
>>> list(t)
[99, 54, 27, 13, -1]
```


Übersicht

- Kurze Wiederholung: Iteratoren
- Generatoren
- Listen & Iteration
  - map, filter
  - lambda
  - List Comprehensions

map

- `map(function, iterable, ...)`
- Wendet die Funktion `function` auf alle Elemente aus `iterable` an ...
- und liefert einen Iterator über die Ergebnisse
- Die Anzahl der übergebenen `iterables` muss mit der Anzahl Parameter der Funktion übereinstimmen

```
def plus3(x):
    return x + 3

>>> lis = [1,2,3]
>>> map(plus3, lis)
<map object at 0xb7470f2c>
>>> list(map(plus3, lis))
[4,5,6]
```


map

```
def max(x, y):
    return x if x > y else y

>>> list(map(max, [15, 20, 25], [24, 18, 12]))
[24,20,25]
>>> list(map(max, [15, 20], [24, 18, 12]))
[24, 20]
```

`map()` bricht ab, sobald das Ende *eines* der `iterables` erreicht wird.

filter

```
def even(x):
    return x % 2 == 0
def odd(x):
    return x % 2 == 1

>>> list(filter(even, [1,2,3,4,5]))
[2,4]
>>> list(filter(odd, [1,2,3,4,5]))
[1,3,5]
```

`filter(function, iterable)` liefert einen Iterator über alle Elemente `elt` aus `iterable`, für die `function(elt)` zu True auswertet.

lambda

- Normalerweise definieren wir Funktionen mit „def …“
  ⇨ Funktionen haben einen Namen
- Mit dem **`lambda`** Schlüsselwort können wir anonyme Funktionen definieren
  - **lambda** ⟨parameters⟩: ⟨expr⟩
- Lambda-Ausdrücke
  - werten zu Funktionen aus. Der Rückgabewert der Funktion ist der Wert von ⟨expr⟩
  - können überall verwendet werden, wo Funktionen erwartet werden (`map`, `filter`, …)
  - sind eingeschränkt auf einen einzelnen Ausdruck

lambda – Beispiele

```
>>> lambda x: x % 2 == 0
<function <lambda> at 0xb72e2dac>
>>> (lambda x: x % 2 == 0)(2)
True
>>> list(filter(lambda x: x % 2 == 0, [1,2,3,4,5]))
[2, 4]
>>> list(map(lambda x, y: x + y, [1,2,3], [4,5,6]))
[5, 7, 9]
>>> list(sorted([('a', 2), ('b', 1)], key=lambda x: x[0]))
[('a', 2), ('b', 1)]
>>> list(sorted([('a', 2), ('b', 1)], key=lambda x: x[1]))
[('b', 1), ('a', 2)]
```


Geschachtelte Funktionen

- Man darf Funktionen innerhalb von Funktionen definieren
  - die eingebettete Funktion kann Variablen in der einbettenden Funktion nur lesen – Werte können **nicht** zugewiesen werden
  - veränderliche Werte wie Listen können aber modifiziert werden
- Funktionen können die eingebettete Funktion per `return` zurückgegeben werden

```
def makeAdd(x):
  def add(y):
    return x + y
  return add

>>> plus3 = makeAdd(3)
>>> plus3(4)
7
```

Geschachtelte Funktionen

- Man darf Funktionen innerhalb von Funktionen definieren
  - das geht natürlich auch mit lambda-Ausdrücken

```
def makeAdd(x):
   return lambda y: x + y

>>> plus3 = makeAdd(3)
>>> plus3(4)
7
```