Programmierkurs Python I

Michaela Regneri

2010-01-21

(Folien basieren auf dem gemeinsamen Kurs mit Stefan Thater)

Übersicht

- Mehr „Abkürzungen“ in Python:
  - `map`
  - List Comprehensions
- Anonyme Funktionen, `lambda`
- Dekoratoren
- Mehr objektorientierte Programmierung
  - statische Methoden und Klassen-Methoden
  - Singletons
  - anonyme Klassen

map

- `map(function,iterable,...)` wendet `function` auf alle Elemente aus `iterable` an und gibt einen Iterator mit den Ergebnissen zurück
- Es müssen so viele Sammel-Objekte übergeben werden, wie `function` Argumente hat

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

> l = range(5)
> l
[0,1,2,3,4]
> list(map(plus3,l))
[3,4,5,6,7]
```


map

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

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

- Ist eine Liste kürzer als die andere, wird aufgehört wenn die kürzere Liste aufhört

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

List Comprehensions

- eine andere kompakte Art, Listen zu definieren.
  Basis-Syntax:

```
[x   for ... in ...]
```

Ausdruck, der eine Variable (x) enthält

for - Schleife (mit Variable (x) irgendwo)

optional mit Bedingung: `[x for x in ... if ...]`

- Einfaches Beispiel: Alle Werte in `l1` kopieren, die größer als 15 sind

```
> l1 = [13, 17, 19, 23]
> [x for x in l1 if x < 15]
[17, 19, 23]
```


List Comprehensions

- Comprehensions können `map()` simulieren:

```
> l = range(5)
> list(map(plus3,l))
[3,4,5,6,7]
```

```
> l = range(5)
> [plus3(x) for x in l]
[3,4,5,6,7]
```

- Comprehensions sind dabei flexibler; sie können auch mit verschachtelten Funktionen umgehen:

```
> l = range(5)
> [plus3(x)*2 for x in l]
[6, 8, 10, 12, 14]
```

```
> l = range(5)
> [plus3(plus3(x)) for x in l]
[6, 7, 8, 9, 10]
```

List Comprehensions

- man darf auch komplexe Comprehensions mit mehreren Variablen schreiben, dann werden alle möglichen Variablenbelegungen durchgerechnet

```
> l1, l2 = [15,20,25], [24,18,12]
> [x + y for x in l1 for y in l2]
[39, 33, 27, 44, 38, 32, 49, 43, 37]
```

- Die Reihenfolge entspricht einer for-Schleifen-Schachtelung von links nach rechts

```
def add(l1,l2):
 ret = []
 for x in l1
  for y in l2
    ret.append(x+y)
 return ret
```


List Comprehensions

- Bsp. mit Tupeln:

```
[(x,x+y) for x in l1 for y in l2 if x+y > 30]
```

- Bedingungen können irgendwo nach dem ersten for-Konstrukt stehen, aber nach Einführung der benutzen Variablen

```
[(x,x+y) for x in l1 if x%2 == 0 for y in l2]
[(x,x+y) for x in l1 for y in l2 if x%2 == 0]
```

```
[(x,x+y) for x in l1 for y in l2 if y%2 == 0]
X[(x,x+y) for x in l1 if y%2 == 0 for y in l2]
```

List Comprehensions

- die erste for-Schleife muss nicht zwangsläufig die Variable(n) davor beinhalten:

```
[x for i in [2,3,5,7] for x in range(101) if x % i == 0]
```

- man darf die Comprehensions auch schachteln (für Matrizen z.B.):

```
> 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]]
```


Verschachtelte Funktionen („nested functions“)

- man darf Funktionen in Funktionen definieren
- die inneren Funktionen dürfen die Variablen der äußeren lesen, aber können sie nicht überschreiben

```
def foo(n):
 x = 0
 def bar():
  print(‘in bar: x=‘, x, ‘n=‘,n)
 bar()
 return x
```

```
> foo(2):
in bar: x= 0 n= 2
2
```

Verschachtelte Funktionen („nested functions“)

- man darf Funktionen in Funktionen definieren
- die inneren Funktionen dürfen die Variablen der äußeren lesen, aber können sie nicht überschreiben

```
def foo(n):
 x = 0
 def bar():
  x += n
  print(‘x:‘), x
 bar()
 return x
```

```
> foo(2):
x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in foo
  File "<stdin>", line 4, in bar
UnboundLocalError: local variable 'x'
referenced before assignment
```


Anonyme Funktionen

- die inneren Funktionen können Hilfsfunktionen sein (aber man verwendet sie meistens nicht so)
- öfter ist die innere Funktion der Rückgabewert der äußeren Funktion
- Funktionen, die nicht explizit mit ihrem Namen definiert (und evtl aufgerufen) werden, heißen *anonyme* Funktionen

```
def makeAdd(n):
 def plus(x):
  return x + n
 return plus
```

```
> makeAdd(3)(4)
7
> p3 = makeAdd(3)
> p3(4)
7
```

(Aufruf & Definition)

anonyme Funktionen

(Definition)

Anonyme Funktionen: `lambda`

- `lambda` ist ein Schlüsselwort zum Erzeugen von anonymen Funktion
- Syntax: `lambda Parameter : Ausdruck`
- der Ausdruck ist der Rückgabewert, Parameter der / die Paremeter der anonymen Funktion
- der Ausdruck muss ein Ausdruck im eigentlichen Sinn sein (siehe 2. Vorlesung; also keine Schleifen, keine Anweisungen wie `yield`)

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

```
> p3 = makeAdd(3)
> p3(4)
7
```


Anonyme Funktionen: `lambda`

- lambda-Ausdrücke können geschachtelt sein:

```
makeAdd = lambda x: lambda y: x + y
```

```
> makeAdd(3)
<function <lambda> at 0x56a30>
```

```
> p3 = makeAdd(3)
> p3(4)
7
```

- mehrere Parameter können mit Kommata getrennt angegeben werden

```
> max = lambda x,y: x if x > y else y
> max(5,6)
6
```

Anonyme Funktionen: `lambda`

- lambda-Ausdrücke können geschachtelt sein:

```
> makeAdd = lambda x: lambda y: x + y
```

das ist erlaubt: Pythons bedingte Ausdrücke (‘conditional expressions‘) sind Ausdrücke, die je je nach Auswertung ihrer Bedingung zu einem anderen Wert auswerten. Doku: http://docs.python.org/3.1/reference/expressions.html#boolean-operations

- mehrere Parameter können mit Kommata getrennt angegeben werden

```
> max = lambda x,y: x if x > y else y
> max(5,6)
6
```


Dekoratoren („decorators“)

- Funktionen dürfen nicht nur Funkionen definieren und zurückgeben, sondern auch Funktionen als Parameter haben
- Funktionen, die Funktionen als Parameter nehmen und verändert zurück geben, heißen *Dekoratoren*

```
def deco(fun):
 def wrap():
  print(‘deko!‘)
  return fun()
 return wrap
```

```
def hello():
 print(‘Hallo Ente!‘)
> hello = deco(hello)
> hello()
‘deko!‘
‘Hallo Ente!‘
```

Dekoratoren

- Python bietet eine alternative Syntax zur Definition dekorierter Methoden; folgende Definitionen sind äquivalent

```
def hello():
 print('Hallo Ente!')
hello = deco(hello)
```

```
@deco
def hello():
 print('Hallo Ente')
```

- Man darf Funktionen auch mehrfach dekorieren:

```
def hello(): (...)
hello = deco1(deco2(deco3(hello)))
```

```
@deco1 @deco2 @deco3
def hello(): (...)
```


Dekoratoren

- Ein sinnvollerer Dekorator mit Funktions-Argumenten:

```
def print_args(fun):
    def wrap(*args):
        i = 1
        for arg in args:
            print('Argument', i, ':', arg)
            i += 1
        return fun(*args)
    return wrap
```

Dekoratoren

- Manchmal brauchen Dekoratoren zusätzliche Parameter
- Die Dekorator-Funktion selbst darf nur die Funktion als Parameter haben
- für zusätzliche Parameter dekoriert man den Dekorator:

```
def string_deco(d_arg):
 def deco(fun):
   def wrap(*args):
     print(d_arg)
     return fun(*args)
   return wrap
 return deco
```

```
@string_deco('Kekse!')
def hello():
  print('Hallo Ente!')
> hello()
 Kekse!
 Hallo Ente!
```


Statische Variablen und Methoden

- eine statische Variable einer Klasse hat immer den gleichen Wert, unabhängig von ihren Instanzen
- statische Variablen sind Klassen-Variablen; sie gehören zur Klasse(n-Objekt) statt zum aktuellen (Instanz-)Objekt

```
class Counted:
  count = 0

  def __init__(self):
    Counted.count +=1
```

Statische Variablen und Methoden

- Statische Methoden haben den gleichen Rückgabewert, unabhängig von Klassen-Instanzen
- wie die entsprechenden Variablen gehören auch sie zur Klasse selbst, nicht zu deren individueller Instanz
- sie nehmen kein *self*-Argument als Parameter
- sie können demnach auch nicht auf objektspezifische (*self.*bla) Variablen zugreifen - wohl aber auf statische Variablen

Statische Variablen und Methoden

- für statische Methoden gibt es einen vordefinierten Dekorator `staticmethod`

```
class Counted:
  _count = 0

  def __init__(self):
    Counted._count +=1

  @staticmethod
  def getcount():
    return Counted._count
```

```
> c1 = Counted()
> c2 = Counted()
> Counted.getcount()
2
> c1.getcount()
2
```

Aufruf mit Klassen-Objekt oder Instanz macht keinen Unterschied

Statische Methoden - Klassenmethoden

- statische Methoden werden - wenn sie nicht überschrieben werden - von allen abgeleiteten Klassen geerbt
- dh. das auch alle abgeleiteten Klassen den gleichen Rückgabewert für diese Methoden haben wie die Basis-Klasse
- für Methoden, die für alle Instanzen einer Klasse gleich sind, aber für jede abgeleitete Klasse verschieden, gibt es *Klassenmethoden*
- Klassenmethoden werden mit dem Dekorator `classmethod` und dem Default-Argument `cls` (statt `self`) defniniert


```
class Counted:
  count = 0

  def __init__(self):
    self.countup()

  @classmethod
  def countup(cls)
    try:
      cls.count +=1
    except UnboundLocalError:
      cls.count = 1

  @classmethod
  def getcount(cls):
    return cls.count
```

Das Singleton-Pattern

- ein Beispiel für ein Design-Pattern und den Einsatz von statischen Methoden ist das Singleton-Pattern
- Singletons sind Klassen, von denen es nur eine einzige Instanz gibt
- eine Methode in der Klasse sorgt dafür, dass man immer genau das selbe Objekt zurück bekommt, wenn man sich eine Instanz erzeugt
- Eine Möglichkeit zur Implementierung:
  - die Instanz selbst liegt in einer statischen Variable
  - die Methode, die die Instanz zurück gibt, ist auch statisch

Das Singleton-Pattern

- Eine Basis-Klasse für Singletons, von der Singleton-Klassen erben könnten:

```
class Singleton:
    __instance = None

    @classmethod
    def getInstance(cls, *args):
        if cls.__instance == None:
            cls.__instance = cls(*args)
        return cls.__instance
```

Anonyme Klassen

- Man darf in Methoden auch Klassen definieren und zurückgeben
- so kann man anonyme Klassen erzeugen

```
def makeNameClass(name):
  class Name:
    def __init__(self):
      self.name = name
    def sayName(self):
      print self.name

  return Name
```

```
> al_c = makeNameClass('Al')
> al1 = al_c()
> al1.sayName()
Al
```


Anonyme Klassen / Metaklassen

- Man kann auch Metaklassen definieren, die ihrerseits Klassen von Klassen definieren
- die allgemeinste Metaklasse in Python ist type - von der können andere Metaklassen erben
- eine Methode __new__ mus implementiert werden, die sagt, was beim Erzeugen einer neuen Klasse mit dieser Metaklasse passiert
- Doku:
  - http://www.python.org/doc/essays/metaclasses/meta-vladimir.txt
  - http://www.ibm.com/developerworks/linux/library/l-pymeta.html

Anonyme Klassen / Metaklassen

- aus http://www.ibm.com/developerworks/linux/library/l-pymeta.html :

  „Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).“

Zusammenfassung

- Mehr Python-Tools für Listen: `map` und Comprehensions
- Mehr Tricks mit und für Funktionen:
  - anonyme Funktionen, `lambda`
  - Dekoratoren
- Weiterführendes zur Objektorientierung:
  - statische Variablen und Methoden
  - Anwendungsbeispiel: Singletons
  - kurzer Ausblick: anonyme Klassen