Programmierkurs Python I

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

Winter 2010/11

Übersicht

- Mehr zum Thema Iteratoren:
  - List Comprehensions
  - Generator-Ausdrücke
- Klassen-Methoden
- Dekoratoren

List Comprehensions

- List-Comprehensions sind eine elegante Alternative zu map, filter, etc.
- Syntax (einfache Variante)
  - `[⟨expr⟩ for ⟨var⟩ in ⟨iterable⟩]`
- Der Ausdruck wertet zu einer Liste aus, die für jedes Element `⟨var⟩` in `⟨iterable⟩` den entsprechenden Wert für `⟨expr⟩` enthält.
- Beispiel:
  - `[x + 1 for x in [1,2,3]]` ⇒ `[2,3,4]`

`map` vs. List Comprehensions

```
>>> list(map(lambda x: x + 3, [1,2,3]))
[4,5,6]

>>> [x + 3 for x in [1,2,3]]
[4,5,6]
```

Die Verwendung von List-Comprehensions führt im Vergleich zu map + lambda häufig zu einfacherem und besser lesbarem Code

List Comprehensions

- List-Comprehensions können mehrere for-Schleifen enthalten.
  - `[x + y for x in [1,2,3] for y in [4,5,6]]`
  - ⇒ `[5,6,7,6,7,8,7,8,9]`
- Der Ausdruck ist (weitgehend) äquivalent zu `result` nach Abarbeiten des Codes:

```
result = []
for x in [1,2,3]:
   for y in [4,5,6]:
      result.append(x + y)
```


List Comprehensions

- List-Comprehensions können Bedingungen enthalten
  - Bedingungen können irgendwo nach dem ersten for-Konstrukt stehen, aber nach Einführung der benutzen Variablen
- Zum Beispiel:
  - `[x for x in [2,3] if x % 2 == 0]`
    ⇒ `[2]`
  - `[x + y for x in [2,3] if x % 2 == 0 for y in [5,6]]`
    ⇒ `[7, 8]`
  - `[x + y for x in [2,3] for y in [5,6] if x % 2 == 0]`
    ⇒ `[7, 8]`

List Comprehensions

- [⟨expr⟩ for ⟨var⟩ in ⟨iterable⟩]
  - ⟨expr⟩ ist ein *beliebiger* Ausdruck
- typischerweise enthält ⟨expr⟩ die Variable ⟨var⟩
  - das muss aber nicht so sein: `[1 for x in range(3)]`
    ⇒ `[1,1,1]`
- man kann Comprehensions auch schachteln
  - `mat = [[1,2,3], [4,5,6], [7,8,9]]`
  - `[[row[i] for row in mat] for i in [0, 1, 2]]`
    ⇒ `[[1, 4, 7], [2, 5, 8], [3, 6, 9]]`

List Comprehensions

- [⟨expr⟩ for ⟨var⟩ in ⟨iterable⟩]
  - Die Variable ⟨var⟩ ist nur innerhalb der List-Comprehension sichtbar (neu in Python 3.x)

```
>>> x = 3
>>> x
3
>>> [x + 1 for x in range(5)]
[1, 2, 3, 4, 5]
>>> x
3
```

Generator Expressions

- Variante der List-Comprehensions:
  - **(⟨expr⟩ for ⟨var⟩ in ⟨iterable⟩)**
  - (erweiterte Syntax entsprechend)
- Statt zu einer Liste wertet dieser Ausdruck zu einem Iterator aus.
- Wenn die Comprehension das einzige Argument in einem Funktionsaufruf ist, kann man die Klammer weglassen
  - `sum(x*x for x in [1,2,3])` ⇒ 14

And now for something completely different ...

Klassen & Objekte (Wdh.)

```
>>> c1 = Counter()
>>> c2 = Counter()
>>> c1.count()
1
>>> c1.count()
2
>>> c2.count()
1
>>> c1.count()
3
```

```
class Counter:
    def __init__(self):
        self.counter = 0
    def count(self):
        self.counter += 1
        return self.counter
```

- Normalerweise werden Daten in Objekt-Instanzen („self“) gespeichert.
- Jede Instanz hat ihre eigenen Daten

Klassen-Variablen

```
>>> o1 = MyClass()
>>> o1.getCount1()
1
>>> o1.getCount2()
1
>>> o2 = MyClass()
>>> o2.getCount1()
2
>>> o1.getCount1()
2
```

```
class MyClass:
    count = 0
    def __init__(self):
        MyClass.count += 1
    def getCount1(self):
        return MyClass.count
    def getCount2(self):
        return self.count
```

- Klassen-Variablen gehören zur Klasse statt zum Objekt
- Sie haben für alle Instanzen (Objekte) den gleichen Wert

Klassen-Variablen

- Zugriff auf Klassen-Variablen:
  - ⟨class⟩.⟨name⟩ = ... (Schreiben)
  - ⟨class⟩.⟨name⟩ oder ⟨object⟩.⟨name⟩ (Lesen)
- Attribut-Zugriff (⟨object⟩.⟨name⟩)
  - erst in der Instanz nachsehen
  - dann in der Klasse
- ⇒ Instanz-Variablen gleichen Namens überdecken („hide“) die Klassen-Variable!

Klassen-Variablen – Achtung

```
>>> o1 = MyClass()
>>> o1.getCount1()
0
>>> o1.getCount2()
1
>>> o2 = MyClass()
>>> o2.getCount1()
0
>>> o2.getCount2()
1
```

```
class MyClass:
    count = 0
    def __init__(self):
        self.count += 1
    def getCount1(self):
        return MyClass.count
    def getCount2(self):
        return self.count
```

- Warum?

Klassen-Methoden

```
class MyDict(dict):
   @classmethod
   def fromKeys(cls, keys, default = None):
      return cls((key, default) for key in keys)

>>> MyDict.fromKeys([1,2,3])
{1: None, 2: None, 3: None}
```

- Neben Klassen-Variablen gibt es auch Klassen-Methoden
- Unterschied zu „normalen“ Methoden: das erste Argument ist die Klasse, nicht die Instanz (self).

Statische Methoden

- Neben Klassen-Methoden gibt es auch statische Methoden
- Diese werden mit `@staticmethod` ausgezeichnet und nehmen keine impliziten Argumente (kein `cls` bzw. `self`).
- Statische Methoden sollte nach Möglichkeit vermieden werden (stattdessen Klassen-Methoden verwenden).

Dekoratoren

- Dekoratoren sind Funktionen, die Funktionen als Argument nehmen und eine Funktion als Wert liefern
- @classmethod ist nur syntaktischer Zucker
- MyDict könnten wir auch so implementieren:

```
class MyDict(dict):
    def fromKeys(cls, keys, default = None):
        return cls((key, default) for key in keys)
    fromKeys = classmethod(fromkeys)
```


Geschachtelte Funktionen (Wdh.)

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

>>> plus3 = makeAdd(3)
>>> plus4 = makeAdd(4)
>>> plus3(4)
7
>>> plus4(4)
8
```

- Man kann Funktionen innerhalb von Funktionen definieren
- Die eingebettete Funktion kann die (lokalen) Variablen der einbettenden Funktion lesen
- Die eingebettete Funktion kann per `return` zurückgegeben werden

Geschachtelte Funktionen (Wdh.)

```
def makeCounter():
    counter = 0
    def count():
        counter += 1
        return counter
    return count

>>> c = makeCounter()
>>> c()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in count
UnboundLocalError: local variable 'counter' referenced before
assignment
```

- Die eingebettete Funktion kann die (lokalen) Variablen der einbettenden nicht setzen!
- Python interpretiert die Anweisung `counter += 1` als Zuweisung an eine lokale Variable der Funktion `count()`

Geschachtelte Funktionen (Wdh.)

```
def makeCounter():
    counter = [0]
    def count():
        counter[0] += 1
        return counter[0]
    return count

>>> c = makeCounter()
>>> c()
1
>>> c()
2
```

- Veränderbare Datentypen wie Listen können aber modifiziert werden
- Warum?

Geschachtelte Funktionen (Wdh.)

```
def makeCounter():
    counter = 0
    def count():
        nonlocal counter
        counter += 1
        return counter
    return count

>>> c = makeCounter()
>>> c()
1
>>> c()
2
```

mit **nonlocal** kann man Variablen als nicht-lokal deklarieren

Dekoratoren

- Dekoratoren sind Funktionen, die
  - Funktionen als Argument nehmen
  - und eine Funktion als Wert liefern.
- Wir können uns eigene Dekoratoren definieren

```
def log(fn):
    def wrap(*args):
        print(fn)
        return fn(*args)
    return wrap
```

Dekoratoren

```
def log(fn):
    def wrap(*args):
        print(fn)
        return fn(*args)
    return wrap
@log
def fun1():
    return 17
@log
def fun2():
    print(fun1())
```

```
>>> fun1()
<function fun1 at 0xb738686c>
17
>>> fun2()
<function fun2 at 0xb71fdeec>
<function fun1 at 0xb738686c>
17
```


Dekoratoren

```
def log(fn):
    def wrap(*args):
        print(fn)
        print('Args:', ', '.join(str(a) for a in args))
        return fn(*args)
    return wrap

@log
def fun1(x, y):
    return x + y
```

```
>>> fun1(1,2)
<function fun1 at 0xb71fd72c>
Args: 1, 2
3
```

Dekoratoren mit Argumenten

- Dekoratoren können Argumente haben
  - Streng genommen sind das keine Dekoratoren, sonder Funktionen, die Dekoratoren als Wert liefern

```
def log(name):
   def deco(fn):
      def wrap(*args):
         print(name)
         print('Args:', ', '.join([...]))
         return fn(*args)
      return wrap
   return deko
```


Klassen und Dekoratoren

- Wie Funktionen können auch Klassen innerhalb von Funktionen definiert und als Wert zurückgeliefert werden
- Das heißt:
  - Wir können auch Dekoratoren für Klassen schreiben.
  - Der Dekorator muss freilich eine Klasse liefern (keine Funktion)

Klassen und Dekoratoren

```
def singleton(cls):
   instance = None
   def getinstance():
      nonlocal instance
      if instance == None:
         instance = cls()
      return instance
   return getinstance

@singleton
class Test:
   def whoami(self): return id(self)
```

```
>>> i1 = Test()
>>> i2 = Test()
>>> i1.whoami()
3072413676
>>> i2.whoami()
3072413676
>>> i1 == i2
True
```