Schlangenbeschwörung für Anfänger

Eine Einführung in Python

Wolfgang Fellger

inf.misc

Einführung

- Begonnen Ende 1989 als Hobbyprojekt des heutigen *BDFL* Guido van Rossum
- Finale Version 1.0 wurde 1994 veröffentlicht
- 2000 erschien Version 2.0, aktuell seit 2008: Version 3.0
- Im Linux-Umfeld heute eine der wichtigsten Skriptsprachen
- Kommerziell im Einsatz unter anderem bei YouTube, Google, ILM
- Verfügbar für viele Betriebssysteme und Plattformen:
  - Linux, Windows, Mac OS, Symbian, iPod, AIX, BeOS, Solaris, ...
  - Nativ (CPython), Java RE (Jython), .NET/Mono (IronPython)

Ziele:

- Einfache, intuitive, aber gleichzeitig mächtige Syntax
- Gut lesbarer Quelltext
- Kurze Entwicklungszeiten ermöglichen, u.a. durch eine große Standardbibliothek (“batteries included”)

- Blockstruktur durch Einrücken
- Wenige, englische Schlüsselwörter
- Seit Python 2 ist die Sprache vollständig objektorientiert, absolut *alles* ist ein Objekt; auch Funktionen, Klassendefinitionen und Module.
- “very high level language”: Automatische Speicherverwaltung, abstrakte Datentypen (nur ein Integer-Typ, Listen und Dictionaries gehören zum Sprachkern)
- Multi-Paradigma-Sprache: Python erlaubt prozedurale, objektorientierte und (eingeschränkt) funktionale Programmierung
- Unicode-Unterstützung
- einfache Anbindung an andere Sprachen

Nachteile

- Langsam
- Erzwungene Blockstruktur
- Einige Konstrukte, wie `do..while` oder `switch`, fehlen
- OOP-Ansatz gewöhnungsbedürftig

IDEs

- eric
- IDLE
- Stani's Python Editor
- PyDev (eclipse-Plugin)
- NetBeans-Erweiterung
- Syntaxhighlighting in fast allen Texteditoren verfügbar

Dokumentation

- Primäre Quelle: http://python.org/doc/
- Python verfügt über eine integrierte Hilfe. Die interaktive Shell liefert mit dem Kommando help(irgendwas) eine Kurz-Doku. Das funktioniert für Module, Funktionen, Typen und einzelne Objekte. help() verfügt auch über einen interaktiven Modus, wenn es ohne Parameter aufgerufen wird.
- Kürzer: dir(object) gibt eine Liste der Attribute und Methoden zurück.

Datentypen

Die Standarddatentypen sind bereits high-level:

- Ganze Zahlen (beliebig groß)
- Fließkommazahlen (IEEE double)
- Komplexe Zahlen
- Byte- und Unicodestrings
- Listen und Tupel
- Mengen
- Dictionaries (“assoziatives Array”, “Hashmap”)
- Dateien

Strings

- Strings können von ’ oder " umschlossen sein
- Long-Strings sind von """ umschlossen und können sich über mehrere Zeilen erstrecken
- Aufeinanderfolgende Literale werden automatisch konkateniert
- Strings sind Objekte und bringen eigene Methoden mit, zum Beispiel `capitalize, split, isdigit, join, partition`
- Maskierungen (`\\, \n, \t`) wie in C
- Alternativ *Raw-Strings*: `len( r'\\' ) == 2`
- Unterscheidung von Unicode- und Bytestrings mit Präfixen 'b' und 'u'. Standard in Python 2 ist Byte-, in Python 3 Unicodestring

Parameterübergabe

- Optionale, benannte und Sammel-Parameter sind möglich:

```
def foo(spam='spam', *egg, **sausage):
    print spam
    print egg
    print sausage

foo('SPAM', 'egg', 'and', 'spam', something='different')
```

- Übergabe ausschließlich per Referenz (wie bei Perl). Um Chaos zu vermeiden, werden alle Typen zwischen zwei Gruppen unterschieden: mutable und immutable.

immutable

Alle Zahlenwerte, Strings und Tupel sind immutable. Sie können nach der Instanzierung nicht mehr verändert (und daher auch bedenkenlos einer Funktion übergeben) werden. Operationen auf diesen Datentypen sind selbstverständlich trotzdem möglich, liefern dann aber eine veränderte *Kopie* des ursprünglichen Objekts.

mutable

Listen, Dictionaries und die meisten anderen Typen sind mutable, d.h. der Zustand der Objekte kann verändert werden. Änderungen in einer Funktion wirken sich also auf den Aufrufer aus; falls dies nicht gewünscht wird, muss eine Kopie manuell erstellt werden.

Gültigkeitsbereiche

Variablen können in Python automatisch erstellt werden und gelten ab dann modul- bzw. funktionsweit.
Python unterstützt verschachtelte Funktionsdefinitionen und damit auch verschachtelte Scopes.
Lesezugriffe sind wie in anderen Sprachen gewohnt möglich, bei Schreibzugriffen muss der Bezeichner allerdings, um zwischen einer lokalen Deklaration zu unterscheiden, vorher angegeben werden:

- `global`, um einen Namen dem globalen Namespace zuzuordnen
- Ab Python 3: `nonlocal`, um den Namen aufwärts suchend zu holen (schlägt fehl wenn es den Bezeichner nicht gibt)

Importierte Module haben ihren eigenen 'Namensraum' (tatsächlich ist ein importiertes Modul ein Objekt mit entsprechenden Eigenschaften).
Um einzelne oder alle Bezeichner eines Moduls in den Namensraum des Moduls zu bringen, kann:

```
from modul import Bezeichner
```

verwendet werden.
Die von einem Modul exportierten Bezeichner können eingeschränkt werden.

Gültigkeitsbereiche - Beispiel

```
a = 21

def f():
    global a
    a *= 2

def g():
    print a*2
```

Beispiele (1)

Das unvermeidliche “Hello World”-Programm:

```
print "Hello World"
```

Beispiele (1)

Das unvermeidliche “Hello World”-Programm:

```
print("Hello World")
```

Beispiele (2)

Aufgabe: Schreibe ein Programm, das vom Benutzer mehrere Wörter einliest, diese sortiert, Duplikate entfernt und das Ergebnis ausgibt.

```
print sorted(set(raw_input().split()))
```

Beispiele (2)

Aufgabe: Schreibe ein Programm, das vom Benutzer mehrere Wörter einliest, diese sortiert, Duplikate entfernt und das Ergebnis ausgibt.

```
print '; '.join(sorted(set(raw_input().split())))
```

Beispiele (3)

Aufgabe: Schreibe ein Programm, das den Benutzer mehrere Zahlen eingeben lässt und diese multipliziert.

```
print reduce(lambda x, y: int(x) * int(y), raw_input().split())
```

Beispiele (4)

Aufgabe: Schreibe ein Programm, das alle zweibuchstabigen Permutationen eines vom Nutzer eingebenen Wortes ausgibt.

```
import itertools
print list(itertools.permutations(raw_input(), 2))
```

Sprachkonstrukte

- Exceptions: Wie üblich
- Kurzschreibweisen für Zuweisungen aus C (a += 1, b *= 2)
- Funktionen, Methoden, Module sind normale Objekte und können damit insbesondere als Parameter übergeben werden
- %-Operator zur Interpretation von Formatstrings:

```
print 'Guten %s, %s!' % (tageszeit, name)
```

- Ternärer Operator:

```
v = True-Ausdruck if Bedingung else False-Ausdruck
```

OOP

```
class C(object):                # Klassendefinition mit Vorfahrenklasse
    a = 0                       # KLASSEN-Attribute ("static")
    b = 1

    def public(self, bar): # 'self' als impliziter erster Parameter
        print self.a
    def _protected(self): # 'protected'-Methode
        print self.b
    def __private(self):  # 'private'-Methode
        print self.c

    def __init__(self):   # Konstruktor
        self.c = 2        # INSTANZ-Attribute

    @staticmethod         # Statische Methoden
    def static():         # (kein self erforderlich)
        C.a *= 2
```

Schleifen

Es gibt nur zwei Schleifenarten: `for` und `while`. `for` ist eigentlich ein foreach und durchläuft ein beliebiges iterierbares Objekt elementweise. Möchte man einen Index durchlaufen, so kann dies mit der `range`-Funktion simuliert werden:

```
for i in range(17, 42):
    print i                 # gibt die Zahlen 17 bis 41 aus
```

In der häufigen Situation, das neben dem Element selbst auch der Index benötigt wird, lässt sich auch die `enumerate`-Funktion verwenden:

```
for index, item in enumerate(someList):
    print index, item   # gibt die Elemente in someList
                        # mit ihrem Index aus
```

Slices

Python ermöglicht einen sehr einfachen Zugriff nicht nur auf einzelne Listenelmente, sondern auch auf Teillisten:

```
aList = [1, 2, 3, 4, 5]
print aList[0]          # Ausgabe: 1
print aList[-2]         # Ausgabe: 4
print aList[2:4]        # Ausgabe: [3, 4]
print aList[2:]         # Ausgabe: [3, 4, 5]
print aList[:2]         # Ausgabe: [1, 2]
print aList[-2:]        # Ausgabe: [4, 5]
```

An Slices kann auch ein beliebiges iterable-Objekt zugewiesen werden, und dabei muss nicht einmal die Anzahl der Elemente übereinstimmen:

```
aList = [1, 2, 3, 4, 5]
aList[:2] = [17, 23]    # aList == [17, 23, 3, 4, 5]
aList[:2] = [42]        # aList == [42, 3, 4, 5]
aList[1:2] = [7, 7, 7]  # aList == [42, 7, 7, 7, 4, 5]
aList[1:] = "spam"      # aList == [42, 's', 'p', 'a', 'm']
```

List Comprehension

List Comprehension ist ein leistungsfähiges funktionales Konzept, das bis auf die Schlüsselwörter von Haskell übernommen wurde. Einfach gesagt können bei einer Listendefinition bereits Berechnungen vorgenommen werden:

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

liefert eine Liste der ersten zehn Quadratzahlen.
Es ist auch möglich, Elemente nur bedingt zu übernehmen:

```
a = 42
[x for x in range(1, a+1) if a % x == 0]
```

liefert die Liste aller Teiler von 42.

List Comprehension - Beispiel

```
def quicksort(items):
    if len(items) > 0:
        pivot = items.pop()
        left  = [i for i in items if i < pivot]
        right = [i for i in items if i >= pivot]
        return quicksort(left) + [pivot] + quicksort(right)
    else:
        return []

print quicksort([2, 3, 5, 8, 1, 4, 7, 6])
```

Operatorenverkettung

Aufgabe: Vergleiche drei boolsche Werte auf Gleichhheit.
Naiver Ansatz:

```
A == B == C
```

C:

```
A == B == C <=>
(A == B) == C =>
(False == False) == False =>
(True) == False => False!
```

Python:

```
A == B == C <=>
A == B and B == C.
```

Python interpretiert Aneinderreihungen von “Relationsoperatoren” immer als eine solche und-verknüpfte Kette. Damit ist zum Beispiel auch der Ausdruck:

```
if 23 < A <= 42:
```

gültig und funktioniert so wie man das von der mathematischen Konvention erwarten würde.

Mehrfach-Zuweisungen

Aufgabe: Tausche zwei Variablen.
C (einfach):

```
tmp = a;
a = b;
b = tmp;
```

C (trickreich):

```
a ^= b;
b ^= a;
a ^= b;
```

Python:

```
b, a = a, b
```

Zuweisungen können an mehrere Variablen auf einmal erfolgen. Als einzige Einschränkung muss die Anzahl der Elemente links und rechts passen; entweder können (wie hier) mehrere Variablen stehen, oder aber jedes iterierbare Objekt verwendet werden.
Weitere Beispiele:

```
a, b, c = 1, 2, 3  # mehrfach-Deklaration und Initialisierung
index, value = meintupel          # 'Auspacken' von Tupeln
for a, b in [(0,1), (1,3), (2,6)]: # Mehrere Schleifenvariablen
a, b = aList[2:], aList[:2]       # Aufteilen einer Liste
```

Dekoratoren

Funktionsdefinitionen können mit Dekoratoren *umwickelt* werden:

```
def logger(f):
    def new(*args):
        print f.func_name, 'aufgerufen mit', args
        return f(*args)
    return new

@logger
def func(a, b):
    print a**2 + b**2
```

Hiermit kann aspektorientiert programmiert, oder die Sprache um Funktionalität erweitert werden.

Beispiel: Properties sind mit Dekoratoren implementiert:

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value
```

Generator-Funktionen

Es kommt häufig vor, dass aus einer Funktion eine Menge von Werten zurückgegeben werden soll. In fast allen Sprachen steht man hier vor dem Problem, dass der Rückgabewert zwangsläufig nur ein einziges Objekt sein darf.
Als Workaround gibt es klassisch zwei Möglichkeiten:

- Man verpackt die einzelnen Werte in einen Container und gibt diesen zurück. Nachteile:
  - (etwas) höherer Aufwand für die Funktion, da sie den Container erzeugen und verwalten muss
  - Die Art des Containers wird von der Funktion festgelegt, obwohl dieser Aspekt meistens gar nicht relevant ist; will der Aufrufer die Daten in einem anderen Container, muss er wandeln
  - Es ist nur möglich, alle Ergebnisse auf einmal zurückzugeben; es gibt keine Zwischenergebnisse, der Container braucht viel Speicher und man ist zwangsläufig auf endliche Mengen beschränkt.

- Der Funktion wird eine Callback-Funktion etc. übergeben, die für jeden errechneten Wert aufgerufen wird. Zwischenergebnisse stehen so sofort zur Verfügung und unendliche Mengen sind möglich; auch wird dem Aufrufer kein bestimmter Container aufgezwungen.
  Nachteile:
  - sehr viel höherer Aufwand für den Aufrufer
  - Kontrollfluss zerfleddert, dadurch schlechte Wartbarkeit
  - je nach Programmiersprache und Implementierung auch höherer Aufwand für die Funktion selbst

Die Python-Lösung:

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

Verwendung:

```
for item in fibonacci():
    print item
```

Gibt der Generator nur eine endliche Menge zurück und will man die Ergebnisse gar nicht iterativ verarbeiten, so kann man sie trivial in eine normale Liste wandeln:

```
items = list(generator())
```

Typsystem

Python verwendet "Duck-Typing". Konkret heißt das, dass der Typ eines Objekts in der Regel keine große Rolle spielt, es müssen nur die geforderten Operationen unterstützt werden (= alles ist generisch). Beispiel:

```
def multiplyAll(numbers):
    result = 1
    for item in numbers:
        result *= item
    return result

print multiplyAll([1, 2, 3])       # Ausgabe: 6
print multiplyAll([0.5, 4, 0.5])   # Ausgabe: 1.0
print multiplyAll([1+1j, 1+1j])    # Ausgabe: 2j
```

Standard-Protokolle

Da der konkrete Typ in der Python-Philosophie also weitestgehend egal ist, gibt es eine Anzahl von Standard-"Schnittstellen", die ein Objekt implementieren kann. Ein solches Objekt kann dann mit allen Sprachkonstrukten wie ein vordefinierter Typ mit dieser Eigenschaft verwendet werden.
Einige solche Protokolle:

iterable

Das Objekt stellt im weitesten Sinne eine aufzählbare Menge dar. Diese kann sogar unendlich groß sein.
Darunter fallen Listen, Tupel, Strings, Dateiobjekte, Generatorfunktionen...

sequence

Im weitesten Sinne ein Array: Objekte haben eine endliche Größe, die mit len() abgefragt werden kann, und erlauben Zugriff über die []-Notation.

mappings

Repräsentiert wie Sequence eine endliche Menge, diese muss aber nicht geordnet sein, also Objekte die sich grob wie ein Dictionary verwenden lassen. Auch hier ist der Zugriff über die []-Notation möglich, aber die Indizes können beliebige Typen haben.

Beispiele (5)

Schreibe ein Programm, das folgende Funktion hat:

- Als einzigen Parameter erhält das Programm einen Dateinamen.
- Das Programm liest die Datei und gibt folgende Statistiken aus:
  - Anzahl der Zeilen
  - Anzahl der Wörter
  - Größe in Bytes

```
import sys

f = open(sys.argv[1], 'r')

lines, words, chars = 0, 0, 0

for line in f.readlines():
    lines += 1
    words += len(line.split())
    chars += len(line)

print "%d | %d | %d" % (lines, words, chars)
```

Standardbibliothek

Die Standardbibliothek bietet Lösungen für viele Standardaufgaben oder Ergänzungen zum Sprachumfang, beispielsweise `collections` mit dem `namedtuple`:

```
from collections import namedtuple

Point = namedtuple('Point', 'x y')

p = Point(2, 3)
print p[0]       # Ausgabe 2
print p.x        # Ausgabe 2
print p          # Ausgabe Point(x=2, y=3)
```

Kommandozeilenparameter verarbeiten

Für die komfortable Verarbeitung von Kommandozeilenparametern gibt es gleich zwei Module in der Standardbibliothek:

- `getopt` orientiert sich bei Syntax und Funktionsumfang an GNU `getopt`
- `optparse` ist eine leistungsfähige Alternative: Das flexible und erweiterbare Modul kann die komplette Interaktion mit dem Benutzer an der Kommandozeile übernehmen, inklusive Syntaxprüfungen und Anzeige von Hilfe.

Binärdaten verarbeiten

Zum Verarbeiten von Binärdaten bietet Python die spezialisierten Module `array` und `struct`:

```
from array import array
print array('f', [1.0, 2.0, 3.14]).tostring()
# Ausgabe '\x00\x00\x80...'

import struct
struct.unpack('!4s4sI', my_binary_data)
# Ausgabe z.B. ['HELO', 'WRLD', 123456]
```

`struct` kann auch mit verschiedenen Byte Orders und Padding umgehen und geht damit über die Möglichkeiten des C-`struct` sogar hinaus.

Weitere interessante Module

- **doctest**: Unit-Tests
- **logging**: Flexibles Logging-System
- **profile**: Engpässe finden
- **unicodedata**: Abruf von Informationen über Unicode-Zeichen
- **time**, **datetime**, **calendar**: Umgang mit Zeit und Daten
- **decimal**: Beliebig große Festkomma-Zahlen
- **itertools**: Hilfreiche Listenfunktionen
- **pickle**, **shelve**: Serialisierung von Objekten
- **readline**: Komfortfunktionen für die Eingabezeile (nur Unix)
- **xml.etree.ElementTree**: Komfortable XML-Implementierung

Grafische Toolkits

- Bindings für praktisch alle gebräuchlichen Toolkits verfügbar:
  - pyGTK
  - pyQt
  - wxPython
  - pyFLTK
  - ...
- In der Standarddistribution allerdings nur `Tkinter` enthalten

Python 3

- Seit Dezember 2008 ist die neue Majorversion verfügbar
- *nicht* abwärtskompatibel; automatisches Übersetzungswerkzeug verfügbar
- `print` ist eine Funktion
- Strikte Trennung zwischen Unicode- und Bytestrings (nicht mehr zuweisungskompatibel)
- Unicode-Strings Standard, Unicode-Bezeichner möglich
- (Unverbindliche) *Annotations* für Funktionsparameter und -rückgabewerte
- List Comprehension jetzt auch für Tupel, Sets und Dictionaries
- Neue Methode der Stringformatierung

Weiteres auf http://docs.python.org/3.0/whatsnew/3.0.html

Annotations

Per *Annotation* können unverbindliche Informationen zur Art eines Parameters oder Rückgabewerts hinterlegt werden:

```
def funktion(a: str, b: int) -> "Ergebnis"
```

- Bisher reine Syntax, keine Auswirkung auf das Programm
- Beliebige Objekte hinterlegbar
- Informationen zur Laufzeit verfügbar über die `func_annotation`-Eigenschaft des entsprechenden Funktionsobjektes

Weiterführendes

Online:

- Offizielle Seite: http://python.org/
- Guido van Rossums Blog: http://neopythonic.blogspot.com/
- Sammlung von Stolpersteinen für Einsteiger:
  - http://www.ferg.org/projects/python_gotchas.html
  - http://wiki.python.org/moin/BeginnerErrorsWithPythonProgramming
  - http://wiki.python.org/moin/MovingToPythonFromOtherLanguages
- http://mindview.net/
- Usenet: comp.lang.python und de.comp.lang.python

Literatur:

- Dive into Python <http://diveintopython.org/>
- Expert Python Programming <http://www.packtpub.com/expert-python-programming/book>

Fragen?

http://wolfgangfellger.de/pythonvortrag/