Einführung in Python

Rebecca Breu

Verteilte Systeme und Grid-Computing
JSC
Forschungszentrum Jülich

Oktober 2007

Inhalt — Anhang

Neues in Python 2.5

Conditional Expressions

Kurze Schreibweise für bedingte Zuweisung. Statt:

```
if zahl<0:
    s = "Negativ"
else:
    s = "Positiv"
```

kann man schreiben:

```
s = "Negativ" if zahl<0 else "Positiv"
```

Das with-Statement

Einige Objekte bieten Kontext-Management an. Damit können try... finally-Blöcke einfacher geschrieben werden:

```
from __future__ import with_statement

with open("test.txt") as f:
    for line in f:
        print line
```

Nach dem with-Block ist das Dateiobjekt stets wieder geschlossen, auch wenn im Block eine Exception auftrat.

Partielle Funktionsanwendung

```
import functools

def add (a, b):
    return a + b

add_ten = functools.partial(add, b=10)
add_ten(42)
```

Flexiblere Funktionen

Funktionsparameter aus Listen und Dictionaries

```
def spam(a, b, c, d):
    print a, b, c, d
```

Man kann positionale Parameter aus Listen erzeugen:

```
>>> args = [3, 6, 2, 3]
>>> spam(*args)
3 6 2 3
```

Man kann Keyword-Paramter aus Dictionaries erzeugen:

```
>>> kwargs = {"c": 5, "a": 2, "b": 4, "d":1}
>>> spam(**kwargs)
2 4 5 1
```

Funktionen mit beliebigen Parametern

```
def spam(*args, **kwargs):
    for i in args:
        print i
    for i in kwargs:
        print i, kwargs[i]
```

```
>>> spam(1, 2, c=3, d=4)
1
2
c 3
d 4
```

Anonyme Funktionen

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

```
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Sinnvoll, wenn einfache Funktionen als Parameter übergeben werden sollen.

Funktionale Techniken mit Listen

List Comprehension

Abkürzende Schreibweise zum Erstellen von Listen aus for-Schleifen. Statt:

```
a = []
for i in range(10):
    a.append(i**2)
```

kann man schreiben:

```
a = [i**2 for i in range(10)]
```

Map

Anwenden einer Funktion auf alle Elemente einer Liste:

```
>>> a = [1.6, 4.0, 81.0, 9.0]
>>> map(math.sqrt, a)
[1.2649110640673518, 2.0, 9.0, 3.0]
>>> map(lambda x: x * 2, a)
[3.2000000000000002, 8.0, 162.0, 18.0]
```

Wenn die Funktion mehr als einen Parameter nimmt, kann je zusätzlichem Parameter eine weitere Liste übergeben werden:

```
>>> map(math.pow, a, [2 for i in a])
[2.5600000000000005, 16.0, 6561.0, 81.0]
```

Filter

Gibt Elemente einer Liste zurück, die nach Anwendung einer Funktion wahr sind:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> filter(lambda x: x % 2, a)
[1, 3, 5, 7, 9]
```

Reduce

Wendet Funktion auf die ersten beiden Elemente an, dann auf das Ergebnis und das nächste Element etc.

```
>>> def add(x, y):
...     return x + y
...
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> reduce(add, a)
45
```

Optionaler Startwert, der vor die Liste gesetzt wird:

```
>>> reduce(add, a, 10)
55
```

Iteratoren, Generatoren, Generator-Audrücke

Iteratoren

Was passiert, wenn `for` auf einem Objekt aufgerufen wird?

```
for i in obj:
    pass
```

- Auf `obj` wird die `__iter__`-Methode aufgerufen, welche einen Iterator zurückgibt
- Auf dem Iterator wird bei jedem Durchlauf `next()` aufgerufen
- Eine `StopIteration`-Ausnahme beendet die for-Schleife

Iteratoren

```
class Reverse:
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> for char in Reverse("spam"):
...     print char,
...
m a p s
```

Generatoren

Einfache Weise, Iteratoren zu erzeugen:

- Werden wie Funktionen definiert
- `yield`-Statement, um Daten zurückzugeben und beim nächsten `next`-Aufruf dort weiterzumachen

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse("spam"):
...     print char,
...
m a p s
```

Generator-Audrücke

Ähnlich zu List Comprehensions kann man anonyme Iteratoren erzeugen:

```
>>> data = "spam"
>>> for c in (data[i] for i in
...               range(len(data)-1, -1, -1)):
...     print c,
...
m a p s
```

Etwas Dynamik

Dynamische Attribute

Erinnerung: Man kann Attribute von Objekten zur Laufzeit hinzufügen:

```
class Empty:
    pass

a = Empty()
a.spam = 42
a.eggs = 17
```

Und entfernen:

```
del a.spam
```

getattr, setattr

Man kann Attribute von Objekten als Strings ansprechen:

```
import math
f = getattr(math, "sin")
print f(x) # sin(x)
```

```
a = Empty()
setattr(a, "spam", 42)
print a.spam
```

Nützlich, wenn man z.B. Attributnamen aus User-Input oder Dateien liest.