Diese Sonderausgabe von **freies**Magazin beschäftigt sich allein mit dem Thema Python-Programmierung. Auf den Wunsch der Leser hin wurden die sechs Artikel von Daniel Nögel, die in den **freies**Magazin-Ausgaben Oktober 2010 bis März 2011 erschienen sind, zusammengefasst und liegen nun als zusammenhängende Kompilation vor. Auf die Art kann jeder Python-interessierte Neuling die erste Schritte in der Python-Welt wagen und lernt so neben den Grundstrukturen und Sprachelementen der Sprache in den fortgeschrittenen Teilen auch die Anbindung an einer Datenbank, um eine Musikverwaltung aufzubauen.

Wir wünschen viel Spaß mit dieser Python-Sonderausgabe.

Ihre **freies**Magazin-Redaktion

© **freies**Magazin CC-BY-SA 3.0 ISSN 1867-7991

Python-Programmierung: Teil 1 – Hallo Welt

von Daniel Nögel

Python erfreut sich seit einiger Zeit großer Beliebtheit bei Anfängern und Fortgeschrittenen. Die Sprache überzeugt durch Einfachheit und die große Zahl mitgelieferter Bibliotheken. Sowohl als Skript-Sprache, etwa für GIMP, als auch bei eigenständigen Projekten ist Python immer häufiger anzutreffen. Bekannte Programme wie PiTiVi oder BitTorrent setzen auf Python und auch im Hintergrund von Google und Youtube soll die mächtige Sprache nicht mehr wegzudenken sein. Dieser Artikel bildet den Anfang einer mehrteiligen Einführung in die Programmiersprache Python.

Python ist mittlerweile bei allen großen Distributionen vorinstalliert – meist in der Version 2.6+. In der Version 3.0 gab es einige größere Änderungen, die hier – wo möglich – bereits übernommen werden. Wo nicht möglich, wird auf die bevorstehende Änderung hingewiesen.

Gleichzeitig sei aber auch angemerkt, dass diese Einführung nicht jede Methode und jede Funktion erörtern kann, die zum Standardrepertoire von Python gehört. Vielmehr soll ein Einblick in die Sprache geliefert werden, der – wo es notwendig ist – hilfreiche und wichtige Methoden kurz anspricht. Für tiefer gehende Einblicke empfiehlt sich beispielsweise die offizielle Dokumentation von Python [1].

Die interaktive Shell

Python-Skripte werden nicht kompiliert, sondern zur Laufzeit von einem Interpreter ausgeführt. Python-Skripte sind somit immer direkt lauffähig. Der Python-Interpreter hat zudem einen interaktiven Modus – hier können Befehle direkt abgesetzt werden.

Dieser interaktive Modus kann in einem Terminal mit dem Befehl

```
$ python
```

gestartet werden.

Hinter der Eingabeaufforderung (>>>) können beliebige Python-Befehle abgesetzt werden.

```
daniel@kassandra: ~
daniel@kassandra:~$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Die interaktive Python-Konsole nach dem Start

Die Befehlszeile

```
>>> print("Hallo Python!")
```

gibt in der nächsten Zeile erwartungsgemäß **Hallo Python!** aus. Auch Berechnungen lassen sich direkt in ihr durchführen:

```
>>> 3+7
10
>>> 7*10
70
>>> 3-7
-4
>>> 8/4
2
>>>8/3
2
```

Die wichtigsten mathematischen Operatoren sind damit gleich bekannt: Die Verwendung von „+“, „-“, „/“ und „*“ sollte keine Schwierigkeiten bereiten. Auffällig erscheint allerdings das letzte Ergebnis. In den Python-Versionen vor 3.0 geht der Python-Interpreter bei Divisionen davon aus, dass der Benutzer eine Ganzzahldivision durchführen möchte. Erst die Eingabe

```
>>> 8/3.0
```

führt zum erwünschten Ergebnis und zeigt auch Nachkommastellen an [2].

Die interaktive Konsole ist ideal, um erste Erfahrungen mit Python zu sammeln. Für größere Projekte empfiehlt sich aber ein Texteditor.

Im Folgenden soll es so gehalten werden, dass Codeblöcke mit Eingabeaufforderung (>>>) immer in der Python-Konsole ausgeführt werden. Zeilen ohne diese Zeichen sind als Ausgabe der Konsole zu interpretieren.

Codeblöcke ohne Eingabeaufforderung sind meist als allgemeine Beispiele und Veranschaulichungen zu verstehen.

Hallo Welt

Ein erstes kleines Skript ist im Texteditor der Wahl schnell erstellt. Anders als in der interaktiven Python-Konsole, wo Eingaben direkt berechnet und auf den Bildschirm ausgegeben werden, benötigt man in eigenständigen Skripten eine Funktion, welche die gewünschten Informationen auf dem Bildschirm ausgibt – dazu dient in Python die **print()**-Funktion. Nur in der interaktiven Konsole kann auf diese Funktion in der Regel verzichtet werden (wie etwa bei den mathematischen Operationen oben).

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

print("Hallo Welt!")
```

Listing 1: hello_world.py

Diese Zeilen werden als **hello_world.py** gespeichert. Die Datei **hello_world.py** wird nun mit dem Befehl

```
$ chmod +x hello_world.py
```

als ausführbar markiert und schließlich mit

```
$ ./hello_world.py
```

oder

```
$ python hello_world.py
```

gestartet. Es erscheint folgende Meldung im Terminal:

```
Hallo Welt!
```

Das erste kleine Python-Skript ist funktionsfähig! Bei der ersten Zeile handelt es sich um die sogenannte Shebang-Zeile (siehe „Shebang – All der Kram“, **freies**Magazin 11/2009 [3]). Hier wird festgelegt, dass die Datei mit dem Python-Interpreter auszuführen ist. Die zweite Zeile informiert den Python-Interpreter über die verwendete Zeichenkodierung. Diese beiden Zeilen sollten in allen Python-Dateien vorhanden sein. Ohne Hinweis auf die Kodierung kann es zu Problemen mit Umlauten in Zeichenketten kommen.

Das erste „nützliche“ Programm

Als nächstes soll ein etwas ambitionierteres Projekt in Angriff genommen werden: Der Benutzer soll seinen Namen eingeben können und diesen dann in einer Box aus Gleichheitszeichen dargestellt bekommen.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

name = raw_input("Hallo! Wie heisst du? ")
name_with_borders = "= {0} =".format(name)
line = "=" * len(name_with_borders)

print(line)
print(name_with_borders)
print(line)
```

Listing 2: your_name.py

Hier gibt es schon Neues zu sehen: In der vierten Zeile wird der Benutzer nach seinem Namen gefragt. Die Funktion **raw_input()** gibt die als Parameter übergebene Zeichenkette auf dem Bildschirm aus und wartet dann auf die Eingabe des Benutzers. Diese wird nach dem Drücken von Enter in der Variable **name** gespeichert (genauer: es wird ein Zeichenketten-Objekt erstellt, auf das die Variable verweist). Erst danach werden die übrigen Zeilen des Skriptes verarbeitet.

Achtung: Ab Python 3 heißt es **input()** und nicht mehr **raw_input()**. Bis Python 3 ist von der Verwendung von **input()** dringend abzuraten, weil die Benutzereingabe direkt vom Interpreter ausgeführt wird – das ist in den allermeisten Fällen nicht erwünscht.

In Zeile 5 wird die Eingabe des Nutzers mit Gleichheitszeichen umgeben. Dazu wird die Zeichenkette **= {0} =** erzeugt und die Zeichenfolge **{0}** mit Hilfe der **format()**-Methode durch den Inhalt der Variable **name** ersetzt. In einem späteren Teil dieser Reihe wird auf diese Art der Ersetzung genauer eingegangen werden. In Zeile 6 wird mit der Funktion **len()** die Länge des Benutzernamens inklusive Gleichheits- und Leerzeichen ermittelt und dann die entsprechende Anzahl von Gleichheitszeichen ausgegeben. Dazu wird hier der *****-Operator benutzt – auch Zeichenketten lassen sich in Python also „multiplizieren“! So entsteht eine Folge von Gleich-

heitszeichen, die so lang ist, wie die Zeichenkette **name_with_borders**.

In Zeile 9 wird die in Zeile 5 erstellte Zeichenkette auf dem Bildschirm ausgegeben, Zeilen 8 und 10 geben jeweils die in Zeile 6 erstellte „Linie“ auf dem Bildschirm aus. Das Gesamtergebnis sieht dann wie folgt aus:

```
Hallo! Wie heisst du? Margot
==========
= Margot =
==========
```

Zeichenketten

Jetzt wurden bereits erste Erfahrungen mit Zeichenketten gesammelt. Diese sollen hier vertieft werden: Zeichenketten müssen immer von Anführungszeichen umschlossen werden. Möglich sind einfache und doppelte Anführungszeichen:

```
name = "Bernd"
name = 'Bernd'
```

Es gibt keinen Unterschied zwischen diesen beiden Varianten – die Verwendung ist letztlich Geschmackssache. Allerdings sollte darauf geachtet werden, dass innerhalb einer Zeichenkette keine äußeren Anführungsstriche vorkommen können:

```
message = "Ich heisse "Bernd"!"
```

führt also zu einem Fehler. Stattdessen kann in solchen Fällen der jeweils andere Anführungsstrich genutzt werden:

```
message = "Ich heisse 'Bernd'!"
```

oder

```
message = 'Ich heisse "Bernd"!'
```

In Python gibt es außerdem aber noch dreifache Anführungsstriche (**"""** oder **'''**). Innerhalb von dreifachen Anführungsstrichen kann man die einfachen und doppelten Anführungszeichen nach Belieben verwenden und sogar Zeilenumbrüche sind möglich:

```
print("""Hier kann ich " und ' nach Belieben einsetzen!
Ausserdem sind sogar Zeilenumbrueche moeglich!""")
```

Wie in vielen anderen Sprachen gibt es in Python natürlich auch die Möglichkeit, Zeichenketten durch Voranstellen eines \ zu „escapen“, d. h. so zu markieren, dass der Interpreter nicht darüber stolpert. Das fehlerhafte Beispiel von oben sähe mit Escape-Zeichen dann so aus:

```
message = "Ich heisse \"Bernd\"!"
```

Mit dem Escape-Zeichen lassen sich auch Zeilenumbrüche in Zeichenketten mit einfachen und doppelten Anführungsstrichen erzwingen – dies wäre sonst nicht möglich:

```
>>> print("Ein \nZeilenumbruch")
Ein
Zeilenumbruch
```

In der Python-Dokumentation finden sich weitere Escape-Sequenzen [4].

Zeichenketten näher betrachtet

Zeichenketten in Python gehören – wie auch Zahlen – zu den unveränderbaren Datentypen. Jede Veränderung an einer Zeichenkette liefert immer eine neue Zeichenkette zurück.

```
>>> text1 = "HALLO!!!"
>>> text2 = text1.lower()
>>> print(text1)
'HALLO!!!'
>>> print(text2)
'hallo!!!'
```

Hier wurden die Großbuchstaben mit der Methode **lower()** in Kleinbuchstaben „umgewandelt“. Tatsächlich bleibt **text1** davon aber unberührt; stattdessen wird eine völlig neue Zeichenkette erzeugt. Natürlich ist es aber möglich, eine Variable direkt neu zuzuweisen, so dass die Unveränderbarkeit von Strings in der Praxis kaum Bedeutung hat:

```
>>> text = "HALLO!!!"
>>> text = text.lower()
>>> print(text)
'hallo!!!'
```

Hier wird zunächst der Variable **text** die Zeichenkette **HALLO!!!** zugewiesen. Durch die Methode **lower()** wird eine neue Zeichenkette mit Kleinbuchstaben erstellt. Nun zeigt die Variable **text** auf die neu erstellte Zeichenkette. Da jetzt keine Variable mehr auf die alte Zeichenkette zeigt, wird diese bei Zeiten automatisch aus dem Speicher gelöscht.

Analog zu **lower()** gibt es mit **upper()** eine Methode, die eine Zeichenkette mit ausschließlich Großbuchstaben erzeugt. Mit **swapcase()** werden kleine Buchstaben zu Großbuchstaben und umgekehrt. Geradezu unerlässlich ist die **replace()**-Methode. Sie ersetzt alle Vorkommen einer gesuchten Zeichenfolge innerhalb einer Zeichenkette:

```
>>> "Ich finde Python doof".replace↷
("doof", "super")
'Ich finde Python super'
```

In der Python-Dokumentation finden sich viele weitere nützliche Zeichenketten-Funktionen [5].

Richtig einrücken

Viele Programmiersprachen kennen bestimmte Kontrollstrukturen, die den Programmfluss in besonderer Weise beeinflussen. Hier ein Beispiel in Pseudocode:

```
zaehler = 1
solange zahler <= 5 wiederhole:
gib zaehler auf dem Bildschirm aus
erhoehe zaehler um 1
gib "fertig" auf dem Bildschirm aus
```

Klar: Hier wird der Zähler von 1 bis 5 hochgezählt und jeweils auf dem Bildschirm ausgegeben. Aber wie oft wird „fertig“ auf den Bildschirm geschrieben? Es wird deutlich, dass dem Interpreter/Compiler irgendwie mitgeteilt werden muss, welche Information noch zur Kontrollstruktur gehört und wo der normale Programmfluss fortgesetzt wird.

Viele andere Programmiersprachen lösen das Problem mit geschweiften Klammern, die Anfang und Ende des auszuführenden Codeblocks markieren. In Python gibt es derartige Klammern nicht – zusammengehörende Codeblöcke müssen gemeinsam eingerückt werden:

```
zaehler = 1
solange zahler <= 5 wiederhole:
    gib zaehler auf dem Bildschirm ↷
    aus
    erhoehe zaehler um 1
gib "fertig" auf dem Bildschirm aus
```

So weiß der Python-Interpreter, dass nur Zeilen 3 und 4 zur Kontrollstruktur gehören. „fertig“ wird nur einmal auf dem Bildschirm ausgegeben. Wäre auch Zeile 5 eingerückt, würde „fertig“ ebenfalls fünfmal ausgegeben.

Das Einrücken ist eine Besonderheit von Python und einigen wenigen anderen Sprachen, die das Lesen des Quelltextes vereinfachen soll: Der Benutzer wird gezwungen, sinnvoll einzurücken! Wie viel eingerückt wird und ob es mit Leerzeichen oder Tabulatoren geschieht, bleibt dem Benutzer überlassen – es muss nur einheitlich sein. Sobald Leerzeichen und Tabulatoren gemischt werden oder an einer Stelle mit vier Leerzeichen eingerückt wird, an anderer aber mit drei, kommt es zu Problemen und das Programm wird sehr wahrscheinlich nicht richtig ausgeführt werden. Allgemein wird das Einrücken mit vier Leerzeichen empfohlen. Fast jeder gängige Texteditor ist heute in der Lage, statt Tabulatoren Leerzeichen einzufügen, so dass dem Benutzer durch die Verwendung von Leerzeichen keinerlei Nachteile entstehen.

for-Schleife

Zuletzt soll hier nun die **for**-Schleife besprochen werden. Mit dieser Schleife kann man beliebige Anweisungen beliebig oft ausführen lassen. Mit folgendem einfachen Beispiel werden zehn Zahlen nacheinander ausgegeben:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

for i in range(0, 10):
    print(i)
print("Fertig")
```

Listing 3: for_schleife.py

Die Ausgabe des obigen Skriptes sieht wie folgt aus:

```
0
1
2
3
4
5
6
7
8
9
Fertig
```

Was passiert in dem obigen Beispiel genau? Die Funktion **range()** liefert (vereinfacht gesagt,

es gibt hier Unterschiede in den verschiedenen Python-Versionen) eine Liste von 0 bis 9 zurück. Die Anweisung `for i in range(0, 10)` lässt sich umgangssprachlich übersetzen mit: „Für jedes Element i in der Liste der Zahlen von 0 bis ausschließlich 10 mache ...“

Die Liste der Zahlen von 0 bis 9 wird also schrittweise durchlaufen. Zunächst wird `i` der Wert `0` zugewiesen und dann die Anweisung `print(i)` ausgeführt. Danach wird `i` der Wert `1` zugewiesen und erneut der Anweisungsblock ausgeführt etc. Man nennt dieses Vorgehen auch „Iteration“: Hier wird also über die von `range()` erzeugte Liste „iteriert“.

An diesem Beispiel wird auch deutlich, warum richtiges Einrücken so wichtig ist: Hätte man die Zeile `print("Fertig")` in Zeile 6 auch eingerückt, wäre diese Anweisung bei jedem Schleifendurchlauf ausgeführt worden – und nicht erst nach dem Durchlaufen der Schleife.

Mit der `for`-Schleife endet der erste Teil der Einführung in Python. Im nächsten Teil werden dann `if`- und `while`-Blöcke sowie Listen besprochen.

LINKS

[1] http://docs.python.org/

[2] http://www.python.org/dev/peps/pep-0238/

[3] http://www.freiesmagazin.de/freiesMagazin-2009-11

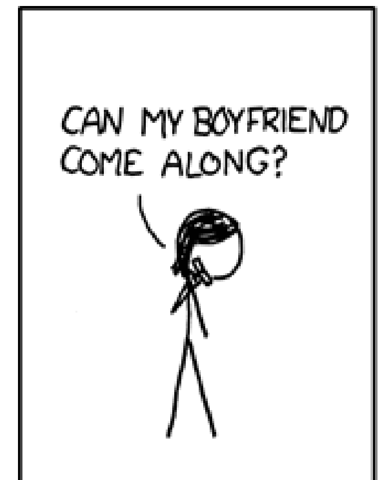
[4] http://www.python.org/doc//current/reference/lexical_analysis.htmli#grammar-token-escapeseq

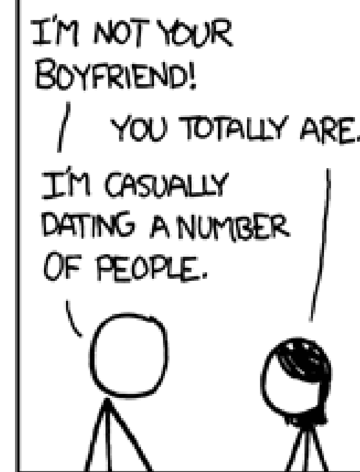
[5] http://docs.python.org/library/stdtypes.html#string-methods

Autoreninformation

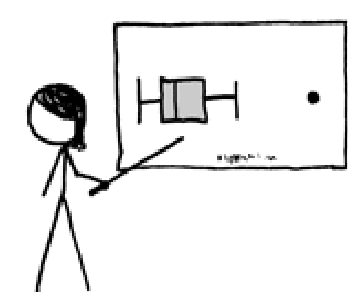
Daniel Nögel (Webseite) beschäftigt sich seit drei Jahren mit Python. Ihn überzeugt besonders die intuitive Syntax und die Vielzahl der unterstützten Bibliotheken, die Python auf dem Linux-Desktop zu einem wahren Multitalent machen.

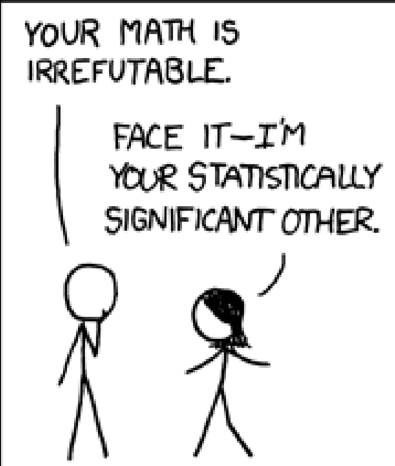
Diesen Artikel kommentieren





BUT YOU SPEND TWICE AS MUCH TIME WITH ME AS WITH ANYONE ELSE. I'M A CLEAR OUTLIER.





„Boyfriend“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/539

Python-Programmierung: Teil 2 – Tiefere Einblicke

von Daniel Nögel

Im vorherigen Teil der Python-Reihe *„Python-Programmierung: Teil 1 – Hallo Welt“* auf Seite 2 wurden erste Erfahrungen mit mathematischen Operatoren, Zeichenketten und `for`-Schleifen gesammelt. Im zweiten Teil sollen nun besonders Listen in Python betrachtet werden. Mit `if` und `while` werden aber auch zwei weitere Kontrollstrukturen vorgestellt.

Korrekturen und Ergänzungen

Im ersten Teil wurde bereits angesprochen, dass manche Spracheigenschaften von Python sich ab Version 3.x geändert haben. Dazu gehören insbesondere die Zeichenketten. Erst ab 3.x arbeitet Python immer mit Unicode-Zeichenketten. Davor muss die Verwendung von Unicode bei der Erstellung von Zeichenketten erzwungen werden. Unterbleibt dies, können schnell schwer zu ermittelnde Probleme auftreten. Auch Zeichenketten aus Dateien und anderen Quellen sollten so früh wie möglich in Unicode umgewandelt werden. Ab Python 3 muss sich der Entwickler darum nicht mehr kümmern.

Eine Unicode-Zeichenkette wird in Python 2.x durch das Voranstellen eines **u** vor die Zeichenkette oder den Aufruf der Funktion **unicode()** erstellt [1] [2] [3]:

```
u"Ich bin ein Unicode-String"
unicode("Auch ich werde eine ↷
Unicode-Zeichenkette")
"Bei Python-Versionen < 3 bin ich ↷
ein normaler Byte-String"
```

Ein weiterer Unterschied zu Python 3.x, der im letzten Teil verschwiegen wurde, ist die Verwendung von **print**. Erst ab Python 3 wird **print** als Funktion verwendet und muss wie folgt aufgerufen werden:

```
>>> print("Hallo Welt")
```

Vor Python 3 wurde **print** als Statement implementiert:

```
>>> print "Hallo Welt"
```

Hinweis: Wie schon im ersten Teil vereinbart, werden Zeilen, die mit >>> beginnen, direkt in die interaktive Konsole von Python eingegeben – dann natürlich ohne die spitzen Klammern.

Die genauen Unterschiede sollen hier weiter keine Rolle spielen. Wichtig ist nur: Nutzer von Python 3.x verwenden **print** als Funktion (mit Klammern), Nutzer von Python 2.x verwenden **print** ohne Klammern.

Da heute noch zumeist Python 2.x verwendet wird und viele Bibliotheken für Python 3.x noch nicht angepasst wurden, werden ab diesem zweiten Teil die hier genannten Ergänzungen berücksichtigt. Allen Zeichenketten wird von nun an also ein **u** vorangestellt, um Unicode-Zeichenketten zu erzeugen. Benutzereingaben werden im Folgenden mit der Funktion **unicode()** ebenfalls in Unicode umgewandelt. Nutzer von Python 3.x müssen das vorangestellte **u** und die Funktion **unicode()** jeweils auslassen – in 3.x wird ja ohnehin immer mit Unicode gearbeitet.

Operatoren

Bevor nun in den nächsten Abschnitten **if**- und **while**-Blöcke behandelt werden, sollen zuerst einige Operatoren besprochen werden. Operatoren sind – vereinfacht gesagt – (mathematische) Vorschriften, durch die aus zwei Objekten ein neues Objekt gebildet wird.

Eine Auswahl von Operatoren in Python

Operator	Typ	Funktion
+, -, *, /	Mathematisch	Addition, Subtraktion, Multiplikation, Division
**	Mathematisch	Potenzierung
<, >, <=, >=	Vergleich	kleiner als, größer als, kleiner als oder gleich, größer als oder gleich
==	Vergleich	gleich
!=	Vergleich	ungleich
=	Zuweisung	weist einen Wert zu
in	Listen-Operator/ Mitgliedschaftstest	testet, ob der rechte Operand Mitglied im linken Operanden ist
and	Bool. Operator	Konjunktion, logisches Und
or	Bool. Operator	Disjunktion, logisches Oder
not	Bool. Operator	Negation, logische Verneinung

Die üblichen mathematischen Operatoren sind sicher ebenso bekannt wie die Vergleichsoperatoren. Für Verwunderung sorgt vielleicht der Divisionsoperator: / liefert bis Python 3 ganzzahlige Ergebnisse, wenn nicht explizit Fließkommazahlen dividiert werden. Erst ab Python 3 gibt dieser Operator Fließkommazahlen zurück, wenn das Ergebnis keine natürliche Zahl ist.

Auch auf den Unterschied des Vergleichsoperators == und des Zuweisungsoperators = soll hingewiesen werden: **x == 3** liefert abhängig von **x** entweder **True** oder **False** zurück. **x = 3** dahingegen weist **x** den Wert 3 zu. Gerade bei Anfängern ist das eine beliebte Fehlerquelle.

Der **in**-Operator kommt bei allen iterierbaren Objekten (also besonders Listen) zur Geltung: Mit ihm lässt sich in Erfahrung bringen, ob ein bestimmter Eintrag in einer Liste vorhanden ist.

Die Booleschen Operatoren [4] **and** und **or** dienen zur Verknüpfung mehrere Wahrheitswerte. Der Ausdruck **3 < 5 and 3 < 2** ist offensichtlich falsch, der Ausdruck **3 < 5 or 3 < 2** dahingegen wahr. Der Operator **not** dreht einen Wahrheitswert schlicht um: Der Ausdruck **3 < 5 and not 3 < 2** ist also ebenfalls wahr.

Eine vollständige Übersicht der Operatoren in Python findet sich unter anderem im kostenlos verfügbaren Buch „A Byte of Python“ [5].

if-Anweisung

if-Blöcke bieten die Möglichkeit, das Ausführen eines bestimmten Code-Teiles von einer oder mehreren Bedingungen abhängig zu machen.

In dem Kopf des **if**-Blockes wird die Bedingung für die Ausführung definiert, also beispielsweise:

```
1 number = 5
2 if number > 3:
3     print u"Zahl groesser als 3"
```

Bei der Definition derartiger Bedingungen sind besonders vergleichende Operatoren wichtig. Im Kopf eines **if**-Blockes können – durch boolesche Operatoren verknüpft – eine ganze Reihe derartiger Vergleiche aneinandergereiht werden:

```
1 number = 20
2 if number > 10 and number < 40:
3     print u"Zahl liegt zwischen ↷
      10 und 40"
```

Durch den Operator **and** müssen beide Vergleiche wahr sein, damit der **if**-Rumpf ausgeführt und die Meldung ausgegeben wird. Verwendet man dahingegen den Operator **or**, muss nur eine der Bedingungen wahr sein:

```
1 good_looking = False
2 rich = True
3 if good_looking == True or rich ↷
  == True:
4     print u"Heirate mich!"
```

Hier wird die Meldung „Heirate mich!“ ausgegeben, wenn die Variable **good_looking** oder die Variable **rich True** ist (oder beide). In Zeile 3 werden die Variablen dazu mit **True** verglichen. Dieser Vergleich mit **True** ist eigentlich immer unnötig. Üblich und schöner zu lesen ist folgende Schreibweise:

```
1 if good_looking or rich:
2     print u"Heirate mich!"
```

Am Ende dieses Abschnitt soll noch kurz auf die Möglichkeit eingegangen werden, mehrere Eventualitäten mit **if** abzudecken:

```
1 if number < 10:
2     print u"Kleiner 10"
3 elif number < 20:
4     print u"Kleiner 20"
5 else:
6     print u"Groesser oder gleich ↷
      20"
```

Das Schlüsselwort **elif** steht für **else if** und gelangt immer dann zur Ausführung, wenn die vorherige **if**-Bedingung nicht erfüllt war. Mit **elif** können – ebenso wie mit **if** – eine Vielzahl von Bedingungen definiert werden.

Wäre **number** beispielsweise 3, wäre die Bedingung in Zeile 1 wahr und Zeile 2 käme zur Ausführung. Wäre **number** aber 11, wäre die Bedingung in Zeile 1 nicht erfüllt und der Interpreter würde die Bedingung in Zeile 3 prüfen. Da diese in diesem Fall wahr wäre, käme Zeile 4 zur Ausführung. Wäre **number** aber nun 40 und entsprechend keine der beiden Bedingungen wahr, käme Zeile 6 zur Ausführung: Das Schlüsselwort **else** ist also immer dann (und nur dann) von Bedeutung, wenn keine der vorherigen **if** oder **elif**-Bedingungen erfüllt wurde.

while-Schleife

Eine weitere wichtige Kontrollstruktur in Python ist die **while**-Schleife. So lange die im Schleifenkopf definierten Bedingungen wahr sind, wird der Schleifenrumpf ausgeführt. Ein sehr einfaches Beispiel ist folgende Endlosschleife:

```
while True:
    raw_input(u"Wie war Ihr Name noch gleich?")
```

Da die Bedingung **True** immer wahr ist, wird die Schleife nie enden. Durch die Tastenkombination Strg + C kann die Ausführung des Programms aber beendet werden.

Sinnvoller ist eine derartige Schleife natürlich, wenn eine Abbruchbedingung definiert wird. Denkbar wäre hier beispielsweise das Sammeln von Namen, bis der Benutzer das Programm durch die Eingabe von **exit** beendet.

```
names = []
running = True
while running:
    user_input = unicode(raw_input(u"Geben Sie einen Namen ein oder 'exit' zum Beenden > "))
    if user_input == u"exit":
        running = False
    else:
        names.append(user_input)
print u"Sie haben folgende Namen eingegeben:"
print names
```

Wichtig ist hier die Funktion **unicode()**: Sie wandelt in Python 2.x die Eingabe des Benutzers in Unicode um. Da in Python 3.x von Haus aus mit Unicode-Zeichenketten gearbeitet wird, gibt es diese Funktion dort nicht mehr.

Hinweis: Nutzer von Python 3 verwenden statt **raw_input** lediglich **input**.

Zwischenfazit: Kontrollstrukturen

Bisher wurde folgende Kontrollstrukturen behandelt: **if**, **for** und **while**. Für diese Strukturen gilt:

- Jede Kontrollstruktur besteht aus einem Kopf, der die Ausführungsbedingungen definiert und einem Rumpf, der ausgeführt werden soll.
- Der Kopf einer Kontrollstruktur wird immer mit einem Doppelpunkt abgeschlossen.
- Der Rumpf einer Kontrollstruktur muss immer um eine Ebene eingerückt werden.
- Die Einrückungen müssen immer gleichmäßig sein.

Kontrollstrukturen können natürlich auch verschachtelt werden. Folgendes Beispiel veranschaulicht dies:

```
if username == u"Bernd":
    if password == u"xy":
        print u"Alles ok"
    else:
        print u"Password falsch"
else:
    print u"Benutzername falsch"
```

Der innere **if**-Block muss also insgesamt eine Ebene eingerückt werden – er gehört ja zum Rumpf des äußeren **if**-Blockes. Der Rumpf des inneren **if**-Blockes muss um zwei Ebenen eingerückt werden.

Jede Verschachtelungsebene muss also durch Einrückung von der vorherigen Ebene getrennt werden.

Weitere Informationen über Kontrollstrukturen finden sich in der Python-Dokumentation [6].

```
geany_run_script.sh
Geben sie einen Namen ein oder 'exit' zum Beenden > Karl
Geben sie einen Namen ein oder 'exit' zum Beenden > Peter
Geben sie einen Namen ein oder 'exit' zum Beenden > Heinz
Geben sie einen Namen ein oder 'exit' zum Beenden > Johannes
Geben sie einen Namen ein oder 'exit' zum Beenden > exit
Sie haben folgende Namen eingegeben:
['Karl', 'Peter', 'Heinz', 'Johannes']
```

Vier verschiedene Namen werden eingegeben.

Listen

In Teil 1 dieser Einführung wurde mit der Funktion **range()** eine Liste von 0 bis 9 generiert. Hier soll nun abschließend näher auf Listen eingegangen werden. Bei Listen handelt es sich um einen Datentyp, der beliebige andere Datentypen verwalten kann (sogar gemischt) – gewissermaßen also

ein Aktenschrank für Zeichenketten, Zahlen und alle möglichen anderen Objekte, die in Python vorkommen (sogar Listen lassen sich in Listen ablegen, so dass verschachtelte Listen möglich sind) [7].

Listen werden in Python mit eckigen Klammern (**[** und **]**) gekennzeichnet. Sie sind sehr leicht zu erstellen:

```
1 >>> persons = []
2 >>> type(persons)
3 <type 'list'>
4 >>> persons = list()
5 >>> type(persons)
6 <type 'list'>
7 >>>persons = [u"Peter", u"Hermann", u"Simon"]
```

In Zeile 1 wird eine leere Liste erstellt und an den Namen **persons** gebunden. In Zeile 2 wird mit der Funktion **type()** der Typ des Objektes, welches an **persons** gebunden ist, ausgegeben. Wie erwartet, handelt es sich dabei um eine Liste. Zeile 4 zeigt die Erzeugung mittels der Funktion **list()**. Das Ergebnis ist das gleiche. In Zeile 7 sieht man, dass man in Python eine Liste direkt befüllen kann. Es werden die drei Unicode-Zeichenketten **Peter**, **Hermann** und **Simon** in die Liste eingetragen.

Wie schon in Teil 1 gezeigt wurde, lässt sich sehr einfach über Listen iterieren. Listen haben aber auch zusätzliche Methoden, die sehr nützlich sein können.

```
1 >>> persons = [u"Peter", u"Hermann", u"Simon"]
2 >>> persons.append(u"Carla")
3 >>> persons.append(u"Hermann")
4 >>> persons
5 [u'Peter', u'Hermann', u'Simon', u'Carla', u'Hermann']
6 >>> persons.remove(u"Hermann")
7 >>> persons
8 [u'Peter', u'Simon', u'Carla', u'Hermann']
```

Mit der Methode **append()** kann ein weiterer Eintrag angehängt werden, wie man in den Zeilen 2 und 3 sehen kann. Zeile 5 zeigt das Ergebnis: Die beiden Unicode-Objekte **Carla** und **Herman** wurden in der Reihenfolge der **append**-Aufrufe an die Liste angefügt.

Analog dazu lassen sich Einträge mit der Methode **remove()** entfernen (Zeile 6). Hierbei sollte beachtet werden, dass jeweils nur das erste Vorkommen von **Hermann** entfernt wird. Gibt es mehrere gleichlautende Einträge, muss **remove()** auch mehrfach ausgeführt werden, etwa wie folgt:

```
1 >>> persons = [u"Peter", u"Hermann", u"Hermann"]
2 >>> while u"Hermann" in persons:
3 >>>     persons.remove(u"Hermann")
4 >>> print persons
5 ['Peter']
```

Wichtig hierbei: Da die Zeichenketten **Hermann** in der Liste Unicode-Objekte sind, sollte auch als Such-Zeichenkette ein Unicode-Objekt angegeben werden, um Fehler zu vermeiden. Wird versucht, einen Eintrag zu entfernen, der gar nicht in der Liste vorhanden ist (etwa **Heidi**), kommt es zu einer Fehlermeldung – hier als Beispiel in der interaktiven Python-Konsole:

```
1 >>> persons = [u"Peter", u"Hermann", u"Simon"]
2 >>> persons.append(u"Carla")
3 >>> persons.remove(u"Hermann")
4 >>> print persons
5 [u'Peter', u'Simon', u'Carla']
6 >>> persons.remove(u"Heidi")
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 ValueError: list.remove(x): x not in list
```

Nach den Veränderungen der Liste in den Zeilen 1 bis 3 ist in Zeile 5 noch alles in Ordnung: **Hermann** wurde aus der Liste gelöscht, **Carla** hinzugefügt. Der Versuch, **Heidi** zu entfernen scheitert: Dieser Eintrag ist in der Liste gar nicht vorhanden. Die Zeilen 7-9 zeigen die Reaktion des Python-Interpreters darauf. In einem späteren Teil dieser Reihe werden Python-Fehler (meist Exceptions genannt) näher behandelt. Hier soll zunächst gezeigt werden, wie vor dem Löschen eines Eintrages überprüft werden kann, ob er sich überhaupt in der Liste befindet:

```
if u"Heidi" in persons:
    persons.remove(u"Heidi")
```

Nun wird mit dem Operator **in** geprüft, ob der Eintrag **Heidi** überhaupt in der Liste existiert. Sehr schön zu sehen ist dabei, wie intuitiv und natürlich Python sein kann.

Listen-Indizes

Beim Umgang mit Listen sollte man wissen, dass Python die Listeneinträge mit einem sogenannten „Index“ verwaltet. Jedem Listeneintrag wird mit 0 beginnend eine eindeutige Zahl zugewiesen. Der erste Eintrag wird also mit 0 angesprochen, der zweite Eintrag mit 1 usw. So ist es sehr leicht, auf einzelne Einträge zuzugreifen:

```
>>> letters = [u"a", u"b", u"c"]
>>> letters[1]
u"b"
```

Damit wird der zweite Listeneintrag ausgelesen – der erste Listeneintrag hat ja den Index 0. Soll von hinten gezählt werden, wird einfach ein negativer Index angegeben:

```
>>> letters[-3]
u"a"
```

Weitere Listen-Methoden

Die gerade besprochenen Indizes spielen auch bei bestimmten Methoden von Listen eine Rolle: So gibt es mit **insert()** und **pop()** die Möglichkeit, Einträge an einer bestimmten Stelle der Liste einzufügen oder zu entfernen:

```
1  >>> letters = [u"a", u"c", u"e"]
2  >>> letters.insert(1, u"b")
3  >>> letters
4  [u"a", u"b", u"c", u"e"]
5  >>> letters.insert(3, u"d")
6  >>> letters
7  [u"a", u"b", u"c", u"d", u"e"]
8  >>> letters.pop()
9  u"e"
10 >>> letters
11 [u"a", u"b", u"c", u"d"]
12 >>> letters.pop(2)
13 u"c"
14 >>> letters
15 [u"a", u"b", u"d"]
```

In Zeile 2 wird mit der **insert()**-Methode **b** an die richtige Stelle der Liste befördert, in Zeile 5 wird mit **d** analog verfahren. Zu beachten ist hier, dass der erste Parameter der **insert()**-Methode immer die gewünschte Position im Index der Liste angibt (daher muss erneut von 0 gezählt werden), der zweite Parameter beinhaltet das einzufügende Objekt. **pop()** löscht das letzte Element aus einer Liste und gibt dieses zurück. Alternativ kann auch ein bestimmter Eintrag aus einer Liste gelöscht werden – dazu wird der entsprechende Index als Parameter angegeben.

Slicing

Sehr wichtig für Listen ist auch das Slicing – also das „Zerschneiden“. Mit dem **slicing**-Operator können einzelne Elemente oder Ausschnitte von Listen ausgelesen werden. Der Operator sieht dabei wie folgt aus:

```
[von:bis]
```

von steht dabei für den Eintrag der Liste, bei dem das Zerschneiden beginnen soll – es wird von 0 gezählt. **bis** steht für den Listeneintrag, *vor* dem das Zerschneiden endet:

```
>>> li = [u"a", u"b", u"c", u"d", u↷
"e"]
>>> li[0:3]
[u"a", u"b", u"c"]
>>> li[2:5]
[u"c", u"d", u"e"]
```

Es ist auch möglich, das Ende des Schnittes vom Ende der Liste aus zu definieren – indem ein negatives Vorzeichen gewählt wird:

```
>>> li[0:-1]
[u"a", u"b", u"c", u"d"]
>>> li[0:-2]
[u"a", u"b", u"c" ]
>>> li[1:-2]
[u"b", u"c"]
```

Abkürzen erlaubt

Soll der erste Schnitt gleich am Anfang der Liste gesetzt werden, muss nicht nicht immer 0 als Startpunkt gesetzt werden:

```
>>> li = [u"a", u"b", u"c", u"d", u↷
"e"]
>>> li[:3]
```

gibt wie gewünscht **[u"a", u"b", u"c"]** zurück. Auch beim zweiten Schnitt kann abgekürzt

© freiesMagazin CC-BY-SA 3.0

werden: Soll dieser hinter dem letzten Listenelement erfolgen, wird ebenfalls keine Angabe gemacht:

```
>>> li[2:]
[u"c", u"d", u"e"]
```

Es ist nicht schwer zu erraten, was der Ausdruck

```
>>> li[:]
```

folglich bewirken muss.

Listen durch Slices verändern

Bisher wurde nur lesend auf verschiedene Listen-Indizes zugegriffen: Die Ursprungsliste wurde dabei jedoch nie verändert. Mit dem Zuweisungsoperator lassen sich aber auch einzelne Indizes überschreiben oder ganze Bereiche einfügen:

```
1  >>> li = [u"a", u"b", u"c"]
2  >>> li[2] = u"e"
3  >>> li
4  [u'a', u'b', u'e']
5  >>> li[2:2] = [u"c", u"d"]
6  >>> li
7  [u'a', u'b', u'c', u'd', u'e']
8  >>> li[3:] = [1, 2, 3]
9  >>> li
10 [u'a', u'b', u'c', 1, 2, 3]
```

Hier wird zunächst eine Liste mit den Buchstaben **a** bis **c** erstellt. Der dritte Eintrag der Liste, wird in Zeile 2 durch **e** ersetzt. In Zeile 4 ist zu sehen, dass die Liste **li** dadurch verändert wurde. In Zeile 5 werden zwischen **b** und **e** zwei weitere Listenelemente eingefügt: Durch den Slice **[2:2]** wird der **Schnitt** direkt vor dem dritten Listenelement gesetzt (Index 2), so dass die Buchstabenreihenfolge wieder stimmt. In Zeile 8 ist schließlich zu sehen, wie ein ganzer Slice der Liste überschrieben wird.

Es ist sehr zu empfehlen, das Slicing und die anderen hier vorgestellten Methoden und Funktionen in der interaktiven Python-Konsole ein wenig zu erproben. Auch die Python-Dokumentation kann wertvolle Hinweise zum Umgang mit Listen liefern [8].

Ein kleines Beispiel

Das folgende Beispiel setzt einige der hier erlernten Techniken ein.

```
1  #!/usr/bin/env python
2  # coding: utf-8
3
4  allowed_tries = 5
5  counter = 1
6
7  users = [u"Karl", u"Willi", u"Joe↷
   "]
8  passwords = [u"karl123", u"↷
   willi456", u"joe789"]
9
10 while counter <= allowed_tries:
11     username = unicode(raw_input(↷
       u"Bitte geben sie ihren ↷
       Benutzernamen ein: "))
12     password = unicode(raw_input(↷
       u"Bitte geben sie ihr ↷
       Passwort ein: "))
13
14     if not username in users:
15         print u"Dieser Benutzer ↷
           existiert nicht!"
16     else:
17         idx = users.index(↷
           username)
18         if passwords[idx] == ↷
           password:
19             print u"Erfolgreich ↷
               eingeloggt!"
20             break
21         else:
22             print u"Sie haben ein↷
                falsches Passwort ↷
               eingegeben"
23
24     counter += 1
25
26     if counter > allowed_tries:
27         print u"Sie haben es zu ↷
           oft versucht!"
```

Listing 1: beispiel.py

Hinweis: Benutzer von Python 3.0 verwenden anstatt **raw_input()** schlicht **input()**.

In den Zeilen 7 und 8 werden zwei Listen definiert: **users** beinhaltet die verschiedenen Benutzer, **passwords** deren Passworte. Dabei gehören immer die Listeneinträge mit dem selben Index-Wert zusammen (also **Karl** und **karl123** etc.).

Die Schleife in diesem Beispiel wird höchstens fünfmal ausgeführt – nach fünf Durchläufen hat **counter** den Wert 6, so dass die Bedingung der **while**-Schleife nicht mehr wahr ist.

In Zeile 14 wird geprüft, ob der Benutzername *nicht* in der Liste vorkommt – in diesem Fall wird die Fehlermeldung in Zeile 15 ausgegeben und der Zähler in Zeile 24 um 1 erhöht. Andernfalls (ab Zeile 16) wird zunächst mit der Methode **`index()`** die Position des Benutzernamens in der Liste **`users`** ermittelt. In Zeile 18 wird das dazugehörige Passwort mit dem vom Benutzer eingegebenen Passwort verglichen. Stimmen beide überein, wird in Zeile 19 eine Meldung ausgegeben und die Schleife in Zeile 20 mit dem neuen Schlüsselwort **`break`** abgebrochen.

Im nächsten Teil dieser Reihe wird auf eine besondere Art der Ersetzung in Zeichenketten („String Substitution“) sowie Module und Funktionen eingegangen.

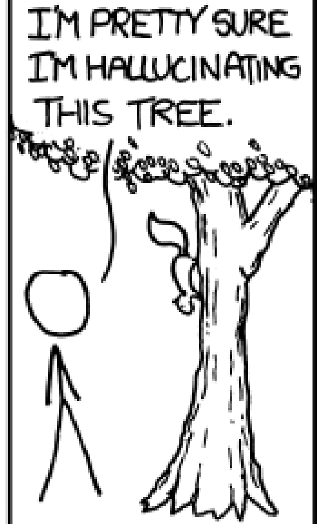
LINKS

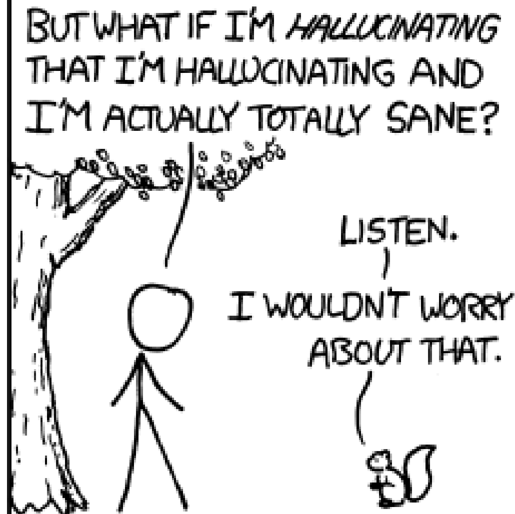
[1] http://docs.python.org/howto/unicode.html
[2] http://wiki.python-forum.de/Von%20Umlauten,%20Unicode%20und%20Encodings
[3] http://wiki.python.de/User%20Group%20M%C3%BCnchen?action=AttachFile&do=view&target=unicode-folien.pdf
[4] http://de.wikipedia.org/wiki/Boolesche_Algebra
[5] http://abop-german.berlios.de/read/operators.html
[6] http://docs.python.org/py3k/reference/compound_stmts.html
[7] http://docs.python.org/faq/design.html#how-are-lists-implemented
[8] http://docs.python.org/tutorial/datastructures.html#more-on-lists

Autoreninformation

Daniel Nögel (Webseite) beschäftigt sich seit drei Jahren mit Python. Ihn überzeugt besonders die intuitive Syntax und die Vielzahl der unterstützten Bibliotheken, die Python auf dem Linux-Desktop zu einem wahren Multitalent machen.

Diesen Artikel kommentieren





„Still No Sleep“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/776

Python-Programmierung: Teil 3 – Funktionen und Module

von Daniel Nögel

Im vorherigen Teil *„Python-Programmierung: Teil 2“* auf Seite 7 wurden Listen, Zeichenketten und die beiden Kontrollstrukturen `if` und `while` behandelt. Dieses Mal werden mit Funktionen und Modulen zwei wichtige Möglichkeiten vorgestellt, um eigene Python-Projekte zu strukturieren und wiederverwendbar zu machen. Zunächst sollen aber noch die versprochenen Ersetzungen bei Zeichenketten besprochen werden. Mit den „Dictionaries“ wird zudem ein weiterer wichtiger Datentyp in Python vorgestellt.

Substitution von Zeichenketten

In den letzten beiden Teilen dieser Reihe wurden schon mehrfach einfache Zeichenketten erstellt. In der Regel möchte man aber nicht nur bloße Zeichenketten ausgeben, sondern bestimmte dynamische Informationen darin transportieren, etwa den Namen des Benutzers. Dies funktioniert in Python so, dass man zunächst Platzhalter in der Zeichenkette definiert und diese später mit der `format()`-Methode gegen den gewünschten Inhalt austauscht (substituiert).

```
>>> message = u"Hallo {0}, du hast {1} Euro im Portemonnaie.".format(u"Karl", 10)
>>> print message
Hallo Karl, du hast 10 Euro im Portemonnaie.
```

Die Methode `format()` ersetzt also die Zeichenfolge `{0}` innerhalb der Zeichenkette durch den ersten Parameter, die Zeichenfolge `{1}` durch den zweiten Parameter usw. Python kümmert sich dabei automatisch um das Umwandeln der Datentypen – so können sehr leicht auch Zahlen in Zeichenketten eingefügt werden, ohne dass sich der Benutzer um irgendwelche Umwandlungen zu kümmern hätte. Folgendes Beispiel dient zur Veranschaulichung:

```
>>> names = [u"Karl", u"Bernd", u"Hannes", u"Ina" ]
>>> for name in names:
...     print u"'{0}' hat {1} Buchstaben".format(name, len(name))
```

Hinweis: Wie in den Artikeln zuvor steht `>>>` für eine Eingabe in der Python-Shell und muss nicht mit eingegeben werden. Mit drei Punkten `...` zeigt die Shell an, dass ein Befehl noch nicht abgeschlossen ist und sich über mehrere Zeilen erstreckt. Diese Punkte müssen ebenfalls nicht mit eingeben werden.

Die Ausgabe des obigen Beispiels lautet:

```
'Karl' hat 4 Buchstaben
'Bernd' hat 5 Buchstaben
'Hannes' hat 6 Buchstaben
'Ina' hat 3 Buchstaben
```

Bisher wurden nur positionale Argumente verwendet, das heißt: `{0}` verweist jeweils auf den ersten Parameter von `format()`, `{1}` auf den zweiten etc. Um die Übersicht zu wahren, ist auch die Angabe von Namen möglich. `format()` erlaubt dabei eine sehr weitreichende Formatierung:

```
>>> for name in names:
...     print u"'{username}' \
...     hat {namelength} \
...     Buchstaben".format(\
...     username=name, \
...     namelength=len(name))
```

So lässt sich etwa festlegen, wie viele Nachkommastellen ausgegeben werden sollen, ob und wie viele Leerzeichen der Zeichenkette vorangestellt werden sollen und vieles mehr [1].

Achtung: In Zeichenketten, die mit `format()` formatiert werden, werden alle geschweiften Klammern als Ersetzungszeichen interpretiert. Folgende Zeile wird also zu einem Fehler führen:

```
>>> print u"{0} mag Klammern wie \
... { oder }".format(u"Bernd")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: u' oder '
```

Hier müssen die letzten beiden Klammern maskiert werden:

```
>>> print u"{0} mag Klammern wie {{ oder }}".format(u"Bernd")
Bernd mag Klammern wie { oder }
```

Doppelte geschweifte Klammern werden also von der **format()**-Methode ignoriert.

Dictionaries

Sogenannte „Dictionaries“ oder „Dicts“ werden in anderen Sprachen oft „Hashes“ oder „assoziative Arrays“ genannt. Wie auch Listen können Dictionaries beliebige andere Datentypen verwalten. Während Listen aber ihre Einträge intern mit fortlaufenden Nummern adressieren (die sogenannten Indizes), können die Einträge in Dictionaries mit Zeichenketten, beliebigen Zahlen oder anderen Datentypen adressiert werden. Somit besteht jedes Dictionary aus zwei wesentlichen Elementen: Schlüsseln (keys) und Werten (values).

Ein leeres Dict wird in Python entweder mit der Funktion **dict()** oder zwei geschweiften Klammern erstellt [2]:

```
>>> persons = dict()
```

und

```
>>> persons = {}
```

sind also äquivalent.

In folgendem Beispiel sollen nun verschiedene Personen und ihr jeweiliges Alter in einer Datenstruktur gespeichert werden. Dies könnte wie folgt aussehen:

```
>>> persons = {u"Peter":18, \
... u"Ilse":87, u"Juergen":33, \
... u"Jutta":25}
```

Wie also auch Listen lassen sich Dicts initial befüllen. Die Namen sind in diesem Beispiel jeweils die Schlüssel, das Alter der dazugehörige Wert. Schlüssel und Wert werden durch einen Doppelpunkt getrennt, mehrere Schlüssel/Wert-Paare durch Kommata.

Um das Alter von Peter aus dem Dict auszulesen, genügt folgender Aufruf:

```
>>> print persons[u"Peter"]
18
```

Es fällt auf: Obwohl Dicts mit geschweiften Klammern erstellt werden, wird – wie auch bei Listen – mit eckigen Klammern auf die Werte zugegriffen. Auch sonst gibt es einige Parallelen zwischen Dictionaries und Listen. Um beispielsweise zu überprüfen, ob der Eintrag **Hans** in einem Dict vorhanden ist, wird ebenfalls der Operator **in** genutzt:

```
>>> if u"Hans" in persons:
...     print persons[u"Hans"]
... else:
...     print u"Der Eintrag Hans \
...     ist nicht vorhanden"
```

Während der in-Operator aber bei Listen das Vorhandensein des *Wertes* **Hans** abfragt, bezieht sich der Operator bei Dicts auf den *Schlüssel* **Hans**. Ebenso wie bei Listen führt der Zugriff auf ein nicht vorhandenes Element/Schlüssel zu einem Fehler. Wie man derartige Fehler sehr leicht abfängt, wird in einem der folgenden Teile besprochen werden.

In manchen Situationen ist es aber vielleicht gar nicht so wichtig, ob ein bestimmter Eintrag nun in einem Dict vorhanden ist oder nicht. Für solche Fälle gibt es die **get()**-Methode von Dicts:

```
1 >>> print persons.get(u"Hans", 15)
2 15
3 >>> print persons.get(u"Peter", 5)
4 18
```

Die Methode **get()** erwartet als ersten Parameter einen beliebigen Schlüssel. Ist der Schlüssel im Dict vorhanden, wird der dazugehörige Wert zurückgegeben. Andernfalls wird der zweite Parameter (in Zeile 1 also 15) zurückgegeben. So lassen sich beispielsweise Standardwerte für nicht vorhandene Schlüssel implementieren.

Gut zu sehen ist, dass der Aufruf in Zeile 3 nicht 5, sondern 18 zurück gibt, denn dieser Wert wurde oben dem Schlüssel **Peter** zugewiesen.

Der zweite Parameter der Methode **get()** ist optional: Er muss nicht angegeben werden. Wird kein zweiter Parameter angegeben, gibt die Methode **None** zurück, wenn der gesuchte Schlüssel im Dict nicht vorhanden ist:

```
>>> print persons.get(u"Anke")
None
```

Natürlich können auch jederzeit weitere Einträge zu Dicts hinzugefügt oder bestehende Einträge verändert werden:

```
1 >>> persons[u"Peter"] = 99
2 >>> print persons[u"Peter"]
```

© freiesMagazin CC-BY-SA 3.0

```
3 99
4 >>> persons[u"Hans"] = 15
5 >>> print persons[u"Hans"]
6 15
```

In Zeile 1 wird das Alter von Peter auf 99 verändert. In Zeile 4 wird der Eintrag **Hans** hinzugefügt.

Dictionaries lassen sich – ebenso wie Listen – auch sehr leicht verschachteln. In folgendem Beispiel wird ein verschachteltes Dict genutzt, um ein kleines Adressbuch zu implementieren:

```
1 >>> addresses = {
2 ...     u"Peter":{u"street":"Musterstr. 16", u"mobile":"0151/123 456"},
3 ...     u"Jutta":{u"street":"Beispielstr. 99", u"mobile":"0151/33 44 55"},
4 ... }
```

Hier wird – zur besseren Lesbarkeit über mehrere Zeilen verteilt – ein verschachteltes Dict erstellt. In Zeile 1 wird dem Namen **addresses** ein Dict zugewiesen. Dieses wird dabei direkt initial befüllt. Den Schlüsseln **Peter** und **Jutta** wird dabei nicht wie oben ihr Alter als Wert zugeteilt, sondern es dienen Dicts als Werte.

Der Aufruf

```
>>> print addresses[u"Peter"]
```

gibt nun das dazugehörige Dict zurück:

```
{u'mobile': '0151/123 456', u'street': 'Musterstr. 16'}
```

Auf dieses Dict kann auch direkt zugegriffen werden:

```
>>> print addresses[u"Peter"][u"street"]
Musterstr. 16
```

Dieser Aufruf irritiert vielleicht zunächst. Etwas verständlicher (aber länger) ist folgende Vorgehensweise:

```
1 >>> peters_data = addresses[u"Peter"]
2 >>> type(peters_data)
3 <type 'dict'>
4 >>> peters_data[u"street"]
5 'Musterstr. 16'
```

In Zeile 1 wird der zum Schlüssel **Peter** gehörige Wert der Variable **peters_data** zugewiesen.

Dieser Wert ist wiederum ein Dict mit den zu **Peter** gehörigen Adressdaten (siehe obiges Beispiel). Zeile 2 und 3 zeigen, dass die Variable **peters_data** auf ein Dict zeigt. In Zeile 4 und 5 wird nun wiederum der Schlüssel **street** dieses zweiten Dicts ausgegeben.

Aus dem Ausdruck

```
>>> print addresses[u"Peter"][u"↷
street"]
```

wird also zunächst

```
{u"street":u"Musterstr. 16", u"↷
mobile":"0151/123 456"}[u"street"]
```

was schließlich auf den Wert **Musterstr. 16** verweist.

Insgesamt eignen sich Dicts hervorragend, um Datenstrukturen wie Wörterbücher oder Adressbücher abzubilden. Überall dort, wo Informationen über bestimmte Schlüssel zugänglich sein sollen, empfiehlt sich die Verwendung von Dictionaries.

Ebenso wie über Listen kann sehr leicht über Dicts iteriert werden. Dabei ist aber zu beachten, dass Dicts keine feste Reihenfolge kennen: Es gibt also keine Garantie dafür, dass die Schlüssel eines Dicts beim Iterieren in der Reihenfolge durchlaufen werden, in der sie erstellt wurden [3].

Funktionen

Funktionen sind ein wichtiges Strukturierungsmerkmal moderner Programmiersprachen. Durch sie können häufig benötigte Arbeitsschritte leicht wiederverwertet werden. Damit dienen sie auch der Lesbarkeit und Wartbarkeit des Quelltextes.

Eine Funktion wird in Python wie folgt deklariert:

```
def say_hallo():
    print u"Hallo"
```

Diese Funktion kann nun einfach mit **say_hallo()** aufgerufen werden und wird bei jedem Aufruf die Meldung **Hallo** auf dem Bildschirm ausgeben. Das Schlüsselwort **def** leitet hierbei die Deklaration einer Funktion ein, danach folgt der Name der Funktion, der nach dem Paar runden Klammern mit einem Doppelpunkt abgeschlossen wird. Zu beachten ist, dass der

Rumpf von Funktionen einzurücken ist – wie bei allen Kontrollstrukturen in Python.

Natürlich handelt es sich bei dem obigen Beispiel noch um eine sehr einfache Funktion. Folgende Funktion greift ein Beispiel aus Teil 1 dieser Reihe wieder auf und wird beliebige Zeichenketten mit einer Box aus Leerzeichen umgeben:

```
1 def boxify(text):
2     text_with_borders = u"= {0} =".format(text)
3     line = len(text_with_borders) * u"="
4
5     output = u"{0}\n{1}\n{2}".format(line, text_with_borders, line)
6     return output
```

In Zeile 1 wird – wie gehabt – eine Funktion definiert. Sie hat den namen **boxify** und erwartet genau einen Parameter (hier: **text**). Es können prinzipiell natürlich beliebig viele Parameter im Funktionskopf definiert werden – getrennt werden sie durch Kommata.

In Zeile 2 wird links und rechts der übergebenen Zeichenkette **text** ein Gleichheits- und Leerzeichen eingefügt, in Zeile 3 wird eine Zeichenkette erstellt, die ausschließlich Gleichheitszeichen enthält.

Zeile 5 wirkt nur auf den ersten Blick kompliziert: Wie bereits in Teil 2 dieser Reihe erörtert wurde, handelt es sich bei der Zeichenfolge **\n** um eine sogenannte Escape-Sequenz, die einen Zeilenumbruch erzeugt. Somit beinhaltet die Zeichenkette **output** drei Zeilen: Die erste Zeile enthält ausschließlich Gleichheitszeichen, die zweite Zeile die übergebene Zeichenkette mit Gleichheitszeichen links und rechts, die dritte Zeile schließlich erneut nur Gleichheitszeichen.

Neu ist das Schlüsselwort **return** – es gibt den nachfolgenden Ausdruck (hier: **output**) zurück.

Wird diese Funktion nun aufgerufen, geschieht zunächst scheinbar nichts. Die Funktion **boxify()** macht keine Bildschirmausgaben, sondern gibt nur eine Zeichenkette zurück. Diese muss also noch an die **print()**-Funktion weitergegeben werden:

```
>>> print boxify(u"Mein Haus, mein ↷
Garten, meine Box")
=====================================
= Mein Haus, mein Garten, meine Box =
=====================================
```

Eine Besonderheit ist bei **return** noch zu beachten: Die Anweisung bricht die Funktion sofort ab und liefert einen Rückgabewert – nachfolgende Codezeilen der Funktion werden nicht mehr ausgeführt:

```
1 def test():
2     print u"Hallo!"
3     return
4     print u"Dies ist ein Test"
5
6 print u"start"
7 test()
8 print u"ende"
```

Dieses Beispiel gibt nur die folgende Meldung aus:

```
start
Hallo!
ende
```

Die Zeile 4 (**Dies ist ein Test**) kommt niemals zur Ausführung. Gut zu sehen ist auch, in welcher Reihenfolge die Anweisungen ausgeführt werden: Zwar wird in den Zeilen 1 bis 4 die Funktion definiert – sie wird aber noch nicht ausgeführt. Es muss also immer zwischen der „Deklaration“ einer Funktion und dem „Aufruf“ derselben unterschieden werden. Zuerst wird hier demnach die Meldung **start** ausgegeben. Dann wird die Funktion aufgerufen und abgearbeitet – die Meldung **Hallo!** erscheint. Durch die Anweisung **return** wird der Programmfluss nun in Zeile 8 fortgesetzt – **ende** wird ausgegeben.

Zu beachten ist, dass Funktionen keine **return**-Anweisung haben müssen – haben sie keine, kehrt der Programmfluss nach dem Abarbeiten des Funktionsrumpfes zum Ausgangspunkt zurück. Der Rückgabewert der Funktion ist dann **None**.

Eine weiteres wichtiges Element von Funktionen sind Standardparameter. Sie werden durch ein Gleichheitszeichen definiert:

```
def say_something(what, who=u"Karl"):
    print u"{0} sagt: '{1}'".format(who, what)

say_something(u"Hi")
say_something(u"Tach auch", u"Bernd")
```

Der Parameter **what** ist obligatorisch – bei jedem Funktionsaufruf muss er angegeben werden, sonst kommt es zu einem Fehler. Der zweite Parameter hingegen ist optional: Wird er nicht angegeben, erhält er automatisch den Wert **Karl**. Entsprechend gibt obiges Skript folgende Ausgabe aus:

```
Karl sagt: 'Hi'
Bernd sagt: 'Tach auch'
```

Wichtig ist: Bei der Funktionsdefinition müssen zuerst immer die obligatorischen Parameter angegeben werden. Die Funktion

```
def say_something(who="Karl", what):
   ...
```

führt also zu einem Fehler beim Versuch, das Skript auszuführen.

Parameter an Funktionen übergeben

In den obigen Beispielen wurden bereits verschiedene Parameter an die Funktionen übergeben. Etwa **Tach auch** und **Bernd** an die Funktion **say_something()**. In dem Beispiel muss sich der Programmierer peinlich genau an die im Funktionskopf definierte Reihenfolge halten. Die Parameter sind also abhängig von der Position – sie sind „positional“.

Als Beispiel sei die folgende (nicht besonders schöne) Funktion gegeben, bei der die Argumente alle in einer bestimmten Reihenfolge angegeben werden müssen:

```
def print_a_lot(name, country, street, adress, mobile, age, sex, hobbies):
    print u"{name} stammt aus {country} ...".format(name=name, country=country)
```

Statt hier nun die erforderlichen Argumente immer in der einzig richtigen Reihenfolge anzugeben, gibt es noch die Möglichkeit, die Argumente per Schlüsselwort zu übergeben:

```
print_a_lot(name=u"Bernd", age=18,
sex=u"m", street=u"Musterstrasse",
adress=18, country=u"Deutschland",
mobile=u"0151-123456789", hobbies=u
"lesen")
```

Diese Art des Aufrufes kann die Lesbarkeit von Quelltexten enorm erhöhen.

```
def print_info(name, country=None, street=None, adress=None, mobile=None,
age=None, sex=None, hobbies=None):
    if age:
        print u"Hallo {0}! Du bist {1} Jahre alt!".format(name, age)
    else:
        print u"Hallo {0}".format(name)
```

Die Funktion **print_info()** unterscheidet sich von **print_a_lot()** darin, dass alle Parameter bis auf **name** optional sind – sie müssen nicht angegeben werden. Wird aber ein Alter übergeben (**age**), wird zusätzlich zum Namen auch das Alter ausgegeben. Der Aufruf ist:

```
print_info(u"Jutta", age=25)
```

Der erste Parameter ist positional: Hier wird der Name übergeben. Da alle anderen Parameter optional sind, werden sie einfach ausgelassen. Lediglich der **age**-Parameter wird noch als Schlüsselwort-Argument übergeben.

In Python sind auch Funktionen ganz normale Objekte. Auch sie lassen sich damit beispielsweise an Namen binden:

```
1 >>> from operator import add
2 >>> add(1, 3)
3 4
4 >>> plus = add
5 >>> plus
6 <built-in function add>
7 >>> plus(1, 3)
8 4
```

Achtung: Wichtig ist hier, dass die Funktion in Zeile 5 ohne runde Klammern an einen Namen gebunden wird. Andernfalls würde die Funktion aufgerufen und das Ergebnis an den Namen gebunden werden. Anders gesagt: Mit Klammern wird eine Funktion aufgerufen, ohne Klam-

mern wird das Funktionsobjekt angesprochen, die Funktion aber nicht ausgeführt.

In Zeile 1 wird zunächst die Funktion **add()** aus dem Modul **operator** importiert (siehe dazu nächster Abschnitt). Die Funktion **add()** addiert – wie der Operator **+** – zwei Zahlen. Sie stellt die gleiche Funktionalität nur als Funktion zur Verfügung (Zeile 3).

In Zeile 5 wird die Funktion **add()** zunächst an den Namen **plus** gebunden. In Zeile 6 und 7 wird gezeigt, dass der Name **plus** noch immer auf die Funktion **add()** verweist – **add** und **plus** sind identisch.

In Zeile 9 wird deutlich, dass man an Namen gebundene Funktionen genau so wie normale Funktionen aufrufen kann. Am Ende dieses Teils wird diese Technik an einem praktischen Beispiel veranschaulicht.

Module

Module bieten eine einfache Möglichkeit, seine Skripte um zusätzliche Funktionen zu erweitern. Es wurde bereits angesprochen, dass Python mit einer Vielzahl zusätzlicher Bibliotheken ausgeliefert wird – diese Bibliotheken heißen in Python Module. Sie werden durch den Befehl **import** in ein eigenes Skript eingebunden [4]:

```
1  import os
2
3  counter = 0
4
5  files = os.listdir(".")
6  for entry in files:
7      if os.path.isfile(entry):
8          counter += 1
9
10 print u"Es gibt {0} Dateien in ↷
   diesem Verzeichnis".format(↷
   counter)
```

In Zeile 1 wird der Interpreter angewiesen, das Modul **os** im jetzigen Skript verfügbar zu machen. Dieses Modul enthält verschiedene Funktionen zum Kopieren, Verschieben oder Löschen von Dateien. Auch das Auflisten des aktuellen Verzeichnisinhaltes gehört dazu [5].

In Zeile 5 wird die Funktion **listdir()** des Modules **os** aufgerufen. Der Parameter **.** verweist auf das aktuelle Verzeichnis. Die Funktion erstellt eine Liste mit allen Dateien und Ordnern des aktuellen Verzeichnisses und gibt diese Liste zurück. Die Variable **files** zeigt nun auf diese Liste.

In Zeile 6 wird eine **for**-Schleife definiert, die über die zuvor erstellte Liste iteriert. In Zeile 7 wird überprüft, ob es sich bei dem jeweiligen Eintrag um eine Datei oder ein Verzeichnis handelt – auch dazu wird eine Funktion aus dem Modul **os** genutzt. Falls es sich um eine Datei handelt, gibt diese Funktion **True** zurück, andernfalls **False**. Nur im ersten Fall ist die **if**-Bedingung wahr und der Zähler wird erhöht.

Nach dem Durchlauf der Schleife setzt der normale Programmfluss fort – die letzte Zeile des Skriptes wird ausgeführt und zeigt die Zahl der Dateien im aktuellen Verzeichnis an.

Es gibt auch eine Möglichkeit, Funktionen aus Modulen so zu importieren, dass sie im Skript direkt verfügbar sind – mit der Anweisung **from MODUL import FUNKTION/OBJEKT**. So könnte es im obigen Beispiel etwa heißen:

```
from os import listdir
```

Allerdings müsste dann auch Zeile 5 angepasst werden: Da durch den **from**-Import die **listdir()**-Funktion direkt im Namensraum [5] des Skriptes verfügbar ist, müsste die Zeile nun wie folgt lauten:

```
files = listdir(".")
```

Das sieht auf den ersten Blick sehr bequem aus, führt aber jetzt in Zeile 7 zu Problemen: Da nur die Funktion **listdir** importiert wurde, ist ein Zugriff auf die Funktion **os.path.isfile** nicht möglich. Diese Funktion müsste nun zusätzlich importiert werden.

Das Importieren von einzelnen Funktionen hat noch andere Nachteile [6]: So erschwert es die Verständlichkeit des Quelltextes, da Fremde nicht wissen, ob mit **listdir** hier die Funktion aus dem **os**-Modul gemeint ist oder vielleicht irgendwo im Quelltext noch eine eigene **listdir**-Funktion zu finden ist, die etwas ganz anderes macht. In aller Regel sollte daher von **from**-Importen abgesehen und die zusätzliche Schreibarbeit in Kauf genommen werden.

Was es nicht alles gibt

Das **os**-Modul erscheint noch recht bodenständig: Bisher wurde damit der Inhalt eines Verzeichnisses aufgelistet und überprüft, ob eine bestimmte Datei ein Ordner oder eine Datei ist. Darüber hinaus gibt es aber Module für grafische Oberflächen [7], für Datenbanken [8], für die Bildbearbeitung [9] oder für mathematische und wissenschaftliche Anforderungen [10]. Es gibt Module, um Spiele zu programmieren [11], Module, um automatisiert Eingaben in Webseiten vorzunehmen [12], Module für Mediendateien [13], für IMAP [14] oder sogar BitTorrent [15]. Nicht alle diese Module gehören dabei zum Standardumfang [16] der Sprache – die fehlenden lassen sich aber leicht über die Paketquellen oder das eigens für Python entwickelte EasyInstall-System installieren.

Module selbst gemacht

Es stellt sich nun natürlich die Frage, wie sich derartige Module selbst erstellen lassen. Die meisten Leser dieser Reihe werden dabei vermutlich schon lange das eine oder andere Python-Modul erstellt haben:

```
1 #! /usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 def boxify(text):
5     text_with_borders = u"= {0} =".format(text)
6     line = len(text_with_borders) * u"="
7
8     output = u"{0}\n{1}\n{2}".format(line, text_with_borders, line)
9     return output
```

Listing 1: box.py

Ein Python-Modul ist zunächst nämlich nichts anderes als eine Textdatei mit der Endung **.py**. Die oben besprochene Funktion **boxify()** soll im Folgenden in ein eigenes Modul ausgegliedert werden.

Diese obigen Zeilen werden in die Datei **box.py** gespeichert. Die ersten beiden Zeilen wurden bereits im ersten Teil dieser Reihe besprochen: Es handelt sich um die Shebang-Zeile und um die Angabe der Zeichenkodierung. Auch die Funktion **boxify()** sollte mittlerweile hinlänglich bekannt sein.

Damit wurde nun das Modul **box** erstellt. Der Modulname entspricht also immer dem Dateinamen abzüglich der Endung **.py**.

```
1 #! /usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import box
5
6 name = unicode(raw_input(u"Bitte ↷
  Namen eingeben: "))
7 print box.boxify(name)
```

Listing 2: myprog.py

Um dieses Modul nun verwenden zu können, wird eine weitere Python-Datei im selben Verzeichnis erstellt: **myprog.py** (siehe oben).

In Zeile 4 wird das selbst erstellte Box-Modul importiert. In Zeile 6 wird der Benutzer zur Eingabe seines Namens aufgefordert. In Zeile 7 schließlich wird die Funktion **boxify()** aus dem **box**-Modul mit dem eingegebenen Namen aufgerufen. Das Resultat wird mit **print** auf dem Bildschirm ausgegeben.

Einschränkungen und Erweiterungen

Der Python-Interpreter hat eine recht genaue Vorstellung davon, in welchen Verzeichnissen sich ein Modul befinden darf [17]. Befindet sich das Modul nicht in einem dieser Verzeichnisse, kommt es zu einem Fehler. Der Python-Interpreter schaut aber zusätzlich auch immer in das Programmverzeichnis – hier also etwa in das Verzeichnis der Datei **myprog.py**. So lange die selbst erstellten Module in diesem Verzeichnis zu finden sind, befindet sich der Anfänger also auf der sicheren Seite.

Eine Erweiterung des Modul-Systems stellen die sogenannten „Packages“ dar [18]. Damit lassen sich verschiedene zusammengehörige Module bündeln und strukturieren. In dieser Einführung wird aber auf eine weitergehende Behandlung dieser Thematik verzichtet. Wer aber größere Bibliotheken programmieren möchte oder eine ganze „Werkzeugsammlung“ in Modulen organisieren will, sollte sich Packages einmal näher ansehen [19].

Nachdem nun Listen, Dictionaries, Zeichenketten, Module, Funktionen und verschiedene Kontrollstrukturen in den ersten drei Teilen dieser Einführung besprochen wurden, sollen im nächsten Teil Klassen vorgestellt werden.

Praktisches Beispiel: Der Taschenrechner

In folgendem Beispiel kommen verschiedene bereits erlernte Techniken zum Einsatz.

```
 1 #!/usr/bin/env python
 2 # -*- coding: utf-8 -*-
 3
 4 from operator import add, sub, ↷
   mul, div
 5
 6 def info(*args):
 7     print "Moegliche Befehle:"
 8     print ",".join(dispatch)
 9     print "Syntax: BEFEHL [↷
       PARAM_A PARAM_B]"
10
11 dispatch = {
12     u"help": info,
13     u"add": add,
14     u"sub": sub,
15     u"mul": mul,
16     u"div": div,
17     u"addiere": add,
18     u"plus": add
19 }
20
21 def parser():
22     while True:
23         user_command = unicode(↷
           raw_input("calc: "))
24         tokens = user_command.↷
           split()
25         command = tokens[0]
26         arg_a, arg_b = None, None
27         if len(tokens) > 1:
28             arg_a = int(tokens↷
               [1])
29             arg_b = int(tokens↷
               [2])
30         print tokens
31         if command == u"quit":
32             return
33         elif command in dispatch:
34             result = dispatch[↷
               command](arg_a, arg_b↷
               )
35             print ">>> {0}".↷
               format(result)
36         else:
37             print "Unbekanntes ↷
               Kommando '{0}'".↷
               format(command)
38             print "Tippe 'help' ↷
               fuer Hilfe."
39
40 if __name__ == "__main__":
41     parser()
```

Listing 3: calc.py

In Zeile 4 werden die Funktionen **add**, **sub**, **mul** und **div** aus dem Modul **operator** importiert. Sie stellen die Funktionalität der Operatoren +, -, * und / als Funktion zur Verfügung (siehe oben).

In Zeile 6 wird die Funktion **info()** definiert. Das Sternchen (*) vor dem Parameter **args** führt dazu, dass die Funktion beliebig viele (positionale) Argumente akzeptiert und diese als Liste **args** bereitstellt. Keine Sorge: Dieses Vorgehen dient in diesem Fall nur dazu, den Taschenrechner unempfindlicher gegen Fehleingaben zu machen. Die Funktion ignoriert alle Parameter und gibt lediglich alle möglichen Befehle durch Kommata getrennt aus (Zeile 8).

In Zeile 11 wird ein Dict definiert und initial befüllt. Den Schlüsseln werden hier Funktionen als Werte zugewiesen. Oder anders: Die Funktionen werden an Schlüssel des Dicts gebunden. Ein Aufruf von **dispatch["plus"](4+1)** ist im Folgenden also identisch mit einem Aufruf von **add(4, 1)**. In Zeile 13, 17 und 18 ist zu sehen, dass die Funktion **add** gleich mehreren Dict-Schlüsseln zugewiesen wird. Jeder Dict-Schlüssel ist dabei ein möglicher Befehl. Der Benutzer wird später also mit **addiere**, **add** und **plus** gleichermaßen addieren können.

Das Herzstück des Taschenrechners ist die Funktion **parse()**, die in Zeile 21 definiert wird. In Zeile 22 wird weiterhin eine Schleife implementiert. Da die Bedingung **True** immer wahr ist, handelt es sich hierbei erst einmal (scheinbar) um eine Endlosschleife.

In Zeile 23 wird eine Benutzereingabe eingelesen. Diese könnte etwa wie folgt aussehen:

```
add 3 7
```

In Zeile 24 wird diese Benutzereingabe durch die Funktion **split()** aufgeteilt. Da diese Funktion hier ohne Parameter aufgerufen wird, teilt sie die Zeichenkette bei allen Leerräumen (Whitespace-Zeichen). Die Eingabe von **add 3 7** würde so zur Liste **["add", "3", "7"]** führen, die dem Namen **tokens** zugewiesen wird.

In Zeile 25 wird das erste Element der Liste (der Befehl) dem Namen **command** zugewiesen. Weiterhin werden zwei Variablen **arg_a** und **arg_b** erstellt. Sie sollen später die Zahlen enthalten, die addiert werden sollen, zeigen aber zunächst nur auf **None**.

Zeile 27 testet, ob die Liste **tokens** mehr als nur ein Element enthält. Ist dies der Fall, werden das zweite und dritte Element der Eingabe (also 3 und 7, wenn man vom obigen Beispiel ausgeht) mit der **int()**-Funktion in Zahl-Objekte umgewandelt und den Namen **arg_a** bzw. **arg_b** zugewiesen. In Zeile 30 wird die Liste **tokens** ausgegeben.

In Zeile 30 ff. findet sich nun eine **if**-Kontrollstruktur. Nach der Eingabe von **quit** soll das Programm beendet werden. Dazu wird mit dem Schlüsselwort **return** die Abarbeitung der Funktion **parse** direkt unterbrochen. Die in Zeile 22 definierte „Endlosschleife“ verfügt damit also doch über eine Abbruchbedingung.

Das Schlüsselwort **elif** in Zeile 33 wurde auch bereits besprochen. Es wird getestet, ob die erste Benutzereingabe (**add** im vorherigen Beispiel) im Dict **dispatch** (Zeile 11) vorhanden ist. In Zeile 34 geschieht nun die eigentlich „Magie“ des Taschenrechners:

```
result = dispatch[command](arg_a, ↷
arg_b)
```

Abhängig vom Inhalt der Variable **command** wird nun der dazugehörige Schlüssel im Dict angesprochen. Da diesen Schlüsseln aber in Zeile 11 ff. Funktionen zugewiesen wurden, wird mit der Anweisung **dispatch[command]** abhänging von **command** eine Funktion angesprochen. Die Klammern hinter dieser Anweisung **(arg_a, arg_b)** enthalten also die Parameter für diese Funktion. Das Ergebnis der jeweiligen Funktion ist nun über die Variable **result** verfügbar.

Was passiert hier also? Der Taschenrechner liest Benutzereingaben in der Form **BEFEHL ZAHL1 ZAHL2** ein. Ist nun **BEFEHL** ein Schlüssel im Dict **dispatch**, wird die dazugehörige Funktion aufgerufen. Der Befehl **addiere** würde somit zum Aufruf der Funktion **add** führen, der Befehl **info** zum Aufruf der Funktion **info**. Die Eingaben **ZAHL1** und **ZAHL2** werden dabei jeweils als Parameter übergeben. Dies ist auch der Grund, warum die in Zeile 6 definierte Funktion **info** mit ***args** beliebig viele Parameter übernimmt. So reagierte sie tolerant auf eventuelle Falschangaben des Benutzers, denn diese werden in jedem Fall an die Funktion übergeben.

In Zeile 35 wird das Ergebnis der aufgerufenen Funktionen formatiert und ausgegeben. Da in der Funktion **info** kein Rückgabewert festgelegt wurde, gibt sie immer **None** zurück.

In Zeile 36-38 wird nun schließlich für den Fall vorgesorgt, dass **BEFEHL** nicht im Dict **dispatch** vorkommt. Dann hat der Benutzer eine ungültige Eingabe gemacht und wird darauf hingewiesen.

Der **if**-Block in Zeile 40 f. stellt schließlich sicher, dass die Funktion **parser()** nur ausgeführt wird, wenn das Python-Skript direkt gestartet wurde. Würde man das Skript als Modul importieren (siehe oben), würde der Taschenrechner nicht automatisch ausgeführt werden.

Natürlich wirkt dieser Taschenrechner auf den ersten Blick sehr kompliziert. Er zeigt aber, warum das Binden von Funktionen an Namen (oder hier: Dict-Schlüssel) sehr sinnvoll sein kann. Ohne diese Technik müsste der Programmierer jede Eingabe von Hand auswerten:

```
if command == "add" or command == "↷
addiere" or command == "plus":
    result = arg_a + arg_b
elif command == "sub":
    result = arg_a - arg_b
elif command == "div":
    result = arg_a / arg_b
elif command == "mul":
    result = arg_a * arg_b
elif command == "info":
    info()
elif command == "quit":
    return
else:
```

```
    print "Unbekanntes Kommando '
    {0}'".format(command)
    print "Tippe 'help' fuer Hilfe
    ."

print ">>> {0}".format(result)
```

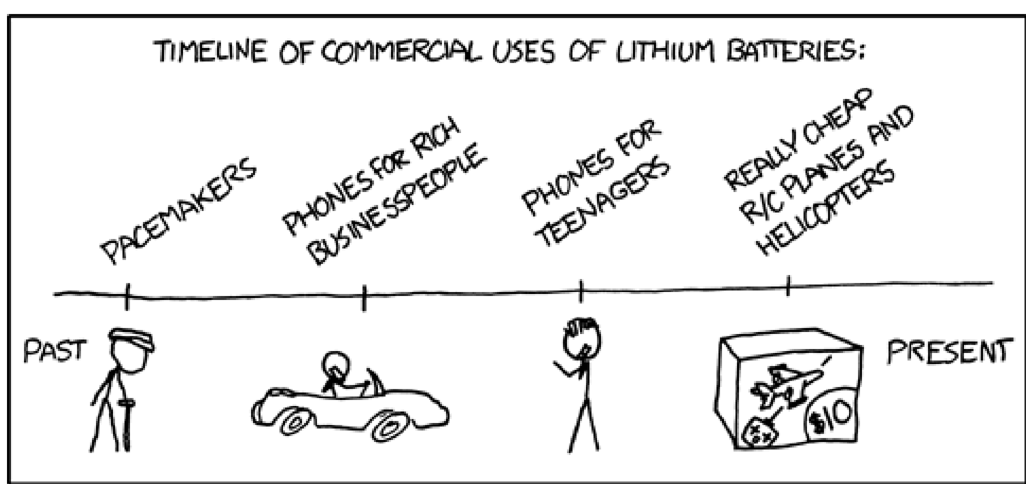
LINKS

[1] http://docs.python.org/library/string.html#format-string-syntax
[2] http://docs.python.org/tutorial/datastructures.html#dictionaries
[3] http://www.python.org/dev/peps/pep-0372/
[4] http://docs.python.org/tutorial/modules.html
[5] http://docs.python.org/library/os.html
[6] http://effbot.org/zone/import-confusion.htm
[7] http://de.wikipedia.org/wiki/Liste_von_GUI-Bibliotheken#Python
[8] http://initd.org/tracker/pysqlite
[9] http://www.pythonware.com/products/pil/
[10] http://www.scipy.org/
[11] http://www.pygame.org/news.html
[12] http://wwwsearch.sourceforge.net/mechanize/
[13] http://pymedia.org/
[14] http://docs.python.org/library/imaplib.html
[15] http://www.rasterbar.com/products/libtorrent/python_binding.html
[16] http://docs.python.org/modindex.html
[17] http://docs.python.org/using/cmdline.html#envvar-PYTHONPATH
[18] http://docs.python.org/tutorial/modules.html#packages
[19] http://docs.python.org/distutils/index.html

Autoreninformation

Daniel Nögel (Webseite) beschäftigt sich seit drei Jahren mit Python. Ihn überzeugt besonders die intuitive Syntax und die Vielzahl der unterstützten Bibliotheken, die Python auf dem Linux-Desktop zu einem wahren Multitalent machen.

Diesen Artikel kommentieren



„Lithium Batteries“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/560

Python-Programmierung: Teil 4 – Klassenunterschiede

von Daniel Nögel

Im dritten Teil dieser Einführung *„Python-Programmierung: Teil 3“* auf Seite 14 wurden mit Funktionen und Modulen zwei elementare Konstrukte zur Strukturierung von Python-Skripten vorgestellt. In diesem Teil soll nun mit Klassen ein weiteres wichtiges Konstrukt besprochen werden.

Objektorientierte Programmierung

Bei der objektorientierten Programmierung (OOP [1]) handelt es sich um ein sogenanntes Programmierparadigma – also um ein bestimmtes Entwurfsmuster beim Programmieren. Bei der OOP wird versucht, Daten und Funktionen zu sinnvollen Einheiten zusammenzufassen und an sogenannte Objekte zu binden.

Neben „Objekten” gibt es zumindest noch drei weitere wichtige Begriffe im Zusammenhang mit objektorientierter Programmierung, die im Folgenden immer wieder auftauchen: „Klassen“, „Attribute“ und „Methoden“.

Als Beispiel sollen für eine Rennsimulation unterschiedliche Autos zur Verfügung stehen. Diese Autos unterscheiden sich in Länge, Gewicht, Geschwindigkeit, Beschleunigung, Farbe etc. Wenn nun 20 unterschiedliche Autos gleichzeitig um die Wette fahren, müssen viele Daten verwaltet werden. Es bietet sich an, diese Daten jeweils zusammenzufassen und zu bündeln. So könnte es beispielsweise eine Objekt „Auto“ mit folgenden Eigenschaften geben:

- Länge
- Gewicht
- Geschwindigkeit
- Beschleunigung
- Farbe

Diese Definition der späteren Auto-Objekte im Quelltext eines Programmes nennt man Klasse. Jedes Objekt wird aus dieser Klasse erstellt – man spricht auch von der Instanz einer Klasse. Klasse und Objekt verhalten sich also stark vereinfacht ausgedrückt so zueinander, wie „Bauplan“ und „Auto auf der Straße“.

Bei Länge, Gewicht und den anderen Angaben handelt es sich offensichtlich um Eigenschaften der jeweiligen Autos. Eigenschaften von Objekten nennt man Attribute. Objekte können aber auch Funktionen besitzen – also etwa **`bremse_ab()`** oder **`gib_gas()`**. Diese Funktionen von Klassen bzw. Objekten nennt man Methoden.

Klassen in Python

Das Beispiel **`BeispielKlasseCar.py`** zeigt, wie man eine (leicht abgewandelte) Auto-Klasse in Python umsetzt.

```
1  class Car(object):
2      def __init__(self, speed=0, max_speed=140, acceleration=5, color="↷
       grey"):
3          self.speed = speed
4          self.max_speed = max_speed
5          self.acceleration = acceleration
6          self.color = color
7
8      def brake(self):
9          self.speed -= self.acceleration
10         print "Current Speed: {0}".format(self.speed)
11
12     def accelerate(self):
13         self.speed += self.acceleration
14         print "Current Speed: {0}".format(self.speed)
15
16     def honk(self):
17         print "Tuuuuut Tuuuut"
```

Listing 1: BeispielKlasseCar.py

© freiesMagazin CC-BY-SA 3.0

In Zeile 1 wird die Klasse definiert. Dies wird durch das Schlüsselwort **class** gekennzeichnet. In Klammern dahinter wird angegeben, von welchen anderen Klassen die neue Klasse erben soll (siehe dazu den nächsten Abschnitt). In älteren Python-Versionen sollte hier immer von **object** geerbt werden – nur dann steht der volle Funktionsumfang der Klassen in Python zu Verfügung. Wer bereits Python >= 3 verwendet, verzichtet auf diese Angabe und hat dennoch den vollen Funktionsumfang der Klassen. Wie jede Kontrollstruktur in Python wird auch der Kopf der Klasse mit einem Doppelpunkt abgeschlossen.

In Zeile 2 wird die Spezialmethode **__init__** definiert. Diese wird aufgerufen, wenn eine Instanz der Klasse (also ein Objekt) erstellt wird. Die Methode **__init__** wird hier mit fünf Parametern definiert: **self**, **speed**, **max_speed**, **acceleration** und **color**.

In Python muss jede Methode (egal ob Spezialmethoden wie **__init__()** oder eigene Methoden wie **brake()**) als erstes den Parameter **self** akzeptieren. Durch diesen Parameter kann später zu jeder Zeit auf die Attribute und Methoden eines Objektes zugegriffen werden. **self** ist also – wie der Name schon andeutet – eine (Selbst)Referenz des späteren Objektes. Das mag zunächst umständlich und lästig wirken – tatsächlich gibt es aber gute Gründe für dieses Vorgehen [2].

Die übrigen Parameter (**speed**, . . . , **color**) sind optional. Sie definieren die Standardwerte für das Auto-Objekt.

In den Zeilen 3 bis 6 werden die Parameter an Klassenattribute gebunden: Erst dadurch haben die späteren Objekte auch tatsächlich die entsprechenden Eigenschaften.

In Zeile 8 wird eine Methode **brake()** definiert. Als Klassenmethode muss auch sie den Parameter **self** akzeptieren. In Zeile 9 wird deutlich warum: Über diesen Parameter kann wieder auf das in Zeile 3 definierte Attribut **speed** zugegriffen werden. Im Fall der Methode **brake()** wird das Attribut **speed** um die Höhe des Attributs **acceleration** reduziert.

In den Zeilen 12 und 16 werden weiterhin noch die Methoden **accelerate()** und **honk()** definiert.

Klassen zum Leben erwecken

Die Klasse **Car** wurde definiert. Bisher gibt es aber noch keine Instanz dieser Klasse. Im Folgendem werden zwei Car-Objekte erstellt:

```
1  >>> car1 = Car()
2  >>> car2 = Car(speed=50)
3  >>> car1.speed
4  0
5  >>> car2.speed
6  50
7  >>> car1.accelerate()
8  Current Speed: 5
9  >>> car1.accelerate()
10 Current Speed: 10
11 >>> car2.brake()
12 Current Speed: 45
13 >>> car1.honk()
14 Tuuuuut Tuuuut
```

In den Zeilen 1 und 2 werden Instanzen der Klasse **Car** erstellt. Dabei wird implizit die Funktion **__init__()** aufgerufen. Für **car1** werden keine Parameter übergeben, sodass die Standard-Parameter greifen (**speed=0**). Für **car2** wird dahingegen **speed=50** übergeben, so dass das **speed**-Attribut dieser Car-Instanz von Anfang an auf **50** gesetzt wird.

Die Zeilen 3 und 5 veranschaulichen dies: Hier wird jeweils das **speed**-Attribut der beiden Car-Objekte ausgelesen. Für **car1** ist es 0, für **car2** erwartungsgemäß 50. In den Zeilen 7 und 9 wird **car1** jeweils beschleunigt. Da die Beschleunigung **acceleration** in Zeile 2 der Klassendefinition mit **5** angegeben wurde, wird das **speed**-Attribut jeweils um 5 erhöht.

In Zeile 11 wird die **brake()** Methode von **car2** aufgerufen. Das **speed**-Attribut vermindert sich damit von 50 auf 45.

Zeile 13 zeigt, dass Klassenmethoden nicht immer Attribute der Klasse manipulieren müssen: Hier „hupt“ **car1**, indem die aufgerufenen Methode eine entsprechende Meldung auf dem Bildschirm ausgibt.

Das Beispiel zeigt, wie aus dem Bauplan **Car** zwei Car-Objekte erstellt und an die Namen **car1** bzw. **car2** gebunden wurden. Diese Objekte sind völlig unabhängig voneinander: Veränderungen am Attribut **speed** von **car1** wirken sich nicht auf **car2** aus und umgekehrt. Theoretisch könnten nun beliebig viele weitere Car-Objekte erstellt werden:

```
>>> ferrari = Car(max_speed=280, acceleration=10, color="red")
>>> trabbi = Car(max_speed=70, acceleration=2, color="blue")
```

Vererbung

Vererbung ist ein weiteres mächtiges Element der OOP: Es ist damit möglich, Attribute und Methoden von anderen Klassen zu „erben“ und weitere Attribute und Methoden hinzuzufügen. So können bequem Klassen konstruiert werden, die aufeinander aufbauen und sich gegenseitig erweitern.

Oben wurde bereits die Klasse **Car** für eine fiktive Rennsimulation definiert. Durch Vererbung lassen sich spezifischere Klassen von Autos erstellen – etwa eine Rennwagenklasse mit hoher Beschleunigung und hoher Endgeschwindigkeit:

```
1  class Racer(Car):
2      def __init__(self, color="red"):
3          Car.__init__(self, speed=0, max_speed=400, acceleration=10, color
           =color)
4          self.current_gear = 0
5
6      def honk(self):
7          print "Tuuuuut Tuuuut Tuuuut"
8
9      def shift_up(self):
10         self.current_gear += 1
11         print "Current gear: {0}".format(self.current_gear)
12
13     def shift_down(self):
14         self.current_gear -= 1
15         print "Current gear: {0}".format(self.current_gear)
```

Listing 2: BeispielKlasseRacer.py

Die Definition von **BeispielKlasseRacer.py** ähnelt sehr der bei der Klasse **Car**. Allerdings wird hier von **Car** geerbt (Zeile 1). Die neue Klasse **Racer** hat also zunächst die gleichen Attribute und Methoden wie die Klasse **Car**. Die Init-Methode unterschiedet sich aber schon deutlich: Sie kennt neben dem Parameter **self** nur noch den Parameter **color**.

Nun wurde zwar von der Klasse **Car** geerbt – diese wurde aber noch nicht initialisiert. Irgendwie soll die Racer-Klasse ja die gleichen Attribute erhalten, wie die Car-Klasse (**speed**, **acceleration** etc.). Dies geschieht in Zeile 3.

Die Init-Methode wird hier deshalb explizit aufgerufen, damit als erster Parameter (**self**) eine Referenz auf das neue Racer-Objekt übergeben werden kann. Nach der Initialisierung in Zeile 3 hat das Racer-Objekt also die Attribute **speed**, **max_speed** und **acceleration** mit den Werten 0, 400 und 10. Außerdem noch das das Attribut **color**.

In Zeile 4 erhält die Klasse Racer ein zusätzliches Attribut. Da die Wagen der Racer-Klasse eine manuelle Schaltung erhalten sollen, wird der gerade eingelegte Gang an das Attribut **current_gear** gebunden.

Die Bedeutung der drei Funktionen **honk()**, **shift_up()** sowie **shift_down()** lassen sich am folgenden Beispiel gut ablesen:

```
1  >>> ferrari = Racer()
2  >>> ferrari.honk()
3  Tuuuuut Tuuuut Tuuuut
4  >>> ferrari.shift_up()
5  Current gear: 1
6  >>> ferrari.accelerate()
7  Current Speed: 10
8  >>> ferrari.accelerate()
9  Current Speed: 20
10 >>> ferrari.shift_up()
11 Current gear: 2
```

Zunächst wird also eine Instanz der Klasse **Racer** erstellt und an den Namen **ferrari** gebunden. Der Aufruf der Methode **honk()** führt bei Instanzen der Car-Klasse zur Ausgabe von **Tuuuuut Tuuuut** – bei Instanzen der Racer-

Klasse aber zu **Tuuuuut Tuuuut Tuuuut**. Die Methode **honk()** der Klasse **Racer** *überschreibt* also die gleichnamige Methode der Elternklasse.

In den Zeilen 4 und 10 kommen die neuen Methoden **shift_up()** und **shift_down()** ins Spiel. Sie *erweitern* die ursprüngliche Car-Klasse also. In den Zeilen 6 und 8 wiederum wird deutlich, dass die Methode **accelerate()** offensichtlich von der Car-Klasse *geerbt* wurde. Obwohl sie in der Racer-Klasse nicht definiert wurde, steht sie zur Verfügung und zeigt das gleiche Verhalten wie in der Car-Klasse.

Es lassen sich noch viele andere Beispiele finden, in denen Klassen und Vererbungen sinnvoll zur Modellierung von Datenstrukturen eingesetzt werden können. Etwa in einem Rollenspiel: Hier würde es vermutlich zunächst die Basisklasse **Charakter** geben – diese hätte Attribute wie **Kampfkraft**, **Leben** oder **Schnelligkeit**. Darauf aufbauend gäbe es Klassen wie **Magier**, **Kämpfer** oder **Drache**, die jeweils unterschiedliche Attribute haben (mehr oder weniger Lebensenergie etc.) und ganz eigene Methoden wie **zaubere()** oder **spucke_feuer()** implementieren.

Klassen im Einsatz

Natürlich sind weder Renn- noch Rollenspiele der primäre Einsatzzweck von Python. Es sind lediglich sehr anschauliche Beispiele dafür, wie man ausgehend von einer Basisklasse verschiedene andere Klassen entwirft.

Ein etwas praxisnäheres Beispiel könnte aber eine Musikverwaltung sein. Auch hier stellt sich die Frage: Wie lässt sich die Funktionalität, die implementiert werden soll, sinnvoll strukturieren und in Klassen kapseln?

Eine Musikverwaltung arbeitet in aller Regel mit einer Datenbank. Hier wäre eine eigene Klasse sicher sinnvoll. Sie hätte Methoden wie **erstelle_datenbank()**, **finde_lied()** oder **trage_lied_ein()**. Der Vorteil dabei: Wenn im gesamten Programm diese Schnittstellen der Datenbankklasse benutzt werden, kann sehr leicht eine zweite Datenbankklasse umgesetzt werden – etwa um auch die Unterstützung von PostgreSQL anzubieten. Da die Schnittstellen (also die verfügbaren Methoden und Attribute) bei beiden Klassen identisch sind, ist ein Austausch ohne Probleme möglich.

Weiterhin benötigt die Musikverwaltung eine grafische Benutzeroberfläche (GUI). Auch diese würde in eine Klasse gekapselt werden und hätte Methoden wie **zeige_dialog()**, **knopf_wurde_geklickt()** oder **eingabefeld_wurde_veraendert()**. Wenn die GUI-Klasse gut entworfen wird, kann später theoretisch sehr leicht eine Ersatz-GUI entwickelt werden, etwa wenn doch Qt [3] statt GTK [4] verwendet werden soll.

Diese Beispiele lassen sich leicht fortspinnen. Die Anbindung an last.fm (Attribute: **benutzername**, **passwort**; Methoden: **uebertrage_gerade_gespieltes_lied()** oder **anmelden()**) ließe sich sicher ebenso sinnvoll in einer Klasse kappseln wie etwa ein Cover-Downloader für Amazon.

Alles ist eine Klasse

In Python haben Klassen aber eine noch viel grundlegendere Bedeutung, denn alle Datentypen (Zeichenketten, Ganz- und Fließkommazahlen, Listen oder Dictionaries) sind in Python Objekte. Es gibt also auch entsprechende Klassen, von denen geerbt werden kann:

```
class BoxiCode(unicode):
    def boxify(self):
        text_with_borders = u"= {0}↷
         =".format(self)
        line = len(↷
        text_with_borders) * u"="

        output = u"{0}\n{1}\n{2}".↷
        format(line, ↷
        text_with_borders, line)
        return output
```

Listing 3: BoxiCode.py

Hier etwa wird von der Klasse **unicode** geerbt. Unicode-Objekte können wie folgt erstellt werden:

```
>>> unicode("Hallo")
u'hallo'
```

Aus dieser Einführung ist folgende Kurzschreibweise bereits bekannt:

```
>>> u"Hallo"
u'Hallo'
```

In beiden Fällen wird also eine Unicode-Zeichenkette mit dem Inhalt **Hallo** erstellt. Beide Aufrufe sind äquivalent.

Die neue Klasse **BoxiCode** erbt von **unicode** und fügt lediglich eine Methode hinzu: das altbekannte **boxify()**.

```
>>> s=BoxiCode("Hallo Welt")
>>> s
u'Hallo Welt'
>>> s.lower()
u'hallo welt'
>>> print s.boxify()
==============
= Hallo Welt =
==============
```

BoxiCode hat offensichtlich sämtliche Methoden von **unicode** geerbt – im Beispiel an der Methode **lower()** zu sehen. Zusätzlich gibt es nun aber eine Methode, um den jeweiligen Text in einer ASCII-Box anzeigen zu lassen.

Auf gleiche Weise könnte man eine Listenklasse erstellen, die zusätzliche Methoden hat. Oder eine Ganzzahlklasse mit zusätzlichen Methoden:

```
class SuperInt(int):
    def iseven(self):
        return self % 2 == 0
```

Die neue Klasse **SuperInt** erbt von **int** und hat die neue Methode **iseven()**. Diese gibt **True** zurück, wenn die jeweilige Zahl ohne Rest durch 2 teilbar ist (**%** ist der Modulo-Operator).

```
1  >>> x = SuperInt(15)
2  >>> y = SuperInt(16)
3  >>> x.iseven()
4  False
5  >>> y.iseven()
6  True
7  >>> z = x+y
8  >>> z.iseven()
9  Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11 AttributeError: 'int' object has no attribute 'iseven'
```

In den Zeilen 1 und 2 werden zwei **SuperInt**-Objekte erstellt. In den folgenden beiden Zeilen wird gezeigt, dass die neue Methode wie erwartet arbeitet: 15 ist ungerade, 16 gerade. In Zeile 7 werden die beiden **SuperInt**-Objekte addiert und das Ergebnis wird an den Namen **z** gebunden. Beim Versuch, auch hier die Methode **iseven()** aufzurufen, kommt es zu einem Fehler.

Offensichtlich handelt es sich bei dem neu erstellten Objekt, das die Summe von **x** und **y** repräsentiert, nicht mehr um ein von **SuperInt** abgeleitetes Objekt. Es handelt sich um ein einfaches **int**-Objekt und das kennt die Methode **iseven()** nicht.

Das hängt mit der Art und Weise zusammen, wie Python Ganzzahl-Objekte addiert. Der Aufruf **3 + 5** wird intern als **3.__add__(5)** interpretiert. Es wird also die Methode **__add__** der links stehenden Ganzzahl aufgerufen. Diese Methode erstellt ein neues int-Objekt. Soll nun stattdessen ein neues **SuperInt**-Objekt erstellt werden, müsste die Methode **__add__** in der **SuperInt**-Klasse neue implementiert werden. Da es aber auch Methoden für Subtraktion, Division, Multiplikationen und viele andere mathematische Operationen gibt, würde dies in viel (fehlerträchtige) Arbeit ausarten.

Dieses Beispiel soll veranschaulichen, dass das Erweitern von Datentypen in Python prinzipiell möglich ist und manchmal durchaus sinnvoll sein kann. Es kann aber auch zu schwer nachvollziehbaren Fehlern führen und erschwert das Lesen des Quelltextes unter Umständen enorm. Methoden wie **boxify()** oder **iseven()** sollten daher immer als unabhängige *Funktionen* umgesetzt werden. Für Anfänger gilt also: Nicht von Basisdatentypen erben, keine „Spezialmethoden“ implementieren!

Die Sache mit den Unterstrichen

Attribute und Methoden in Python kommen bisweilen mit allerlei Unterstrichen daher. Es gibt drei Varianten, die dabei zu unterscheiden sind. Da es dabei häufig zu Missverständnissen und Fehlern kommt, sollen die drei Varianten kurz angeschnitten werden:

Zwei Unterstriche vor und nach dem Methodennamen (`__foo__`)

Hierbei handelt es sich um spezielle Methoden, die auf bestimmte Aufrufe reagieren. So wurde bereits erörtert, dass der erste Aufruf von **`MeineKlasse()`** die **`__init__`**-Methode der entsprechenden Klasse aufruft. Die Methode **`__add__`** wiederum wird aufgerufen, wenn die Klasse links von einem +-Operator steht:

```
x = MeineKlasse()
x + 3
```

wird also als

```
x = MeineKlasse()
x.__add__(3)
```

interpretiert. Analog gibt es Methoden für viele andere mathematische Operatoren. Auch Operatoren für Listen oder Dictionaries gibt es:

```
x = MeineKlasse()
x["test"]
```

ruft intern etwa die Methode **`__getitem__()`** auf:

```
x = MeineKlasse()
x.__getitem__("test")
```

Kurz und bündig erklärt: Von doppelten Unterstrichen umgebene Methodennamen stehen für Spezialmethoden von Klassen. Entsprechend sollten sie auch nur in diesem Zusammenhang genutzt werden.

Durch die Spezialmethoden ist es möglich, eigene Klassen wie Listen, Zeichenketten oder Zahlen agieren zu lassen bzw. auf die entsprechenden Operatoren zu reagieren. Eine Übersicht verschiedener Spezialmethoden findet sich in der Python-Dokumentation [5].

Einfacher Unterstrich vor Attributen (`self._foo`)

Hierbei handelt es sich lediglich um eine Konvention, um ein Attribut als „privat“ zu markieren. Python kennt aber – anders als andere Programmiersprachen – keine richtigen privaten Variablen [6]. Daher ist ein einfacher Unterstrich eher als Hinweis oder als eine Erinnerung zu verstehen: “Dieses Attribut ist privat und kein Teil der offiziellen Schnittstelle.“

Doppelter Unterstrich vor Attributen, höchstens ein Unterstrich danach (`self.__foo_` bzw. `self.__foo`)

```
class Test(object):
    def __init__(self):
        self._privat = "hallo"
        self.__sehrprivat = "welt"

class Test2(Test):
    def __init__(self):
        Test.__init__(self)
        self.__sehrprivat = "noch ein test"

        print self._Test__sehrprivat
        print self._Test2__sehrprivat
```

Listing 4: TestKlassen.py

Auch hiermit werden private Variablen markiert. Im Unterschied zum vorherigen Abschnitt werden diese aber zur Laufzeit umbenannt. Die Umbenennung von Attributen mit zwei Unterstrichen wird in Python „Name Mangling“ genannt [7]. Durch diese Umbenennung soll verhindert werden, dass Klassen, die voneinander erben, versehentlich mit ihren Attributen die Attribute der Elternklasse überschreiben:

```
>>> x = Test2()
welt
noch ein test
>>> x.__dict__
{'_Test2__sehrprivat': 'noch ein ↷
test', '_Test__sehrprivat': 'welt',↷
 '_privat': 'hallo'}
```

Das Spezialattribut **`__dict__`** referenziert ein Dict mit allen Attributen einer Klasse. Hier ist deutlich zu erkennen, was Name Mangling bewirkt: den jeweiligen Attributen wird der Klassenname vorangestellt. So können Eltern- und Kindklasse (**`Test`** und **`Test2`** also) die gleichen Attribut haben, ohne dass das Attribut der Elternklasse überschrieben wird.

Aber auch das Name Mangling ist keine Sicherheitsfunktion. Rein private Attribute gibt es in Python nicht.

Bemerkungen und Ausblick

Die hier besprochenen Beispiele zeigen einige der grundlegenden Mechanismen von Klassen in Python

auf. Python bietet darüber hinaus viele Erweiterungen, die die Klassen noch mächtiger und nützlicher machen – etwa mehrfache Vererbung [8] oder sogenannte „Properties“ [9].

Im nächsten Teil dieser kleinen Einführung soll der erste Grundstein für eine Musikdatenbank gelegt werden. Auch wird die Fehlerbehandlung in Python ein Thema sein.

LINKS

[1] http://de.wikipedia.org/wiki/Objektorientierte_Programmierung

[2] http://neopythonic.blogspot.com/2008/10/why-explicit-self-has-to-stay.html

[3] http://de.wikipedia.org/wiki/Qt

[4] http://de.wikipedia.org/wiki/GTK+

[5] http://docs.python.org/reference/datamodel.html#basic-customization

[6] http://docs.python.org/tutorial/classes.html#tut-private

[7] https://secure.wikimedia.org/wikipedia/en/wiki/Name_mangling#Name_mangling_in_Python

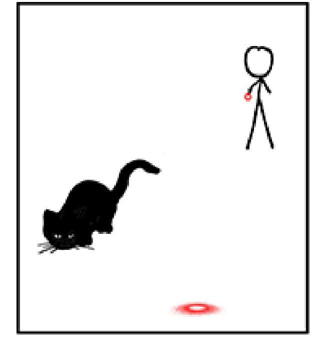
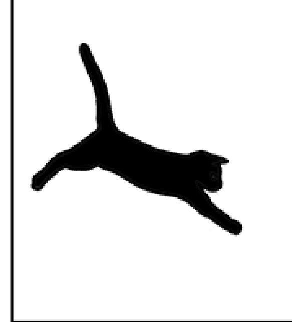
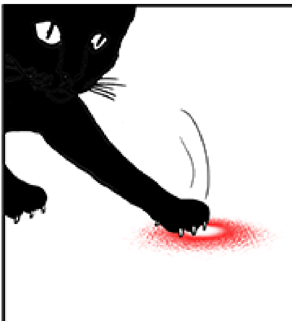
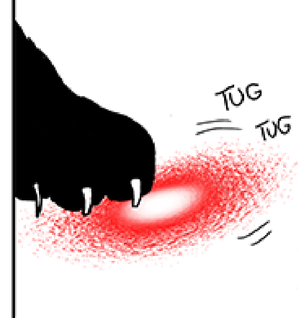
[8] http://www5.in.tum.de/persons/ferstlc/python_kapitel_12_002.htm#mja9ad55f483dad0b289bb6a13fc9dd3fa

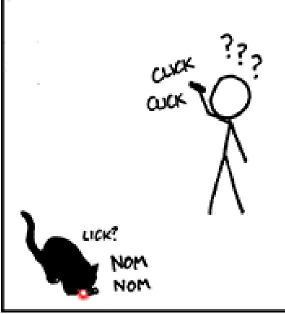
[9] http://realmike.org/blog/2010/07/18/introduction-to-new-style-classes-in-python/

Autoreninformation

Daniel Nögel (Webseite) beschäftigt sich seit drei Jahren mit Python. Ihn überzeugt besonders die intuitive Syntax und die Vielzahl der unterstützten Bibliotheken, die Python auf dem Linux-Desktop zu einem wahren Multitalent machen.

Diesen Artikel kommentieren



„Laser Pointer“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/729

Python-Programmierung: Teil 5 – In medias res

von Daniel Nögel

Im vorherigen Teil dieser Einführung *„Python-Programmierung: Teil 4“* auf Seite 24 wurden Klassen besprochen. Mit diesem Teil soll nun ein Einstieg in die praktische Programmierung in Angriff genommen werden. Zunächst wird dazu aber noch auf die Fehlerbehandlung in Python eingegangen.

Fehlerbehandlung

Fehler werden in Python „Exceptions“ oder „Ausnahmen“ genannt. Sie treten beispielsweise auf, wenn eine Datei nicht geöffnet werden kann, ein Schlüssel einer Liste abgefragt wird, der gar nicht existiert oder auf Variablen zugegriffen wird, die noch nicht bekannt sind. Tritt ein Fehler auf, wird das Skript sofort beendet; in der Konsole erscheint eine Fehlermeldung:

```
>>> names = [u"Peter", u"Isabell"]
>>> names.remove(u"Karla")
Traceback (most recent call last):
  File "<stdin>", line 1, in <
  module>
ValueError: list.remove(x): x not
in list
```

In diesem Beispiel wird zunächst eine Liste mit Namen definiert. Beim Versuch, den Eintrag **Karla** zu entfernen, tritt ein Fehler auf, da dieser Eintrag sich gar nicht in der Liste befindet. Um diese spezielle Situation exakt zu erfassen, wirft Python eine Ausnahme vom Typ **ValueError**. Python kennt bereits in seiner Standardbibliothek viele Typen von Fehlern [1]. Zusätzlich gibt Python noch die Zeilennummer und eine kurze Fehlerbeschreibung aus.

Nun ist es in den meisten Fällen nicht erwünscht, dass das Programm einfach endet und eine kryptische Fehlermeldung ausgibt. Daher ist es möglich, Stellen, an denen Fehler auftreten können, mit folgender Syntax zu umgeben:

```
1 try:
2     names.remove(u"Karla")
3 except ValueError:
4     print "Eintrag nicht in der
      Liste"
```

Bei dem Versuch, die Anweisung in Zeile 2 auszuführen, wird wie oben auch ein **ValueError** ausgelöst. Statt aber das Programm abzubrechen, überprüft der Interpreter, ob einer der **except**-Ausdrücke (es sind mehrere möglich) diesen Fehler abfängt. Das ist in Zeile 3 der Fall; der Programmfluss wird daher an dieser Stelle fortgesetzt. Grob ließe sich **try... except...** also mit *versuche... im Fehlerfall mache...* übersetzen.

Es lassen sich auch mehrere mögliche Fehler in einem except-Block abfangen:

```
1 try:
2     names.remove(u"Karla")
3 except (SyntaxError, ValueError)
  as error:
4     print u"Folgender Fehler ist
      aufgetreten {0}".format(error
      )
```

Hier wird also auf einen möglichen **SyntaxError** ebenso reagiert wie auf einen **ValueError**. Das Konstrukt **except ... as error** führt dazu, dass der aufgetretene Fehler im Rumpf des **except**-Blocks als **error** zur Verfügung steht. Sprich, der aufgetretene Fehler wird an den Namen **error** gebunden. Obwohl Fehler-Objekte keine Zeichenketten sind, können sie in vielen Situationen wie solche verwendet werden:

```
>>> error = ValueError("Hilfe")
>>> str(error)
'Hilfe'
```

In Zeile 4 des zweiten Beispiels oben geschieht im Grunde genommen Ähnliches: Das Fehler-Objekt wird implizit in eine Zeichenkette umgewandelt, sodass auf der Konsole einige Details zum Fehler erscheinen.

else und finally

Wie auch **if**-Blöcke kennen **try... except**-Blöcke das **else**-Statement. Der Rumpf eines **else**-Blockes wird ausgeführt, wenn im **try**-Block *kein* Fehler aufgetreten ist. Der **finally**-Block schließlich kommt immer zur Ausführung, egal ob ein Fehler auftrat oder nicht:

```
l = [u"Peter", u"Isabell"]
try:
```

© freiesMagazin CC-BY-SA 3.0

```
    names.remove(u"Karla")
except ValueError:
    print "Karla war gar nicht in ↷
    der Liste."
else:
    print "Karla wurde aus der ↷
    Liste entfernt."
finally:
    print "Mir doch egal, ob Karla ↷
    in der Liste war oder nicht!"
```

Abhängig davon, ob Karla nun in der Liste vorhanden war oder nicht, erscheint entweder die Meldung:

```
Karla war gar nicht in der Liste.
Mir doch egal, ob Karla in der ↷
Liste war oder nicht!
```

oder

```
Karla wurde aus der Liste entfernt.
Mir doch egal, ob Karla in der ↷
Liste war oder nicht!
```

Ästhetik des Scheiterns

Die Ausnahmebehandlung in Python gilt als sehr mächtig und wird oft gegenüber manuellen Tests bevorzugt. Statt also von Hand zu überprüfen, ob eine Datei vorhanden ist und ob der Benutzer die nötigen Rechte hat, diese Datei zu lesen, würde ein Python-Programmierer es einfach auf einen Versuch ankommen lassen und die Datei öffnen. Tritt ein Fehler auf, wird dieser durch entsprechende **except**-Blöcke abgefangen und dort darauf reagiert.

Auftretende Fehler werden gewissermaßen von „unten nach oben“ durchgereicht:

```
1  def innen():
2      return int("asd")
3
4  def mitte():
5      return innen()
6
7  def aussen():
8      return mitte()
9
10 print aussen()
```

Der Umwandeln einer Zeichenkette wie **asd** in eine Ganzzahl wird zu einem **ValueError** führen. Die Frage lautet nun: An welcher Stelle muss dieser Fehler nun abgefangen werden?

Tatsächlich gibt es theoretisch vier Möglichkeiten: Der Fehler tritt zunächst in der Funktion **innen()** auf. Dort könnte er mit einem **try... except...**-Block abgefangen werden. Wird der Fehler dort nicht behandelt, wird er an die übergeordnete Funktion **mitte()** weitergegeben. Geschieht auch hier keine Fehlerbehandlung, überprüft der Interpreter, ob auf nächsthöherer Ebene (also in **aussen()**) eine Fehlerbehandlung stattfindet. Ist auch dies nicht der Fall, besteht noch die Möglichkeit, den Fehler in Zeile 10 abzufangen. Erst wenn an all diesen Stellen keine Fehlerbehandlung stattgefunden hat, bricht der Interpreter die Abarbeitung des Skripts ab.

Viele Anfänger mögen sich jetzt die Frage stellen: Wieso vermeidet man nicht einfach Fehler, indem man dafür sorgt, dass die möglichen Ausnahmen nicht auftreten können? Folgender Code-Schnipsel soll das verdeutlichen:

```
# ein Dictionary
persons = {u"Peter": "m", u"Karla":↷
 "w"}

# irgend wo spaeter im Code
if u"Peter" in persons:
    gender = persons[u"Peter"]
```

Man könnte an dieser Stelle natürlich auch einen **KeyError** abfangen:

```
try:
    gender = persons[u"Peter"]
except KeyError:
    pass
```

Diese kleine Situation zeigt das Dilemma. Oftmals ist nicht unbedingt klar und direkt ersichtlich, wie man ein solches Problem optimal löst. In Python wird das EAFP-Prinzip (frei übersetzt: „Erst schießen, dann fragen.“) dem LBYL-Paradigma (in etwa: „Schau, bevor du abspringst“) vorgezogen [2]. EAFP würde offensichtlich für die zweite Variante sprechen, außerdem spricht dafür, dass das Abklopfen aller Eventualitäten den Code unter Umständen sehr aufblähen kann. Andererseits handelt es sich bei Exceptions um *Ausnahmen*. Sollte es – wieso auch immer – sehr wahrscheinlich sein, dass der Schlüssel (oben: **Peter**) an dieser Stelle nicht verfügbar ist, so wäre die Lösung mittels Test auf Vorhandensein sinnvoller, da es expliziter die erwartete Situati-

on ausdrückt. Entscheidend für die gute Nutzung von Exceptions ist also die Situation im Code. Über die Vor- und Nachteile von EAFP und LBYL hat Oran Looney einen sehr anschaulichen Artikel verfasst [3]. Weitere allgemeine Hinweise zur Fehlerbehandlung in Python finden sich in der offiziellen Dokumentation [4].

Musikverwaltung

Die bisher vorgestellten Funktionen und Möglichkeiten Pythons sollen nun in einem etwas umfassenderen Beispiel – einer kleinen Musikverwaltung – veranschaulicht werden. In den nächsten beiden Teilen dieser Reihe wird dieses Beispiel dann sukzessive ausgebaut. Den Anfang macht ein kleines Modul, das Verzeichnisse rekursiv nach Musikdateien durchsucht und deren Tags in einer Liste von Dictionaries abbildet.

ID-Tags lesen

Zum Auslesen der ID3-Tags wird die Python-Bibliothek **mutagen** [5] eingesetzt. In Ubuntu ist es als Paket **python-mutagen** in den Paketquellen verfügbar und kann darüber bequem nachinstalliert werden.

Das Modul zum Auslesen der MP3s soll **readmp3s** heißen. Entsprechend wird die Datei readmp3s.py angelegt und mit folgenden Inhalt befüllt:

```
1 #!/usr/bin/env python
2 # coding:utf-8
3
4 import os
5 import sys
6 from mutagen.easyid3 import EasyID3
7 import mutagen.mp3
8 import mutagen.oggvorbis
```

Listing 1: readmp3s-imports.py

Neben Shebang und Zeichenkodierung finden sich hier fünf Importe: Das Modul **os** beinhaltet diverse betriebsystemabhängige Schnittstellen. Das Modul **sys** wird genutzt, um auf die Parameter des späteren Programmes zugreifen zu können. Der Import

```
from mutagen.easyid3 import EasyID3
```

erscheint zunächst kompliziert. Es wird aber lediglich vom Submodul **easyid3** des Paketes **mutagen** die Klasse **EasyID3** in den Namensraum des Skriptes importiert. In Teil 3 dieser Einführung (siehe **freies**Magazin 12/2010 [6]) wurde von derartigen **from**-Importen abgeraten, **mutagen** wurde aber bewusst so geschrieben, dass das gezielte Importieren einzelner Subbibliotheken möglich und sinnvoll ist.

Mit **import mutagen.mp3** wird schließlich noch das Submodul **mp3** in den Namensraum des Skriptes importiert. Der letzte Import verfährt parallel mit dem Submodul **oggvorbis**. Diese beiden Module stellen später Schnittstellen zu den ID3- bzw. Vorbis-Tags bereit.

Als nächstes wird eine Funktion implementiert, welche die Tags der Audiodateien ausliest und zurückgibt. Treten Fehler auf, gibt die Funktion **None** zurück.

```
1  def get_infos(path):
2
3      path = path.decode("utf-8")
4
5      if path.lower().endswith(".mp3"):
6          audio = mutagen.mp3.MP3(path, ID3=EasyID3)
7      else:
8          audio = mutagen.oggvorbis.OggVorbis(path)
9
10     length = audio.info.length
11
12     infos = {"path":path, "length":length}
13     for tag in ["title", "artist", "album"]:
14         content = audio.get(tag, [None])[0]
15         infos[tag] = content
16
17     return infos
```

Listing 2: readmp3s-get_infos.py

Der Kopf der Funktion hält keine Überraschungen bereit: Die Funktion **get_infos** erwartet nur einen Parameter **path**. In Zeile 3 werden die Pfadangaben in Unicode umgewandelt – dies ist schon allein in Hinblick auf die SQLite-Datenbank zu empfehlen. Nun ist es immer möglich, dass einige Dateinamen fehlerhaft kodiert wurden (das berühmte Fragezeichen im Dateinamen). In diesem Fall würde eine **UnicodeDecodeError** auftreten. Dieser Fehler wird in dieser Funktion nicht

behandelt, daher muss er also an anderer Stelle abgefangen werden.

In den Zeilen 6 und 8 wird abhängig von der Dateiendung (besser wäre natürlich eine Überprüfung des MIME-Types [7]) eine Instanz der Klasse **mutagen.mp3.MP3** oder **mutagen.oggvorbis.OggVorbis** erstellt. Es wird jeweils der Pfad zur Audiodatei übergeben. Der Zusatz **ID3=EasyID3** in Zeile 6 sorgt weiterhin dafür, dass eine vereinfachte Schnittstelle auf die ID3-Tags bereitsteht. Die etwas kryptischen ID3-Tags (**TPE1**, **TALB**) können so einfach als **artist** bzw. **album** angesprochen werden. Die erstellten MP3- und OggVorbis-Objekte werden jeweils an den Namen **audio** gebunden und verhalten sich in vielerlei Hinsicht ähnlich. So kann in Zeile 10 bequem die Dauer der Audiodatei ausgelesen werden – unabhängig davon, um welches Format es sich dabei handelt.

In Zeile 12 wird zunächst ein neues Dict mit dem Pfad und der Dauer erstellt und an den Namen **infos** gebunden. Die Schleife in Zeile 13 iteriert über die Liste mit Tags, die ausgelesen werden sollen. Die Objekte **mutagen.mp3.MP3** und **mutagen.oggvorbis.OggVorbis** verhalten sich wie ein Dict, sodass mit **audio["title"]** der Titel der jeweiligen Datei abgefragt werden könnte. Da aber nicht jede Audiodatei zwangsläufig jedes Tag hat, könnten hier Fehler auftreten. An dieser Stelle erspart man sich weitere Fehlerbehandlungen und nutzt die **get**-Methode der Dicts. Allerdings gibt Mutagen nicht einfach Zeichenketten für die gewünschten Tags aus. Es werden immer Listen zurückgegeben, sodass man statt **Künstlername** die Liste **["Künstlername"]** erhält. Daher wird am Ende von Zeile 14 mit **[0]** das erste Listenelement ausgelesen – das ist in den meisten Fällen völlig ausreichend. Hier lauert freilich auch eine kleine Stolperfalle: Die **get**-Methode arbeitet bekanntlich mit einem Standard-Parameter, der zurückgegeben wird, falls das Dict den angefragten Wert nicht enthält. Dieser Wert ist in dem Fall **None**. Da die Rückgabe der **get**-Methode aber als Liste behandelt wird (**[0]**), muss auch **None** zunächst in eine Liste gepackt werden.

Nach dem Durchlaufen der Schleife wird in Zeile 17 das Dict an die aufrufende Funktion übergeben.

Hinweis: In den Zeilen 10 und 14 werden **audio.info.length** und die Rückgabe von **audio.get()** an die Namen **length** bzw. **content** gebunden. Dieser Zwischenschritt dient hier lediglich der Veranschaulichung und der Übersichtlichkeit. Für gewöhnlich würde man diesen Zwischenschritt auslassen:

```
{"path":path, "length":audio.info.length}
...
infos[tag] = audio.get(tag, [None])[0]
```

Verzeichnis rekursiv auslesen

Nun wird noch eine Funktion benötigt, die die MP3s auf der Festplatte findet und an die Funktion **get_infos()** übergibt. Diese Funktion soll **read_recursively()** heißen (siehe nächste Seite).

Auch hier zunächst nichts Neues: Die Funktion kennt einen Parameter **path**. Damit soll später das Verzeichnis übergeben werden, welches rekursiv durchsucht werden soll. In Zeile 3 wird eine Liste erstellt und an den Namen **audio_infos** gebunden. Damit sollen später sämtliche von **get_infos()** erzeugten Dicts mit den Tags aufbewahrt werden. **counter** und **error_counter** sollen lediglich zählen, wie viele Audiodateien bereits verarbeitet wurden und bei wie vielen es zu Fehlern kam.

In Zeile 7 wird eine neue Funktion eingeführt: **os.walk()** durchläuft ein Verzeichnis rekursiv. Für jedes Verzeichnis gibt die Funktion dabei den Pfad des aktuellen Verzeichnisses (**root**), die Verzeichnisse in dem aktuellen Verzeichnis (**dirs**) und die Dateien im jeweiligen Verzeichnis (**files**) an. Bei **os.walk()** [8] handelt es sich um einen sogenannten Generator. Das ist der Grund, warum die Funktion in einer Schleife verwendet werden kann [9].

Da für die Musikdatenbank nur Dateien von Interesse sind, genügt es, in Zeile 8 über die Dateiliste eines jeden Verzeichnisses zu iterieren. In Zeile 9 wird für jede Datei geprüft, ob sie mit **.mp3** oder **.ogg** endet. Da die Dateien theoretisch auch auf **.mP3** oder **.OgG** enden könnten, werden die Dateinamen in Kleinbuchstaben verglichen. Auch hier gilt: Die Abfrage der Dateiendung ist keine sonderlich befriedigende Lösung – für das erste kleine Projekt sollte dieses Vorgehen aber ausreichend sein.

```
1  def read_recursively(path):
2
3  audio_infos = []
4  counter = 0
5  error_counter = 0
6
7  for root, dirs, files in os.walk(path):
8      for fl in files:
9          if fl.lower().endswith(".mp3") or fl.lower().endswith(".ogg"):
10             path = os.path.join(root, fl)
11             try:
12                 infos = get_infos(path)
13             except (UnicodeDecodeError, mutagen.mp3.HeaderNotFoundError, ↷
               mutagen.id3.error) as inst:
14                 print u"\n\nSKIPPING reading this file:"
15                 print path
16                 print u"ERROR: {0}\n".format(inst)
17                 error_counter +=1
18             else:
19                 audio_infos.append(infos)
20                 counter +=1
21         print "\rScanning: {0} Files ok, {1} Files broken".format(counter↷
           , error_counter),
22 print
23 return audio_infos
```

Listing 3: readmp3s-read_recursively.py

Da in der Liste **files** nur Dateinamen vorhanden sind, wird in Zeile 10 der komplette Pfad erzeugt. Die Funktion **os.path.join()** kümmert sich dabei darum, dass die korrekten Schrägstriche verwendet werden (Linux und Windows unterscheiden sich in dieser Hinsicht). Durch **os.path.join** wird weiterhin sicher gestellt, dass es nicht versehentlich zu doppelten Schrägstrichen im Pfad kommt. In Zeile 12 schließlich wird der so erstellte Pfad an die Funktion **get_infos()** übergeben und das Ergebnis an den Namen **info** gebunden.

In Zeile 11 wird aber zunächst die Fehlerbehandlung eingeleitet. Die in der Funktion **get_infos()** nicht behandelten Fehler sollen also an dieser Stelle abgefangen werden. In Zeile 13 werden dabei drei Fehler erwartet: Der bereits erwähnte **UnicodeDecodeError** findet sich ebenso wie die Fehler **mutagen.mp3.HeaderNotFoundError** und **mutagen.id3.error**. Während der **UnicodeDecodeError** auftritt, wenn eine Zeichenkette nicht in Unicode umgewandelt werden konnte und damit für die später verwendete SQLite-Datenbank ungeeignet ist, tritt der **HeaderNotFoundError** auf, wenn Mutagen in einer MP3-Datei keinen MP3-Header finden konnte. **mutagen.id3.error** schließlich fängt viele weitere Mutagen-Fehler ab. Dies erspart dem Entwickler die Arbeit, jeden einzelnen möglichen Fehler im Vorfeld zu definieren. Ihm genügt es (wie in diesem Beispiel) zu wissen, dass Mutagen – warum auch immer – eine bestimmte Datei nicht verarbeiten konnte.

In den Zeilen 14-17 findet die Fehlerbehandlung statt. Der Fehler wird in der Konsole ausgegeben und der Fehlerzähler wird um 1 erhöht. Nur wenn kein Fehler auftritt (Zeile 18) wird in Zeile 19 die Rückgabe von **get_infos()** an die Liste **audio_infos** angehängt. In diesem Fall wird **counter** um 1 erhöht.

Eine letzte Besonderheit findet sich in Zeile 21. Die Zeichenkette dort beginnt mit **\r**. Diese Escape-Sequenz steht für den sogenannten Wagenrücklauf. Dadurch wird der Cursor im Terminal wieder auf den Anfang der Zeile gesetzt. Nachfolgende Ausgaben mit **print** würden die vorherigen Ausgaben überschreiben. Da **print** eine Zeile im Normalfall mit einem Zeilenumbruch abschließt, wird dies hier mit dem Komma am Ende der Zeile unterbunden.

Der Wagenrücklauf und das Unterbinden des Zeilenumbruches sorgen dafür, dass die Konsole nicht mit Textausgaben überflutet wird: Die Anzahl der verarbeiteten Dateien wird immer in die selbe Zeile geschrieben.

Hinweis: In Python >= 3 müsste Zeile 17 wie folgt aussehen:

```
print ("\rScanning: {0} MP3s ok, ↷
{1} MP3s broken".format(counter, ↷
error_counter), end="")
```

Kleiner Test

Obwohl die Musikdatenbank noch in weiter Ferne ist, kann zumindest das Modul **readmp3s** schon einmal getestet werden. Dazu wird am Ende des Skripts noch folgende **if**-Block ergänzt:

```
1 if __name__ == "__main__":
2     try:
3         path = sys.argv[1]
4     except IndexError:
5         print "Usage:"
6         print "readmp3s.py ↷
          DIRECTORY"
7     else:
8         mp3s = read_recursively(↷
          path)
```

Listing 4: readmp3s-main.py

Die erste Zeile überprüft dabei, ob die Datei als Modul oder als eigenständiges Skript geladen wurde. Nur im letzteren Fall kommen die folgenden Zeilen zur Ausführung. In Zeile 3 wird der zweite Parameter des Skriptes aus der Liste **sys.argv** gelesen. Der erste Parameter ist immer der Dateiname (hier also **readmp3s.py**). Falls der Benutzer keine Parameter übergeben und die Liste nur einen Eintrag hat, wird es in Zeile 3 zu einem **IndexError** kommen, der aber in Zeile 4 abgefangen wird. Im Fehlerfall wird dann ein kurzer Hinweis auf den erforderlichen Parameter gegeben. In Zeile 8 wird die Funktion **read_recursively()** aufgerufen. Wie bereits erwähnt kommt dieser **else**-Block nur zur Ausführung, wenn zuvor kein Fehler auftrat.

Das Skript wird nun das angegebene Verzeichnis durchsuchen, die ID-Tags auslesen und über die Anzahl lesbarer und nicht-lesbarer MP3s informieren. Keine Sorge: Das Skript ist bisweilen etwas empfindlich, was die Zeichenkodierung und die MP3-Header betrifft und nicht alle als defekt gemeldeten MP3s sind tatsächlich unbrauchbar.

Im nächsten Teil dieser Reihe wird die Musikdatenbank um eine SQLite-Datenbank ergänzt.

LINKS

[1] http://docs.python.org/library/exceptions.html

[2] http://docs.python.org/glossary.html

[3] http://oranlooney.com/lbyl-vs-eafp/

[4] http://docs.python.org/tutorial/errors.html

[5] http://code.google.com/p/mutagen/

[6] http://www.freiesmagazin.de/freiesMagazin-2010-12

[7] http://de.wikipedia.org/wiki/MIME-Type

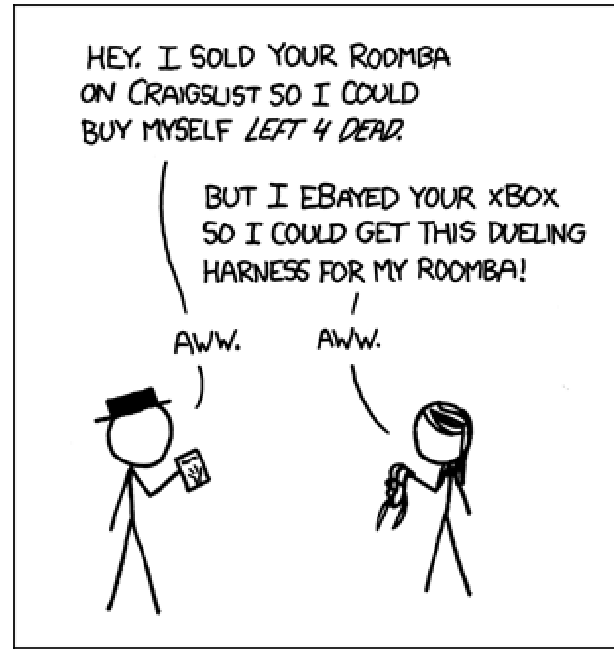
[8] http://docs.python.org/library/os.html#os.walk

[9] http://de.wikibooks.org/wiki/Python-Programmierung:_Funktionen#Generatoren_und_yield

Autoreninformation

Daniel Nögel (Webseite) beschäftigt sich seit drei Jahren mit Python. Ihn überzeugt besonders die intuitive Syntax und die Vielzahl der unterstützten Bibliotheken, die Python auf dem Linux-Desktop zu einem wahren Multitalent machen.

Diesen Artikel kommentieren



„Theft of the Magi“ © by Randall Munroe (CC-BY-NC-2.5), http://xkcd.com/506

Python-Programmierung: Teil 6 – Datenbank für die Musikverwaltung

von Daniel Nögel

Im fünften Teil dieser Reihe *„Python-Programmierung: Teil 5“* auf Seite 24 wurde die Fehlerbehandlung in Python besprochen. Darüber hinaus wurde der Grundstein für eine kleine Musikverwaltung gelegt. Bisher wurden Funktionen implementiert, die ein gegebenes Verzeichnis rekursiv nach Musikdateien durchsuchen und dann deren ID-Tags auslesen. In diesem Teil soll die Musikverwaltung um eine Datenbank erweitert werden, die bequem durchsucht werden kann.

Die Datenbank

Als Datenbank wird in diesem Fall SQLite eingesetzt [1]. Bei SQLite handelt es sich um ein SQL-Datenbanksystem, das ohne Server-Software auskommt und daher auch neben der SQLite-Programmbibliothek selbst keine weitere Software auf dem Zielrechner erfordert. SQLite unterstützt viele SQL-Sprachbefehle, ist aber in einigen Bereichen simpler gehalten als beispielsweise MySQL.

Für die Verwendung in Python muss neben der SQLite-Programmbibliothek (`sqlite3`) noch die entsprechende Python-Schnittstelle installiert werden. Diese findet sich in Ubuntu beispielsweise im Paket **python-sqlite2**.

Das im letzten Teil erstellte Python-Skript soll nun um eine Datenbankanbindung erweitert werden. Wer bei den Ergänzungen den Überblick verliert, kann das fertige Skript musicdb.py auch direkt herunterladen und dort die Änderungen nachvollziehen.

Die neuen Importe

Zunächst müssen einige Importe ergänzt werden:

```
import sqlite3

import subprocess
from optparse import OptionParser

import codecs
from textwrap import dedent
from random import shuffle
```

Hier werden eine ganze Reihe neuer Module und Funktionen eingeführt. **`sqlite3`** stellt schlicht die Schnittstelle zum SQLite-Datenbanksystem bereit [2].

Bei **`subprocess`** handelt es sich um ein Modul, mit dem aus Python heraus andere Prozesse gestartet werden können. Darüber hinaus können mit dem Modul Signale an diese Prozesse gesendet werden oder STDOUT bzw. STDERR ausgelesen werden. Auch das Schreiben nach STDIN ist mit **`subprocess`** möglich [3]. In diesem Skript wird es später lediglich benötigt, um das Medienwiedergabeprogramm Totem zu starten und einige MP3s damit abzuspielen.

Das Modul **`optparse`** hält verschieden Werkzeuge bereit, um die Optionen und Argumente von Skripten auszuwerten. Auch lassen sich damit recht einfach Übersichten der möglichen Optionen erstellen [4].

Neu ist das Modul **`codecs`** [5]. Mit dessen Funktion **`open()`** kann später bequem eine Datei mit einer beliebigen Zeichenkodierung geschrieben werden.

Die Funktion **`dedent`** aus dem Modul **`textwrap`** wird später im Skript dazu genutzt, eine mehrzeilige, eingerückte Zeichenkette ohne Einrückungen ausgeben zu können [6].

Einzig das Modul **`random`** sollte aus den vorherigen Teilen dieser Reihe bereits bekannt sein. Es stellt verschiedene Zufallsfunktionen zur Verfügung. Die Funktion **`shuffle`** durchmischt eine gegebene Liste schlicht, sodass sich damit beispielsweise eine Wiedergabeliste durchmischen ließe [7].

Die Datenbankanbindung

Als nächstes soll nun die Datenbankanbindung des Skripts erstellt werden. Diese besteht aus zwei Klassen: einer Cursor-Klasse und einer Datenbank-Klasse. Beide Klassen verfügen aber über eine Neuerung, die hier kurz erläutert werden soll.

Das Schlüsselwort `with`

Das Schlüsselwort **`with`** ist ein Sprachelement, das es seit Python 2.5 gibt [8]. Es wird besonders häufig beim Arbeiten mit Dateien eingesetzt,

weshalb es auch an diesem Beispiel erörtert werden soll. Für gewöhnlich wird beim Umgang mit Dateien ein Konstrukt wie das folgende benötigt:

```
1 handler = open("datei.txt", "r")
2 try:
3     print handler.read()
4 finally:
5     handler.close()
```

In Zeile 1 wird dabei eine Datei geöffnet und das daraus resultierende Datei-Objekt an den Namen **handler** gebunden. In Zeile 3 wird der Inhalt der Datei ausgegeben. Der **try...finally**-Block stellt sicher, dass das Datei-Objekt anschließend in jedem Fall geschlossen wird – auch wenn beim Auslesen der Datei in Zeile 3 eventuell Fehler aufgetreten sind. Die Konstruktion „Vorbereiten, Bearbeiten, Aufräumen“ ist allerdings auch in anderen Zusammenhängen so häufig anzutreffen, dass ab Python 2.5 mit **with** eine deutliche Vereinfachung für derartige Fälle eingeführt wurde:

```
1 with open("datei.txt", "r") as ↷
  handler:
2     print handler.read()
```

Auch hier wird zunächst die Datei **open.txt** zum Lesen geöffnet und das daraus resultierende Datei-Objekt an den Namen **handler** gebunden. Allerdings erfolgt die Zuweisung des Namens hier durch das Schlüsselwort **as**. In Zeile 2 wird – wie gehabt – der Inhalt der Datei ausgegeben. Damit sind die beiden Schritte „Vorbereiten“ und „Bearbeiten“ abgehandelt. Das Aufräumen erfolgt mit dem Verlassen der Kontrollstruktur, also am Ende des with-Blocks. Es muss nicht explizit durchgeführt werden. Wie funktioniert das?

Seit Python 2.5 gibt es zwei neue spezielle Methoden, die Objekte implementieren können: **__enter__** und **__exit__**. Die Methode **__enter__** ist für die Vorbereitung zuständig und wird implizit beim Betreten des **with**-Blocks aufgerufen. Der Rückgabewert dieser Methode wird dann an den mit **as** angegebenen Namen gebunden – oben also an den Namen **handler**. Die Methode **__exit__** wird beim Verlassen des **with**-Blocks aufgerufen – unabhängig davon, ob ein Fehler aufgetreten ist oder nicht.

Diese Konstruktion wird im Folgenden auch für die Datenbank verwendet. Da die SQLite-Schnittstelle für Python von Haus aus noch keinen Gebrauch von diesem neuen Sprachmittel macht, wird es hier selbst implementiert.

Der Cursor

SQLite kennt sogenannte **Cursor**, mit denen Datensätze in Datenbanken ausgelesen und bearbeitet werden [9]. In der Regel folgt auch das Arbeiten mit Cursorn dem Schema „Vorbereiten, Bearbeiten, Aufräumen“, weshalb sich hier die Verwendung der **with**-Anweisung empfiehlt.

Das bisherige Skript aus dem letzten Teil wird nun um diese Klasse ergänzt:

```
class Cursor(object):
    def __init__(self, connection):
        self.connection = connection

    def __enter__(self):
        self.cursor = self.connection.cursor()
        return self.cursor
```

```
    def __exit__(self, type, value, traceback):
        self.cursor.close()
```

Es handelt sich hierbei letztlich nur um einen sogenannten „Wrapper“, der die Verwendung von SQLite-Cursorn mit **with** ermöglicht. Später könnte ein Aufruf wie folgt aussehen:

```
1 with Cursor(connection) as cursor:
2     cursor.machetwas()
```

connection ist dabei eine Referenz auf ein SQLite-Connection-Objekt, das eine offene Datenbankverbindung verwaltet. In Zeile 1 wird nun zunächst eine Instanz der Cursor-Klasse erstellt und **connection** als Parameter übergeben. In der Methode **__init__** wird die Referenz auf das Connection-Objekt an das Attribut **self.connection** gebunden. Erst danach wird durch den **with**-Block die Methode **__enter__** des Cursor-Objektes aufgerufen. Hier wird nun das SQLite-Cursor-Objekt durch den Aufruf **self.connection.cursor()** erstellt. Das Objekt wird an das Attribut **self.cursor** gebunden und mittels des Schlüsselwortes **return** zurückgegeben. Da der Rückgabewert der Methode **__enter__** in **with**-Blöcken an den hinter **as** angegebenen Namen gebunden wird, steht nun eine Referenz auf das SQLite-Cursor-Objekt unter dem Namen **cursor** zur Verfügung. In Zeile 2 des Beispiels kann so bequem auf das SQLite-Cursor-Objekt zugegriffen werden. Mit dem Verlassen des **with**-Blocks würde schließlich die Methode **__exit__** aufgerufen werden, in der das Cursor-Objekt korrekt geschlossen wird.

Noch einmal zur Erinnerung: Die Cursor-Klasse fungiert hier lediglich als Wrapper. Ohne sie sähe jeder Zugriff auf SQLite-Cursor wie folgt aus:

```
cursor = connection.cursor()
try:
    cursor.machetwas()
finally:
    cursor.close()
```

Die Datenbankklasse

Als nächstes soll nun die eigentliche Datenbankanbindung realisiert werden. Aus Gründen der Übersichtlichkeit wird der Code nur verlinkt: musicdb-databaseconnector.py.

Auch die Klasse `DatabaseConnector` kennt die Methode `__enter__` und `__exit__` und ist damit auf die Verwendung mit `with` ausgelegt. In der Methode `__enter__` werden dabei alle relevanten Vorbereitungen für die Erstellung der Datenbankverbindung getroffen. Zunächst wird der Ordner des Skriptes per

```
os.path.dirname(os.path.abspath((↷
__file__)))
```

ermittelt und an den Namen `path` gebunden [10]. In dem selben Ordner soll nun auch die Datenbank `music.db` abgelegt werden. Existiert diese Datenbank noch nicht, wird sie mit der Methode `create_database()` erstellt. Anschließend wird durch den Aufruf

```
self.connection = sqlite3.connect(↷
self.db_path)
```

eine Verbindung zu dieser Datenbank erstellt und das Connection-Objekt an das Attribut `self.connection` gebunden. Mit der Zeile

```
self.connection.row_factory = ↷
sqlite3.Row
```

wird SQLite angewiesen, die Ergebnisse in Form von Row-Objekten [11] darzustellen. Dadurch lassen sich die Ergebnisse von Datenbankabfragen später sehr leicht über Schlüsselworte ansprechen.

Neben der Methode `__enter__`, die die Datenbankverbindung schlicht wieder schließt, kennt die Klasse `DatabaseConnector` weiterhin noch die Methoden `search`, `get_all_songs`, `insert_songs` und `count`. Hier werden jeweils verschiedene Datenbankzugriffe getätigt; es kommt jeweils die oben vorgestellte Cursor-Klasse zum Einsatz. Die Bedeutung der einzelnen SQL-Abfragen sollte weitestgehend bekannt oder aus dem Kontext ersichtlich sein. Im Rahmen dieser Einführung ist eine tiefergehende Auseinandersetzung mit der SQL-Syntax leider nicht möglich. Eine Übersicht der SQL-Befehle findet sich allerdings auf der Homepage des SQLite-Projektes [12]. Dennoch sollen einige Anmerkungen zu kleinen Besonderheiten gemacht werden.

In der Methode `search` werden dem Suchbegriff Prozentzeichen voran- und nachgestellt, außerdem werden Leerzeichen durch Prozentzeichen ersetzt. Hierbei handelt es sich lediglich um die SQL-typischen Wildcard-Zeichen. Am Ende der Methode `insert_songs` findet sich außerdem dieser Aufruf:

```
cursor.executemany(sql, songs)
```

Damit wird eine Liste von mehreren Liedern in die Datenbank eingetragen. **songs** ist in diesem Fall eine Liste von Tupeln nach folgendem Schema:

```
[
 ("Kuenstler", "Titel", "Album", "↷
 Laenge", "Pfad"),
 ("Kuenstler", "Titel", "Album", "↷
 Laenge", "Pfad"),
 ...
]
```

So können sehr effizient große Mengen von Titelinformationen in die Datenbank übernommen werden. Durch das anschließende

```
connection.commit()
```

werden die Änderungen übernommen. Dieser Aufruf muss immer dann erfolgen, wenn schreibend auf die Datenbank zugegriffen wurde. Ansonsten wären die Änderungen bei der nächsten Abfrage noch nicht sichtbar.

Wiedergabelisten erstellen

Die Musikverwaltung soll in der Lage sein, Lieder, die auf eine Suche passen, in eine Wiedergabeliste zu speichern. Dazu wird das Skript um folgende drei Funktionen erweitert:

```
def generate_extended_playlist(songs):
    playlist = [u"#EXTM3U\n"]
    for id, artist, title, album, length, path in songs:
        playlist.append("#EXTINF:{0},{1} - {2}\n".format(int(length), ↷
        artist,
                        title))
        playlist.append("{0}\n".format(path))
    return u"".join(playlist)

def generate_simple_playlist(songs):
    return u"\n".join(hit["path"] for hit in songs)

def dump(playlist, path, encoding="utf-8"):
    with codecs.open(path, "w", encoding=encoding) as fh:
        fh.write(playlist)
```

Die ersten beiden Funktionen erwarten jeweils eine Liste **songs**. Diese Songs sind im Fall dieses Skripts eigentlich Row-Objekte, wie sie von der Methode **search** des DatabaseConnectors zurückgegeben werden. Diese Row-Objekte verhalten sich dahingehend wie Listen und Dicts, als dass sie den Zugriff über Schlüsselworte ebenso ermöglichen, wie über Listen-Indizes.

Die Funktion **generate_simple_playlist** erstellt nun schlicht eine Zeichenkette mit Pfadangaben, getrennt durch einen Zeilenumbruch (**\n**). Die Zeile mutet zunächst recht kompliziert an:

```
return u"\n".join(hit["path"] for ↷
hit in songs)
```

Dies ist aber nur eine kompaktere und effizientere Variante für folgenden Code (siehe Abschnitt „Kleine Aufgabe“ unten):

```
paths = []
for hit in songs:
    paths.append(hit["song"])
return u"\n".join(paths)
```

Die so erstellte Zeichenkette mit Pfandangaben kann als einfache m3u-Liste gespeichert werden. Sollen außerdem noch Meta-Informationen in der Wiedergabeliste gespeichert werden, muss eine erweiterte m3u-Liste erstellt werden. Dies geschieht durch die Funktion **generate_extended_playlist**. Auch hier wird eine Zeichenkette erstellt, die später als Wiedergabeliste gespeichert werden kann. Allerdings wird dabei zunächst der Umweg über eine Liste gegangen: Jeder Eintrag in der Liste repräsentiert später eine Zeile. Mit

```
playlist = [u"#EXTM3U\n"]
```

wird die Liste **playlist** direkt initial befüllt, sodass die spätere Wiedergabeliste in der ersten Zeile die Information enthält, dass es sich um eine erweiterte Wiedergabeliste handelt. In der folgenden **for**-Schleife werden die einzelnen Lieder durchlaufen. Für jedes Lied wird dabei ein Listen-Eintrag mit Meta-Informationen (**#EXTINF**) und ein Listeneintrag mit der dazugehörigen Pfadangabe erstellt. Erst in der letzten Zeile der Funktion wird aus der Liste **playlist** mit Hilfe der Methode **join** eine Zeichenkette, die direkt zurückgegeben wird.

Die dritte Funktion **dump** schreibt schließlich eine gegebene Zeichenkette (in diesem Fall die Wiedergabelisten) in eine Datei. Statt der Funktion **open** kommt dabei allerdings die gleichnamige Funktion aus dem Modul **codecs** zum Einsatz. Diese Funktion hat den Vorteil, dass die gewünschte Ziel-Kodierung direkt wählbar ist (hier UTF-8).

Startbedingungen

In einem letzten Schritt wird nun der Block

```
if __name__ == "__main__":
    ...
```

komplett durch musicdb-main.py ersetzt.

Darin wird zunächst eine Instanz des OptionParser erzeugt und an den Namen **parser** gebunden. In den folgenden Zeilen werden die verschiedenen Optionen definiert, die das Skript später kennt. Die Methode **add_option** fügt jeweils eine weitere Option hinzu und definiert die

dazugehörigen Schalter (beispielsweise **-s** oder **--scan**), das Schlüsselwort, über das die Option später ausgelesen wird (**dest**) und einen kurzen Hilfetext (**help**). Es gibt eine Reihe weiterer Möglichkeiten, die einzelnen Optionen genauer zu spezifizieren (etwa **metavar** oder **type**, vgl. Dokumentation [13]).

In der Zeile

```
options, args = parser.parse_args()
```

dieses Codeblocks wird der OptionParser mit **parse_args()** angewiesen, die Parameter, mit denen das Skript gestartet wurde, auszuwerten. Im Ergebnis werden die Optionen an den Namen **options** gebunden, die Argumente an den Namen **args**.

Ab der Zeile

```
if not (options.search or options.↷
shuffle) ...
```

beginnt die Auswertung der Parameter. Durch diese und die danach folgenden drei Zeilen wird eine Fehlermeldung ausgegeben, wenn der Nutzer die Schalter **-t** oder **-p** übergeben hat, ohne mit **--shuffle** oder **--find** eine Auswahl von Musikstücken zu treffen. **parser.error()** gibt die übergebene Zeichenkette aus und beendet das Skript.

In der Zeile

```
with DatabaseConnector() as ↷
database:
```

kommt die oben implementierte Klasse **DatabaseConnector** zum Einsatz; sie wird im **with**-Block an den Namen **database** gebunden. In der nächsten Zeile wird geprüft, ob mit den Schaltern **-s** oder **--scan** ein Verzeichnis übergeben wurde. Ist dies der Fall, enthält **options.add** eine Pfadangabe als Zeichenkette und der Ausdruck **options.add** ist wahr. In diesem Fall wird das angegebene Verzeichnis mit der im letzten Teil umgesetzten Funktion **read_recursively()** ausgelesen. Die so gefundenen Titelinformationen werden mit der Methode **insert_song()** in die Datenbank geschrieben. Anschließend wird noch die Gesamtzahl der Titel in der Datenbank auf der Konsole ausgegeben.

Ab der Zeile

```
elif options.search or options.↷
shuffle:
```

werden die Optionen **--shuffle** und **-f** bzw. **--find** behandelt. Im Fall einer Suche wird die Zeichenkette zunächst dekodiert und an den Namen **searchterm** gebunden. Die so erstellte Unicode-Zeichenkette kann dann an die Suchfunktion der Datenbank übergeben werden. Das Ergebnis wird an den Namen **songs** gebunden. Im Alternativfall ohne Suche wird eine Zufallsfunktion umgesetzt. Dazu werden zunächst mit **get_all_songs()** alle Lieder aus der Datenbank ausgelesen und zufällig angeordnet. Durch

```
songs = songs[:options.shuffle]
```

wird dann ein Ausschnitt dieser Gesamtliste an den Namen **songs** gebunden. Die Größe des Ausschnitts hat der Benutzer zusammen mit der Option **--shuffle** übergeben.

Dieses Vorgehen ist natürlich nicht besonders effizient: Wer 40.000 Lieder in seiner Datenbank hat, möchte sicher nicht, dass alle diese Lieder zunächst ausgelesen werden, nur um 100 zufällige Lieder davon auszuwählen. Sehr viel eleganter wäre hier eine Lösung via SQL:

```
SELECT * FROM mp3s ORDER BY RANDOM↷
() LIMIT 20
```

Damit werden direkt 20 zufällige Lieder aus der Datenbank ausgelesen. Die im Skript umgesetzte Variante sollte in der Praxis also eher nicht eingesetzt werden. Sie besticht aber in diesem Fall dadurch, dass sie durch die Verwendung des **random**-Modules und die Nutzung von Slices Techniken einsetzt, die in den vorherigen Teilen bereits diskutiert wurden.

In den beiden **if**-Blöcken von **options.search** wurden die ausgewählten Lieder jeweils an den Namen **songs** gebunden. Für das weitere Vorgehen ist es nicht von Bedeutung, wie die Auswahl der Lieder zustande kam. Mit **if songs:** wird lediglich geprüft, ob überhaupt Lieder ausgewählt wurden (eine leere Liste wird als **falsch** ausgewertet). Nachfolgend wird eine Wiedergabeliste erstellt und gespeichert, falls die entsprechend Option **options.playlist** gesetzt wurde. Dabei wird der vom Nutzer angegebene Dateiname um

`.m3u` erweitert, falls er nicht darauf endet. Am Ende kommt das **`subprocess`**-Modul zum Einsatz, wenn die Option **`-t`** bzw. **`--totem`** gesetzt wurde. Die Funktion **`Popen`**, die Totem ausführen soll, erwartet die Parameter in Form einer Liste. Diese wird initial mit **`totem`** und **`--enqueue`** befüllt und danach um die Pfadangaben der ausgewählten Musikstücke erweitert. So entsteht eine Liste nach dem Schema:

```
["totem", "--enqueue", "song1.mp3",↷
 "song2.mp3"]
```

Die entsprechende Befehlszeile in der Shell sähe wie folgt aus:

```
$ totem --enqueue song1.mp3 song2.mp3
```

Totem wird also mit der Option **`--enqueue`** gestartet, die alle folgenden Musikstücke an die Wiedergabeliste anhängt.

Schließlich deckt dieser Codeblock noch zwei weitere Eventualitäten ab: Der vorletzte **`else`**-Block ist von Bedeutung, wenn die Liste **`songs`** leer ist, also beispielsweise die Suche keine Treffer ergab. Das letzte **`else`** gibt einen Nutzungshinweis auf der Konsole aus, wenn weder die Optionen **`add`**, **`search`** noch **`shuffle`** gesetzt wurden.

Kleine Aufgabe

Mehrfach wurden kleine Fragestellungen oder Aufgaben erbeten, die die Leser bis zum Erscheinen des nächsten Teils zu lösen hätten. Im Abschnitt „Startbedingungen" oben wird auf eine mögliche Alternative zur derzeitigen Shuffle-Funktion hingewiesen. Interessierte Leser könnten versuchen, die dort vorgeschlagenen Änderungen zu implementieren, indem sie die Klasse **`DatabaseConnector`** um eine **`shuffle()`**-Methode erweitern und das Skript so anpassen, dass diese Methode statt der jetzigen Variante zur Auswahl der Zufallstitel eingesetzt wird.

Wer sich darüber hinaus noch vertiefend mit dem Skript beschäftigen möchte, kann sich die Funktion **`generate_simple_playlist`** einmal näher ansehen. Der Einzeiler im Funktionsrumpf hat es bei näherer Betrachtung in sich, dort kommt ein sogenannter „Generator-Ausdruck" zum Einsatz [14].

Schlusswort

Mit diesem sechsten Teil hat die einführende Vorstellung von Python eine kleine Zäsur erreicht. In Form der Musikdatenbank wurde mit Hilfe der vorgestellten Technologien und Werkzeuge erstmals ein (etwas) größeres Projekt in Angriff genommen, das zum Experimentieren und Erweitern einlädt.

Die Python-Reihe soll auch weiterhin fortgesetzt werden, allerdings wird sich der Abstand der einzelnen Teile etwas vergrößern und ab sofort voraussichtlich zweimonatlich erscheinen.

LINKS

[1] http://www.sqlite.org/
[2] http://docs.python.org/library/sqlite3.html
[3] http://docs.python.org/library/subprocess.html
[4] http://docs.python.org/library/optparse.html
[5] http://docs.python.org/library/codecs.html
[6] http://docs.python.org/library/textwrap.html#textwrap.dedent
[7] http://docs.python.org/library/random.html#random.shuffle
[8] http://effbot.org/zone/python-with-statement.htm
[9] http://docs.python.org/library/sqlite3.html#sqlite3.Cursor
[10] http://docs.python.org/library/os.path.html#module-os.path
[11] http://docs.python.org/library/sqlite3.html#sqlite3.Row
[12] http://www.sqlite.org/lang.html
[13] http://docs.python.org/library/optparse.html
[14] http://www.python.org/dev/peps/pep-0289/

Autoreninformation

Daniel Nögel (Webseite) beschäftigt sich seit drei Jahren mit Python. Ihn überzeugt besonders die intuitive Syntax und die Vielzahl der unterstützten Bibliotheken, die Python auf dem Linux-Desktop zu einem wahren Multitalent machen.

Diesen Artikel kommentieren

Impressum

freiesMagazin erscheint als PDF und HTML einmal monatlich.

ISSN 1867-7991

Erscheinungsdatum: 17. April 2011

Kontakt

E-Mail	redaktion@freiesMagazin.de
Postanschrift	freiesMagazin c/o Dominik Wagenführ Beethovenstr. 9/1 71277 Rutesheim
Webpräsenz	http://www.freiesmagazin.de/

Redaktion

Frank Brungräber
Thorsten Schmidt
Dominik Wagenführ (Verantwortlicher Redakteur)

Satz und Layout

Ralf Damaschke
Yannic Haupenthal
Nico Maikowski
Matthias Sitte

Korrektur

Daniel Braun
Stefan Fangmeier
Mathias Menzer
Karsten Schuldt
Stephan Walter

Veranstaltungen

Ronny Fischer

Logo-Design

Arne Weinberg (GNU FDL)

Dieses Magazin wurde mit LaTeX erstellt. Mit vollem Namen gekennzeichnete Beiträge geben nicht notwendigerweise die Meinung der Redaktion wieder. Wenn Sie freiesMagazin ausdrucken möchten, dann denken Sie bitte an die Umwelt und drucken Sie nur im Notfall. Die Bäume werden es Ihnen danken. ;-)

Soweit nicht anders angegeben, stehen alle Artikel, Beiträge und Bilder in freiesMagazin unter der Creative-Commons-Lizenz CC-BY-SA 3.0 Unported. Das Copyright liegt beim jeweiligen Autor. freiesMagazin unterliegt als Gesamtwerk ebenso der Creative-Commons-Lizenz CC-BY-SA 3.0 Unported mit Ausnahme der Inhalte, die unter einer anderen Lizenz hierin veröffentlicht werden. Das Copyright liegt bei Dominik Wagenführ. Es wird erlaubt, das Werk/die Werke unter den Bestimmungen der Creative-Commons-Lizenz zu kopieren, zu verteilen und/oder zu modifizieren. Das freiesMagazin-Logo wurde von Arne Weinberg erstellt und unterliegt der GFDL. Die xkcd-Comics stehen separat unter der Creative-Commons-Lizenz CC-BY-NC 2.5 Generic. Das Copyright liegt bei Randall Munroe.