Kurzeinführung in Python

Andreas Rottmann, arott@cosy.sbg.ac.at

7. April 2003

Zusammenfassung

Dieser Artikel ist eine Kurzeinführung in Python und liegt umfangmäßig zwischen der Python Introduction [1] und dem Python-Tutorial [2].

Inhaltsverzeichnis

1 Allgemeines

Python ist eine portable, interpretierte, Objekt-orientierte Programmiersprache. Es wurde Anfang der 1990er von Guido van Rossum am Stichting Mathematisch Centrum (CWI, siehe `<http://www.cwi.nl/>`) in den Niederlanden entwickelt. Van Rossum ist noch immer Python's Haupt-Autor, aber es enthält viele Beiträge anderer.

Die Sprache hat eine elegante (aber nicht über-vereinfachte) Syntax; einige mächtige high-level Datentypen sind eingebaut. Python kann systematisch durch Module erweitert werden, diese können entweder in Python oder aber auch in einer kompilierten Sprache wie C oder C++ geschrieben werden. Module können neue Funktionen, Klassen und Variablen bereitstellen.

Standardmäßig steht schon eine große Anzahl von Modulen zur Verfügung, von Regular Expressions über Zugriff auf Betriebssystemfunktionen (z.B. Threading) bis zu Netzwerksupport (HTTP, XML-RPC, ...). Zusätzlich sind jede Menge Module verfügbar, die nicht im Standardumfang enthalten sind, wie etwa die Python Imaging Library.

Der Python-Interpreter kann auch in eigene Programme eingebettet werden, um diese mit einer Programmiersprache zu versehen (a'la *Emacs Lisp* oder *Visual Basic for Applications*).

2 Ein erster Einblick

Ein Gefühl für eine neue Programmiersprache bekommt man wohl am besten anhand von erklärtem Beispielcode. Hier also erstmal das unvermeidliche „Hello World!"-Programm:

```
print "Hello World!" # Ein Kommentar
```

Viel gibt es bei diesem ein-Zeiler noch nicht zu erklären; Strings werden in Python mit `"` oder `'` begrenzt, Kommentare mit `#` eingeleitet und vom Ende der Zeile begrenzt.

`print` ist ein Python-Schlüsselwort und gibt die übergebenen Parameter von einem Zeilenumbruch gefolgt aus.

Nun eine Python-Funktion, die eine Tabelle invertiert:

```
1 def invert(table):
2     index = {}                # empty dictionary
3     for key in table.keys():
4         value = table[key]
5         if not index.has_key(value):
6             index[value] = [] # empty list
```

```
7           index[value].append(key)
8       return index
```

In Zeile 1 wird mit dem Schlüsselwort **def** die Funktionsdefinition eingeleitet, Funktionsname und Parameter (in Klammern) folgen. Der Doppelpunkt ist in Python das Zeichen für einen Block-Beginn, wobei der Block durch die Einrückung definiert wird.

Zeile 2 weist der Variable `index` das leere Dictionary zu. Python besitzt eine dynamische Typbindung, das heißt der Variablentyp muss nicht extra deklariert werden und wird implizit durch die Zuweisung festgelegt. Der einer Variablen kann sich auch ändern, auch wenn das meist nicht als besonders sauberer Programmierstil betrachtet wird:

```
foo = 1           # foo hat jetzt den Typ 'Integer'
foo = 'A String' # foo hat jetzt den Typ 'String'
```

Zeile 3 startet eine **for**-Schleife über die Schlüssel des Parameters `table`. Die (u.a. auch in C++ und Java gebräuchliche) Notation `table.keys()` bedeutet die Auswahl des Attributs `keys` des Objekts `table` und Aufruf desselben ohne Parameter, also ein Methodenaufruf. Innerhalb des Schleifenkörpers nimmt nun `key` alle Werte der von der `keys` Methode zurückgegeben Liste an.

Die eckigen Klammern zur Indizierung (für Dictionaries sind u.a. auch Strings als Indizes zulässig) dürften ebenfalls geläufig sein.

In der letzten Zeile wird dann der `index` als Funktionswert mittels **return** zurückgegeben. Erfolgt kein **return**, so ist der Funktionwert implizit das spezielle Objekt `None`.

3 Datentypen

3.1 Zahlen

Python unterstützt an einfachen Datentypen Ganzzahlen (auch beliebiger Größe), Realzahlen und komplexe Zahlen.

Die meisten folgenden Beispiele sind die Ausgabe einer interaktiven Python-Sitzung, `>>>` ist der Interpreter-Prompt, die Ausgabe erscheint ohne `>>>` in der nächsten Zeile, falls vorhanden.

Rechnen mit Ganzzahlen (`**` ist der Potenzierungs-Operator):

```
>>> 3 * 3
6
>>> 300 ** 10
590490000000000000000000000L
```

Ganze, reelle und komplexe Zahlen können gemischt werden, wobei das Suffix `j` den

Imaginärteil kennzeichnet. Alternativ kann die eingebauten Funktion `complex(real, imag)` benutzt werden:

```
>>> (5 + 30j) / 5j
(6-1j)
>>> complex(5, 30) / complex(0, 5)
(6-1j)
>>> 5.0 ** 2
25.0
```

Konvertierungen erledigen die eingebauten Funktionen `int()`, `float()` und `long()`:

```
>>> int(5.5)
5
>>> long(3e+30)
2999999999999999778178897805312L
>>> float(3000000000000000000000000000000L)
2.9999999999999998e+30
```

Im obigen Beispiel sieht man recht deutlich die Ungenauigkeit der Computerrepräsentation von Gleitkommazahlen.

3.2 Strings

Python kann natürlich auch mit Strings umgehen, die entweder in einfache oder doppelte Hochkomma gesetzt werden können. Der Backslash (\) dient wie in C als Escape-Zeichen:

```
>>> 'Sie sagte "Hallo!".'
'Sie sagte "Hallo!".'
>>> "Was, des hat's g'sagt?"
"Was, des hat's g'sagt?"
>>> 'Tja, so war\'s.'
"Tja, so war's."
```

Für Zeilenumbrüche bietet Python im wesentlichen drei Möglichkeiten: Escape-Sequenzen (wie in C), Raw Strings (die die Bedeutung des Backslash verändern) und Triple-Quoted-Strings. Details findet man im Python Tutorial [2] und der Python Reference [4].

Die Operatoren `+` und `*` sind auch für Strings zulässig:

```
>>> 'ab' + 'cdef'
'abcdef'
>>> 'ab' * 5
'abababababab'
```

Strings können indiziert werden, wobei Python keinen eigenen Zeichen-Datentyp definiert; ein Zeichen ist einfach ein String mit Länge eins. Die Indizes beginnen wie in C mit 0. Substrings können in *Slice-Notation* angegeben werden: zwei Indizes, durch einen Doppelpunkt getrennt.

```
>>> word = 'Help'
>>> word[0]
'H'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Die Indizes können bei Slices weggelassen werden – gleichbedeutend mit 0 oder der Stringlänge. Auch negative Indizes sind erlaubt – sie entsprechen der Stringlänge abzüglich des Index-Betrages.

```
>>> word = 'Help'
>>> word[-1]
'p'
>>> word[:2]
'He'
>>> word[2:]
'lp'
```

Strings können in Python nicht direkt verändert werden; Zuweisungen an einzelne Elemente oder Substrings sind also nicht möglich.

```
>>> word = 'Help'
>>> word[0] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

3.3 Listen

Eine Liste wird als Komma-separierte und von eckigen Klammer eingeschlossene Liste ihrer Elemente notiert. Die Elemente müssen nicht vom gleichen Typ sein.

Listen verhalten sich ähnlich wie Strings:

```
>>> list1 = ['A', 'little', 'list']
>>> list1[0]
'A'
>>> list2 = [1, 2, 3]
>>> list3 = list1 + list2
```

```
['A', 'little', 'list', 1, 2, 3]
>>> list3[2:4]
[ 'little', 'list', 1, 2]
```

Listen sind im gegensatz zu String veränderbar, und haben auch dementsprechende Methoden:

```
>>> list = [1, 2, 3]
>>> list[2] += 3
>>> list.append('a')
[1, 2, 5, 'a']
```

Die Zuweisung an Teilbereiche ist ebenso möglich, und das kann sogar die Länge der Liste verändern.

```
>>> list = [1, 2, 3]
>>> list[:0] = [-2, -1, 0] # Am Anfang einfuegen
>>> list
[-2, -1, 0, 1, 2, 3]
```

3.4 Dictionaries

Im Abschnitt 2 wurden bereits Dictionaries verwendet, hier noch ein kleines Beispiel um die Möglichkeiten zu illustrieren:

```
1  >>> tel = {'jack': 4098, 'sape': 4139}
2  >>> tel['guido'] = 4127
3  >>> tel
4  {'sape': 4139, 'guido': 4127, 'jack': 4098}
5  >>> tel['jack']
6  4098
7  >>> del tel['sape']
8  >>> tel['irv'] = 4127
9  >>> tel
10 {'guido': 4127, 'irv': 4127, 'jack': 4098}
11 >>> tel.keys()
12 ['guido', 'irv', 'jack']
13 >>> tel.has_key('guido')
14 1
```

In Zeile 7 wird hier die **del**-Anweisung dazu verwendet um ein Element zu löschen.

4 Kontrollstrukturen

4.1 Verzweigungen

Die vielleicht bekannteste Anweisung ist die **if**-Anweisung, z.B.:

```
>>> if x < 0:
...       x = 0
...       print 'Negativ; auf null geaendert'
... elif x == 0:
...       print 'Zero'
... elif x == 1:
...       print 'Single'
... else:
...       print 'More'
...
```

Es kann keine oder mehrere **elif**-Teile geben, und der **else**-Teil ist optional. Das Schlüsselwort **elif** ist eine Kurzschreibweise für else if und ist nützlich, um exzessive Einrückungen zu vermeiden. Eine Sequenz der Art **if** ... **elif** ... **elif** ... ist ein Ersatz für `switch`- oder `case`-Anweisungen in anderen Sprachen.

4.2 Schleifen

Python unterstützt eine **while**- und eine **for**-Schleife. Die Syntax der **while**-Schleife dürfte aus der einen oder anderen Programmiersprache bekannt sein; Python macht hier keine Ausnahme:

```
while not quit:
    print 'Hallo!'
    ...
```

Die **for**-Schleife iteriert über die Elemente einer Sequenz:

```
for x in [1, 2, 3, 4]:
    print x
```

In Verbinding mit der **for**-Schleife ist die eingebaute Funktion `range()` sehr praktisch. Sie generiert eine Sequenz, die einer arithmetischen Aufzählung entspricht:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 20, 5)
[0, 5, 10, 15]
```

Das obige **for**-Beispiel lässt sich also kürzer auch so schreiben:

```
for x in range(1, 5):
    print x
```

4.3 Funktionen

Das Beispiel im Abschnitt 2 definierte schon eine Funktion. Die Syntax erlaubt Vorgabewerte:

```
1 def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
2     while 1:
3         ok = raw_input(prompt)
4         if ok in ('y', 'ye', 'yes'): return 1
5         if ok in ('n', 'no', 'nop', 'nope'): return 0
6         retries = retries - 1
7         if retries < 0: raise IOError, 'refusenik user'
8         print complaint
```

Das obige Beispiel verwendet einige noch nicht besprochene Features von Python, wie die Möglichkeit Statements ohne Einrückung direkt nach einen Doppelpunkt zu stellen (Zeilen 4, 5 und 7) oder **raise** (Zeile 7), das eine Exception auslöst (siehe Abschnitt 4.4).

4.4 Exceptions

Python unterstützt Exceptions, so wie auch C++ oder Java. Ein Beispiel zur Exception-Behandlung:

```
>>> numbers = [0.3333, 2.5, 0, 10]
>>> for x in numbers:
...     print x,
...     try:
...         print 1.0 / x
...     except ZeroDivisionError:
...         print '*** has no inverse ***'
...
0.3333 3.00030003
2.5 0.4
0 *** has no inverse ***
10 0.1
```

Der **try**: ... **except**:-Block entspricht **try** { ... } **catch** { ... } in C++ und Java. Ausgelöst werden Exceptions mit **raise**:

```
if something_gone_wrong:
    raise RuntimeError, 'Something has gone wrong'
```

Die Argumente der **raise**-Anweisung sind im allgemeinen eine Klasse (siehe Abschnitt 5), optional gefolgt von Argumenten.

5 Objektorientierte Programmierung

Python unterstützt natürlich Objektorientierte Programmierung (OOP) (u.a. Polymorphie, (Mehrfach-) Vererbung, ...). Hier nur ein kurzes Beispiel, um ein Gefühl dafür zu vermitteln, wie OOP in Python aussieht; im Python-Tutorial [2] findet man eine viel ausführlichere Behandlung.

```
1  class Greeter:
2      def __init__(self):
3          self.message = 'Hello World!'
4      def greet(self):
5          print self.message
6
7  class Shouter(Greeter):
8      def greet(self):
9          print self.message.upper()
10
11 greeter = Greeter() # Instanzierung
12 greeter.greet()
13
14 greeter = Shouter()
15 greeter.greet()
```

Hier werden zwei Klassen, `Greeter` und `Shouter`, definiert, jeweils mit einer Methode `greet()`. `Greeter` besitzt einen Konstruktor (`__init__`). `Shouter` ist von `Greeter` abgeleitet.

Alle Methoden deklarieren die Instanzvariable (konventionellerweise `self` genannt) explizit, im Gegensatz zu C++ und Java, wo der **this**-Pointer bei jeder (nichtstatischen) Methode implizit vorhanden ist.

Nach den Klassendefinitionen wird zuerst ein `Greeter`-Objekt instanziert, dann wird seine Methode `greet()` aufgerufen. Anschließend wird dies mit `Shouter` wiederholt.

Literatur

[1] An Introduction to Python,
`http://www.python.org/doc/Introduction.html`

[2] Python Tutorial,
`http://www.python.org/doc/current/tut/tut.html`

[3] Python Library Reference,
`http://www.python.org/doc/current/lib/lib.html`

[4] Python Language Reference,
`http://www.python.org/doc/current/ref/ref.html`