Python

Life Is Better Without Braces

Jochen Schulz

FH NORDAKADEMIE Elmshorn

Moderne Softwaretechnologien
26.9.2005

Was ist Python?

Python ist eine

- objektorientierte
- interaktive
- interpretierte

Programmiersprache für Windows, Unix und Macintosh, OS/2, Amiga...

Und...

Python ist...

- umfangreich („batteries included“)
- erweiterbar
- dynamisch
- portabel
- einfach

und macht Spaß!

Wer macht Python?

- ab 1990: Guido van Rossum
- seit Version 2.1: Python Software Foundation
- öffentliche Diskussion von *Python Enhancement Proposals* (PEPs)
- GvR „Benevolent Dictator for Life“ (BDFL)

Wer nutzt Python?

- Google
- www.ZEIT.de (Zope, Plone)
- die NORDAKADEMIE (Mailman)
- verfügbar bei vielen Webhostern
- MoinMoin (Wiki-Software)
- Nokia (Developer Kit für Telefone der *Series 60* verfügbar)
- ...

Verzweigungen

```
if a < b:
  foo()
elif a > b:
  bar()
else:
  baz()
```

- Kein case-Konstrukt vorhanden!

Schleifen

```
while a < b:
  foo()
```

```
for elem in [-5, 23, 42]:
  foo(elem)
```

- Keine *Repeat*-Schleife vorhanden!

Funktionen

```
def function(param):
  [...]
  return something
```

- `return` ist optional, aber guter Stil. Soll kein Wert zurückgegeben werden, kann `return` auch allein stehen. In diesem Fall wird implizit `None` zurückgeliefert.

Funktionsparameter

- Ein Funktionsparameter kann als *optional* gekennzeichnet werden, indem ein Default-Wert angegeben wird:

```
def func(param1, param2=0):
```

- Gültige Aufrufe von `func()`:

```
func(1, 4)
func(1)   # der zweite Parameter
          # ist implizit 0
func(param2=4, param1=1)
```

- Default-Parameter werden nur einmal ausgewertet!

Klassen

```
class C(object):

  classattr = 42

  def __init__(self, param):
    self.attr = param

  def method(self, param):
    print param

  def __private_method(self, param):
    return
```

Objekte und Attributzugriff

```
obj = C(23)
obj.attr
C.classattr
obj.classattr
obj.method('hallo')
```

```
23
42
42
hallo
```

Klassen

- Instanzvariablen werden generell in der `__init__`-Methode benannt – können aber jederzeit (auch außerhalb der Klasse!) zugewiesen werden.
- Es gibt keine Klassenmethoden – stattdessen verwendet man Funktionen auf Modulebene
- Klassenattribute können auch über Instanzen der Klasse angesprochen werden.
- Private Attribute werden durch zwei _ zu Beginn des Attributnames gekennzeichnet. Zugriff wird nicht effektiv verhindert, sondern durch *name mangling* und Konvention *discouraged.*

Ausnahmen

```
try:
  myfile = open('filename')
except IOError, (no, msg):
  print ``I/O error(%s): %s'' % (no, msg)
else:
  myfile.close()

class MyException(Exception):
  pass

raise MyException, 'Mein Fehler!'
```

None

- Repräsentiert „Nichts“, ähnlich `null` in Java
- Ist ein *Singleton*, d.h. es gibt immer nur ein Objekt dieses Typs (ermöglicht Prüfung auf Objektidentität: `foo is None`)

Zahlen

- `bool`: `True` oder `False`. Es existiert jeweils immer nur ein Objekt beider Werte
- `int`: ganze Zahlen
  Literal: `1 10 32391`
- `long`: große ganze Zahlen, beliebig lang
  Literal: `538478744543253245 23L`
- `float`: Fließkommazahlen
  Literal: `0.5`
- `complex` : Fließkommazahl + Imaginärteil
  Literal: `(3.5+23.7J)`

Sequenzen

- `tuple`: Listen fester Länge (immutable)
  Literal: `(a, b, c)`
- `str` : Zeichenketten (intern Tupel, also auch immutable)
  Literal: `'abc' ``abc'' ``````abc''''''`
- `list`: Listen variabler Länge beliebigen Inhalts.
  Literal: `[a, b, c]`
- Listenzugriff per Index, Zählung beginnt bei Null. Wird ein negativer Index angegeben, wird die Länge der Liste aufaddiert.
- Zugriff auf Ausschnitte über *Slicing*.

Listenzugriff

```
>>> a = [1, 2, 3, 4]
>>> a[0]
1
>>> a[-1]
4
>>> a[1:3]
[2, 3]
>>> a[2:]
[3, 4]
>>> a[:]       # flache Kopie der Liste
[1, 2, 3, 4]
```

Mappings

- Einziger Mapping-Type: `dict`
- Dictionaries verhalten sich wie Sequenzen, allerdings kann statt über Indizes über (fast) beliebige Keys auf die Elemente zugegriffen werden.
- Einschränkung: Keys müssen immutable sein.
- Die Reihenfolge der Elemente ist undefiniert – damit ist Slicing natürlich unmöglich.

```
d = { 'i02b79' : Student('Meier'),
      'i02b80' : Student('Petersen') }
print d['i02b79']
print d['abcdef']
```

```
Student Meier
Traceback (most recent call last):
File "<stdin>", line 1, in ?
KeyError: 'abcdef'
```

Die Funktion `filter()`

- gleiche Semantik wie in *Lisp*
- Rückgabewert sind alle Elemente einer Liste, für die eine gegebene Funktion `True` zurückliefert.
- Aufruf: `filter(function, list)`

filter() Beispiel

```
f = lambda x: x % 2 == 0
l = (34, 21, 432, 3, 4)

print filter(f, l)
```

```
(34, 432, 4)
```

Die Funktion map()

- gleiche Semantik wie in *Lisp*
- Rückgabewert ist eine neue Liste mit dem Ergebnis einer gegebenen Funktion für jedes Element einer Liste.
- Aufruf: `map(function, list)`

map() Beispiel

```
f = lambda x: x % 2 == 0
l = (34, 21, 432, 3, 4)

print map(f, l)
```

```
(True, False, True, False, True)
```

Die Funktion reduce()

- gleiche Semantik wie in *Lisp*
- Reduziert eine Liste auf einen einzelnen Wert. Es wird eine gegebene Funktion von links nach rechts auf alle Elemente einer Liste angewandt.
- Aufruf:
  `reduce(function, list[, initial])`

reduce() Beispiel

```
f = lambda x, y: x + y
l = (34, 21, 432, 3, 4)

print reduce(f, l)
print reduce(f, (), 5)
```

```
((((34+21) + 432) + 3) + 4) = 494
5
```

Hinweise zur Benutzung

- `filter()`, `map()`, `reduce()` und `lambda` werden wahrscheinlich in Python 3.0 nicht mehr als built-in verfügbar sein (PEP 3000).
- Statt komplizierter Lambda-Ausdrücke benutzt man besser Inline-Funktionen.
- Für den trivialen Fall `reduce(lambda x, y: x+y, list, 0)` kann man auch die eingebaute Funktion `sum(list)` benutzen.

List Comprehensions

- Ersetzen `filter()` und `map()` und vollständig
- Haben ihren Ursprung in *Haskell*
- Sind oft besser lesbar und flexibler als die obigen Methoden
- Form:

```
[expr for elem in list [condition]]
```

List Comprehensions: Beispiel I

```
l = (34, 21, 432, 3, 4)

[x for x in l if x % 2 ==  0]
```

```
(34, 432, 4)
```

- Entspricht genau der Anwendung von `filter()`

List Comprehensions: Beispiel II

```
l = (34, 21, 432, 3, 4)

[x % 2 == 0 for x in l]
```

```
(True, False, True, False, True)
```

- Entspricht genau der Anwendung von `map()`

Generator-Funktionen: Beispiel I

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

for char in reverse('Lotti'):
  print char,
```

```
 i t t o L
```

Generator-Funktionen: Beispiel II

```
def fibonacci():
  a, b = 0, 1
  while True:
    yield a
    a, b = b, a+b

iterator = iter(fibonacci())
for i in range(0, 13):
  print iterator.next()
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144
```

Generator-Funktionen

- Generatoren sind zustandsbehaftete Funktionen.
- Statt `return` geben sie Ihr Ergebnis mit `yield` zurück.
- Wiederholte Aufrufe können unterschiedliche Ergebnisse liefern.
- Generatoren können beliebig viele Elemente zurückgeben.
- Im Gegensatz zu List Comprehensions erzeugen sie nicht eine ganze Liste, sondern jedes Element einzeln (weniger Speicherbedarf!).

Generator-Expressions

Eine Mischung aus List Comprehensions und Generator-Funktionen:

- Erzeugen ein Element zur Zeit
- Haben eine einfache Syntax zur Erzeugung:

```
(x**2 for x in (2, 6, 12, 32))
```

```
[4, 36, 144, 1024]
```

Simulation von built-in Types

Das Verhalten von Objekten bei Anwendung der Standardsyntax kann angepasst werden:

- Vergleiche:
  `< > <= >= == !=`
- arithmetische Operationen:
  `+ - * / % and or ...`
- Zuriff per Index oder Key:
  `obj[idx] del obj[idx]`
- Iteration:
  `for elem in obj`

Vergleiche: Beispiel

```
class Rectangle(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def area():
    return self.x * self.y

  def __lt__(self, other):
    return self.area() < other.area()

  def __gt__(self, other):
    return self.area() > other.area()
```

Vergleiche: Beispiel

```
a = Rectangle(2, 4)
b = Rectangle(4, 3)
print a < b
print a > b
```

```
True
False
```

Arithmetik: Beispiel

```
class Number(object):

  def __init__(self, value):
    self.value = value

  def __add__(self, other):
    return Number(self.value + other.value)

  def __sub__(self, other):
    return Number(self.value - other.value)
```

Arithmetik: Beispiel

```
a = Number(5)
b = Number(7)
c = a + b
d = a - b
print type(c)
print c.value
print d.value
```

```
<class '__main__.Number'>
12
-2
```

Listenverhalten: Beispiel

```
class Zenturie(object):

  def __init__(self):
    self.studenten = {}

  def __len__(self):
    return len(self.studenten)

  def __iter__(self):
    for key in self.studenten.iterkeys():
      yield key
```

Listenverhalten: Beispiel

```
def __getitem__(self, key):
  return self.studenten[key]

def __setitem__(self, key, value):
  self.studenten[key] = value

def __delitem__(self, key)
  del self.studenten[key]
```

Listenverhalten: Beispiel

```
myZenturie = Zenturie()

myZenturie['1862'] = 'Fuerst v. Bismarck'
myZenturie['1949'] = 'Konrad Adenauer'

for student in myZenturie:
  print myZenturie[student]
```

```
Fuerst v. Bismarck
Konrad Adenauer
```

Listenverhalten: Beispiel

```
print len(myZenturie)

del MyZenturie['1862']

print len(myZenturie)
```

```
2

1
```

Demo

Zusammenfassung

Python hat Schwächen:

- keine Typsicherheit
- Performance
- keine wirklich privaten Attribute
- keine Interfaces

Zusammenfassung

Python hat Stärken:

- Keine Typsicherheit :)
- Schnell zu erlernen
- Hohe Produktivität
- Sehr gute Lesbarkeit
- Portabilität
- Dynamik
- Offene Entwicklung, freie Dokumentation
- Kostenfrei

Wofür kann ich Python nicht benutzen?

- Anwendungen, bei denen Performance entscheidend ist
- Anwendungen auf Embedded Devices
- Hardwarenahe Programmierung

Wofür kann ich Python benutzen?

- Ersatz für Shellskripte
- CGI-Programme
- Portable Anwendungen
- Rapid Prototyping
- Anwendungen zwischen 10 und 10.000.000 Zeilen

Anlaufstellen

- docs.python.org – offizielle Dokumentation, inkl. Tutorial, Sprachbeschreibung und Moduldokumentation
- DiveIntoPython.org – sehr gutes Buch für Python- Einsteiger mit Programmiererfahrung (auch gedruckt erhältlich)
- CafePy.com – enthält einige gute Artikel über Python-Internas