And now for something completely different...

Python

Dirk Loss

29. November 2001
Seminar „Alternative Programmiersprachen“
Universität Münster

Drei Vorträge über Python

Vortrag 1: Einführung in Python

(Dirk Loss)

Vortrag 2: Möglichkeiten und Einsatzbereiche von Python

(Hendrik Maeder)

Vortrag 3: Vergleich von Python mit PHP

(Marcel Lüttmann)

Eigenschaften von Python

- interpretiert, interaktiv
- dynamisch typisiert, aber streng getypt
- objektorientiert
- high-level-Datentypen
- Open Source, portabel
- erweiterbar, einbettbar
- ausführlich dokumentiert
- vielfältige Bibliotheken
- aktiv weiterentwickelt

Entwicklungsgeschichte

1989	Guido van Rossum, Amsterdam, entwickelt Python während des Amoeba-Projekts
1991	erstes Release
...	
1999	Python 1.5
2000	Python 2.0
2001	Python 2.1
heute	Python 2.2beta2

- Python Software Foundation
- Anwender z.B. Disney, NASA, SGI, Infoseek...
- Software-Projekte: Zope, Mailman, BSCW...

Einführendes Beispiel

```
n = 10                          # n wird integer
while n <= 13:
    if n % 2 == 0:
        print n, "gerade"
    else:
        print n, "ungerade"
    n = n + 1
```

- bekannte Operatoren und Kontrollstrukturen
- keine Deklarationen von Variablen
- Blöcke durch Einrückung definiert
  (keine Klammern, keine Semikolons)

Ausführen von Python-Code

- Code in Moduldatei

```
/home/dirk/$ python test.py
```

- interaktive Eingabe

```
Python 2.1.1 (#20, Jul 20, 2001)
>>> print 3*4
12
```

- als Shell-Skript

```
#!/usr/bin/python
print 'Hello world!'
```

- aus einer IDE

IDLE, PythonWin, Komodo, BlackAdder

- aus einem C-Programm

```
#include "Python.h"
Py_initialize;
PyRun_SimpleString("print 4");
```

- und andere Möglichkeiten...

Operatoren und Kontrollstrukturen

Operatoren

`+,-,*,/,and, or, not, ==, !=, <, >, **, in, is ...`

Besonderheiten:

Bereichstest: `1 < a < 5`

Mehrfachzuweisung: `a = b = 4`

Kontrollstrukturen

`if, while, for, break, continue`

Besonderheiten:

for iteriert über Sequenz `for a in [1,2,3,4]: print a`

kein switch bzw. case, kein do while

optionaler else-Zweig hinter `for` und `while`

Eingebaute Datentypen

- Integer `42, 0x26, 052`
- Double `3.1415926535987931`
- String `'Python'`
- Tupel `(3, 4.0)`
- Liste `['spam', 4, 1.0]`
- Dictionary `{'egg':23, 'ham':40}`
- Long Integer `2323299094L`
- Complex `(-1+4j)`
- Unicode String `u"Hello\u0020world!"`

- kein Array, Boolean, Char
- aber: neue Datentypen sind über Module in C realisierbar

Strings

- Strings sind unveränderliche Sequenzen von Zeichen

```
s = 'eggs'
len(s)                    # 4   (eingebaute Funktion)
s[0]                      # 'e'
s[-1]                     # 's'
s[1]='a'                  # TypeError!
```

- Slicing

```
s = 'sliceofspam'
s[2:5]                      # 'ice'
s[:5]                       # 'slice'
s[-4:]                      # 'spam'
```

- Sonstiges

```
'a' * 99999                 # 'aaaaaaaaaaaaaaaaaa...'
"eggs".capitalize()         # 'EGGS'
"The %s who say %s' % ('knights', 'Ni!')
'Neue Zeile\n'              # \n ist Newline
```

Tupel

- Tupel sind Sequenzen

```
t = (0, "python", 1.5)          # heterogen
t[1]                            # 'python'
t[1:3]                          # ('python', 1.5)

v = (3, (1,2,3), 7)             # verschachtelbar
v[1]                            # (1,2,3)
```

- Ein- und Auspacken von Tupeln

```
a, b, c = 4, "Python", -1.0
x, y = y, x                     # tauscht Werte
```

- Tupel sind unveränderlich

Listen

- Listen sind veränderliche Sequenzen

```
L = ['spam', 3.14, 42]
L[1] = 'eggs'                   # ['spam', 'eggs', 42]
L[0:2] = [4,3,2,1]              # [4,3,2,1,42]

L.sort()                        # [1,2,3,4,42]
L.append('a')                   # [1,2,3,4,42,'a']
L.pop()                         # 'a' (Liste als Stack)

42 in L                         # 1    (true)
range(4)                        # [0,1,2,3]
```

- List comprehensions

```
[x*x for x in [1,2,3,4] if x * x > 6]      # [9, 16]
```

Dictionaries

- erweiterbare Hash-Tabellen bzw. assoziative Arrays

```
dict = {'schmitz': 5325, 'meier': 2345,
            'huber': 'kein Telefon'}

dict['schmitz']         # 5325
dict.keys()             # ['meier','schmitz','huber']
dict.values()           # [2345, 5325, 'kein Telefon']
dict.has_key('meier')       # 1   (true)
dict['mueller'] = 9876      # neu einfügen
del dict['mueller']         # wieder entfernen


for key in dict.keys():
    print key, "--", dict[key]
```

- Zugriff über eindeutigen Schlüssel
- heterogen, ungeordnet, veränderlich

Dateien

```
myfile = open('text.txt', 'r')      # zum Lesen öffnen
s = myfile.readlines()              # Liste von Zeilen
print "Anzahl Zeilen:", len(s)
myfile.close()                  # Schließen (optional)
```

- weitere Methoden auf Dateien, z.B.:

```
s = myfile.read()          # in einen String einlesen
s = myfile.read(n)         # n Bytes einlesen
s = myfile.readline()      # nächste Zeile
s = myfile.seek(pos)       # Zugriff über Position
```

- Standardeingabe/-ausgabe

```
myfile = sys.stdin
myfile = sys.stdout
```

Exceptions (1)

- Abfangen von Laufzeitfehlern

```
try:
    myfile = open('text.txt','r')
except IOError, data:
    print 'Oeffnen nicht moeglich: ', data
else:
    print 'Alles ok.'
```

- Eigene Exceptions

```
myException = 'Fehler'          # String als Exception

...

if notOK: raise myException

...
```

Exceptions (2)

- Alternativ: finally

```
try:
   doSomething()
finally:                        # fängt Exception nicht ab
   print "wird immer ausgeführt"
```

- Klassen als Exceptions:

  eigene Exceptions ableiten von Basisklasse "Exception"

- Assertions: `assert test, data` entspricht:

```
if __debug__:
    if not test:
        raise AssertionError, data
```

Funktionen

```
def factorial(n):
    "Fakultaet von n"          # optionaler Doc-String
    result = 1L                # Long Integer
    while n > 1:
        result = result * n
        n = n - 1
    return result            # optional, sonst 'None'
```

- keine Deklaration von Argumenttypen
- Default-Parameter und überzählige Argumente möglich

  ```
  def factorial(n=10)
  ```

  ```
  def factorial(n,*others)
  ```

- "call-by-assignment" (~ call-by-reference)
- Funktionen höherer Ordnung, anonyme Funktionen, `apply`

Objekte

- „Everything is an object“:

  Zahlen, Strings, Listen, Tupel, Dictionaries, Dateien, Funktionen, Klassen, Module...

- Objekte besitzen

Identität	`id(obj)`
Typ	`type(obj)`
Inhalt	

- Ein Ausdruck berechnet nicht Werte, sondern liefert ein Objekt

```
id("Python")                    # 135181080
type("Python")                  # <type 'string'>
id(5+4)                         # 125140036
type(5+4)                       # <type 'int'>
```

Namen (1)

- Zuweisungen verändern nicht Speicherzellen, sondern binden Namen an Objekte

```
a = 42
id(a)                                   # 135139460
type(a)                                 # <type 'int'>
```

- Namen sind Verweise auf Objekte

```
b = a
id(b)                                   # 135139460
b is a                                  # 1  (true)
```

- Namen können neuen Objekten zugewiesen werden, die alte Bindung geht dann verloren

```
a = [1,4,5]                 # a weist auf Liste
a = a[2]                    # a weist auf Element der Liste
def a(n):return n           # a weist auf Funktion
type(a)                     # <type 'function'>
```

Namen (2)

- Unterschied zwischen dem Binden von Namen und dem Verändern von Objektinhalten

```
a = [1, 2]
b = a
b[0] = [5]          # erstes Element der Liste ändern
print b             # [[5], 2]
print a             # [[5], 2]


a = [1,2]
b = a
b = [5]             # neues Objekt [5] an b binden
print b             # [5]
print a             # [1, 2]
```

Namensräume (namespaces)

- Die Zuordnungen von Namen zu Objekten werden in Namensräumen gespeichert
- Namensräume kann man sich als Dictionaries vorstellen
- Folgende Objekte besitzen eigene Namensräume

  Funktionen

  Module

  Klassen, Instanzen

- Namen qualifizieren:

```
objekt.name = wert
wert = objekt.name
```

Geltungsbereiche (scopes)

- "Wo werden unqualifizierte Namen gesucht?"
  1. lokal — im Namensraum der Funktion/Methode/Klasse
  2. global — im Namensraum des Moduls (d.h. der Datei)
  3. built-in — im eingebauten Namensraum
- Namen werden grundsätzlich angelegt im lokalen Geltungsbereich
- Ausnahme: Anweisung `global var1, [var2] ...`

  anlegen im globalen Geltungsbereich

  suchen im globalen und eingebauten Geltungsbereich
- Eingebaute und globale Namen können von lokalen verdeckt werden!

Geltungsbereiche - Beispiel

```
# globaler Geltungsbereich
g = 1
a = 3
b = 15

def func(x):
    # lokaler Geltungsbereich
    global g
    g = b + 1
    a = x * 2
    return a

print func(15)        # 30
print a               # 3
print g               # 16
print b               # 15
print x               # NameError: name 'x' not defined
print func            # <function 'func' at 009970C>
```

Verschachtelte Geltungsbereiche (nested scopes)

- Geltungsbereiche waren bis v2.0 nicht verschachtelbar

```
x = 2.0
def outer():
    x = 2.2
    def inner()
        print x
    inner()

outer()             # 2.0   (in v2.0)
                    # 2.2   (in v2.2)
```

- In v2.1 sind nested scopes optional zuschaltbar
- Ab v2.2 sind nested scopes standardmäßig aktiviert

Module

- Module sind Python-Dateien (*.py) oder Erweiterungen in C

```
import math                 # importiert den Namen 'math'
print math.pi               # 3.1415926535926535

from math import exp        # import math; exp = math.exp
exp(1)                      # 2.718281828590451

from math import *          # alle Namen
```

- Beim ersten Importieren des Moduls wird der Code ausgeführt (Seiteneffekte möglich)
- Dynamisches Nachladen von Modulen `reload()`
- Modulpakete

Klassen (1)

```
class Counter:
    def __init__(self):           # Konstruktor
        self.value = 0            # Attribut der Instanz
    def inc(self, n):             # Methode
        self.value = self.value + n

c = Counter()                     # Instanz erzeugen
print c.value                     # 0
c.inc(3)                          # Methode aufrufen
Counter.inc(c, 3)                 # Alternativschreibweise
print c.value                     # 6
c.value = 7                       # Attribute sind public
```

- Bei Aufruf der Methode über die Instanz wird implizit eine Referenz auf die Instanz als erstes Argument an die Methode übergeben

- Beim Aufruf der Methode über die Klasse muss eine Instanz explizit als erstes Argument übergeben werden

Klassen (2)

```
class Derived(Counter):
    def inc(self, n = 1):        # Methode überschreiben
        Counter.inc(self, n)     # kein 'super'
    def dec(self, n = 1):        # Klasse erweitern
        self.value -= n
    def __repr__(self):          # Operator overloading
        print 'Zaehlerstand =', self.value

d = Derived()
d.inc()
print d                          # 'Zaehlerstand = 1'
```

- Instanzattribute: `self.value`
- Klassenattribute: `Klasse.value`
- lokale Variablen einer Methode: `value`

Klassen (3)

- Mehrfachvererbung möglich `class MyClass(A, B)`
- Eingebaute Datentypen sind keine Klassen (erst ab v2.2)

  Workaround: Hüllklassen `UserList, UserDict, ...`
- keine Klassenmethoden (erst ab v2.2)
- Information hiding nur per Konvention
  (keine privaten Attribute):

  "Namensverstümmelung" verhindert ungewollte Namenskollisionen:

  Der Name `__x` in der Klasse `MyClass` wird automatisch zu `_MyClass__x` umgesetzt

Zwischenfazit

- vertraute Kontrollstrukturen und Operatoren
- angenehme Syntax
- leistungsfähige Datentypen bereits eingebaut
- objektorientiert

– gewöhnungsbedürftiges Namensraumkonzept
– kein strenges Information Hiding

- gut für kleine Projekte und Rapid Prototyping

In der nächsten Woche...

- Vortrag 2: (Hendrik Maeder)

  Standard-Library

  Funktionale Programmieraspekte

  Java-Integration

  GUIs

  Python-Speicherverwaltung

- Vortrag 3: (Marcel Lüttmann)

  Vergleich von Python mit PHP

Fragen?

"A language that makes it hard to write elegant code
makes it hard to write good code." (Eric S. Raymond)

Quellen

- Lutz, M.: „Einführung in Python“, O'Reilly 2000
- Lutz, M.: „Python – kurz und gut“, O'Reilly, 1998
- van Rossum, G.; Drake, F.: „Python Tutorial“
  http://www.python.org/doc/current/tut/tut.html
- van Rossum, G.; Drake, F.: „Python Library reference“
  http://www.python.org/doc/current/lib/lib.html
- van Rossum, G.; Drake, F.: „Python Reference Manual“
  http://www.python.org/doc/current/ref/ref.html
- Koch, B.: „Einführung in Python“
  http://www.vorlesungen.uni-osnabrueck.de/informatik/altprog00/python/