Python unter Linux: ALLES

Python unter Linux
Wikibooks

Vorwort

Herzlich Willkommen zu **Python unter Linux**. Dieses Buch möchte Ihnen eine Hilfe bei der Benutzung von Python unter unix-ähnlichen Betriebssystemen sein. Wir legen Wert auf vollständige Skripte, die Sie per Copy&Paste als Grundlage eigener Übungen verwenden dürfen. Alle Skripte wurden unter verschiedenen Linuxdistributionen geprüft, eine Haftung für eventuelle Schäden wird jedoch ausgeschlossen. In diesem Buch wird davon ausgegangen, dass Python in der Version ab 2.7 installiert wurde. Für die Installation selbst wird auf die zahlreichen Referenzen und HOWTOs verwiesen.

Dieses Buch richtet sich an Einsteiger in die Programmierung von Python. Grundsätzliche Computerkenntnisse sollten vorhanden sein. Sie sollten wissen, was Dateien sind, wie Sie eine Textdatei erstellen und wie Sie eine Shell aufrufen, um die Programmbeispiele zu testen. Ebenfalls sollten Sie mit Dateirechten vertraut sein. Wenn Sie schon Programmiererfahrungen haben, sind Sie ein optimaler Kandidat für dieses Buch. Wir erklären nicht jedes kleine Detail. Manche Dinge bleiben offen. Andere Dinge mag der Autor schlicht als zu trivial – oder zu schwierig – angesehen haben, um es zu erklären. Suchen Sie dann eine Erklärung an anderen Stellen und erwägen Sie, uns daran teilhaben zu lassen. Zu Themen, die in diesem Buch behandelt wurden, gibt es auch außerhalb von Wikibooks Informationen. Diese werden im Kapitel Weiterführende Informationen gesammelt.

Zur Benutzung dieses Buches ein Spruch, der Laotse falsch zugeschrieben [1] wird: „*Sag es mir - und ich werde es vergessen. Zeige es mir – und ich werde mich daran erinnern. Beteilige mich – und ich werde es verstehen.*“

Wir wünschen Ihnen viel Spaß mit dem Buch und mit Python.

Das Autorenteam

Erste Schritte

Hallo, Welt!

Ein einfaches *Hallo, Welt!*-Programm sieht in Python folgendermaßen aus:

```
#!/usr/bin/python
print "Hallo, Welt!"
```

Speichern Sie diese Datei unter zum Beispiel erstes.py und geben Sie der Datei Ausführungsrechte über **`chmod u+x erstes.py`**. Anschließend können Sie das Programm mit ./erstes.py in einer Linux-Shell ausführen:

Ausgabe

```
user@localhost:~$ ./erstes.py
Hallo, Welt!
```

Die erste Zeile enthält den sogenannten Shebang, eine Zeile, die in der Shell den passenden Interpreter für das folgende Skript aufruft. Die zweite Zeile enthält die erste Python-Anweisung. Der Text innerhalb der Hochkommata wird ausgegeben. print fügt ein New-Line-Zeichen an.

```
#!/usr/bin/python
print "Hallo,",
```

```
print "Welt!"
```

Dieses Skript erzeugt die gleiche Ausgabe, jedoch sorgt das Komma dafür, daß statt eines New-Line-Zeichens ein Leerzeichen eingefügt wird.

Ausgabe

user@localhost:~$ ./erstes1.py
Hallo, Welt!

Python können Sie auch im interaktiven Modus benutzen. Diese Art, Python aufzurufen eignet sich besonders dann, wenn Sie kleine Tests vornehmen wollen:

Ausgabe

user@localhost:~$ python
Python 2.7.3 (default, Feb 27 2014, 20:00:17)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hallo, Welt"
Hallo, Welt
>>>

>>> ist die Eingabeaufforderung. Wenn Sie diese sehen, dann können Sie direkt loslegen. Drücken Sie am Ende der Eingabe **Return**, so wird die Programmzeile ausgeführt.

Umlaute

Falls Sie jetzt schon mit dem Code experimentiert haben, wird Ihnen unter Umständen aufgefallen sein, dass die Ausgabe der Umlaute nicht korrekt war: Sie bekamen eine Fehlermeldung zu sehen. Folgender Code behebt das Problem:

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-
print "Über Sieben Brücken..."
```

Ausgabe

user@localhost:~$ ./umlaute.py
Über Sieben Brücken...

Die Zeile coding: iso-8859-1 enthält die Zeichensatzkodierung, die Sie im gesamten Quellcode verwenden dürfen. Welche Kodierung Sie verwenden, hängt von den von Ihnen benutzten Programmen ab. Statt **iso-8859-1** können Sie selbstverständlich auch **utf-8** oder eine andere von Ihnen bevorzugte Codierung verwenden.

Eingaben

Eingaben auf der Shell werden mit dem Befehl raw_input() entgegengenommen:

```
#!/usr/bin/python
print "Wie ist Ihr Name?",
Name = raw_input()
print "Ihr Name ist", Name
```

Ausgabe

user@localhost:~$ ./eingabe1.py
Wie ist Ihr Name? **Rumpelstilzchen**

Ihr Name ist Rumpelstilzchen

Name ist hierbei eine Variable von Typ *String*. Sie muss nicht explizit vereinbart werden, sondern wird erzeugt, wenn sie gebraucht wird.

Man kann raw_input() ein Argument mitgeben. Dadurch erspart man sich die erste print-Anweisung. Die Ausgabe ist dieselbe wie im ersten Beispiel.

```
#!/usr/bin/python
Name = raw_input("Wie ist Ihr Name? ")
print "Ihr Name ist", Name
```

Weitere Ausgaben

Neben Zeichenketten können Sie auch Zahlen eingeben, diese Zahlen werden von raw_input() jedoch als Zeichenketten behandelt, eine Konversion macht aus ihnen ganze Zahlen:

```
#!/usr/bin/python
Alter = int(raw_input("Wie alt sind Sie? "))
print "Sie sind", Alter, "Jahre alt."
```

Ausgabe

user@localhost:~$ ./ausgabe1.py
Wie alt sind Sie? **100**
Sie sind 100 Jahre alt.

Alter soll eine ganze Zahl werden, deswegen schreiben wir um das Ergebnis der Eingabe eine Konvertierungsaufforderung. int(String) erzeugt aus einem String eine Zahl, wenn der String nur aus Ziffern besteht.

Beim Experimentieren mit diesem Programm fällt auf, daß es bei der Eingabe von nicht-Ziffern abstürzt:

Ausgabe

user@localhost:~$ ./ausgabe1.py
Wie alt sind Sie? **abc**
Traceback (most recent call last):
  File "ausgabe1.py", line 2, in <module>
    Alter = int(raw_input("Wie alt sind Sie? "))
ValueError: invalid literal for int() with base 10: 'abc'

Diesem Problem wollen wir in einem späteren Kapitel auf die Spur kommen, wenn es um Ausnahmebehandlung geht. Konvertierungen von Datentypen besprechen wir im nächsten Kapitel.

Eine andere Art, die Ausgabe dieses Programmes zu formatieren sind Formatstrings:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
Name = raw_input("Wie heißen Sie? ")
Alter = int(raw_input("Wie alt sind Sie? "))
print "Sie heißen %s und sind %d Jahre alt." % (Name, Alter)
```

Ausgabe

user@localhost:~$ ./ausgabe2.py
Wie heißen Sie? **Rumpelstilzchen**
Wie alt sind Sie? **109**
Sie heißen Rumpelstilzchen und sind 109 Jahre alt.

Hier bedeutet ein %s einen String, %d eine ganze Zahl. Die Variablen, welche die so erzeugten Lücken füllen, werden am Ende des Strings nachgeliefert, wobei das Prozentzeichen die Argumentenliste einleitet. Die dem String übergebenen Argumente werden in Klammern als so genanntes Tupel, darauf kommen wir noch zu sprechen, angeführt.

Das Prozentzeichen hat insgesamt drei Bedeutungen: Als mathematische Operation ist es ein Modulo-Operator, als Trennzeichen in Formatstrings leitet es die Argumentenliste ein und als Formatierungssymbol zeigt es an, dass nun ein Argument folgt.

Elementare Datentypen

Strings können explizit durch ein vorrangestelltes u als UTF-8-Strings erklärt werden, Strings, in denen Steuerzeichen vorkommen, die als Text ausgedruckt werden sollen, muss hingegen ein r vorrangestellt werden:

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

UTFText = u'Dies\u0020ist\u0020UTF-8'
RoherText = r'Noch\tein\tText\nMit\u0020Steuerzeichen!'
NormalerText = 'Ein\tText\nMit Steuerzeichen!'

print UTFText
print RoherText
print NormalerText
```

Ausgabe

```
user@localhost:~$ ./ed1.py
Dies ist UTF-8
Noch\tein\tText\nMit\u0020Steuerzeichen!
Ein Text
Mit Steuerzeichen!
```

Innerhalb von *UTFText* werden UTF-8-Zeichen, in diesem Fall das Leerzeichen, eingefügt. Der Text *RoherText* wird mitsamt Steuerzeichen ausgegeben, während bei *NormalerText* die Steuerzeichen interpretiert werden. Es werden hier ein Tab und eine Neue Zeile eingefügt.

Zahlen werden so behandelt, wie man es sich naiv vorstellt:

```
#!/usr/bin/python

a = 3
b = 7
print a + b, a - b, b % a, b / a

c = 3.14
print "%f**2 = %f" % (c, c**2)
```

Ausgabe

```
user@localhost:~$ ./ed2.py
10 -4 1 2
3.140000**2 = 9.859600
```

Der Ausdruck b % a ist die Modulo-Operation, der Ausdruck c**2 hingegen berechnet das Quadrat von *c*.

Funktionen und Module

Funktionen dienen dazu, wiederkehrenden Code zu gliedern und so die Programmierung zu vereinfachen. Meist werden in Funktionen Dinge berechnet, oft geben Funktionen das Rechnungsergebnis wieder preis. Eine solche Rechnung ist s = sin(3,14):

```
#!/usr/bin/python

import math

s = math.sin(3.14)
print s
```

Ausgabe

```
user@localhost:~$ ./fumu.py
0.00159265291649
```

Hier wird das gesamte Modul math importiert. Einzig die Sinusfunktion wird gebraucht, das Ergebnis der Rechnung wird zuerst in die Variable s übernommen und anschließend auf dem Bildschirm dargestellt. Auf Module werden wir noch detailliert in einem eigenen Kapitel zu sprechen kommen.

Kommentare

Kommentare helfen dabei, den Quellcode leichter zu verstehen. Als Faustregel soll soviel wie nötig, so wenig wie möglich im Quellcode dokumentiert werden. *Wie* ein Programm funktioniert, sollte ersichtlich sein, *was* aber ein Abschnitt *warum* tut, soll dokumentiert werden.

```
#!/usr/bin/python

"""Das folgende Programm berechnet
 die Summe zweier Zahlen"""

# Zahlen eingeben
a = raw_input("Bitte eine Zahl eingeben: ")
b = raw_input("Bitte noch eine Zahl eingeben: ")

# in ganze Zahlen konvertieren
aZahl = int(a)
bZahl = int(b)

# Summe berechnen und ausgeben
summeZahl = aZahl + bZahl
print summeZahl
```

Der erste Kommentar wird mit """ eingeleitet und geht so lange, auch über mehrere Zeilen, bis wieder """ vorkommt. Kommentare, die mit # beginnen gehen bis zum Ende der Zeile.

Einrückungen

Eine Besonderheit von Python ist, dass zusammengehöriger Code gleich gruppiert wird. Blöcke werden anders als in anderen Programmiersprachen nicht etwa durch Schlüsselwörter[2] oder durch spezielle Klammern[3] gekennzeichnet, sondern durch Einrückung.

Als Vorgriff auf das Kapitel Kontrollstrukturen zeigen wir ihnen, wie zwei print-Anweisungen gruppiert werden.

```
#!/usr/bin/python

a = 0
if a < 1:
    print "a ist kleiner als 1"
    print "noch eine Ausgabe"
```

Die beiden print-Anweisungen gehören zusammen, weil sie die gleiche Einrückung haben. Beide werden nur ausgeführt, wenn der zu if gehörende Ausdruck *wahr* ist.

Zusammenfassung

Sie haben nun die grundlegenden Merkmale von Python kennengelernt und können bereits einfache Programme schreiben. In den folgenden Kapitel werden wichtige Grundlagen für die nächsten Schritte behandelt, zum Beispiel Datentypen und Kontrollstrukturen, Funktionen, Module und Objekte. Damit wird es möglich, komplexere Programme zu schreiben, die eine Strukturierung in sinnvolle Abschnitte, Wiederholungen und Entscheidungen ermöglichen. Es folgen Zugriff auf Dateien und reguläre Ausdrücke, mit denen man sein Linux schon sinnvoll erweitern kann. Danach sehen wir weiter … ;-)

Anmerkungen

[1] http://de.wikiquote.org/wiki/Laotse

[2] z. B. *begin* und *end* in Pascal

[3] Die geschweiften Klammern in C, C++ und Java sind typisch für viele Sprachen

---

Datentypen

In diesem Kapitel beleuchten wir die Datentypen von Python und geben typische Operationen an, mit denen Variablen dieses Typs modifiziert werden können. Ein wichtiges Prinzip in Python ist das **Duck-Typing**[1]. Solange eine Variable jede Operation unterstützt, die von ihr gefordert wird, ist Python der Typ der Variablen egal. Recht nützlich ist die Eigenschaft, eine Ente in einen Frosch zu verwandeln, sofern das möglich ist. Diese Zauberei wird *Konvertierung* genannt. Kommen wir aber erst zu den Datentypen.

Boolean

Dieser Datentyp repräsentiert **Wahrheitswerte** aus der Menge True und False. Werte dieses Datentyps werden zumeist bei Anfragen zurückgegeben, wie zum Beispiel: *Enthält folgende Zeichenkette ausschließlich Ziffern?*. Die passende String-Methode hierzu lautet übrigens isdigit(). Ergebnisse werden zumeist im Zusammenhang mit Kontrollstrukturen, über die Sie im nächsten Kapitel mehr erfahren, verwendet. Wahrheitswerte kann man mit Operatoren verknüpfen.

Die *logischen Verknüpfungen* haben wir in folgender Tabelle zusammengefasst, wobei *a* und *b* Bool'sche Variablen sind, die nur die Werte False und True annehmen können:

a	b	not a	a and b	a or b	a ^ b (xor)
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

Ein Ausdruck wird so schnell wie es geht ausgewertet. Ist am Anfang eines komplizierten Ausdrucks schon klar, dass der Ausdruck einen bestimmten Wahrheitswert erhält, dann wird nicht weiter ausgewertet. Zum Beispiel wird True or (A and B) zu True ausgewertet, ohne dass der Ausdruck (A and B) berücksichtigt wird. Dieses Vorgehen ist insbesondere wichtig im Zusammenhang mit Funktionen, denn A und B können auch Funktionen sein, die in diesem Fall nicht aufgerufen werden. Gleiches gilt für False and (Ausdruck). Hier wird (Ausdruck) ebenfalls nicht berücksichtigt.

Zahlen

Zahlen sind Grundtypen in Python. Es gibt verschiedene Sorten Zahlen unterschiedlicher Genauigkeit. Hier folgt als Vorgriff auf die Erläuterungen ein gemeinsames Beispiel:

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

# Int
a = 3
print a

# Long
b = 2**65
print b

# Fließkommazahl
c = 2.7
d = 1.3
print c // d

# komplexe Zahl
d = 5.0 + 2.0j
print d**2.0
```

Die Ausgabe:

Ausgabe

```
user@localhost:~$ ./zahlen1.py
3
36893488147419103232
2.0
(21+20j)
```

Int

Ganze Zahlen haben Sie schon in den ersten Schritten kennen gelernt. Hier folgt nochmal eine Zusammenfassung der Dinge, die Sie mit ganzen Zahlen tun können:

Operation	Bedeutung
abs(Zahl)	berechnet den Absolutbetrag der Zahl
bin(Zahl)	Liefert die binäre Darstellung der Zahl
divmod(Zahl1, Zahl2)	berechnet ein 2-Tupel, wobei der erste Teil die ganzzahlige Division und der zweite Teil die Modulo-Operation aus den beiden Zahlen ist.
hex(Zahl)	Liefert die hexadezimale Darstellung der Zahl
oct(Zahl)	Oktale Darstellung der Zahl
chr(Zahl)	liefert das Zeichen mit dem "ASCII"-Code *Zahl*, z.B. chr(97) liefert 'a'.
/, //	Ganzzahlige Division
%	Modulo-Operation
+, -, *, **	Addieren, Subtrahieren, Multiplizieren und Potenzieren

Long

Dieser Typ wird verwendet, wenn es sich um sehr große Zahlen handelt. So können Sie in Python problemlos a=2**65 berechnen, das Ergebnis wird in eine Zahl vom Typ **long** konvertiert. Zahlen dieses Typs wird manchmal ein **L** nachgestellt. Die Operationen auf solche Zahlen sind die gleichen wie oben beschrieben.

Float

Dieser Typ repräsentiert Fließkommazahlen. Diese Zahlen werden mit einem Dezimalpunkt notiert. Folgende Operationen sind auf Fließkommazahlen definiert:

Operation	Bedeutung
round(Zahl)	rundet die Zahl kaufmännisch zur nächst größeren/kleineren ganzen Zahl. Das Ergebnis wird als Fließkommazahl dargestellt.
/	Normale Division
//	entspricht der ganzzahligen Division
%	Modulo-Operation
+, -, *, **	Addieren, Subtrahieren, Multiplizieren und Potenzieren

Complex

Komplexe Zahlen[2] bestehen aus einem 2-Tupel, wobei sich ein Teil *Realteil* und ein anderer *Imaginärteil* nennt. Man kann komplexe Zahlen nicht aufzählen oder entscheiden, welche von zwei gegebenen Zahlen größer ist als die andere. Dargestellt werden solche Zahlen in Python als Summe von Real- und Imaginärteil, beispielsweise C=5.0 + 2.0j.

Folgende Dinge kann man mit komplexen Zahlen tun:

Operation	Bedeutung
abs(C)	Berechnet den Absolutbetrag der komplexen Zahl C
C.real, C.imag	liefert den Realteil, Imaginärteil der Zahl C zurück.
+, -, *, /, **	Addieren, Subtrahieren, Multiplizieren, Dividieren und Potenzieren

Strings

Ein String ist eine Folge von Zeichen. Anders als in anderen Sprachen kann man einen String nicht mehr verändern. Man muss dazu neue Strings erzeugen, welche den Ursprungsstring beinhalten:

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

# Gibt den String aus
print "Hallo, Welt!"
# Fuegt beide Strings zusammen, ohne Leerzeichen...
print "Hallo" + "Ballo"
# ... selbiges mit Leerzeichen
print "Hallo", "Ballo"
# Viermal "Hallo"
print "Hallo"*4

s = """



Dieses ist ein langer String, er geht
über mehrere Zeilen. Es steht zwar nichts interessantes
darin, aber darauf kommt es auch nicht an."""
print s
```

Ausgabe

```
user@localhost:~$ ./string1.py
Hallo, Welt!
HalloBallo
Hallo Ballo
HalloHalloHalloHallo

Dieses ist ein langer String, er geht
über mehrere Zeilen. Es steht zwar nichts interessantes
darin, aber darauf kommt es auch nicht an.
```

Strings können also zusammengesetzt werden, wobei gerade die Variante "Hallo" * 4, einen neuen String zu erzeugen, ungewohnt aber direkt erscheint. Lange Strings werden ebenso gebildet wie ein Kommentar, mit dem Unterschied, dass er einer Variable zugewiesen wird. Tatsächlich sind Kommentare, die mit """ gebildet werden nichts anderes, als anonyme Strings.

Auf String-Objekte kann man Methoden anwenden. Diese geben oft einen Ergebnis-String zurück und lassen das Original unverändert:

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

s = "dies ist mein kleiner String"
print s.capitalize()
print s.replace("kleiner", "kurzer")
print s.upper()
```

Ausgabe

user@localhost:~$ ./string2.py
Dies ist mein kleiner string
dies ist mein kurzer String
DIES IST MEIN KLEINER STRING

Einige der Methoden, die auf ein String-Objekt anwendbar sind, finden sich in folgender Tabelle:

Funktion	Beschreibung
capitalize()	Erzeugt einen neuen String, dessen erster Buchstabe groß geschrieben ist.
count(Substring, *Anfang*, *Ende*)	Zählt die Vorkommen von *Substring* im Objekt. *Anfang* und *Ende* sind optional
find(Substring, *Anfang*, *Ende*)	Findet das erste Vorkommen von *Substring*, gibt den Index innerhalb des Strings oder -1 zurück. *Anfang* und *Ende* sind optional.
lower()	Gibt einen String zurück, der nur aus Kleinbuchstaben besteht
replace(Alt, Neu, *Anzahl*)	Ersetzt im String *Alt* gegen *Neu*. Nur die ersten *Anzahl* Vorkommen werden ersetzt. *Anzahl* ist optional.
strip(*Zeichen*)	Entfernt die Vorkommen von *Zeichen* am Anfang und am Ende vom String. Wenn *Zeichen* nicht angegeben wird, so wird Leerzeichen angenommen.
upper()	Gibt einen String zurück, der nur aus Großbuchstaben besteht.
isalnum()	True, wenn der String aus Ziffern und Buchstaben besteht
isalpha()	True, wenn der String aus Buchstaben besteht.
isdigit()	True, wenn der String aus Ziffern besteht.
startswith(Prefix, *Anfang*, *Ende*)	True, wenn am Anfang des Strings *Prefix* vorkommt. Die Optionalen Parameter *Anfang* und *Ende* begrenzen den Suchbereich.
split(*Trenner*)	Zerlegt einen String in einzelne Worte. Trenner ist optional. Ohne Angabe wird bei Leerzeichen, Tabulatoren und Zeilenumbruechen getrennt.
join(Liste)	Fügt eine Liste von Strings mit diesem String wieder zusammen. Trenner.join(s.split(Trenner)) sollte genau den String s wiedergeben.

Mehr über Strings erfahren Sie im Abschnitt Sequenzen.

Unicode-Strings und Zeichenkodierung

Strings in Python werden intern immer als Unicode-Codepoints verarbeitet. Ein Codepoint ist eine festgelegte Zahl, die genau einem Zeichen zugeordnet ist. Der Unicode-Standard selbst definiert zudem Transformationsformate wie zum Beispiel UTF-8, die wiederum die tatsächliche Byte-Darstellung für die Speicherung festlegen. Wichtig zu verstehen ist, dass ein Unicode-Codepoint nicht mit einer UTF-8-Byte-Darstellung gleichzusetzen ist. Sie haben normalerweise unterschiedliche Werte für ein bestimmtes Zeichen.

Strings werden in zwei Schritten verarbeitet. Es gibt einen Dekodierungs- bzw. einen Kodierungsschritt. Für beide Schritte stehen analog die Funktionen decode() und encode() zur Verfügung. Beide Funktionen sind in der Klasse unicode und str zu finden.

Im ersten Schritt wird die **Bytefolgen-Dekodierung** der Python-Quelldatei in die Unicode-Codepoint-Darstellung durchgeführt. Über die in der Quelldatei angegebene Kommentarzeile # -*- coding: UTF-8 -*- wird auf die Standardzeichendekodierung hingewiesen, die zur automatischen Dekodierung hin zur internen Unicode-Codepoint-Darstellung benutzt werden soll. Dazu ist vom Programmierer sicherzustellen, dass die Datei auch in der gleichen Byte-Kodierung (hier UTF-8) abgespeichert ist, wie in der Kommentarzeile angegeben.

Im zweiten Schritt ist die Unicode-Codepoint-Darstellung in die **Zielkodierung** für den String zu bringen. Das geschieht über die encode()-Funktion und ist nur dann notwendig, wenn eine abweichende Zielkodierung zur Coding-Kommentarzeile angestrebt wird.

Die folgende Beispieldatei zeichenkodierung.py demonstriert die Bytefolgendarstellung der unterschiedlichen Zeichenkodierungen:

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

UnicodeText=u'ÄÜÖ€'
UTF_8Text='ÄÜÖ€'
ISO_8859_1TextIgnore=u'ÄÜÖ€'.encode('iso-8859-1', 'ignore')
ISO_8859_1TextReplace='ÄÜÖ€'.decode('UTF-8').encode('iso-8859-1',
'replace')

def bytefolge(kommentar, zeichenkette):
    for zeichen in zeichenkette:
        print "%x" % ord(zeichen),
    print "(%s)" % kommentar

bytefolge("Unicode-Codepoint-Folge", UnicodeText)
bytefolge("UTF-8-Bytefolge", UTF_8Text)
bytefolge("ISO-8859-1-Bytefolge (€ ignoriert)", ISO_8859_1TextIgnore)
bytefolge("ISO-8859-1-Bytefolge (€ ersetzt)", ISO_8859_1TextReplace)
```

Ausgabe

```
user@localhost:~$ ./zeichenkodierung.py
c4 dc d6 20ac (Unicode-Codepoint-Folge)
c3 84 c3 9c c3 96 e2 82 ac (UTF-8-Bytefolge)
c4 dc d6 (ISO-8859-1-Bytefolge (€ ignoriert))
c4 dc d6 3f (ISO-8859-1-Bytefolge (€ ersetzt))
```

Es sei darauf hingewiesen, dass das Eurozeichen (€) in der Zeichenkodierung ISO-8859-1 nicht vorhanden ist. Deshalb werden im Beispiel zwei verschiedene Strategien zur Behandlung dieser Situation gezeigt. Im ersten Fall

wird das €-Zeichen ignoriert und erhält somit auch keine Byterepräsentation in der ausgegebenen Bytefolge. Im Zweiten wird es ersetzt durch das Standard-Ersatzzeichen für ISO-8859-1, das Fragezeichen (0x3f).

Zeichen

Zeichen kann man als Spezialfall von Strings auffassen. Sie sind Strings der Länge Eins. Wir führen sie hier auf, um zwei in späteren Kapiteln benötigte Funktionen aufzuführen, nämlich ord(Zeichen) und chr(Zahl). ord(Zeichen) liefert eine Zahl, die der internen Darstellung des Zeichens entspricht, während chr(Zahl) ein Zeichen zurückliefert, welches zur angegebenen Zahl passt.

```
#!/usr/bin/python

print "'A' hat den numerischen Wert", ord('A')
print "Das Zeichen mit der Nummer 100 ist '" + chr(100) + "'"
```

Ausgabe

```
user@localhost:~$ ./zeichen1.py
'A' hat den numerischen Wert 65
Das Zeichen mit der Nummer 100 ist 'd'
```

Listen

Listen sind beschreibbare Datentypen, die dazu dienen, zur selben Zeit Elemente beliebigen Typs aufzunehmen.

```
#!/usr/bin/python

Werte = ['Text', 'Noch ein Text', 42, 3.14]
print Werte
print "Anzahl der Elemente: ", len(Werte)
```

Ausgabe

```
user@localhost:~$ ./listen1.py
['Text', 'Noch ein Text', 42, 3.1400000000000001]
Anzahl der Elemente: 4
```

Die Funktion len() bestimmt die Anzahl der Elemente der Liste, in diesem Fall 4. Listen können auch Listen enthalten, auch sich selbst.

Hinzugefügt werden Werte mit dem +-Operator und den Funktionen append() und insert():

```
#!/usr/bin/python

Werte = ['Text', 42]
Werte += ['foo']
Werte.append(3)
Werte.insert(2, 111)
print Werte
```

Ausgabe

```
user@localhost:~$ ./listen2.py
['Text', 42, 111, 'foo', 3]
```

wobei man diese Funktionen **Methoden** nennt, da sie Bestandteil der Listenobjekte sind. Die Methode insert(2, 111) fügt an die zweite Stelle die Zahl 111 ein. Die Zählung beginnt mit der 0, das nullte Element ist die Zeichenkette *Text*.

Löschen lassen sich Listenelemente ebenfalls:

```
#!/usr/bin/python

Werte = [1, 2, 'Meier', 4, 5]
print Werte.pop()
print Werte.pop(0)
Werte.remove('Meier')
print Werte
```

Ausgabe

```
user@localhost:~$ ./listen3.py
5
1
[2, 4]
```

Die Methode pop() liefert ohne Argument das letzte Element der Liste und entfernt es anschließend. Mit Argument wird das N-te Element gelöscht. Die Methode remove() entfernt das Element mit dem angegebenen Schlüssel.

Listen lassen sich auch per Funktion erzeugen. Eine solche Funktion ist range(von, bis, step). Sie erwartet einen Startwert sowie einen Endwert und baut daraus eine Liste, wobei der Endwert nicht Teil der entstehenden Liste ist. Die Parameter *von* und *step* sind optional und geben den Anfangswert wie auch die Schrittweite an, mit der Listenelemente zwischen dem Start- und dem Endwert eingefügt werden.

```
#!/usr/bin/python

l1 = range(1, 10)
l2 = range(1, 10, 3)
print l1
print l2
```

Ausgabe

```
user@localhost:~$ ./listen4.py
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 4, 7]
```

In unserem Beispiel enthält *l1* alle Werte von 1 bis 9, *l2* hingegen enthält lediglich 1, 4 und 7, weil wir eine Schrittweite von 3 vereinbart haben. Die range()-Funktion legt übrigens die gesamte Liste im Speicher an, weswegen man sich vor Gebrauch unbedingt sicher sein sollte, genug davon zu haben. Eine Liste mit mehreren Millionen Einträgen zu erzeugen dauert eine Weile, wie der Autor dieser Zeilen während eines recht eng terminierten Vortrags feststellen musste, als er statt xrange() (dazu kommen wir später) range() schrieb.

Mehr über Listen erfahren Sie im Abschnitt Sequenzen.

Tupel

Tupel lassen sich, anders als Listen, nicht verändern. Sie sind damit besonders geeignet, um Konstanten zu repräsentieren. Ansonsten ähneln sie Listen aber sehr:

```
#!/usr/bin/python

Werte = ('Text', 'Noch ein Text', 42, 3.14)
print min(Werte)
print max(Werte)
print "Anzahl der Elemente: ", len(Werte)
```

Ausgabe

```
user@localhost:~$ ./tupel1.py
3.14
Text
Anzahl der Elemente: 4
```

Ein Tupel wird in runde Klammern geschrieben. min() bestimmt das Minimum eines Tupels, max() das Maximum. Enthält ein Tupel Text, so wird dieser alphabetisch verglichen.

Mehr über Tupel erfahren Sie im nächsten Abschnitt Sequenzen.

Sequenzen

Zu Sequenzen zählen die hier behandelten Strings, Listen und Tupel. Lediglich Listen lassen sich ändern, alle anderen Sequenztypen sind konstant. Grund genug, einige Details zusammenzufassen.

Sequenzen können aufgezählt werden. Das erste Element hat den Index 0, das letzte Element den Index len(Sequenz)-1. Sequenzen können ihrerseits wieder Sequenzen aufnehmen. Eine Besonderheit ist, dass man Teilsequenzen, so genannte **Slices**, bilden kann:

```
#!/usr/bin/python

text = "Dies ist mein String"
print "Grossbuchstaben: ", text[0], text[14]
print "Verb: ", text[5:8]
print "Erstes Wort: ", text[:4]
print "Letztes Wort: ", text[14:]
```

Ausgabe

```
user@localhost:~$ ./seq1.py
Grossbuchstaben: D S
Verb: ist
Erstes Wort: Dies
Letztes Wort: String
```

An diesem Beispiel sieht man, dass man einzelne Zeichen direkt adressieren kann, wobei man den Index in eckige Klammern setzt, wie auch einen Bereich der Sequenz ansprechen kann, indem Anfang und Ende durch Doppelpunkt getrennt werden. Anfang und Ende vom Slice sind optional. Steht vor dem Doppelpunkt kein Wert, so ist der Anfang der Sequenz gemeint. Analoges gilt für das Ende.

Wenn Sie diese Art der Adressierung verstanden haben, fällt es Ihnen sicher leicht, die negative Adressierung ebenfalls zu verstehen:

```
#!/usr/bin/python

sequenz = (1, 2, 'a', 'b', 'c')
print "Buchstaben: ", sequenz[-3:]
print "Ziffern: ", sequenz[:-3]
```

Ausgabe

user@localhost:~$ ./seq2.py
Buchstaben: ('a', 'b', 'c')
Ziffern: (1, 2)

Buchstabenindizes werden hier vom Ende gezählt. Der Index *-1* ist das *c*, der Index *-2* ist das *b* und so weiter. Die Buchstaben werden also vom drittletzten zum letzten hin ausgegeben. Ziffern hingegen werden vom Nullten zum ebenfalls drittletzten ausgegeben, wobei ebendieses nicht mit ausgegeben wird.

Auf Sequenzen sind gemeinsame Operatoren und Methoden definiert. Folgende Tabelle gibt einen Überblick:

Funktion	Beschreibung	Beispiel
S * n	Erzeugt eine Sequenz, die aus der n-fachen Aneinanderreihung von S besteht.	(3,) * 10, "Hallo " * 2
min(), max()	Bestimmt das Minimum/Maximum der Sequenz	min((1, 2, 3)), max(['a', 'b', 'c'])
a in S	True, wenn Element *a* in der Sequenz *S* vorkommt	3 in [0] * 7
S + T	Die Sequenzen *S* und *T* werden aneinandergehängt	(1, 2, 3) + (2,)
len(S)	Länge, Anzahl der Elemente von S	len("Hallo, Welt!")

Set

Der Typ *set* beschreibt Mengen. Elemente einer Menge sind ungeordnet und einzigartig. Alle Sets unterstützen die üblichen Mengenoperationen:

```
#!/usr/bin/python

s1 = set('abc')
s2 = set('bcd')
print 's1 = ', s1, ' s2 = ', s2
# Differenzmenge
print 's1 - s2 = ', s1 - s2
# Vereinigungsmenge
print 's1 | s2 = ', s1 | s2
# Schnittmenge
print 's1 & s2 = ', s1 & s2
# Symmetrische Differenz
print 's1 ^ s2 = ', s1 ^ s2
# Enthalten
print "'a' in s1 = ", 'a' in s1
# Maechtigkeit der Menge
print "len(s1) = ", len(s1)
```

Ausgabe

user@localhost:~$ ./set1.py
s1 = set(['a', 'c', 'b']) s2 = set(['c', 'b', 'd'])
s1 - s2 = set(['a'])
s1 | s2 = set(['a', 'c', 'b', 'd'])
s1 & s2 = set(['c', 'b'])
s1 ^ s2 = set(['a', 'd'])
'a' in s1 = True
len(s1) = 3

Die Operation in liefert als Ergebnis True wenn ein Element in der Menge vorkommt.

Folgende Methoden lassen sich von einem Set-Objekt ansprechen:

Methode	Beschreibung
s1.difference(s2)	Differenzmenge, s1 - s2
s1.intersection(s2)	Schnittmenge, s1 & s2
s1.issubset(s2)	Teilmenge, s1 <= s2
s1.issuperset(s2)	Obermenge, s1 >= s2
s1.union(s2)	Vereinigungsmenge, s1 | s2
s1.symmetric_difference(s2)	Symmetrische Differenz, s1 ^ s2
s.add(E)	Fügt dem Set s das Element E hinzu
s.remove(E)	Entfernt das Element E aus dem Set

Dictionaries

Der Typ Dictionary stellt eine Zuordnung zwischen Schlüsseln und Werten her. Er ist genau wie ein Wörterbuch zu verstehen, wo als Schlüssel zum Beispiel ein englischer Begriff und als Wert ein deutschsprachiger Begriff aufgeführt ist. Selbstverständlich lassen sich auch Telefonnummern oder Gehälter auf diese Weise ordnen:

```
#!/usr/bin/python

personal = {'Tom' : 32000, 'Beate' : 44000, 'Peter' : 10000}
print personal
print "Tom verdient %d Euro pro Jahr" % (personal['Tom'])
```

Ausgabe

user@localhost:~$ ./dict1.py
{'Peter': 10000, 'Beate': 44000, 'Tom': 32000}
Tom verdient 32000 Euro pro Jahr

Dictionaries können modifiziert werden. Darüber hinaus bietet die Methode keys() die Möglichkeit, sich alle Schlüssel anzeigen zu lassen.

```
#!/usr/bin/python

personal = {'Tom' : 32000, 'Beate' : 44000, 'Peter' : 10000}
print personal
print "Susi kommt dazu...",
personal['Susi'] = 10000
print personal.keys()
```

```
print "Peter hat einen anderen Job gefunden..."
del personal['Peter']
print personal.keys()
print "Tom bekommt mehr Geld: ",
personal['Tom'] = 33000
print personal
```

Ausgabe

```
user@localhost:~$ ./dict2.py
{'Peter': 10000, 'Beate': 44000, 'Tom': 32000}
Susi kommt dazu... ['Susi', 'Peter', 'Beate', 'Tom']
Peter hat einen anderen Job gefunden...
['Susi', 'Beate', 'Tom']
Tom bekommt mehr Geld: {'Susi': 10000, 'Beate': 44000, 'Tom': 33000}
```

Elemente kann man hinzufügen, in dem man einen neuen Schlüssel in eckigen Klammern anspricht, und diesem einen Wert zuweist. Mit delete() lassen sich Schlüssel/Wert-Paare löschen. Die Methode keys() zeigt alle Schlüssel eines Dictionaries als Liste an. Folgende Tabelle zeigt darüberhinaus noch einige gebräuchliche Dictionary-Methoden:

Funktion	Beschreibung
get(Schlüssel)	Liefert den Wert für *Schlüssel*
has_key(Schlüssel)	*True*, wenn Schlüssel vorkommt
items()	Gibt den Inhalt als Liste von Tupeln zurück
pop(Schlüssel)	Gibt den Wert für Schlüssel zurück, entfernt dann *Schlüssel/Wert*
keys()	liefert alle Schlüssel als Liste
values()	analog zu keys(), liefert alle Werte als Liste

Im Kapitel Datenbanken lernen Sie eine Datenbank kennen, die wie ein Dictionary funktioniert.

Besonderheiten beim Kopieren

Beim Kopieren von Variablen gibt es eine Besonderheit. Die Kopien verweisen wieder auf die Originale. Versucht man nun, eine Kopie zu verändern, verändert man gleichzeitig das Original, wie folgendes Beispiel zeigt:

```
#!/usr/bin/python

liste1 = [1, 2, 3]
liste2 = liste1
liste2 += [5]

print liste1
print liste2
```

Statt zweier unterschiedlicher Listen bekommen wir dieses erstaunliche Ergebnis:

Ausgabe

```
user@localhost:~$ ./kopieren1.py
[1, 2, 3, 5]
[1, 2, 3, 5]
```

Wir haben also gar keine Kopie bearbeitet, sondern nur die *einzig vorhandene* Liste, auf die sowohl liste1 als auch liste2 nur *verweisen*.

Lösung des Problems: liste2 = liste1[:]. Ein Slice über die komplette Liste. Für andere Datentypen (aber auch Listen) gibt es aus dem Modul[3] copy die Funktion copy(). Dictionaries haben eine eigene copy()-Methode. All dieses erzeugt aber nur eine *flache* Kopie. Sollen auch z. B. Listen mit kopiert werden, die in einer Liste enthalten sind, verwendet man die Funktion copy.deepcopy(). Hierzu noch ein Beispiel:

```
#!/usr/bin/python
import copy

tupel1 = (1, 2, [3, 4], 5)
tupel2 = copy.deepcopy(tupel1)
#tupel2 = copy.copy(tupel1)    #  zum Testen des unterschiedlichen
Verhaltens auskommentieren
tupel2[2].append("foo")

print tupel1
print tupel2
```

Die beiden Tupel sind nun also eigenständige Kopien inklusive der enthaltenen Liste, wie uns die Programmausgabe beweist:

Ausgabe

```
user@localhost:~$ ./kopieren2.py
(1, 2, [3, 4], 5)
(1, 2, [3, 4, 'foo'], 5)
```

Das Beispiel zeigt auch nochmal, dass zwar Tupel unveränderlich sind, aber enthaltene veränderliche Datentypen gleichwohl auch veränderlich bleiben. Die Problematik des Kopierens ist also nicht auf veränderliche Datentypen beschränkt.

Konvertierung

Wie wir in der Einführung schon festgestellt haben, kann man einige Datentypen ineinander umwandeln. Aus einem String kann zum Beispiel eine Zahl werden, wenn der String nur Ziffern enthält. Andernfalls wird eine Fehlermeldung beim Versuch der Konvertierung ausgegeben. Die folgende Tabelle enthält einige Konvertierungsfunktionen:

Funktion	Konvertiert von	Konvertiert nach	Beispiel
int()	String, float	ganze Zahl	int("33")
float()	String, int	Fließkommazahl	float(1)
unicode()	String, Zahl	Unicode String	unicode(3.14)
ord()	Zeichen	ganze Zahl	ord('A')

Typenabfrage

Den Typ einer Variablen kann man mit Hilfe der Funktion type() abfragen:

```
#!/usr/bin/python
print type(3)
print type('a')
print type(u"Hallo, Welt")
print type(("Hallo", "Welt"))
print type(["Hallo", "Welt"])
print type({"Hallo" : 1, "Welt" : 2})
```

Ausgabe

```
user@localhost:~$ ./typenabfrage1.py
<type 'int'>
<type 'str'>
<type 'unicode'>
<type 'tuple'>
<type 'list'>
<type 'dict'>
```

Konstanten

Konstanten in Python sind nichts als spezielle Vereinbarungen. Man schreibt sie groß, dann wissen alle an einem Programm beteiligten Personen, dass dies eine Konstante ist. Das folgende Beispiel zeigt, wie das geht:

```
#!/usr/bin/python

KONSTANTE = 3

print "Meine Konstante ist:", KONSTANTE
```

Eine Ausgabe kann hier entfallen, das Programm ist schlicht zu trivial. Diese Art der textuellen Vereinbarung wird uns noch im Kapitel Rund um OOP begegnen. Eine Konstante ist also nicht anderes als eine Variable, bei der Programmierer *vereinbaren*, sie groß zu schreiben und nach der Initialisierung nicht mehr zu ändern. Dieses ist ein Teil des *Python way of coding*, der oft von weniger Strenge und Formalismus geprägt ist.

Zusammenfassung

Sie haben jetzt einen Überblick über das Typensystem von Python. Vom Prinzip her braucht man sich über den Typ meistens keine Gedanken machen, sollte jedoch in Spezialfällen darüber Bescheid wissen. Einer Variablen weist man Werte zu und schon steht der Typ fest. Den Typ einer Variablen kann man zur Laufzeit ändern und abfragen.

Anmerkungen

[1] Wenn etwas so aussieht wie eine Ente (oder ein Datentyp), so geht wie eine Ente und so quakt wie eine Ente, warum soll es dann keine sein?

[2] Um mehr Hintergrundinformationen über komplexe Zahlen zu bekommen, empfehlen wir ihnen das Wikibuch Komplexe Zahlen.

[3] Mehr über Module und deren Anwendungen findet sich im Kapitel Python unter Linux: Module

Kontrollstrukturen

Bis jetzt sind Sie in der Lage, einfache Programme mit Ein- und Ausgabe und einfachen Berechnungen zu schreiben. Für größere Programme wird aber die Möglichkeit benötigt, Funktionen nur unter gewissen Bedingungen oder mehrfach durchzuführen. Für diesen Zweck gibt es die Kontrollstrukturen, nämlich bedingte Ausführung und Schleifen.

if

Die if-Anweisung ist die einfachste Anweisung, mit der man abhängig von einer Bedingung eine Aktion auslösen kann:

```
#!/usr/bin/python

a = 0
if a < 1:
    print "a ist kleiner als 1"
```

Ausgabe

```
user@localhost:~$ ./if1.py
a ist kleiner als 1
```

Hier wird, wie in den ersten Schritten erläutert, eine Besonderheit von Python deutlich, nämlich die Einrückung. Alle gleich eingerückten Codezeilen gehören zum selben Block. Als Einrückungszeichen kann man Tabulatoren und Leerzeichen verwenden, wobei man niemals mischen sollte. Verwenden Sie vier Leerzeichen, wenn Sie konform zum Standard sein wollen.

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

a = 0
if a < 1:
    print "a ist kleiner als 1"
    print "Dies gehört auch noch zum Block"
print "Dies gehört nicht mehr zur IF-Anweisung"
```

Ausgabe

```
user@localhost:~$ ./if2.py
a ist kleiner als 1
Dies gehört auch noch zum Block
Dies gehört nicht mehr zur IF-Anweisung
```

Selbstverständlich können Bedingung in der if-Anweisung auch zusammengesetzt werden:

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

print "Bitte geben Sie eine Zahl ein:",
zahl = int(raw_input())

if zahl > 0 and zahl % 2 == 0:
```

```
    print "Gerade Zahl."
    if (90 <= zahl <= 100) or zahl == 50:
        print "Zahl ist zwischen 90 und 100 (inklusive), oder die Zahl
ist 50"
```

Ausgabe

user@localhost:~$ ./if3.py
Bitte geben Sie eine Zahl ein: **22**
Gerade Zahl.

Zusammengesetzte Anweisungen müssen nach logischen Regeln geklammert werden, die Einrückung muss angepasst werden.

Details

Einige Ausdrücke werden automatisch zu false ausgewertet, ohne dass wir extra darauf testen müssen. Dies betrifft:

- das "Nichts-Objekt" None
- die Konstante False
- die Nullwerte aller nummerischen Typen, also z.B. 0, 0L, 0.0, 0j
- den leeren String
- das leere Tupel ()
- die leere Liste []
- das leere Dictionary {}
- die leeren Mengen set() und frozenset()

Entsprechend werden Variablen "mit Inhalt" zu true ausgewertet.

if-elif-else

Entscheidungen sind oft geprägt von mehreren Wahlmöglichkeiten, die man allesamt abfangen kann. Mit einer einfachen Fallunterscheidung wollen wir beginnen:

```
#!/usr/bin/python

print "Bitte geben Sie eine Zahl ein:",
a = int(raw_input())

if a % 2 == 0:
  print "%d ist gerade" % a
else:
  print "%d ist ungerade" % a
```

Ausgabe

user@localhost:~$ ./iee1.py
Bitte geben Sie eine Zahl ein: **12**
12 ist gerade

Das Schlüsselwort else leitet den Codeblock ein, welcher Ausgeführt wird, wenn die Bedingung in der if-Anweisung nicht zutrifft.

Wählt man nun aus vielen verschiedenen Dingen aus, wie bei einer Mahlzeit im Restaurant, kommt man mit dieser recht trivialen Fallunterscheidung nicht weiter, es sei denn, man schachtelt sie. Python bietet mit elif eine weitere Möglichkeit an:

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

print "Menü a la Carte:"
print "1 für Suppe"
print "2 für Hähnchenschnitzel mit Pommes"
print "3 Ich möchte nun zahlen"
print "Ihre Auswahl>",
auswahl = int(raw_input())

if auswahl == 1:
  print "Bitte sehr, Ihre Suppe"
elif auswahl == 2:
  print "Hähnchenschnitzel mit Pommes... bitte sehr."
elif auswahl == 3:
  print "Ihre Rechnung kommt sofort"
else:
  print "Das ist aber keine gute Wahl!"
```

Ausgabe

user@localhost:~$ ./iee2.py
Menü a la Carte:
1 für Suppe
2 für Hähnchenschnitzel mit Pommes
3 Ich möchte nun zahlen
Ihre Auswahl> **2**
Hähnchenschnitzel mit Pommes... bitte sehr.

elif ist die Kurzschreibweise von *else if*. Die angehängte else-Anweisung wird nur dann ausgeführt, wenn keine vorherige Bedingung zutrifft.

if -- ganz kurz

Es gibt eine einzeilige Variante der *if-else*-Kontrollstruktur. Die Syntax ist gedacht für Fälle, in denen in beiden Entscheidungszweigen derselben Variablen etwas zugewiesen werden soll, wie folgt:

```
#!/usr/bin/python

a = 22
text = ""
if a % 2 == 0:
    text = "gerade"
else:
    text = "ungerade"
print text
```

Die Syntax der Kurzschreibweise[1] ist anders als gewohnt, die Abfrage wird so zusagen in der Mitte vorgenommen:

```
#!/usr/bin/python
```

```
a = 22

text = "gerade" if a % 2 == 0 else "ungerade"
print text
```

Ausgabe

user@localhost:~$ ./ifkurz1.py
gerade

Der else-Zweig darf hierbei nicht entfallen.

Nur am Rande: Ein weiterer Weg, einen Einzeiler draus zu machen, kann mit folgender Notation erreicht werden.

```
#!/usr/bin/python

a = 22

text = ("gerade", "ungerade")[a % 2]
print text
```

Vergleichsoperatoren in der Übersicht

Die folgende Tabelle zeigt die von Python unterstützten Vergleichsoperatoren.

Operator	Beschreibung	Beispiel	Beispielausgabe
==	Testet auf Werte-Gleichheit	"Hallo" == "Welt"	False
!=	Testet auf Werte-Ungleichheit	"Hallo" != "Welt"	True
is	Testet auf Objekt-Gleichheit	type("Hallo") is str	True
is not	Testet auf Objekt-Ungleichheit	type("Hallo") is not int	True
<	Testet auf kleineren Wert	4 < 4	False
<=	Testet auf kleineren oder gleichen Wert	4 <= 4	True
>	Testet auf größeren Wert	"b" > "a"	True
>=	Testet auf größeren oder gleichen Wert	5.9 >= 6	False

Diese Vergleichsoperatoren können Sie mit Hilfe der logischen Operatoren aus dem Kapitel Datentypen, Boolean miteinander verknüpfen, um komplexere Ausdrücke zu formen.

Möchte man eine Variable darauf testen, ob sie innerhalb eines Intervalles liegt, kann man dies auf einfache Weise hinschreiben:

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

x = 4

if 10 > x >= 2:
    print "x ist kleiner als 10 und größer/gleich 2"
```

Ausgabe

user@localhost:~$ ./ifdoppelt.py
x ist kleiner als 10 und größer/gleich 2

Diese Syntax ist eine Kurzschreibweise von if x < 10 and x >=2.

for-Schleife

for-Schleifen dienen dazu, einen Codeblock eine bestimmte Anzahl mal wiederholen zu lassen, wobei diese Anzahl zu beginn der Schleife feststeht. Hierbei wird über Sequenzen iteriert, es werden keine Variablen hochgezählt. Die Sequenz, über die iteriert wird, darf sich nicht zur Laufzeit ändern.

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

for person in ["Herr Müller", "Frau Meier", "Tina Schulze"]:
    print person, "lernt Python!"
```

Ausgabe

```
user@localhost:~$ ./for1.py
Herr Müller lernt Python!
Frau Meier lernt Python!
Tina Schulze lernt Python!
```

Die Sequenz ist hier eine Liste, es könnte aber auch ein Tupel oder ein String sein. Manchmal möchte man einen Index mitlaufen lassen, der die Nummer eines Eintrages der Sequenz angibt. Dies ermöglicht uns die Funktion enumerate():

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

for nummer, person in enumerate(["Herr Müller", "Frau Meier", "Tina
Schulze"]):
    print "%s lernt als %d. Person Python!" % (person, nummer + 1)
```

Ausgabe

```
user@localhost:~$ ./for2.py
Herr Müller lernt als 1. Person Python!
Frau Meier lernt als 2. Person Python!
Tina Schulze lernt als 3. Person Python!
```

enumerate() liefert den Index wie auch den Eintrag bei jedem Schleifendurchlauf. Da wir die Personen von 1 durchzählen wollen, der Index jedoch bei 0 beginnt, haben wir die Ausgabe etwas angepasst.

Selbstverständlich funktionieren Schleifen auch mit der schon bekannten Funktion range(). Eine Variante dieser Funktion nennt sich xrange(Von, Bis, Step), wobei die Argumente *Von* und *Step* optional sind. Der Vorteil dieser Funktion liegt darin, dass ein solcher Wertebereich nicht als Liste im Speicher angelegt werden muss, Sie können also beliebig große Werte verwenden ohne Nachteile im Speicherverbrauch befürchten zu müssen:

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

for zahl in xrange(0, 10, 2):
    print zahl, "ist eine gerade Zahl"
```

Ausgabe

user@localhost:~$ ./for3.py
0 ist eine gerade Zahl
2 ist eine gerade Zahl
4 ist eine gerade Zahl
6 ist eine gerade Zahl
8 ist eine gerade Zahl

Mit Hilfe der for-Schleife und einem Dictionary kann man die Häufigkeit von Zeichen einer Eingabe ermitteln:

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

# Texteingabe
text = raw_input("Bitte geben Sie einen Text ein: ")
ergebnis = {}

# Anzahl der jeweiligen Zeichen bestimmen
for zeichen in text:
    ergebnis[zeichen] = ergebnis.get(zeichen, 0) + 1

# Anzahl der Zeichen insgesamt
anzahl_zeichen = len(text)
print "Es wurden", anzahl_zeichen, "Zeichen eingegeben, davon sind",

# Anzahl verschiedener Zeichen
anzahl = len(ergebnis)
print anzahl, "verschieden."

# Statistik der Zeichen ausdrucken
for key in ergebnis.keys():
    haeufigkeit = float(ergebnis[key]) / anzahl_zeichen * 100.0
    print "Das Zeichen '%s' kam %d mal vor. Häufigkeit: %4.1f%%" % \
            (key, ergebnis[key], haeufigkeit)
```

Ausgabe

user@localhost:~$ ./for4.py
Bitte geben Sie einen Text ein: **Hallo, Welt**
Es wurden 11 Zeichen eingegeben, davon sind 9 verschieden.
Das Zeichen 'a' kam 1 mal vor. Häufigkeit: 9.1%
Das Zeichen ' ' kam 1 mal vor. Häufigkeit: 9.1%
Das Zeichen 'e' kam 1 mal vor. Häufigkeit: 9.1%
Das Zeichen 'H' kam 1 mal vor. Häufigkeit: 9.1%
Das Zeichen 'l' kam 3 mal vor. Häufigkeit: 27.3%
Das Zeichen 'o' kam 1 mal vor. Häufigkeit: 9.1%
Das Zeichen ',' kam 1 mal vor. Häufigkeit: 9.1%
Das Zeichen 't' kam 1 mal vor. Häufigkeit: 9.1%
Das Zeichen 'W' kam 1 mal vor. Häufigkeit: 9.1%

ergebnis definieren wir als ein anfangs leeres Dictionary. Anschließend bestimmen wir, wie oft ein bestimmtes Zeichen eingegeben wurde. Hierzu bedienen wir uns der Methode get(), welche uns entweder die bisherige Anzahl

eines eingegebenen Zeichens ausgibt oder 0 als voreingestellten Wert, wenn das Zeichen bisher nicht vorkommt. Die Häufigkeit eines Zeichens ist die Anzahl eines bestimmtes Zeichens geteilt durch die Gesamtzahl der Zeichen. Die Formatierung der print-Ausgabe ist hierbei noch ungewohnt. Die Formatanweisung %4.1f besagt, dass wir 4 Stellen der Zahl insgesamt ausgeben wollen, davon eine Nachkommastelle. Das doppelte Prozentzeichen hingegen bewirkt die Ausgabe eines einzelnen Prozentzeichens.

while-Schleife

Ist die Anzahl der Schleifendurchläufe nicht von vorneherein fest, so bietet sich eine while-Schleife an. Mit dieser lässt sich das obige Restaurant-Beispiel so umschreiben, dass Gäste bestellen können, bis sie satt sind:

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

satt = False
rechnung = 0

while not satt:
    print "Menü a la Carte:"
    print "1 für Suppe (2 Euro)"
    print "2 für Hähnchenschnitzel mit Pommes (4 Euro)"
    print "3 Ich möchte nun Zahlen"
    print "Ihre Auswahl>",
    auswahl = int(raw_input())
    if auswahl == 1:
        print "Bitte sehr, Ihre Suppe"
        rechnung += 2
    elif auswahl == 2:
        print "Hähnchenschnitzel mit Pommes... bitte sehr."
        rechnung += 4
    elif auswahl == 3:
        print "Ihre Rechnung beträgt %d Euro" % rechnung
        satt = True
    else:
        print "Das ist aber keine gute Wahl!"
```

Ausgabe

```
user@localhost:~$ ./while1.py
Menü a la Carte:
1 für Suppe (2 Euro)
2 für Hähnchenschnitzel mit Pommes (4 Euro)
3 Ich möchte nun Zahlen
Ihre Auswahl> 1
Bitte sehr, Ihre Suppe
Menü a la Carte:
1 für Suppe (2 Euro)
2 für Hähnchenschnitzel mit Pommes (4 Euro)
3 Ich möchte nun Zahlen
Ihre Auswahl> 2
```

Hähnchenschnitzel mit Pommes... bitte sehr.
Menü a la Carte:
1 für Suppe (2 Euro)
2 für Hähnchenschnitzel mit Pommes (4 Euro)
3 Ich möchte nun Zahlen
Ihre Auswahl> **3**
Ihre Rechnung beträgt 6 Euro

Selbstverständlich können Sie den Schleifenkopf auf die gleiche Weise wie if-Anweisungen mit Bedingungen bestücken. Wichtig ist nur, dass die Schleife läuft, so lange die Bedingung im Schleifenkopf zu True ausgewertet wird.

break und continue

Die beiden Schlüsselwörter break und continue brechen Schleifen ab oder führen an den Schleifenkopf zurück. Sie werden üblicherweise bei sehr großen Schleifenkörpern eingesetzt, wenn an einer Stelle deutlich wird, dass die aktuelle Iteration entweder die letzte ist oder der Rest des Schleifenkörpers unnötig ist und übersprungen werden soll. Man kann jede Schleife, die diese Schlüsselwörter enthält auch so umformulieren, dass diese nicht benötigt werden, jedoch führt ihr maßvoller Einsatz oft zu einem übersichtlicheren Code.

```
#!/usr/bin/python

while True:
  antwort = raw_input("Soll ich fortfahren? (J/N) ")
  if antwort in ('n', 'N'):
    break
```

Ausgabe

user@localhost:~$ ./break1.py
Soll ich fortfahren? (J/N) **j**
Soll ich fortfahren? (J/N) **j**
Soll ich fortfahren? (J/N) **n**

Hier wird der Schleifenkörper mindestens einmal ausgeführt. Man spricht in diesem Zusammenhang von einer *nicht-abweisenden* Schleife, da die Abbruchbedingung am Schleifenende[2] erfolgt. break kann aber selbstverständlich auch an jeder anderen Stelle innerhalb des Schleifenkörpers stehen.

```
#!/usr/bin/python

for buchstabe, zahl in [('a', 1), ('b', 2), (3, 3), ('d', 4)]:
    if type(buchstabe) is not str:
        continue
    else:
        print "Der Buchstabe", buchstabe, "hat den Zahlenwert", zahl
```

Ausgabe

user@localhost:~$ ./continue1.py
Der Buchstabe a hat den Zahlenwert 1
Der Buchstabe b hat den Zahlenwert 2
Der Buchstabe d hat den Zahlenwert 4

Das Tupel (3, 3) soll hier nicht ausgegeben werden. Der Vergleich auf Typengleichheit führt dazu, dass nur Zeichenketten und deren Werte ausgegeben werden. continue führt hier wieder zurück an den Schleifenanfang, wenn das erste Tupelelement keine Zeichenkette ist.

Schleifen-Else

Schleifen können, wie Verzweigungen auch, ein else enthalten. Dieses wird bei for-Schleifen ausgeführt, wenn keine weiteren Iterationen mehr durchgeführt werden können, beispielsweise weil die zu iterierenden Liste abgearbeitet ist. Bei while-Schleifen wird der else-Block ausgeführt, wenn die Schleifenbedingung zu False ausgewertet wird. break führt bei Schleifen nicht(!) zu einem Übergang in den else-Block. Folgendes Beispiel zeigt einen Einsatzzweck von else bei einer for-Schleife:

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


# user-UID-dict
users = {'anne' : 500, 'paul' : 501, 'martina' : 502}

print "Sucht nach einem Benutzernamen zu einer angegebenen UID."
uid = int(raw_input('Bitte geben Sie die UID ein: '))

for name in users.keys():
    if uid == users[name]:
        print "Die UID", uid, "gehört zu ", name
        break
else:
    print "Tut mir leid, es wurde kein passender Benutzername zu dieser
 UID gefunden."
```

Ausgabe

user@localhost:~$ ./forelse1.py
Sucht nach einem Benutzernamen zu einer angegebenen UID.
Bitte geben Sie die UID ein: **501**
Die UID 501 gehört zu paul
...weiterer Aufruf...
Bitte geben Sie die UID ein: **1000**
Tut mir leid, es wurde kein passender Benutzername zu dieser UID gefunden.

Das Programm sucht nach einem User, dessen User-ID angegeben wurde. Findet es diesen, so gibt es den Namen aus. Falls es keinen Benutzer mit dieser User-Id gibt, so wird der else-Block ausgeführt.

Das folgende Programm gibt den Hexdump einer Eingabe aus. Eine leere Eingabe beendet das Programm per break, das Wort **Ende** hingegen nutzt den else-Teil der Schleife:

```
#!/usr/bin/python

eingabe = ""
while eingabe != "Ende":
    eingabe = raw_input("Geben Sie etwas ein: ")
    if eingabe == "":
```

```
        break
    else:
        for c in eingabe:
            print hex(ord(c)),
        print
else:
    print "Das war es, vielen Dank"
```

Ausgabe

user@localhost:~$ ./whileelse1.py
Geben Sie etwas ein: **Test**
0x54 0x65 0x73 0x74
Geben Sie etwas ein: **Ende**
0x45 0x6e 0x64 0x65
Das war es, vielen Dank

Try-Except

Im Zusammenhang mit Konvertierungen ist es uns schon bei den ersten Programmen aufgefallen, daß mal etwas schief gehen kann. Der Nutzer eines Programmes soll eine Zahl eingeben, gibt aber stattdessen seinen Namen ein. Schon bricht Python die Verarbeitung ab:

```
#!/usr/bin/python

x = int(raw_input("Bitte geben Sie eine Zahl ein: "))
print x
```

Ausgabe

user@localhost:~$ ./except1.py
Bitte geben Sie eine Zahl ein: **Zehn**
Traceback (most recent call last):
  File "./except1.py", line 3, in <module>
    x = int(raw_input("Bitte geben Sie eine Zahl ein: "))
ValueError: invalid literal for int() with base 10: 'Zehn'

Python kennt hier eine Möglichkeit, einen Codeblock probeweise auszuführen und zu schauen, ob die Abarbeitung erfolgreich ist. Ist sie es nicht, wird eine so genannte **Exception** ausgelöst, die Bearbeitung des Codeblockes wird abgebrochen und zu einer Stelle im Code gesprungen, die diesen Fehler abfängt:

```
#!/usr/bin/python

try:
    x = int(raw_input("Bitte geben Sie eine Zahl ein: "))
    print x
except:
    print "Ein Fehler ist aufgetreten, macht aber nichts!"
print "Hier geht es weiter"
```

Ausgabe

user@localhost:~$ ./except2.py
Bitte geben Sie eine Zahl ein: **Zehn**
Ein Fehler ist aufgetreten, macht aber nichts!
Hier geht es weiter

Was immer nun der Nutzer eingibt, er bekommt auf jeden Fall eine Ausgabe. Die Anweisung, welche die Zahl ausgeben soll wird jedoch nur ausgeführt, wenn die Konvertierung erfolgreich war. War sie es nicht, wird der Codeblock unterhalb von except abgearbeitet. In jedem Fall wird die letzte print-Anweisung ausgeführt.

Wird statt einer Zahl ein Buchstabe versuchsweise konvertiert, gibt es einen so genannten *ValueError*, den Sie gesehen haben, als Sie das erste Beispiel dieses Abschnitts mit Buchstabendaten ausgeführt haben. einige der Exceptions kann man gezielt abfangen, wie folgendes Beispiel demonstriert:

```
#!/usr/bin/python

try:
  x = int(raw_input("Bitte geben Sie eine Zahl ein: "))
  print x
except ValueError:
  print "Das war keine Zahl!"
except:
  print "Irgendein anderer Fehler"
```

Ausgabe

user@localhost:~$ ./except3.py
Bitte geben Sie eine Zahl ein: **Zehn**
Das war keine Zahl!

Geben Sie nun eine Buchstabenfolge ein, wird der Block unterhalb except ValueError ausgeführt. Um einen anderen Fehler zu provozieren, geben Sie **STRG+C** ein, es wird dann der Block ausgeführt, der die allgemeine Exception bearbeitet.

Try-Finally

Für manche schweren Fehler reicht except nicht aus, das Programm muss abgebrochen werden. In diesem Fall ermöglicht uns finally, noch letzte Aufräumarbeiten durchzuführen, wie zum Beispiel eine Datei zu schließen oder schlicht einen Grund für das Scheitern anzugeben:

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

try:
  x = 1 / 0
  print x
finally:
  print "Oh, eine Division durch Null! Ich räume noch eben auf..."
print "Hierhin komme ich nicht mehr!"
```

Ausgabe

user@localhost:~$ ./finally1.py
Oh, eine Division durch Null! Ich räume noch eben auf...
Traceback (most recent call last):

```
  File "./finally1.py", line 5, in <module>
    x = 1 / 0
ZeroDivisionError: integer division or modulo by zero
```

Die letzte Zeile des Codes wird nie erreicht. Nach der Division durch Null wird eine Exception geworfen, die Meldung ausgegeben und der Programmlauf abgebrochen. finally wird immer ausgeführt, auch dann, wenn kein Fehler auftrat. Darauf können wir uns verlassen. Man kann try-except und try-finally gemeinsam verwenden, wie folgendes Beispiel zeigt:

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

print "Berechnet die Zahl 1/x "
wert = 0
try:
  x = raw_input("Geben Sie eine Zahl ein: ");
  wert = 1.0 / float(x)
except ValueError:
  print "Fehler: Das war keine Zahl"
except KeyboardInterrupt:
  print "Fehler: Steuerung-C gedrückt"
finally:
  print "Danke für die Zahl"

print wert
```

Hier können Sie Fehler produzieren, in dem Sie **STRG+C**, Buchstaben oder **0** eingeben. In jedem Fall wird der Programmteil ab finally ausgeführt.

Assert

Mit assert(Bedingung) machen Sie Zusicherungen, die im weiteren Verlauf des Programmes gelten. Sollten diese Zusicherungen nicht erfüllt sein, die Bedingung also zu False ausgewertet werden, wird eine Exception geworfen und das Programm bricht ab, wie folgendes Beispiel zeigt:

```
#!/usr/bin/python



text = raw_input("Bitte geben Sie Text ein: ")
assert(text != "")
print text
```

Wenn Sie beim Programmlauf keine Eingabe machen, sondern nur mit **Return** bestätigen, erhalten Sie folgende Ausgabe:

Ausgabe

```
user@localhost:~$ ./assert1.py
Bitte geben Sie Text ein:
Traceback (most recent call last):
  File "./a.py", line 5, in <module>
    assert(text != "")
```

AssertionError

Hier wird eine Exception geworfen, das Programm bricht ab. Solche Zusicherungen baut man überall dort in Programme ein, wo es unsinnig wäre, ohne diese Zusicherung fortzufahren.

Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie den Ablauf von Programmen steuern und beeinflussen können. Steuerungsmöglichkeiten sind das Verzweigen und wiederholte Ausführen von Programmteilen. Ebenfalls können Sie nun Fehler abfangen und notwendige Arbeiten kurz vor dem unvermeidlichen Ende ihres Programmes durchführen lassen.

Anmerkungen

[1] Diese Notation ist einzigartig und insbesondere verschieden von dem in der Programmiersprache C (http://de.wikipedia.org/wiki/C_(Programmiersprache)) bekannten Bedingungsoperator, bei dem die Bedingung der Zuweisung vorhergeht:

```
#!/usr/bin/python
print "Hallo, Welt!"
```

[2] In C/C++/Java ist dies analog zu einer *do...while(bedingung)*-Schleife

---

Funktionen

Funktionen gliedern den Programmtext, gestalten den Code übersichtlich und ersparen dem Programmierer wertvolle Entwicklungszeit. Ebenfalls sind sie eine sehr gute Schnittstelle, um im Team gemeinsam an Aufgaben zu arbeiten.

Funktionen

Funktionen werden wie folgt definiert und aufgerufen:

```
#!/usr/bin/python

def HalloWelt():
    print "Hallo, Welt!"

HalloWelt()
HalloWelt()
```

Ausgabe

```
user@localhost:~$ ./funk1.py
Hallo, Welt!
Hallo, Welt!
```

Dem Schlüsselwort def folgt der Funktionsname. Die Anweisungen der Funktion folgt als Block. Funktionen können Parameter haben und diese nutzen:

```
#!/usr/bin/python

def HalloWelt(anzahl):
    if anzahl > 0:
        for i in xrange(0, anzahl):
            print "Hallo, Welt!"
```

```
HalloWelt(3)
```

Ausgabe

```
user@localhost:~$ ./funk2.py
Hallo, Welt!
Hallo, Welt!
Hallo, Welt!
```

Eine der Aufgaben von Funktionen ist es, Werte zurückzuliefern, wobei jeder Typ zurückgegeben werden kann. Wir beschränken uns in den Beispielen auf Zahlen:

```
#!/usr/bin/python

def summe(a, b, c):
    wert = a + b + c
    return wert

print summe(1, 7, 3)
```

Ausgabe

```
user@localhost:~$ ./funk3.py
11
```

Die folgende Funktion berechnet für jeden String die Buchstabensumme, wobei ein *A* für 1 steht, *B* für 2 und so fort. Kleinbuchstaben werden zunächst in Großbuchstaben umgewandelt. Danach wird die Summe ermittelt, wobei nur die Buchstaben *A*-*Z* gezählt werden:

```
#!/usr/bin/python
def StringWert(s):
    s = s.upper()
    summe = 0
    for zeichen in s:
        wert = ord(zeichen) - ord('A') + 1
        if wert > 0 and wert <= 26:
            summe += wert
    return summe

print StringWert("ABBA")
print StringWert("Hallo, Welt!")
```

Ausgabe

```
user@localhost:~$ ./funk4.py
6
108
```

Die Methode upper() kennen Sie schon, sie liefert einen String zurück, welcher nur aus Großbuchstaben besteht. Über diesen String wird iteriert. Mit ord() erhalten wir einen Zahlenwert für den aktuellen Buchstaben. Von diesem Zahlenwert ziehen wir den Zahlenwert des ersten Buchstabens (*A*) ab, und da wir nicht von 0, sondern von 1 beginnen wollen, müssen wir zur Werteberechnung noch 1 hinzu addieren.

Funktionen kann man auch ineinander schachteln, wie folgendes Beispiel zeigt:

```
#!/usr/bin/python

def Hypothenuse(Kathete1, Kathete2):
    def Quadriere(x):
        return x * x

    def Summiere(a, b):
        return a + b

    Hyp = Summiere(Quadriere(Kathete1), Quadriere(Kathete2))
    return Hyp

print "Das Hypothenusenquadrat lautet:", Hypothenuse(1, 2)
```

Ausgabe

```
user@localhost:~$ ./funk5.py
Das Hypothenusenquadrat lautet: 5
```

Außerhalb der Funktion Hypothenuse() sind die Funktionen Quadriere() und Summiere() unbekannt.

Funktionsdokumentation

Funktionen können und sollten dokumentiert werden. Schreibt man mit passender Einrückung einen **Doc-String**, also einen anonymen mehrzeiligen String, in die Funktion, so kann man an anderer Stelle auf ihn Bezug nehmen. Es werden für jedes angelegte Objekt einige Zusatzinformationen generiert und abgespeichert.

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

def Collatz(parm):
    """Bestimmt das nächste Collatz-Folgeelement vom Parameter parm.
    Das nächste Folgeelement ist 3*parm + 1, wenn parm ungerade ist,
    sonst parm/2 """
    if parm % 2 != 0:
        return 3 * parm + 1
    else:
        return parm / 2

print Collatz.__doc__
```

Ausgabe

```
user@localhost:~$ ./fdoc1.py
Bestimmt das nächste Collatz-Folgeelement vom Parameter parm.
  Das nächste Folgeelement ist 3*parm + 1, wenn parm ungerade ist,
  sonst parm/2
```

Mit __doc__ wird auf eine automatisch erstellte und belegte Variable zugegriffen. Diese Variable wird erzeugt, wenn die Funktion angelegt wird. Falls keine Dokumentation angegeben wurde, enthält __doc__ einen leeren String.

Parameter

Nachdem wir nun die Grundlagen der Funktionen behandet haben, wollen wir uns detailliert mit Parametern beschäftigen.

Parameter kann man mit Werten vorbelegen:

```
#!/usr/bin/python

def HalloWelt(anzahl=3):
    print
    if anzahl > 0:
        for i in xrange(0, anzahl):
            print "Hallo, Welt!"

HalloWelt(1)
HalloWelt()
```

Ausgabe

```
user@localhost:~$ ./param1.py
Hallo, Welt!
Hallo, Welt!
Hallo, Welt!
Hallo, Welt!
```

Man gibt hierbei in der Parameterliste einen voreingestellten Wert an, den man beim Funktionsaufruf auch überschreiben kann. Hat man mehrere Parameter, kann man einzelne von ihnen vorbelegen und im konkreten Aufruf auch vertauschen:

```
#!/usr/bin/python

def summe(a=3, b=2):
    return a + b

print summe()
print summe(a=4)
print summe(b=4)
print summe(b=2, a=1)
```

Ausgabe

```
user@localhost:~$ ./param2.py
5
6
7
3
```

Variable Parameter

Gerade bei Funktionen wie der Summenberechnung ist es praktisch, eine variable Anzahl an Parametern zu haben. Dadurch werden recht praktische Funktionen möglich, und das schreiben neuer Funktionen für jede Anzahl an Parametern entfällt:

```
#!/usr/bin/python

def summe(*list):
    s = 0
    for element in list:
        s += element
    return s

print summe()
print summe(1)
print summe(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Ausgabe

```
user@localhost:~$ ./varparam1.py
0
1
45
```

Der Parameter **list* ist hierbei ein Tupel, das abhängig von der Anzahl der im Aufruf erfolgten Argumente entweder leer (*()*) ist, oder die Argumente *(1)* und *(1, 2, 3, 4, 5, 6, 7, 8, 9)* enthält. Anschließend brauchen wir zur Summenberechnung nur noch über dieses Tupel zu iterieren.

Neben der Tupel-Form variabler Argumente gibt es noch die Dictionary-Form:

```
#!/usr/bin/python

def ZeigeListe(**liste):
    print liste

ZeigeListe(a1=1, a2=2, a3=3)
ZeigeListe(Name="Schmitz", Vorname="Elke", Postleitzahl=44444)
```

Ausgabe

```
user@localhost:~$ ./varparam2.py
{'a1': 1, 'a3': 3, 'a2': 2}
{'Name': 'Schmitz', 'Vorname': 'Elke', 'Postleitzahl': 44444}
```

Hier wird lediglich ein Dictionary aufgebaut, welches jeweils aus dem Argumentenwort und -wert besteht.

Globale und lokale Variablen

Haben funktionslokale Variablen den gleichen Namen wie Variablen, die außerhalb der Funktion definiert wurden, so werden globalere Variablenwerte weder lesend noch schreibend beim Zugriff berührt:

```
#!/usr/bin/python

wert = 42
print wert

def wertetest():
    wert = 12
    print wert

wertetest()
print wert
```

Ausgabe

```
user@localhost:~$ ./global1.py
42
12
42
```

Neue Variablen werden so lokal es geht erzeugt. Hat eine neue Variable innerhalb einer Funktion den gleichen Namen wie eine andere Variable außerhalb, so wird nur die innere Variable genutzt.

Möchte man es anders haben, muss man explizit den Zugriff auf die globale Variable anfordern:

```
#!/usr/bin/python

wert = 42
print wert

def wertetest():
    global wert
    wert = 12
    print wert

wertetest()
print wert
```

Ausgabe

```
user@localhost:~$ ./global2.py
42
12
12
```

Das Schlüsselwort global sorgt hier für den Zugriff auf die außerhalb der Funktion definierte globale Variable. Bitte beachten Sie, dass Zugriffe auf globale Variablen die Lesbarkeit des Codes vermindern.

Funktionen auf Wertemengen

Hat man eine Funktion geschrieben, die ein einzelnes Argument verarbeitet und möchte diese Funktion nun auf eine ganze Liste von Werten anwenden, so bietet sich die Funktion map an. Diese nimmt ein Funktionsargument wie auch eine Liste auf, wendet die Funktion auf jedes Element dieser Liste an und gibt eine Liste als Ergebnis zurück. Folgendes Beispiel verdeutlicht dies:

```
#!/usr/bin/python

def quadriere(x):
    return x * x

quadratzahlen = map(quadriere, [1, 2, 3, 4, 5, 6])
print quadratzahlen
```

Ausgabe

```
user@localhost:~$ ./map1.py
[1, 4, 9, 16, 25, 36]
```

Die Funktion *quadriere()* berechnet für jedes Element der Liste von 1 bis 6 die Quadratzahl und gibt eine Liste mit Quadratzahlen zurück. Selbstverständlich kann dieses konkrete Problem auch mit Hilfe einer for-Schleife gelöst werden, was man benutzt ist meist mehr eine Geschmacksfrage.

lambda

Eine mit lambda erzeugte Funktion ist anonym, sie hat keinen Namen und wird nur in einem bestimmten Kontext genutzt, wie zum Beispiel mit map:

```
#!/usr/bin/python

quadratzahlen = map(lambda x: x * x, [1, 2, 3, 4, 5, 6])
print quadratzahlen
```

Ausgabe

```
user@localhost:~$ ./lambda1.py
[1, 4, 9, 16, 25, 36]
```

Nach dem Schlüsselwort lambda folgt bis zum Doppelpunkt eine durch Kommata getrennte Aufzählung von Argumenten, hinter dem Doppelpunkt beginnt die Anweisung. lambda-Funktionen lassen sich auch nutzen, um während des Programmlaufes neue Funktionen zu erzeugen. Das folgende Beispiel demonstriert, wie eine Quadrat- und eine Wurzelfunktion neu erzeugt werden:

```
#!/usr/bin/python

def Exponential(z):
    return lambda x: x**z

quadriere = Exponential(2)
wurzel = Exponential(0.5)

a = quadriere(2)
print a
```

```
b = wurzel(a)
print b
```

Ausgabe

user@localhost:~$ ./lambda2.py
4
2.0

Die Funktion *Exponential()* erwartet ein Argument, mit dem die lambda-Funktion erzeugt wird. Die Funktion gibt nicht etwa den Wert dieser neuen Funktion zurück, sondern die neue Funktion selbst. So erzeugt *quadriere = Exponential(2)* eine neue Quadratfunktion, die man auch sogleich anwenden kann.

Listen erzeugen sich selbst

Wo wir gerade dabei waren, mit map() Listen zu erzeugen, wird vielleicht auch folgende Syntax[1] etwas für Sie sein. Lehnen Sie sich zurück und genießen Sie die unglaubliche Vorstellung, wie eine Liste sich selbst erzeugt:

```
#!/usr/bin/python

liste = [x * x for x in xrange(1, 10)]
print liste
```

Ausgabe

user@localhost:~$ ./comprehension1.py
[1, 4, 9, 16, 25, 36, 49, 64, 81]

Diese Liste wird aufgebaut, indem alle Werte, die xrange() liefert, quadriert werden. Wir haben es hier also wieder mit einer Liste von Quadratzahlen zu tun. Anders als bei der for-Schleife steht hier der Funktionskörper vor dem Schleifeniterator. Diese Code ist aber noch nicht alles, was wir Ihnen bieten können:

```
#!/usr/bin/python

liste = [x * x for x in xrange(1, 10) if x % 2 == 0]
print liste
```

Ausgabe

user@localhost:~$ ./comprehension2.py
[4, 16, 36, 64]

Nun haben wir es mit einer Liste von Quadratzahlen zu tun, die aus der Menge der geraden Zahlen gebildet wurden. Das in der Liste nachgestellte if sorgt hier für eine Auswahl der Werte, die in die Vorschrift zur Listenbildung übernommen werden.

Um alle 3er-Tupel einer Liste auszugeben, also alle Kombinationen einer 3-Elementigen Liste aufzuzählen, dient folgendes Programm:

```
#!/usr/bin/python

Liste = ['1', '2', '+']
Kreuz = [(a, b, c)  for a in Liste for b in Liste for c in Liste]
print Kreuz
```

Ausgabe

```
user@localhost:~$ ./comprehension3.py
[('1', '1', '1'), ('1', '1', '2'), ('1', '1', '+'), ('1', '2', '1'), ('1', '2', '2'), ('1', '2', '+'), ('1', '+', '1'), ('1', '+', '2'), ('1', '+', '+'), ('2', '1', '1'), ('2', '1', '2'), ('2', '1', '+'), ('2', '2', '1'), ('2', '2', '2'), ('2', '2', '+'), ('2', '+', '1'), ('2', '+', '2'), ('2', '+', '+'), ('+', '1', '1'), ('+', '1', '2'), ('+', '1', '+'), ('+', '2', '1'), ('+', '2', '2'), ('+', '2', '+'), ('+', '+', '1'), ('+', '+', '2'), ('+', '+', '+')]
```

Wie wir sehen, können wir Listen aus Tupel aus anderen Listen erzeugen.

Solche Listen können auch mit der Funktion filter erzeugt werden. Dieser übergibt man eine Funktion, die für Argumente bool'sche Werte zurückliefert und eine Liste, über die iteriert werden soll. Zurück erhält man eine Liste mit all jenen Werten, für die die Funktion True liefert:

```
#!/usr/bin/python

def durch3teilbar(x):
  return x % 3 == 0

print filter(durch3teilbar, range(10))
```

Ausgabe

```
user@localhost:~$ ./comprehension4.py
[0, 3, 6, 9]
```

Die Funktion durch3teilbar() ergibt True für alle Werte, die durch 3 teilbar sind. Nur noch diese Werte verbleiben in der übergebenen Liste.

Generatoren

Funktionen wie range() und xrange() erzeugen Objekte, über die sich iterieren lässt, im einfachsten Fall Listen oder Tupel. Eine andere Art iterierbarer Objekte sind Generatoren, um die es hier geht.

Generatoren mit yield

Ein Generator wird wie eine Funktion erzeugt und erstellt eine Folge von Werten. Einen Generator kennen Sie bereits, nämlich xrange(). Die Folge wird Elementeweise bereitgestellt mit yield():

```
#!/usr/bin/python

def MeinGenerator():
    yield(1)
    yield(2)
    yield(3)

for i in MeinGenerator():
    print i
```

Ausgabe

```
user@localhost:~$ ./yield1.py
1
2
3
```

yield() liefert beim ersten Aufruf des Generators den ersten Wert zurück und stoppt dann die Ausführung. Es wird hierbei also keine Liste erzeugt, sondern jeder Zugriff auf den Generator liefert den nächsten von yield()

bereitgestellten Wert.

Werte in Generatoren lassen sich bequem in for-Schleifen erzeugen:

```
#!/usr/bin/python

def rueckwaerts(text):
    length = len(text)
    for i in xrange(length):
        yield(text[length - i - 1])

for c in rueckwaerts("Hallo, Welt!"):
    print "\b%c" % c,

print
```

Ausgabe

```
user@localhost:~$ ./yield2.py
!tleW ,ollaH
```

Hier wird der Text rückwärts ausgegeben, wobei yield() angefangen vom letzten Zeichen jedes Zeichen des Textes zurückliefert. Da das Komma bei der print-Anweisung ein Leerzeichen einfügt, müssen wir mit einem Backspace (*\b*) dafür sorgen, dass dieses wieder entfernt wird.

Tipp:
Die oben benutzte Funktion dient als einfaches Beispiel zur Demonstration von Generatoren. Wenn Sie sich aber fragen, wie eine Zeichenkette einfach rückwärts dargestellt werden kann, dann hängen Sie [::-1] an eine String-Variable oder an ein String-Literal. Das gleiche Ergebnis wie oben wäre über "Hallo, Welt!"[::-1] ebenfalls möglich.

Generatorexpressions

Auch für Generatoren gibt es wieder eine abkürzende Schreibweise:

```
#!/usr/bin/python

genex = (i * i for i in xrange(5))

for wert in genex:
    print wert
```

Ausgabe

```
user@localhost:~$ ./genex1.py
0
1
4
9
16
```

Die Syntax ist ähnlich wie bei List Comprehensions, jedoch werden runde Klammern verwendet.

Zusammenfassung

Wir haben gezeigt, wie man Funktionen definiert, Werte zurückgibt und Funktionen mit variablen Parameterlisten schreibt. Der Gebrauch von lokalen und globalen Variablen wurde erläutert wie auch die Anwendung der Funktionen auf Listen mit Hilfe von map(). Als *Syntaxzucker* gaben wir einen Einblick in anonyme Funktionen. List Comprehensions und Generatoren rundeten das Thema ab.

Anmerkungen

[1] Diese Syntax nennt man " **List Comprehension** "

Module

Module dienen dazu, zusammengehörige Funktionen und Klassen in Dateien zu gruppieren. Auf diese Weise lassen sich größere Quelltextsammlungen thematisch organisieren.

Aufbau eines Modules

Ein Modul besteht aus einer Folge von Quelltextzeilen. Da Module nicht ausgeführt werden müssen, sparen wir uns die bei den bisherigen Programmen benutzte Interpreterdeklaration in der ersten Zeile:

```
def HalloWelt():
    print "Hallo, Welt!"

def addiere(a, b):
    return a + b
```

Ein Programm, welches dieses Modul nutzt könnte wie folgt aussehen:

```
#!/usr/bin/python
import modul1

print modul1.addiere(1, 2)
```

Ausgabe

```
user@localhost:~$ ./mod1.py
3
```

Der Präfix vor den Funktionen, die man aus einem Modul aufruft, nennt man **Namensraum**. Man kann beliebig viele Namensräume durch Module erzeugen, und kommt so bei der Benennung von Funktionen nicht durcheinander, wenn zwei Funktionen den selben Namen tragen.

Es kann auf alle vom Modul exportierten Elemente zugegriffen werden; es gibt keine privaten oder besonders geschützte Funktionen oder Variablen.

Importieren im Detail

Folgendes Modul kann auf verschiedene Weisen eingebunden werden:

```
def multipliziere(a, b):
    return a * b

def print_mult(a, b):
    print multipliziere(a, b)
```

Es kann bei längeren Modulnamen umständlich erscheinen, immer wieder bei jeder importierten Funktion den Modulnamen davorzuschreiben. Eine Abkürzung bietet das lokale Umbenennen:

```
#!/usr/bin/python
import modul2 as X

X.print_mult(3, 3)
```

Ausgabe

```
user@localhost:~$ ./mod2.py
9
```

Hier wird das Modul *modul2* unter dem Namen *X* bekanntgemacht. Auf alle Funktionen des Modules kann ab sofort mit dem neuen Namen zugegriffen werden.

Den neuen Namensraum kann man aber auch so importieren, dass man auf die Angabe des Namensraumes verzichten kann:

```
#!/usr/bin/python
from modul2 import *

print_mult(3, 3)
```

Ausgabe

```
user@localhost:~$ ./mod3.py
9
```

Es werden alle Funktionen aus dem Modul importiert, als würden sie lokal vereinbart worden sein. Der Stern dient dabei als Platzhalter für alle im Modul vorliegenden Deklarationen.

Einzelne Funktionen aus dem Modul lassen sich über die explizite Angabe der Funktionen in der import-Anweisung einbetten, alle anderen Deklarationen sind unbekannt:

```
#!/usr/bin/python
from modul2 import print_mult

print_mult(3, 3)
```

Ausgabe

```
user@localhost:~$ ./mod4.py
9
```

Bindet man aus einem Modul mehrere Funktionen ein, so sind sie mit Kommata zu separieren, beispielsweise:

```
#!/usr/bin/python
from modul2 import print_mult, multipliziere.
```

Namensräume haben den Vorteil, dass man mehrere Funktionen haben kann, die den gleichen Namen tragen. Durch die Nennung des Namensraumes in der import-Anweisung ist immer klar, welche Funktion man meint. Diesen Vorteil verliert man, wenn man den *Sternchenimport* benutzt.

Inhalt von Modulen

Module enthalten von sich aus schon eine Reihe vorbelegter Variablen, die man nutzen kann. Das folgende Beispiel zeigt den Zugriff auf zwei solcher Variablen, nämlich __name__ und __doc__. Die Variable __name__ enthält den Namen der aktuell ausgeführten Funktion, dies ist bei Modulen **__main__**. __doc__ hingegen enthält einen Dokumentationsstring, sofern dieser am Anfang des Modules festgelegt wurde:

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

"""Dieses Modul enthält Funktionen
rund um erste mathematische Anweisungen. """

def addiere(a, b):
    return a + b

def multipliziere(a, b):
    return a * b

print __name__
print __doc__
```

Ausgabe

```
user@localhost:~$ ./modinhalt1.py
__main__
Dieses Modul enthält Funktionen
rund um erste mathematische Anweisungen.
```

Beachten Sie bitte, dass dieses Modul Ausführungsrechte benötigt. Nutzen kann man diese Konstanten, um eine Dokumentation ausgeben zu lassen oder den Inhalt des Moduls mit der Funktion dir() anzeigen zu lassen:

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

"""Dieses Modul enthält Funktionen
rund um erste mathematische Anweisungen. """

def addiere(a, b):
    return a + b

def multipliziere(a, b):
    return a * b

if __name__ == "__main__":
    print __doc__
    print dir()
```

Ausgabe

```
user@localhost:~$ ./modinhalt2.py
Dieses Modul enthält Funktionen
rund um erste mathematische Anweisungen.
['__builtins__', '__doc__', '__file__', '__name__', 'addiere', 'multipliziere']
```

Wenn dieses Modul aufgerufen wird, dann gibt es den Docstring aus, gefolgt von dem Inhalt des Moduls. Mit dieser Technik verhindert man, dass versehentlich das falsche Modul ausgeführt wird und man keine Programmausgabe erhält. Außerdem ist es bequem, so auf seine Module zugreifen zu können. dir() gibt den gesamten Inhalt des Moduls aus. Neben den Namen der Funktionen werden einige Variablen, die automatisch erzeugt wurden, aufgeführt. dir() können Sie auf alle Objekte, also auch Funktionen, Strings, Zahlen, Listen und so fort anwenden.

Ausserhalb des Moduls, kann man auf diese Modulvariablen ebenfalls zugreifen, wobei dann ihr Inhalt anders lautet. Zum Beispiel gibt:

```
#!/usr/bin/python
import modul3b

print modul3b.__name__
```

den String **modul3b** aus. Die Anweisungen aus diesem Modul werden nicht berührt, da das Modul ja nicht ausgeführt wird.

Pakete

Pakete sind (thematische) Zusammenfassungen von Modulen in einer Verzeichnisstruktur. Hierbei werden mehrere Modul-Dateien hierarchisch gegliedert in einem Verzeichnis abgelegt. Hinzu kommt pro Verzeichnis-Ebene eine Initialisierungsdatei. Diese Initialisierungsdatei enthält für diese Ebene die gemeinsame Dokumentation und übernimmt Initialisierungsaufgaben. Der Inhalt dieser Datei wird beim Import gelesen und ausgeführt. Es ist hierbei nicht notwendig, dieser Datei Ausführungsrechte zu geben. Ein typischer Paket-Verzeichnisbaum kann folgendermaßen aussehen:

```
mathe/
mathe/__init__.py

mathe/addiere/
/mathe/addiere/addiere.py
/mathe/addiere/__init__.py

mathe/subtrahiere/
/mathe/subtrahiere/subtrahiere.py
/mathe/subtrahiere/__init__.py
```

Die Initialisierungsdatei kann leer sein oder folgenden Aufbau haben (mathe/addiere/__init__.py):

```
"""Die hiesigen Module dienen zum Addieren von Zahlen"""

print "Initialisierung von mathe/addiere/*"
```

Das war es schon. Die print-Anweisung wird ausgeführt, wenn Module aus dieser Ebene importiert werden.

Der Aufbau von Modulen unterscheidet sich hierbei nicht von dem oben gesagten. Der Vollständigkeit fügen wir hier noch ein Modul ein (mathe/addiere/addiere.py):

```
def add(a, b):
    return a+b
```

Ein Programm, welches Module aus einem Paket benutzt sieht so aus:

```
#!/usr/bin/python

import mathe.addiere.addiere

print mathe.__doc__
print "-"
print mathe.addiere.__doc__

print mathe.addiere.addiere.add(2, 2)
```

Ausgabe

```
user@localhost:~$ ./paket1.py
Initialisierung von mathe/addiere/*
Dieses Paket enthält einige Mathematik-Module
-
Die hiesigen Module dienen zum Addieren von Zahlen
4
```

Dieses Programm gibt die Dokumentation der Module aus und benutzt eine in einem Paketmodul definierte Funktion. Beachten Sie bitte, dass die Ausgabe beim Testen der hier vorgestellten Dateien von unserer Darstellung abweicht. Der Grund ist, dass wir die Datei mathe/__init__.py nicht vorgestellt haben.

Paketmodule erkennt man an ihrer *Punkt-Darstellung* beim Import. Es wird hier aus dem mathe-Paket im Verzeichnis addiere/ das Python-Modul addiere.py importiert und daraus die Funktion add() aufgerufen. Man kann pro Verzeichnisebene mehrere Module haben.

Pakete werden ihnen im weiteren Verlauf des Buches an vielen Stellen begegnen.

Zusammenfassung

Module unterscheiden sich in ihrem Aufbau nicht von anderen Python-Dateien. Sie gliedern zusammengehörige Codeteile und können auf mehrere Weisen eingebunden werden. Eine reine import-Anweisung erzeugt einen neuen Namensraum, von denen man nie genug haben kann. Module kann und sollte man mit einem Docstring versehen. Es gibt viele vorbelegte Variablen rund um Module, von denen wir einige hier kennengelernt haben. Module kann man zu Paketen gruppieren.

Rund um OOP

Objektorientierte Programmierung ist die Vereinheitlichung der Konzepte

- **Kapselung**: Ein Objekt fasst alle benötigten Bestandteile zusammen, und verbirgt vor dem Nutzer den internen Aufbau. Stattdessen werden öffentliche Schnittstellen exportiert, auf die der Nutzer zugreifen kann.
- **Vererbung**: Ein Objekt kann Eigenschaften einer anderen Klasse im definierten Umfang erben. Einzelne Teile können dabei überschrieben werden.
- **Polymorphie**: Bezeichnet das Überladen von Operatoren und Methoden um eine einheitliche Schnittstelle für verschiedene Datentypen und Parameter zu haben.

Die typische Darstellung eines Objektes ist in Form einer Klasse. Funktionen, die innerhalb einer Klasse vereinbart werden, nennt man **Methoden**, Klassenvariablen nennt man **Attribute**. Wird ein Objekt erzeugt, so kann es unter Umständen eine Aktion wie das Initialisieren aller Daten ausführen. So eine Methode nennt man **Konstruktor**. Klassen speichert man typischerweise in eigenen Modulen.

Aufbau einer Klasse

Die einfachste Klasse tut gar nichts und enthält keinerlei Deklarationen:

```
#!/usr/bin/python

class TuNichts:
    pass

objekt = TuNichts()
print dir(objekt)
print "Typ:", type(objekt)
```

Ausgabe

```
user@localhost:~$ ./klasse1.py
['__doc__', '__module__']
Typ: <type 'instance'>
```

Gefolgt vom Schlüsselwort class steht der Name der Klasse. Ein Objekt erzeugt man, in dem es aufgerufen wird. pass steht für: *Tue wirklich gar nichts*. Der Inhalt dieses Objektes kann mit dir() angezeigt werden. Diesen Typ Klasse nennt man **Classic Classes**. Sie sind mittlerweile veraltet, aber immer noch brauchbar und vor allem aus Gründen der Kompatibilität Standard. Hier das gleiche Beispiel als **New-style Class**:

```
#!/usr/bin/python

class TuNichts(object):
    pass

objekt = TuNichts()
print dir(objekt)
print "Typ:", type(objekt)
```

Ausgabe

```
user@localhost:~$ ./klasse2.py
['__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__', '__weakref__']
Typ: <class '__main__.TuNichts'>
```

Die Elternklassen werden in Klammern nach dem Klassennamen notiert. Wenn es mehrere Elternklassen sind, werden sie durch Kommata separiert. Unsere Klasse *TuNichts* erbt nun die Eigenschaften von *object*. Zur Vererbung kommen wir noch. Klassen neuen Typs haben einige Vorteile gegenüber Klassen alten Typs, wie zum Beispiel, dass sie sich nahtlos ins Typenkonzept von Python einfügen und einige typische Methoden mitbringen. Alle weiteren Beispiele in diesem Kapitel werden mit New-style-Klassen besprochen, die alte Variante sollten Sie aber wenigstens kennen.

Etwas mehr kann schon folgende Klasse:

```
#!/usr/bin/python

class TuEtwas(object):
    def __init__(self, x):
        self.x = x

    def printX(self):
        print self.x

objekt = TuEtwas(42)
objekt.printX()
```

Ausgabe

```
user@localhost:~$ ./klasse3.py
42
```

Diese Klasse enthält zwei Methoden, nämlich *__init__()* und *printX()*. *__init__()* wird aufgerufen, wenn ein Objekt der Klasse angelegt wird. Diese Methode muss nicht explizit aufgerufen werden. Das Argument self bezieht sich auf die Klasse selbst. *self.x* ist also eine Variable, die in der Klasse angelegt ist (Attribut), nicht als lokale Variable in den Methoden. Ruft eine Methode eine andere auf, so muss ebenfalls self vor den Methodenaufruf geschrieben werden.

Die folgende Klasse implementiert eine physikalische Größe, die Temperatur. Temperaturen werden in Grad Fahrenheit, Kelvin oder Grad Celsius gemessen, wobei wir nur die letzten beiden Einheiten berücksichtigen. Die Klasse speichert alle Temperaturen in Kelvin.

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

class Temperatur(object):
    def __init__(self, Wert, Einheit = "K"):
        if Einheit == "K":
            self.temperatur = Wert
            self.einheit = Einheit
        elif Einheit == "°C":
            self.temperatur = Wert + 273
            self.einheit = "K"
        else:
            self.temperatur = 0
            self.einheit = "K"

    def umrechnen(self, Einheit = "K"):
        if Einheit == "K":
            return self.temperatur
        elif Einheit == "°C":
            return self.temperatur - 273

    def ausgeben(self, Einheit = "K"):
        if Einheit in ("K", "°C"):
            print "Die Temperatur beträgt %.2f %s" %
```

```
(self.umrechnen(Einheit), Einheit)
        else:
            print "unbekannte Einheit"


T1 = Temperatur(273, "K")
T1.ausgeben("°C")
T1.ausgeben("K")
T1.ausgeben("°F")
```

Ausgabe

```
user@localhost:~$ ./klasse4.py
Die Temperatur beträgt 0.00 °C
Die Temperatur beträgt 273.00 K
unbekannte Einheit
```

Die Klasse enthält drei Methoden, nämlich *__init__()*, den Konstruktor, *umrechnen()*, eine Methode, die von Kelvin in andere Einheiten umrechnen kann und die Methode *ausgeben()*, welche die aktuelle Temperatur als String ausgibt. Die Attribute dieser Klasse sind *temperatur* und *einheit*. In dieser Klasse sind alle Attribute öffentlich sichtbar und veränderbar.

Privat - mehr oder weniger

Um gegenüber den Nutzern einer Klasse anzudeuten, dass bestimmte Methoden oder Attribute privat sind, also nicht nach außen exportiert werden sollen, stellt man ihnen einen Unterstrich (_) vorran. Echt privat sind solcherart gekennzeichneten Attribute nicht, es handelt sich hierbei mehr um eine textuelle Vereinbarung, wie folgendes Beispiel zeigt:

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

class Quadrat(object):
    def __init__(self, kantenlaenge = 0.0):
        if kantenlaenge < 0.0:
            self._kantenlaenge = 0.0
        else:
            self._kantenlaenge = kantenlaenge

    def flaeche(self):
        return self._kantenlaenge * self._kantenlaenge

Q1 = Quadrat(10)
print Q1.flaeche()
Q1._kantenlaenge = 4
print Q1.flaeche()
```

Ausgabe

```
user@localhost:~$ ./privat1.py
100
16
```

Es kann hier der Kantenlänge ein Wert zugewiesen werden, was vom Autor der Klasse sicher nicht beabsichtigt war. Das Attribut *_kantenlaenge* ist nur durch die Vereinbarung geschützt, an die man sich als Nutzer einer Klasse im Allgemeinen halten sollte.

Um noch stärker anzudeuten, dass Attribute privat sind, werden zwei Unterstriche verwendet. Auch hier kann man der Kantenlänge einen Wert zuweisen, er wird jedoch nicht berücksichtigt:

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

class Quadrat(object):
    def __init__(self, kantenlaenge = 0.0):
        if kantenlaenge < 0.0:
            self.__kantenlaenge = 0.0
        else:
            self.__kantenlaenge = kantenlaenge

    def flaeche(self):
        return self.__kantenlaenge * self.__kantenlaenge

Q1 = Quadrat(10)
print Q1.flaeche()
Q1.__kantenlaenge = 4
print Q1.flaeche()
```

Ausgabe

```
user@localhost:~$ ./privat2.py
100
100
```

In beiden Fällen ist die Fläche immer gleich, der Wert der Kantenlänge verändert sich trotz Zuweisung nicht. Das liegt daran, dass es *Q1.__kantenlaenge* gar nicht gibt. Bei der Zuweisung *Q1.__kantenlaenge = 4* wird ein neues Klassenattribut erzeugt. Das eigentlich als privat deklarierte Attribut *__kantenlaenge* versteckt sich nun hinter dem Ausdruck *_Quadrat__kantenlaenge*. Dies können Sie leicht mit Hilfe der dir()-Funktion selbst überprüfen:

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

class Quadrat(object):
    def __init__(self, kantenlaenge = 0.0):
        if kantenlaenge < 0.0:
            self.__kantenlaenge = 0.0
        else:
            self.__kantenlaenge = kantenlaenge

    def flaeche(self):
        return self.__kantenlaenge * self.__kantenlaenge


Q1 = Quadrat(10)
```

```
print "Die Fläche beträgt: ", Q1.flaeche()
Q1.__kantenlaenge = 4
print "Q1.__kantenlaenge: ", Q1.__kantenlaenge
print "Die echte Kantenlänge: ", Q1._Quadrat__kantenlaenge
print "Q1 hat folgenden Inhalt:\n", dir(Q1)
```

Ausgabe

```
user@localhost:~$ ./privat3.py
Die Fläche beträgt: 100
Q1.__kantenlaenge: 4
Die echte Kantenlänge: 10
Q1 hat folgenden Inhalt:
['_Quadrat__kantenlaenge', '__class__', '__delattr__', '__dict__', '__doc__', '__getattribute__', '__hash__', '__init__', '__kantenlaenge', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__str__', '__weakref__', 'flaeche']
```

Wie Sie sehen, haben wir mit der Zuweisung *Q1.__kantenlaenge = 4* ein neues Attribut erzeugt, denn offenbar wurde dieses ja nicht für die Flächenberechnung herangezogen.

Details

Machen Sie sich bitte keine allzu großen Gedanken über private Attribute und Methoden. Der *Python way of coding* ist da weniger streng als andere Sprachen. Stellen sie allem, was nicht nach außen hin sichtbar sein soll, einen Unterstrich voran, dann ist es privat genug. Diese Vereinbarung reicht Python-Programmierern im allgemeinen aus.

Getter und Setter

In realen Programmen ist es sehr wichtig, einen konsistenten Zugriff auf die Klasse zu haben, so dass bei der Änderung von Attributen auch weiterhin alle Klassenattribute korrekte Werte haben. Probleme in diesem Bereich entstehen oft dadurch, dass Nutzer einer Klasse die Implementation nicht kennen oder kennen sollen. Der Zugriff auf Attribute erfolgt dann durch spezielle Methoden, die lediglich Attributwerte modifizieren oder zurückgeben. Diese nennt man *getter* und *setter*, sie holen oder setzen Werte. In objektorientierten Programmiersprachen, die keinen direkten Zugriff auf Klassenattribute ermöglichen, findet man daher oft triviale Methoden vor, die lediglich Werte setzen oder zurückgeben. Auch in folgendem Beispiel verwenden wir zwei solche eigentlich überflüssigen *getter*. Sie dienen dazu, Ihnen das Konzept vorzuführen und auf weitere Details später einzugehen.

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

class Quadrat(object):
    def __init__(self, kantenlaenge = 0.0):
        self.set_kantenlaenge(kantenlaenge)

    def _berechne_flaeche(self):
        self.__flaeche = self.__kantenlaenge * self.__kantenlaenge

    def get_kantenlaenge(self):
        return self.__kantenlaenge

    def set_kantenlaenge(self, kantenlaenge):
        if kantenlaenge < 0.0:
```

```
            self.__kantenlaenge = 0.0
        else:
            self.__kantenlaenge = kantenlaenge
        self._berechne_flaeche()

    def get_flaeche(self):
        return self.__flaeche

Q1 = Quadrat(10)
print Q1.get_flaeche()
Q1.set_kantenlaenge(12)
print Q1.get_flaeche()
```

Ausgabe

```
user@localhost:~$ ./getset1.py
100
144
```

Hier ist die Fläche ein weiteres Attribut, welches durch die Methode *_berechne_flaeche()* berechnet wird. Bei jedem Aufruf, der die Kantenlänge ändert, wird die Fläche neu bestimmt. In dieser Klasse gibt es zwar die Möglichkeit, die Kantenlänge zu modifizieren, nicht jedoch die Fläche. Ein direkter Zugriff auf die Attribute sollte dringend unterbleiben, deswegen wurden diese als *stark privat* gekennzeichnet. Möchte man sicher gehen, dass jeder Zugriff auf Attribute über die get- und set-Methoden abgewickelt wird, jedoch trotzdem auf den bequemen Zugriff per Objektvariablen zugreifen, so kann die Funktion property() helfen. Sie akzeptiert mindestens eine get- und set-Methode und liefert einen Namen für den Zugriff auf das Attribut:

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

class Quadrat(object):
    def __init__(self, kantenlaenge):
        self.set_kantenlaenge(kantenlaenge)

    def _berechne_flaeche(self):
        self._flaeche = self._kantenlaenge * self._kantenlaenge

    def get_kantenlaenge(self):
        return self._kantenlaenge

    def set_kantenlaenge(self, kantenlaenge):
        if kantenlaenge < 0.0:
            self._kantenlaenge = 0.0
        else:
            self._kantenlaenge = kantenlaenge
        self._berechne_flaeche()

    def get_flaeche(self):
        return self._flaeche
```

```
    kantenlaenge = property(get_kantenlaenge, set_kantenlaenge)
    flaeche = property(get_flaeche)


Q1 = Quadrat(12)
print "Kantenlänge = ", Q1.kantenlaenge, " Fläche = ", Q1.flaeche
Q1.kantenlaenge = 9
print "Kantenlänge = ", Q1.kantenlaenge, " Fläche = ", Q1.flaeche
```

Ausgabe

```
user@localhost:~$ ./getset2.py
Kantenlänge = 12 Fläche = 144
Kantenlänge = 9 Fläche = 81
```

In diesem Code haben wir alle interessanten Getter und Setter eingefügt und am Ende der Klasse miteinander verwebt. Für das Attribut *kantenlaenge* sind also die Methoden *get_kantenlaenge()* und *set_kantenlaenge()* verantwortlich, die ihrerseits auf Attribute wie *self._kantenlaenge* zugreifen. Das Attribut *flaeche* hingegen kann nur gelesen werden, da der Setter für dieses Attribut fehlt. Properties werden innerhalb der Klasse, jedoch außerhalb aller Methoden deklariert.

Statische Methoden

Mit statischen Methoden lassen sich Methodensammlungen anlegen, die sich nicht auf das jeweilige Objekt beziehen, sondern allgemeingültig formuliert sind. Damit nutzen sie auch Anwendern der Klasse, die kein Klassenobjekt benötigen.

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

class Rechteck(object):
    def __init__(self, kante1, kante2):
        self._kante1 = kante1
        self._kante2 = kante2

    def berechne_flaeche(a, b):
        return a * b

    flaeche = staticmethod(berechne_flaeche)

print Rechteck.flaeche(3, 5)
```

Ausgabe

```
user@localhost:~$ ./statisch1.py
15
```

Hier wird eine solche statische Methode namens *flaeche()* definiert. Die dahinterliegende Methode *berechne_flaeche* darf kein self-Argument mitführen, denn dieses bezieht sich ja gerade auf die Instanz der Klasse selbst. Es braucht ebenfalls kein Objekt von Typ *Rechteck* deklariert zu werden.

Selbiges Beispiel funktioniert auch noch etwas einfacher - wer hätte das gedacht. Hierbei bedienen wir uns einer so genannten **Funktionsdekoration**[1].

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

class Rechteck(object):
    def __init__(self, kante1, kante2):
        self._kante1 = kante1
        self._kante2 = kante2

    @staticmethod
    def berechne_flaeche(a, b):
        return a * b


print Rechteck.berechne_flaeche(3, 5)
```

Ausgabe

```
user@localhost:~$ ./statisch2.py
15
```

@staticmethod ist so eine **Funktionsdekoration**. Vor eine Methode geschrieben bewirkt sie, dass die nun folgende Methode als statisch anzusehen ist, also auch unabhängig vom erzeugten Objekt nutzbar ist.

Polymorphie

Polymorphie - das Überladen von Operatoren und Methoden - funktioniert in einem recht begrenzten Umfang. Einige Beispiele haben Sie dazu schon im Kapitel über Funktionen, Variable Parameter, kennengelernt. In diesem Abschnitt wollen wir uns daher auf das Überladen von Operatoren beschränken. Das folgende Programm überlädt die Operatoren für die Addition und die Multiplikation. Darüberhinaus ermöglicht uns das Programm, den Absolutbetrag eines Objektes zu bestimmen wie auch dieses Objekt in einen String zu verwandeln. Das Programmbeispiel implementiert dabei die physikalische Größe der Kraft, die in diesem Fall aus zwei Komponenten besteht, einer horizontalen Richtung und einer vertikalen.

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

import math

class Kraft(object):
    def __init__(self, kx, ky):
        self._kx = kx
        self._ky = ky

    def __add__(self, F):
        x, y = F.get()
        return Kraft(self._kx + x, self._ky + y)

    def __mul__(self, zahl):
        return Kraft(self._kx * zahl, self._ky * zahl)
```

```
    def get(self):
        return (self._kx, self._ky)

    def __abs__(self):
        f2 = self._kx ** 2 + self._ky ** 2
        return math.sqrt(f2)

    def __str__(self):
        return "Die Kraft ist (%.2f, %.2f), die Länge ist %.2f" %
(self._kx, self._ky, abs(self))


F1 = Kraft(3, 4)
F2 = F1 * 3
print F1
print F2
print F1 + F2
```

Ausgabe

```
user@localhost:~$ ./poly11.py
Die Kraft ist (3.00, 4.00), die Länge ist 5.00
Die Kraft ist (9.00, 12.00), die Länge ist 15.00
Die Kraft ist (12.00, 16.00), die Länge ist 20.00
```

Wann immer ein Objekt mit einem anderen multipliziert werden soll, wird die entsprechende __mul__()-Methode des Objektes aufgerufen. Dies machen wir uns hier zu Nutze und definieren diese Methode einfach um. Selbiges geschieht bei der Addition, hier wird __add__() umdefiniert. Beide Funktionen geben ein neues Kraftobjekt zurück. Die __str__()-Methode wird immer aufgerufen, wenn ein Objekt in einen String umgewandelt werden soll, die Methode __abs__() wird aufgerufen, wenn der Absolutbetrag eines Objektes angefragt wird. Der Absolutbetrag unseres Objektes ist schlicht seine Länge. Es gibt noch mehr Operatoren und sonstige Klassenmethoden, die man individuell für Klassen definieren kann. Hierzu finden Sie im Internet Anleitungen.

Vererbung

Mit Vererbung lassen sich aus recht allgemeinen Klasse spezialisierte Klassen erzeugen, die alle Eigenschaften der Elternklasse besitzen. Folgendes Beispiel verdeutlicht das Vorgehen:

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

class Rechteck(object):
    def __init__(self, seite1, seite2):
        self._seite1 = seite1
        self._seite2 = seite2

    def flaeche(self):
        return self._seite1 * self._seite2

class Quadrat(Rechteck):
```

```
    def __init__(self, seite):
        Rechteck.__init__(self, seite, seite)

Q1 = Quadrat(4)
print Q1.flaeche()
```

Ausgabe

```
user@localhost:~$ ./vererb1.py
16
```

Es wird zunächst eine allgemeine *Rechteck*-Klasse implementiert, die über die Fähigkeit verfügt, ihren Flächeninhalt zu berechnen. Die Spezialklasse *Quadrat* erbt sodann alle Eigenschaften von *Rechteck*, insbesondere die Fähigkeit, ihre Fläche nennen zu können. Die Elternklassen werden bei der Deklaration einer Klasse durch Kommata getrennt in die Klammern geschrieben. Für jede Basisklasse muss dann der Konstruktor aufgerufen werden, in unserem Fall *Rechteck.__init__(self, seite, seite)*. Ein Klasse kann auch von mehreren Eltern erben. Als Beispiel dient eine studentische Hilfskraft in einer Firma. Diese ist selbstverständlich ein echter Studierender, hat also eine Matrikelnummer, jedoch auch ein echter Mitarbeiter der Firma, bezieht also ein Gehalt:

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

class Student(object):
    def __init__(self, MatNummer):
        self._matnummer = MatNummer

    def __str__(self):
        return "Matrikelnummer: %d" % self._matnummer


class Mitarbeiter(object):
    def __init__(self, Gehalt):
        self._gehalt = Gehalt

    def __str__(self):
        return "Gehalt: %d" % self._gehalt


class Hilfskraft(Student, Mitarbeiter):
    def __init__(self, Gehalt, MatNummer):
        Mitarbeiter.__init__(self, Gehalt)
        Student.__init__(self, MatNummer)

    def gehalt(self):
        return self._gehalt

    def matnummer(self):
        return self._matnummer

Hans = Hilfskraft(500, 12345)
```

```
print Hans.gehalt(), Hans.matnummer()
print Hans
```

Ausgabe

user@localhost:~$ ./vererb2.py
500 12345
Matrikelnummer: 12345

Die Elternklassen werden durch Kommata getrennt bei der Definition der Klasse aufgeführt. Wie wir bei der Ausführung des Codes sehen können, berücksichtigt die *print Hans*-Anweisung lediglich die __str__()-Methode der Klasse *Student*. Vertauschen Sie in der Klassendefinition die Reihenfolge von *Student, Mitarbeiter*, dann stellen Sie fest, dass immer diejenige __str__()-Methode der zuerst genannten Elternklasse ausgeführt wird. Möchte man beide oder eine bestimmte Methode der Eltern ausführen, so muss man eine eigene Methode schreiben:

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

class Student(object):
    def __init__(self, MatNummer):
        self._matnummer = MatNummer

    def __str__(self):
        return "Matrikelnummer: %d" % self._matnummer


class Mitarbeiter(object):
    def __init__(self, Gehalt):
        self._gehalt = Gehalt

    def __str__(self):
        return "Gehalt: %d" % self._gehalt


class Hilfskraft(Student, Mitarbeiter):
    def __init__(self, Gehalt, MatNummer):
        Mitarbeiter.__init__(self, Gehalt)
        Student.__init__(self, MatNummer)

    def __str__(self):
        return Mitarbeiter.__str__(self) + " " + Student.__str__(self)

Hans = Hilfskraft(500, 12345)
print Hans
```

Ausgabe

user@localhost:~$ ./vererb3.py
Gehalt: 500 Matrikelnummer: 12345

Auf Elternklassen greift man über die Klassennamen (*Student* oder *Mitarbeiter*) zu, nicht über Objekte.

Zusammenfassung

In diesem Kapitel haben Sie einen Überblick über die Fähigkeiten der objektorientierten Programmierung mit Python gewonnen. Sie wissen nun, wie man Klassen aufbaut und kennen einige typische Techniken im Umgang mit Methoden und Attributen. Ebenso haben wir gezeigt, wie in Python Vererbung, eine wichtige Technik der OOP, funktioniert.

Anmerkungen

[1] engl.: function decorator

Dateien

Dateien dienen zum dauerhaften Speichern von Informationen. Sie können erzeugt und gelöscht werden, ihnen kann man Daten anhängen oder vorhandene Daten überschreiben. Selbstverständlich sind die Daten lesbar, sie können auch ausführbar sein. Es können sowohl Textdateien wie auch zeichen- oder blockorientierte Dateien bearbeitet werden.

Öffnen und Schließen von Dateien

Um eine Datei zu erzeugen, reicht es, sie zum Schreiben zu öffnen:

```
#!/usr/bin/python

datei = open("hallo.dat", "w")
datei.close()
```

Ausgabe

```
user@localhost:~$ ./datei1.py ; ls *.dat
hallo.dat
```

In diesem Beispiel wird eine leere Datei mit dem Namen hallo.dat erzeugt. Gibt es eine solche Datei, wird sie überschrieben, dafür sorgt das Dateiattribut `"w"` (write). Eine Datei wird zum Lesen geöffnet, wenn man das Attribut `"r"` (read) benutzt. Das Attribut `"a"` (append) hingegen würde die Datei erzeugen, falls sie noch nicht existiert, sonst hingegen öffnen und Daten an das Ende anhängen.

Lesen und Schreiben

Über vorhandene Dateien lässt sich leicht iterieren, wie folgendes Beispiel zeigt. Es werden nützliche Informationen aus der Datei /etc/passwd angezeigt:

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

datei = open("/etc/passwd", "r")
for zeile in datei:
    liste = zeile[:-1].split(":")
    print "Benutzer '%s' benutzt die Shell %s" % (liste[0], liste[6])
datei.close()
```

Ausgabe

user@localhost:~$ ./pw.py
Benutzer 'root' benutzt die Shell /bin/bash
Benutzer 'daemon' benutzt die Shell /bin/sh
Benutzer 'bin' benutzt die Shell /bin/sh
...

Es wird bei dem Beispiel über jede Zeile in der Datei iteriert. Die Zeilen werden um das Zeilenende (Newline-Zeichen) verkürzt (zeile[:-1]) und dann nach Doppelpunkten aufgeteilt (split(":")).

Dateien können auch mit den Methoden write() und read() beschrieben und gelesen werden:

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

datei = open("hallo.dat", "w")
datei.write("Hallo Welt\n\n")
datei.write("Foo Bar\n")
datei.close()

datei = open("hallo.dat", "r")
x = datei.read()
datei.close()
print x
```

Ausgabe

user@localhost:~$ ./datei2.py
Hallo Welt
Foo Bar

Die Methode write() schreibt die Daten nacheinander in die Datei, Zeilenumbrüche müssen explizit angegeben werden. read() hingegen liest die gesamte Datei ein und erzeugt daraus einen einzelnen String.

Um ein anderes Objekt als einen String zu speichern und zu lesen, muss man sich schon sehr anstrengen. Das folgende Beispiel schreibt eine Liste von Tupeln, die aus einem Namen und einer Zahl bestehen. Anschließend wird diese Datei eingelesen:

```
#!/usr/bin/python

studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

datei = open("studenten.dat", "w")
for student in studenten:
    s = str(student) + '\n'
    datei.write(s)
datei.close()

datei = open("studenten.dat", "r")
for z in datei:
    name, matnum = z.strip('()\n ').split(',')
    name = name.strip("\' ")
    matnum = matnum.strip()
    print "Name: %s  Matrikelnummer: %s" % (name, matnum)
```

```
datei.close()
```

Ausgabe

```
user@localhost:~$ ./datei4.py
Name: Peter Matrikelnummer: 123
Name: Paula Matrikelnummer: 988
Name: Freddi Matrikelnummer: 851
```

Tupel werden genau so geschrieben, wie sie da stehen, denn sie werden auch so in einen String umgewandelt. Je Tupel wird eine Zeile geschrieben. Liest man nun eine solche Zeile wieder ein, erhält man als String den Wert *('Peter', 123)*. Dieser wird mit der Methode strip() verändert, und zwar werden alle Klammern am Anfang und Ende des Strings wie auch Leerzeichen und Zeilenumbrüche entfernt. Man erhält also den String *Peter', 123*. Dieser String wird mit der Methode split() in zwei Teile geteilt, wobei das Komma als Trennzeichen dient. Die so entstandene Liste wird je Teil um führende Anführung- und Leerzeichen bereinigt. strip() ohne Argumente entfernt Leerzeichen. Erst jetzt haben wir zwei Strings, die den ursprünglichen Elementen der Tupel ähnlich sehen. Leichter wird es mit Pickle gehen.

Dateien mit Pickle

Den Funktionen, die Pickle anbietet, kann man wirklich nahezu jedes Objekt anbieten, Pickle kann es speichern. Ob es sich um Zahlen, Listen, Tupel oder ganze Klassen handelt ist Pickle dabei gleich. Der Aufruf ist recht einfach, es gibt eine Funktion zum **pickeln** und eine zum **entpickeln**:

```
#!/usr/bin/python

import pickle

studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

datei = open("studenten.dat", "w")
pickle.dump(studenten, datei)
datei.close()

datei = open("studenten.dat", "r")
meine_studenten = pickle.load(datei)
datei.close()

print meine_studenten
```

Ausgabe

```
user@localhost:~$ ./pickle1.py
[('Peter', 123), ('Paula', 988), ('Freddi', 851)]
```

Der Funktion dump() übergibt man die Daten sowie einen Dateihandle. Pickle kümmert sich dann um das Schreiben. Das Laden passiert mit load(). Diese Funktion nimmt lediglich das Dateiobjekt auf.

Dateien mit json

JSON (http://json.org) ist das bessere Pickle, oder zumindest meist die bessere Wahl. JSON ist *das* Format, wenn es um den Datenaustausch innerhalb von Webanwendungen geht. Aber es lässt sich genau so gut abseits des Webs nutzen. Pickle ist pythonspezifisch, JSON dagegen versteht sich mit jeder nennenswerten Programmiersprache. Hier das Pickle-Skript umgeschrieben für JSON:

```
#!/usr/bin/python

import json

studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

datei = open("studenten.json", "w")
json.dump(studenten, datei, indent=4)
datei.close()

datei = open("studenten.json", "r")
meine_studenten = json.load(datei)
datei.close()

print meine_studenten
```

Ausgabe

```
user@localhost:~$ ./json1.py
u'Peter', 123], [u'Paula', 988], [u'Freddi', 851
```

Da JSON keine Tupel kennt, werden die Name-Nummer-Tupel zu Listen. Und Pythons JSON-Strings sind immer Unicode. Neben Listen, Zeichenketten und Zahlen können noch Dictionarys, True, False und None gespeichert werden.

Ein weiterer Vorteil von JSON offenbart sich beim Blick in die Datei studenten.json:

```
[
    [
        "Peter",
        123
    ],
    [
        "Paula",
        988
    ],
    [
        "Freddi",
        851
    ]
]
```

Das ist so ziemlich das Optimum an Lesbarkeit. dump()s optionales Argument indent bestimmt dabei die Einrücktiefe. Ohne diese Angabe wird das ein, immer noch gut lesbarer Einzeiler.

Zeichenorientierte Dateien

Zeichenorientierte Dateien sind ein Strom aus einzelnen Bytes, die sich nicht notwendigerweise als Text darstellen lassen. Um einzelne Zeichen aus einer Datei zu lesen, übergeben wir der Methode read() die Anzahl der auf einmal zu lesenden Zeichen.

Als ein Beispiel zeichenorientierter Dateien benutzen wir ein Programm, welches uns Lottozahlen zufällig vorhersagt. Hierzu lesen wir unter Linux und anderen unixähnlichen Betriebssystemen die Datei /dev/random aus, welche recht gute Zufallszahlen erzeugt. Diese werden in einem set gespeichert. Wir benutzen hier ein set, damit wir eindeutige verschiedene Werte aufnehmen können. Sobald wir sechs verschiedene Zahlen im Bereich 1 bis 49 haben, werden diese in eine Liste konvertiert, sortiert und angezeigt:

```
#!/usr/bin/python

datei = open("/dev/random", "r")

lottozahlenSet = set()

while len(lottozahlenSet) < 6:
    zahl = ord(datei.read(1))
    if 0 < zahl < 50:
        lottozahlenSet.add(zahl)

datei.close()

lottozahlen = list(lottozahlenSet)
lottozahlen.sort()

print lottozahlen
```

Ausgabe

```
user@localhost:~$ ./lottozahlen.py
[5, 6, 18, 19, 23, 41]
```

Es wird aus der zeichenorientierten Datei /dev/random jeweils ein einzelnes Zeichen gelesen, in eine Zahl verwandelt und in das lottozahlenSet eingefügt. Die sortierte Liste wird ausgegeben.

Blockorientierte Dateien

Manche Dateien sind so aufgebaut, dass sie mehrfach den gleichen Datentyp speichern, wie zum Beispiel Folgen von Fließkommazahlen, Soundinformationen einer Wave-Datei, Login-Informationen wie in /var/log/wtmp oder auch feste Dateiblöcke wie in /dev/sda. Solche Dateien nennt man *blockorientiert*.

Numerische Blöcke

Dateien, die einfache Folgen numerischer Werte aufnehmen können, werden bequem mit dem Modul Array bearbeitet. Das folgende Beispiel legt eine solche Datei an und speichert zwei Fließkommazahlen in ihr. Diese Werte werden anschließend wieder gelesen und ausgegeben. Bitte beachten Sie, dass der Inhalt der in diesem Beispiel angelegten Datei keine Textdarstellung der gespeicherten Zahlen ist.

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

```
import array

# Array anlegen
fa = array.array("f")
fa.append(1.0)
fa.append(2.0)

# In datei schreiben
datei = open("block1.dat", "w")
fa.tofile(datei)
datei.close()

# Aus Datei lesen
datei = open("block1.dat", "r")
ra = array.array("f")
ra.fromfile(datei, 2)
datei.close()

# Ausgeben
for zahl in ra:
    print zahl
```

Ausgabe

```
user@localhost:~$ ./block1.py
1.0
2.0
```

Ein Array wird mit array(code) angelegt. Eine kleine Auswahl an Codes stellen wir ihnen in dieser Tabelle vor:

Typ	Code
Zeichen	**c** und **u** (Unicode)
int	**i**
long	**l**
float	**f**

Es können nur Werte mit dem angegebenen Typ, in unserem Beispiel Fließkommazahlen, gespeichert werden. Die Methode append() fügt dem Array eine Zahl hinzu, mit tofile(dateihandle) wird das gesamte Array geschrieben.

Liest man ein solches Array wieder ein, benutzt man die Methode fromfile(dateihandle, anzahl). Hier muss man neben der Datei noch die Anzahl der Werte wissen, die man auslesen möchte.

Zur Ausgabe können wir einfach in bekannter Weise über das Array iterieren.

Strukturierte Blöcke

Während man es bei Log-Dateien meistens mit Textdateien zu tun hat, gibt es hin und wieder Systemdateien, die einen blockorientierten Aufbau haben. Beispielsweise ist die Datei /var/log/lastlog eine Folge von Blöcken mit fester Länge[1]. Jeder Block repräsentiert den Zeitpunkt des letzten Logins, das zugehörige Terminal und den Hostnamen des Rechners, von dem aus man sich angemeldet hat. Der Benutzer mit der User-ID **1000** hat den 1000sten Block. Der Benutzer **root** mit der User-ID **0** den 0ten Block.

Hat man einmal herausgefunden, wie die Datei aufgebaut ist, und welche Datentypen hinter den einzelnen Einträgen stecken, kann man sich den Blocktyp mit Pythons Modul struct nachbauen. Hierfür ist der Formatstring gedacht, der im folgenden Beispiel in kompakter Form den Datentyp eines **lastlog**-Eintrags nachahmt. Wir bestimmen den letzten Anmeldezeitpunkt des Benutzers mit der User-ID 1000 und ahmen so einen Teil des Unix-Befehles **lastlog** nach:

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

import struct

format = "i32s256s"
userID = 1000

blockgroesse = struct.calcsize(format)
datei = open("/var/log/lastlog", "r")
datei.seek(userID * blockgroesse)
eintrag = datei.read(blockgroesse)
b = struct.unpack_from(format, eintrag)
datei.close()

print "Zeitstempel:", b[0]
print "Terminal:", str(b[1])
print "Hostname:", str(b[2])
```

Ausgabe

```
user@localhost:~$ ./lastlog1.py
Zeitstempel: 1241421726
Terminal: :0
Hostname:
```

Der Formatstring ist i32s256s. Hinter dieser kryptischen Abkürzung verbirgt sich ein Datentyp, der einen Integer enthält (**i**), dann eine Zeichenkette (**s**) von 32 Byte Länge gefolgt von einer weiteren Zeichenkette, die 256 Zeichen lang ist.

Wir bestimmen die Blockgröße mit der Methode calcsize(Formatstring). Diese Blockgröße wird in Bytes gemessen. Anschließend wird die Datei /var/log/lastlog geöffnet und mit seek(Position) bis zu einer bestimmten Stelle in der Datei vorgespult.

Ein Eintrag der Datei wird gelesen und mit Hilfe der Methode unpack_from(Formatstring, String) in ein 3-Tupel konvertiert. String-Daten in diesem Tupel müssen explizit konvertiert werden, da man ansonsten angehängte NULL-Bytes mitführt.

Der Zeitstempel als Unix-Zeit wird zusammen mit den anderen Angaben ausgegeben.

Binärdateien

Unter Binärdateien verstehen wir Dateien, die nicht ausschließlich aus Text bestehen, sondern insbesondere auch nicht darstellbare Zeichen enthalten. Solche Dateien haben wir in den vorgenannten Abschnitten zu blockorientierten Dateien schon kennen gelernt. Technisch macht Linux keinen Unterschied zwischen Text- und Binärdateien.

Mit den uns bekannten Funktionen können wir leicht Informationen aus solchen Dateien extrahieren. Das folgende Beispiel zeigt, wie wir die Anzahl der Kanäle und die Kennung einer Wave-Datei[2] ermitteln.

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

datei = open("test.wav", "rb")
datei.seek(8)
text = str(datei.read(4))
print "Kennung:", text

datei.seek(22)
# nur das höherwertige Byte auslesen
kanaele = ord(datei.read(1))
datei.close()

print "Die WAV-Datei hat", kanaele, "Kanäle"
```

Ausgabe

```
user@localhost:~$ ./binary1.py
Kennung: WAVE
Die WAV-Datei hat 1 Kanäle
```

Die Datei wird im Modus "rb" geöffnet. "b" steht dabei für "binary". Da Linux (anders als zum Beispiel Windows) keinen Unterschied zwischen Text- und Binärdateien macht, wäre das "b" nicht nötig. Aber Plattformunabhängigkeit ist eine von Pythons hervorstechenden Merkmalen, die Sie auch bei der Programmierung anstreben sollten. So läuft das Skript fehlerlos auf *jedem* Betriebssystem.

Die gesuchten Informationen, Kennung und Kanäle, stehen an bestimmten Positionen in der Datei, die wir mit seek() anspringen. Anschließend werden die Kennung wie auch die Anzahl der Kanäle gelesen und ausgegeben.

Bitte beachten Sie, dass diese Art auf Medienformate zuzugreifen nur das Konzept verdeutlichen soll. Zu den meisten Medienformaten gibt es eigene Module, die den Aufbau eines Formates kennen und somit einen gut dokumentierten Zugriff auf die Inhalte bieten. Für den hier dargestellten Zugriff auf das Medienformat WAVE bietet Python das Modul wave an.

Verzeichnisse

Wenn man vom Umgang mit Dateien redet, gehören Verzeichnisse unweigerlich dazu. Die beiden häufigsten Funktionen in diesem Zusammenhang sind wohl, zu überprüfen ob ein Verzeichnis existiert und es gegebenenfalls anzulegen. Ein Beispiel:

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

import os
```

```
datei = "test.txt"
verzeichnis = "test"

pfad = verzeichnis + os.sep + datei
if not os.path.exists(verzeichnis):
    os.mkdir(verzeichnis)
offen = open(pfad, "w")
offen.write("hallo")
offen.close()
```

Ausgabe

```
user@localhost:~$ ./folders1.py; ls test
test.txt
```

Das Programm erzeugt einen Pfad, der aus einem Verzeichnis, einem Trennzeichen und einem Dateinamen besteht. Das Trennzeichen erfährt man mit os.sep, unter unixähnlichen Betriebssystemen ist dies meistens der Schrägstrich (/). Die Funktion os.path.exists() prüft, ob eine Datei oder ein Verzeichnis vorhanden ist. Wenn nicht, dann wird mit os.mkdir() ein neues erzeugt und anschließend eine Datei hineingeschrieben.

Eine andere Anwendung ist herauszufinden, in welchem Verzeichnis wir aktuell sind und dann alle enthaltenen Verzeichnisse und Dateien ab dem übergeordneten Verzeichnis aufzulisten. Dieses Programm implementiert den Befehl **ls -R ..**:

```
#!/usr/bin/python

import os

print "Wir sind hier->", os.getcwd()

os.chdir("..")
for r, d, f in os.walk(os.getcwd()):
  print r, d, f
```

Ausgabe

```
user@localhost:~$ ./folders2.py
Wir sind hier-> /home/tandar/x/y
/home/tandar/x ['y'] ['datei1.txt']
/home/tandar/x/y [] ['datei2.txt']
```

Mit der Funktion getcwd() finden wir das aktuelle Verzeichnis heraus. Anschließend können wir mit chdir(Pfad) in das übergeordnete oder ein anderes Verzeichnis wechseln. Für jenes Verzeichnis rufen wir walk(Pfad) auf, und erhalten eine Menge von 3-Tupeln, die aus einem Basispfad, einer Liste mit enthaltenen Verzeichnissen und einer Liste mit enthaltenen Dateien besteht. Diese Angaben geben wir im Programm schlicht aus. walk() berücksichtigt übrigens keine symbolischen Links auf Verzeichnisse. Sind solche zu erwarten, kann walk(top=Pfad, followlinks=True) benutzt werden.

Zusammenfassung

In diesem Kapitel haben Sie einen Überblick über die Verwendung von Dateien und Ordnern mit Python bekommen. Textdateien lassen sich unkompliziert lesen und schreiben, bei anderen Datentypen als Text bietet sich Pickle an. Block- und zeichenorientierte Dateien lassen sich ebenfalls bequem lesen und beschreiben, wobei im Fall der blockorientierten Dateien der Aufbau bekannt sein muss. Die meisten dateibasierten Operationen lassen sich mit Funktionen aus dem Modul os ansprechen, dessen Inhalt wir im Kapitel Überblick über vorhandene Module näher beleuchten.

---

[1] Siehe lastlog(8) und utmp.h auf ihrem Computer

[2] Das Dateiformat ist unter (http://www.hib-wien.at/leute/wurban/informatik/waveformat.pdflhttp://www.hib-wien.at/leute/wurban/informatik/waveformat.pdf) gut dokumentiert.

Reguläre Ausdrücke

Reguläre Ausdrücke helfen beim Finden von Textstellen. Sie werden angegeben in Form von Mustern wie "*Ich suche eine Folge von Ziffern am Anfang einer Textzeile*". Für "*Ziffernfolge*" und "*Zeilenanfang*" gibt es Kurzschreibweisen und Regeln, wie diese zu kombinieren sind. Um die Funktionen und Methoden rund um reguläre Ausdrücke benutzen zu können, müssen wir das Modul re einbinden.

Finde irgendwo

Haben wir Textzeilen, in denen beispielsweise Telefonnummern und Namen aufgelistet sind, so können wir mit Hilfe von regulären Ausdrücken die Namen von den Telefonnummern trennen - zum Beispiel um nach einer bestimmten Telefonnummer in einer langen Textdatei zu suchen:

```
#!/usr/bin/python

import re

s = "Peter 030111"

m = re.search(r"(\w+) (\d+)", s)

length = len(m.groups())
for i in xrange(length + 1):
    print "Group", i, ":", m.group(i)
```

Ausgabe

```
user@localhost:~$ ./re1.py
Group 0 : Peter 030111
Group 1 : Peter
Group 2 : 030111
```

Haben wir eine Textzeile, die das Tupel enthält, so können wir mit der search()-Funktion nach den einzelnen Tupel-Elementen suchen. Hierbei ist "\w" ein einzelnes Zeichen aus der Menge der Buchstaben, der Modifizierer "+" bedeutet "*mindestens ein*" Zeichen zu suchen. Die Klammern gruppieren diese Ausdrücke. Der erste Ausdruck in Klammern bedeutet also: Finde eine nicht leere Zeichenkette aus Buchstaben und nenne es die erste Gruppe mit Gruppenindex 1. (\d+) sucht nach einer nicht leeren Ziffernfolge und gibt dieser Gruppe den Index 2, da sie an zweiter Stelle steht. Zwischen der Buchstabenfolge und der Ziffernfolge muss sich exakt ein Leerzeichen befinden. search() liefert ein so genanntes **Match-Objekt**, wenn etwas gefunden wurde, sonst None. groups() liefert ein Tupel

mit allen gefundenen Gruppen, mit group() kann man auf einzelne Gruppen zugreifen, wobei der Index 0 eine Zusammenfassung aller Gruppen als einzelnen String anbietet.

Nun muss man nicht alles gruppieren, nach dem man sucht. Manches Teile eines Ausdruckes möchte man schlicht verwerfen, um an die relevanten Informationen eines Textes zu kommen:

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

import re

s ="""Opa hat am 10. 12. 1903 Geburtstag.
Oma wurde am 14. 07. geboren.
Mutti, deren Eltern sich am 10.02.1949 über die Geburt freuten, hat
heute selber Kinder.
Vater hat am 3. 4. 1947 Geburtstag"""

for line in s.splitlines():
    m = re.search(r"^(\w+)\D*(\d*)\.\s?(\d*)\.\s?(\d*)", line)
    print line
    print m.groups()
```

Ausgabe

```
user@localhost:~$ ./re2.py
Opa hat am 10. 12. 1903 Geburtstag.
('Opa', '10', '12', '1903')
Oma wurde am 14. 07. geboren.
('Oma', '14', '07', )
Mutti, deren Eltern sich am 10.02.1949 über die Geburt freuten, hat heute selber Kinder.
('Mutti', '10', '02', '1949')
Vater hat am 3. 4. 1947 Geburtstag
('Vater', '3', '4', '1947')
```

Die for-Schleife wird über jede Zeile ausgeführt, wobei die Methode splitlines() einen Text in einzelne Zeilen zerlegt. In diesem Beispiel interessieren uns der Name, der dem Zeilenanfang als erstes Wort folgt, der Geburtstag, Geburtsmonat und das Geburtsjahr, welches auch fehlen darf. Der Zeilenanfang wird mit einem Dach (^) abgekürzt. Die erste Gruppe beinhaltet ein Wort, in dem Fall den Namen.

Ausdrücke im Überblick

Hier ein Ausschnitt aus der Sprache der regulären Ausdrücke. Alternativen werden durch Kommas getrennt.

Ausdruck	Bedeutung
.	Der Punkt passt auf jedes Zeichen
^	Anfang einer Zeichenkette oder Zeile
$	Ende einer Zeichenkette oder Zeile
[abc]	Passt auf ein Zeichen aus der Menge "abc"
[a-z]	Passt auf ein Zeichen aus der Menge "a bis z"
A|B	Passt auf den regulären Ausdruck A oder auf B
\A	Anfang einer Zeichenkette
\d, [0-9]	Eine einzelne Ziffer
\D, [^0-9]	Jedes Zeichen **außer** der Ziffer
\s	Leerzeichen wie Tab, Space, ...
\S	Jedes Zeichen **außer** dem Leerzeichen
\w	Buchstaben, Ziffern und der Unterstrich
\W	Alle Zeichen **außer** solchen, die in \w definiert sind
\Z	Das Ende einer Zeichenkette

Neben einzelnen Zeichen kann auch die Anzahl der vorkommenden Zeichen begrenzt werden. Folgende Tabelle gibt einen Überblick:

Ausdruck	Bedeutung	Beispiel
*	Der Ausdruck kommt mindestens 0-mal vor.	.*
+	Der Ausdruck kommt mindestens 1-mal vor.	\d+
?	Der Ausdruck kommt 0- oder 1-mal vor.	(LOTTOGEWINN)?
{n}	Der Ausdruck kommt n-mal vor	\w{3}
{m, n}	Der Ausdruck kommt mindestens m-mal, höchstens n-mal vor	\w{3, 10}
*?, +?, ??	die kürzeste Variante einer Fundstelle wird erkannt	.*?, .+?, .??

Gieriges Suchen

Um den Unterschied zwischen den verschiedenen Quantoren zu verdeutlichen, suchen wir in einem angegebenen String mit verschiedenen Kombinationen von Quantoren nach Ausdrücken, die auf beiden Seiten von einem "e" begrenzt werden. Wir bilden dazu drei Gruppen, die unterschiedlich lange Bereiche finden:

```
#!/usr/bin/python
import re

zeile = "Dies ist eine Zeile"

m = re.search(r"(.*)(e.*e)(.*)", zeile)
print m.groups()

m = re.search(r"(.*?)(e.*?e)(.*)", zeile)
print m.groups()
```

```
m = re.search(r"(.*?)(e.*e)(.*?)", zeile)
print m.groups()
```

Ausgabe

```
user@localhost:~$ ./gierig1
('Dies ist eine Z', 'eile', '')
('Di', 'es ist e', 'ine Zeile')
('Di', 'es ist eine Zeile', '')
```

Wie wir sehen können, finden alle drei Ausdrücke Text, jedoch unterschiedlich lang. Die Variante (.*?) findet möglichst wenig Text, während (.*) möglichst lange Fundstellen findet. Solche Quantoren werden als *gierig* bezeichnet.

Matching

Während search() im gesamten übergebenen String nach einem Ausdruck sucht, ist match() auf den Anfang des Strings beschränkt. Suchen wir in einer Datei wie /var/log/syslog nach Ereignissen vom 26. Januar, können wir das leicht mit match() erledigen:

```
#!/usr/bin/python
import re

datei = open("/var/log/syslog", "r")
print "Am 26. Januar passierte folgendes:"
for zeile in datei:
    m = re.match(r"Jan 26 (.*)", zeile)
    if m != None:
        print m.group(1)
datei.close()
```

Ausgabe

```
user@localhost:~$ ./match1
Am 26. Januar passierte folgendes:
10:08:19 rechner syslogd 1.5.0#2ubuntu6: restart.
10:08:21 rechner anacron[4640]: Job `cron.daily' terminated
10:12:42 rechner anacron[4640]: Job `cron.weekly' started
...
```

Es wird hier nach allen Zeichen gesucht, die der Zeichenkette "Jan 26 " folgen. Dies ist der Anfang der Zeile. Gibt es solche Zeichen, werden nur diese ausgegeben.

Ich will mehr!

Gruppen muss man nicht aufzählen, man kann sie auch benennen, wie folgendes Skript zeigt:

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

import re

wunsch ="Chef, ich brauche mehr Geld. 1000 oder 2000 Euro halte ich für
```

```
 angebracht!"

m = re.search(r"(?P<geld>\d{4})", wunsch)

print m.groups()
print m.group('geld')
```

Ausgabe

user@localhost:~$ ./re3.py
('1000',)
1000

Die Gruppe (?P<geld>A) gibt dem eingeschlossenen regulären Ausdruck einen Namen. Es wird nach einer genau 4-stelligen Zahl gesucht, und auch gefunden. Dieses Beispiel offenbart aber schon eine Schwierigkeit mit search(): Wir finden nur das erste Vorkommen des Ausdrucks (*1000*), nicht jedoch den Zweiten.

Hier die verbesserte Version, die wirklich alle Wunschvorstellungen zu finden in der Lage ist:

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

import re
wunsch ="Chef, ich brauche mehr Geld. 1000 oder 2000 Euro halte ich für
 angebracht!"

liste = re.findall(r"\d{4}", wunsch)
print liste
```

Ausgabe

user@localhost:~$ ./re4.py
['1000', '2000']

findall() wird genauso aufgerufen wie search(), findet jedoch auch mehrfache Vorkommen der Ausdrücke im Text.

Modi

Sowohl search() wie auch match() erlauben als dritten Parameter einen Wert, der die Bedeutung des angegebenen regulären Ausdrucks spezifiziert.

Die folgende Tabelle gibt einen Überblick über diese Flags:

Flag	Bedeutung
IGNORECASE	Groß- und Kleinschreibung wird nicht unterschieden
LOCALE	\w, \W, \b, \B, \s und \S werden abhängig von der gewählten Spracheinstellung behandelt
MULTILINE	Beeinflusst ^ und $. Es werden der Zeilenanfang und das -ende beachtet.
DOTALL	Der Punkt (.) soll *jedes* Zeichen finden, auch Zeilenwechsel
UNICODE	\w, \W, \b, \B, \d, \D, \s und \S werden als Unicode-Zeichen behandelt
VERBOSE	Erlaubt unter anderem Kommentare in regulären Ausdrücken, Leerzeichen werden ignoriert. Damit lassen sich *schöne* Ausdrücke mit Kommentaren zusammensetzen.

Hier ein Beispiel, wie man diese Flags anwendet. Es wird nach einer IP-Adresse gesucht, die sich über mehrere Zeilen erstreckt. Zwischen zwei Zahlen kann sowohl ein Punkt stehen wie auch ein Punkt gefolgt von einem

Neue-Zeile-Zeichen. Wir benutzen daher das Flag DOTALL, um den Punkt auch dieses Zeichen suchen zu lassen. Nebenbei zeigen wir, wie man den Aufzählungsquantor ({1,3}) benutzt.

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

zeile = """Dieses sind mehrere Zeilen, es geht
hier um die Darstellung einer IP-Adresse 192.168.
10.1. Diese wird über mehrere Zeilen verteilt."""

m = re.search(r"(\d{1,3}).+?(\d{1,3}).+?(\d{1,3}).+?(\d{1,3}).", zeile,
 re.DOTALL)
if m is None:
    print "nicht gefunden"
else:
    print m.groups()
```

Ausgabe

```
user@localhost:~$ ./modi1.py
('192', '168', '10', '1')
```

Wie wir sehen, wird die IP-Adresse korrekt gefunden.

Gruppen

Man kann in regulären Ausdrücken Gruppen bilden. Einige von diesen Gruppen haben Sie schon kennen gelernt. In diesem Abschnitt geben wir einen Überblick über Möglichkeiten, Ausdrücke zu gruppieren. Folgende Gruppen werden unterstützt:

Gruppe	Bedeutung
(?:*Ausdruck*)	Diese Gruppe wird ignoriert. Man kann sie nicht per groups() herauslösen, ebenfalls kann man sie nicht innerhalb der Ausdrücke referenzieren.
(?P<*Name*>*Ausdruck*)	Weist einer Gruppe einen Namen zu. Siehe das Beispiel Ich will mehr!
(?P=*Name*)	Diese Gruppe wird ersetzt durch den Text, der von der Gruppe mit dem angegebenen Namen gefunden wurde. Damit lassen sich innerhalb von Ausdrücken Referenzen auf vorherige Suchergebnisse bilden.
(?#*Inhalt*)	*Inhalt* wird als Kommentar behandelt.
A(?=*Ausdruck*)	*A* passt nur dann, wenn als nächstes *Ausdruck* folgt. Sonst nicht. Dieses ist also eine Art Vorschau auf kommende Suchergebnisse
A(?!*Ausdruck*)	*A* passt nur dann, wenn *Ausdruck* **nicht** als nächstes folgt.
(?<=*Ausdruck*)*A*	*A* passt nur dann, wenn *Ausdruck* vor *A* kommt.
(?<!*Ausdruck*)*A*	*A* passt **nicht**, wenn *Ausdruck* vor *A* kommt.
(?(*Name*)*A1*|*A2*)	Der Ausdruck *A1* wird nur dann als Suchmuster benutzt, wenn eine Gruppe mit Namen *Name* existiert. Sonst wird der (optionale) Ausdruck *A2* benutzt.

Wir wollen nun den Gebrauch einiger regulärer Ausdrücke demonstrieren:

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

```
zeilen = ("Peter sagt: Peter, gib mir den Ball", "Hans sagt: Peter,
sprichst Du mit dir selbst?")

print "Leute, die gerne mit sich selbst sprechen:"

for z in zeilen:
    m = re.match(r"(?P<Person>\w+) sagt: (?P=Person),", z)
    if m is not None:
        print m.group('Person')
```

In diesem Beispiel wird eine Gruppe Person erzeugt. In der ersten Zeile ist der Inhalt dieser Gruppe *Peter*, in der zweiten *Hans*. (?P=Person) nimmt nun Bezug auf den Inhalt dieser Gruppe. Steht an der Stelle der betreffenden Zeile ebenfalls das, was der Inhalt der ersten Gruppe war (also *Peter* oder *Hans*), dann passt der gesamte reguläre Ausdruck. Sonst nicht. In diesem Fall passt nur die erste Zeile, denn es kommt zweimal *Peter* vor.

Etwas komplizierter ist der Ausdruck im folgenden Beispiel. Es wird die Datei /etc/group untersucht. Es sollen all jene Gruppen aufgelistet werden, die Sekundärgruppen für User sind. Diese Zeilen erkennt man an:

```
cdrom:x:24:haldaemon,tandar
floppy:x:25:haldaemon,tandar
```

Es sind jedoch Zeile, die keine User beinhalten, zu ignorieren, wie die folgenden:

```
fax:x:21:
voice:x:22:
```

Ein Programm, das diese Zusammenstellung schafft, kann mit regulären Ausdrücken arbeiten. Nur jene Zeilen, die hinter dem letzten Doppelpunkt Text haben, sollen berücksichtigt werden. Der Gruppenname wie auch die Benutzernamen sollen in einer eigenen Ausdrucks-Gruppe gespeichert werden.

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

datei = open("/etc/group", "r")

print "User mit Sekundärgruppen"

for zeile in datei:
    m = re.match(r"(?P<Group>\w+)(?=:x:\d+:\S+):x:\d+:(?P<Mitglieder>.*)", zeile)
    if m is not None:
        print "User", m.group('Mitglieder'), "ist/sind in der Gruppe",
m.group('Group')

datei.close()
```

Ausgabe

```
user@localhost:~$ ./gruppen2.py
User mit Sekundärgruppen
User haldaemon,tandar ist/sind in der Gruppe cdrom
```

User haldaemon,tandar ist/sind in der Gruppe floppy
... weitere Ausgabe...

Der Ausdruck (?P<Group>\w+)(?=:x:\d+:\S+) besagt, dass nur dann wenn nach dem letzten Doppelpunkt noch Text folgt, der Gruppe mit dem Namen *Group* der Unix-Gruppenname zugewiesen werden soll. Es wird hier also eine Vorschau auf kommende Suchergebnisse betrieben. Diese Suchergebnisse werden jedoch nicht gespeichert. Anschließend wird der Zwischenbereich im Ausdruck :x:\d+: konsumiert und mit (?P<Mitglieder>.*) die Mitgliederliste aufgenommen.

Weiterführende Hinweise

Reguläre Ausdrücke, wie wir sie in diesem Kapitel besprochen haben, sind recht kurz und wurden nur auf wenige Textzeilen angewendet. Hat man aber wirklich die Aufgabe, große Logdateien zu untersuchen, bietet es sich an, Ausdrücke vorzubereiten. Ausdrücke können mit der Methode compile(Ausdruck, Flags) vorbereitet werden, um dann mit match() und search() abgearbeitet zu werden. Solche kompilierten Ausdrücke sind schneller.

Folgendes Beispiel demonstriert das Vorgehen, wobei wir hier /var/log/messages (oder /var/log/syslog unter z.B. Ubuntu) auswerten:

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

datei = open("/var/log/messages", "r")
exp = re.compile(r"Jan 27 (?P<Zeit>\d+:\d+:\d+) .*? (?P<Ereignis>.*)")

print "Wann passierte heute was?"

for zeile in datei:
    m = exp.match(zeile)
    if m is not None:
        print m.group('Zeit'), m.group('Ereignis')
datei.close()
```

Anders als bei in vorherigen Abschnitten vorgestellten Methoden wird hier zuerst ein Objekt erzeugt, welches dem kompilierten Ausdruck entspricht. Dessen Methoden sind ebenfalls unter anderem match() und search(). exp.match(zeile) ist hier analog zu sehen zu re.match(Ausdruck, Zeile).

Zusammenfassung

In diesem Kapitel wurden die Möglichkeiten regulärer Ausdrücke in Python vorgestellt. Im Kapitel Netzwerk lernen Sie weitere nützliche Anwendungen kennen.

Halbzeit

Wenn Sie es bis hierhin geschafft haben, dann haben Sie Python kennengelernt. Lehnen Sie sich also zurück, genießen Sie eine Tasse Kaffee oder Tee und freuen Sie sich über das Erlernte.

Später werden wir Ihnen einen Ausblick auf vorhandene Module geben, aber jetzt noch nicht.... ;-)

Überblick über Module

In diesem Kapitel geben wir einen Überblick über Module, die meistens mitinstalliert werden. Anschließend wird eine Methode gezeigt, mit der man selbst mehr über unbekannte (aber auch vermeintlich bekannte) Module herausfinden kann.

Modul cmath

Dieses Modul beinhaltet mathematische Funktionen zur Arbeit mit komplexen Zahlen. In Python haben komplexe Zahlen einen eigenen Typ, complex. Aufgeschrieben werden sie zum Beispiel als 5+3j[1].

Ein Beispiel:

```
#!/usr/bin/python

import cmath

z=5+4j
print z
print cmath.sqrt(z)
```

Ausgabe

```
user@localhost:~$ ./datei3.py
(5+4j)
(2.38779440462+0.837593050781j)
```

Modul datetime

Das Modul *datetime* ist für die Verwaltung von Datum und Zeit zuständig. Dazu werden die Klassen time, date, datetime (Zusammenfassung von time und date) sowie timedelta (Zeitdifferenzen) und tzinfo (Zeitzoneninformationen) definiert. Die Verwendung von date wird in folgendem Programm demonstriert:

```
#!/usr/bin/python

from datetime import date

t=date(2008, 4, 10)
print "Datum allgemein: " + str(t)
print "Datum auf deutsch: " + str(t.day) + "." + str(t.month) + "." +
str(t.year)
```

Ausgabe

```
user@localhost:~$ ./date.py
Datum allgemein: 2008-04-10
Datum auf deutsch: 10.4.2008
```

Hier folgt eine kurze Übersicht über einige der vorhandenen Methoden:

Date-Funktionen	Bedeutung
fromtimestamp(timestamp)	Liefert das Datum passend zum Unix-Zeitstempel. Zeitstempel kann man von time.time() (Modul time) geliefert bekommen.
today()	Liefert das heutige Datum
year, month, day	(Klassenattribute) Repräsentieren das Jahr, den Monat und den Tag

Time-Funktionen	Bedeutung
hour, minute, second, microsecond	(Klassenattribute) Repräsentieren Stunde, Minute, Sekunde und Mikrosekunde
isoformat()	Liefert die Zeit im ISO8601-Format: HH:MM:SS.microsekunden
strftime(format)	Liefert die Zeit unter Verwendung von Formatangaben. Beispiele: • **%f**: Mikrosekunden • **%H**: Stunden im 24-Stunden-Format • **%M**: Minuten • **%S**: Sekunden

Modul getopt

Funktionen innerhalb von getopt behandeln das Thema Kommandozeilenargumente:

Funktion	Bedeutung
getopt(Argumente, kurzeOptionen, langeOptionen)	Liest Kommandozeilenoptionen, wie sie in *Argumente* übergeben wurden und vergleicht sie mit *kurzeOptionen* (-f, -h, ...) und *langeOptionen* (--file, --help, ...). Es werden zwei Listen zurückgegeben, eine enthält alle empfangenen Optionen einschließlich der Argumente, die andere Liste enthält alle Kommandozeilenargumente, denen keine Option vorangestellt wurde.
gnu_getopt(Argumente, kurzeOptionen, langeOptionen)	Wie oben, jedoch dürfen Options- und nicht-Optionsargumente gemischt werden.

langeOptionen ist optional und enthält eine Liste mit Strings, die als lange Optionen interpretiert werden. Ein nachgestelltes Gleichheitszeichen bewirkt, dass diese Option ein Argument erwartet. *kurzeOptionen* ist ein String, der die Buchstaben zu den passenden Optionen beinhaltet. Ein einem Buchstaben nachgestellter Doppelpunkt bewirkt, dass diese Option ein Argument erwartet. Kommandozeilenparameter werden typischerweise von der Variablen sys.argv bereitgestellt, wobei sys.argv[0] der Skriptname ist. Beispiel:

```
#!/usr/bin/python

import getopt
import sys

print "Alle Argumente: ", sys.argv

shortOptions = 'hf:'
longOptions = ['help', 'filename=', 'output=']

def usage():
    print sys.argv[0], "--help --filename=meinedatei
[--output=ausgabedatei] ..."
    print sys.argv[0], "-h -f meinedatei ..."


opts = []
args = []
try:
    opts, args = getopt.getopt(sys.argv[1:], shortOptions, longOptions)
except getopt.GetoptError:
    print "ERR: Mindestens eine der Optionen ist nicht verfuegbar"
    usage()
    sys.exit()

print "Optionenargumente:", opts
print "Nicht-Optionen-Argumente:", args

for o, a in opts:
    if o == "--help" or o == "-h":
        print "HELP"
        usage()
    elif o == "--filename" or o == "-f":
        print "Filename, Bearbeite Datei:", a
```

```
    elif o == "--output":
        print "Output, Dateiname:", a

for a in args:
    print "Weiteres Argument, keine Option: ", a
```

Ausgabe

```
user@localhost:~$ ./getopt1.py --output test --filename test1 -f test2 test3
Alle Argumente: ['./getopt1.py', '--output', 'test', '--filename', 'test1', '-f', 'test2', 'test3']
Optionenargumente: [('--output', 'test'), ('--filename', 'test1'), ('-f', 'test2')]
Nicht-Optionen-Argumente: ['test3']
Output, Dateiname: test
Filename, Bearbeite Datei: test1
Filename, Bearbeite Datei: test2
Weiteres Argument, keine Option: test3
```

Falls eine Option eingegeben wurde, die nicht Bestandteil der kurzen oder Langen Optionsliste ist, dann wird die Exception getopt.GetoptError geworfen. Die Optionen `-f`, `--filename` und `--output` erwarten weitere Argumente.

Modul math

Das Modul *math* enthält mathematische Funktionen und Konstanten. Die folgende Tabelle zeigt einen Auschnitt:

Funktion	Bedeutung
e	Der Wert der Eulerkonstante 2.718...
pi	Der Wert der Kreiszahl pi 3.14...
sin(), cos()	Sinus, Kosinus eines Winkels im Bogenmaß
degrees()	Rechnet einen Winkel vom Bogenmaß in Grad um
radians()	Umrechnung Grad -> Bogenmaß
log(), exp()	Logarithmus, Exponentialfunktion

Ein Beispiel:

```
#!/usr/bin/python

import math

print math.e
print math.sin(math.pi/2)
```

Ausgabe

```
user@localhost:~$ ./math.py
2.71828182846
1.0
```

Modul os

Dieses Modul beinhaltet Variablen und Funktionen, die stark vom eingesetzten Betriebssystem (Linux, Mac, OpenBSD, ...) abhängen.

Beispiel:

```
#!/usr/bin/python

import os

print "Arbeite unter " + os.name
print "Dieses Programm ist unter " + os.curdir + os.sep + "ostest" +
os.extsep + "py" + " zu erreichen"
```

Ausgabe

user@localhost:~$ ./ostest.py
Arbeite unter posix
Dieses Programm ist unter ./ostest.py zu erreichen

Anmerkung: Dieses Programm gibt unter verschiedenen Betriebssystemen unterschiedliche Ergebnisse aus.

Hier folgen einige nützliche Funktionen aus dem Modul:

Dateien und Verzeichnisse

Funktion	Bedeutung
chdir(pfad)	Wechselt in das angegebene Verzeichnis
chmod(pfad, modus)	Ändert die Modi (Lesen, Schreiben, Ausführen) der Datei mit dem Dateinamen *pfad*.
chown(pfad, uid, gid)	Ändert die Besitzrechte der Datei mit dem Namen *pfad* auf die der angegebenen UID und GID
close()	Datei wird geschlossen
curdir, pardir	(Konstanten) Kürzel für das aktuelle Verzeichnis (meist ".") oder das übergeordnete Verzeichnis ("..")
getcwd()	Liefert das aktuelle Arbeitsverzeichnis
listdir(pfad)	Erzeugt eine Liste mit dem Inhalt des angegebenen Verzeichnisses *pfad*.
lseek(fd, pos, how)	Positioniert den Dateizeiger der geöffneten Datei *fd* an die Position *pos*. *how* bestimmt, wie positioniert wird: • SEEK_SET: relativ zum Anfang der Datei (voreingestellt) • SEEK_CUR: relativ zur aktuellen Position • SEEK_END: relativ zum Ende der Datei
name	(Konstanten) Kennung für das Betriebssystem (posix, os2, riscos, ...)
open(file, flags[, mode])	*file*: Dateiname, *flags*: Modi der Art • O_RDONLY: nur Lesen • O_WRONLY: nur schreiben • O_RDWR: lesen und schreiben • O_APPEND: anhängen • O_CREAT: Datei wird erzeugt • O_EXCL: Datei wird erzeugt, wenn es sie noch nicht gibt. Falls es die Datei gibt, liefert open() einen Fehler. • O_TRUNC: Datei wird geleert Diese Flags können verodert werden. *mode*: Dateirechte, wie z. B. "0660"
read(fd, n)	Liest *n* Zeichen aus der offenen Datei *fd*.

remove(pfad), rmdir(pfad)	Entfernt eine Datei oder ein Verzeichnis (rekursiv).
write(fd, s)	Schreibt einen String *s* in die offene Datei *fd*.
sep, extsep	(Konstanten) Trennzeichen für Pfadnamen ("/") und Dateiendungen (".")
tmpfile()	Öffnet eine temporäre Datei im Modus "w+"

Prozessmanagement

Funktion	Bedeutung
fork()	Erzeugt einen Kindprozess. Liefert *0*, wenn dieses der Kindprozess ist, sonst die Prozessnummer (PID) des Kindes.
kill(pid, signal)	Sendet das Signal *signal* an den Prozess mit der angegebenen *pid*.
nice(wert)	Fügt dem aktuellen Prozess den Nicewert *wert* hinzu.
system(befehl)	Führt einen externen Befehl aus.

Modul os.path

Dieses Modul enthält Funktionen, die Auskunft über Dateien geben. Man kann Änderung- und Zugriffszeiten abfragen, feststellen, ob ein Pfad eine (existierende) Datei oder ein Verzeichnis ist und vieles mehr.

Funktion	Bedeutung
exists(Pfad)	Liefert True, wenn *Pfad* existiert und das Programm das Recht hat, diesen Pfad einzusehen.
expanduser(~username)	Erzeugt einen Pfad, der das Home-Verzeichnis von Benutzer *username* ist.
getatime(Pfad), getmtime(Pfad)	Liefert die letzte Zugriffszeit oder die Änderungszeit
isfile(Pfad)	True, wenn der Pfad eine Datei darstellt
join(Pfad1, Pfad2, ...)	Fügt Pfade zusammen, wobei *Pfad1* ein Verzeichnis sein kann, *Pfad2* ein Verzeichnis innerhalb von *Pfad1* oder eine Datei.

Modul random

Das Modul *random* stellt Zufallsfunktionen zur Verfügung. Ein kleines Beispiel würfelt eine Zahl zwischen 1 und 6:

```
#!/usr/bin/python

from random import randint

# Zufallszahl ermitteln und als Wuerfelergebnis nehmen.
def print_random():
  wuerfel = randint(1, 6)
  print "Der Wurf ergibt " + str(wuerfel)

# 5mal wuerfeln
for i in range(0, 5):
  print_random()
```

Ausgabe

user@localhost:~$ ./wuerfeln.py
Der Wurf ergibt 2
Der Wurf ergibt 6
Der Wurf ergibt 5
Der Wurf ergibt 6
Der Wurf ergibt 2

Anmerkung: Dieses Programm gibt jedes mal andere Ergebnisse aus.

Modul readline

Mit dem Modul *readline* kann eine Autovervollständigung im Stil der Bash in die Texteingabe eingebaut werden. Mit einer geeigneten Funktion zur Vervollständigung angefangener Wörter braucht man nur noch die Tabulatortaste drücken, um bereits eindeutige Eingaben zu vervollständigen.

Modul sys

Hier folgt eine Auswahl interessanter Funktionen und Variablen innerhalb vom Modul *sys*.

Funktion	Bedeutung
argv	Liste von Kommandozeilen, die dem Skript mitgegeben wurden
exit(Arg)	Der optionale Wert *Arg* wird ausgegeben, oder, wenn es eine Zahl ist, an den aufrufenden Prozess (zum Beispiel die bash) übergeben.
exitfunc	Diese Variable enthält den Wert einer Funktion, die als letzte vor dem Programmende aufgerufen wird.
path	Liste aller Modulsuchpfade
platform	Ein String, der die Plattform, also das Betriebssystem, identifiziert.
stdin	Eingabekanal (Datei)
version	Die aktuelle Python-Version

Ein Beispielprogramm, wie man mit exit() und exitfunc umgeht:

```
#!/usr/bin/python

import sys

def LetzteWorte():
    print "Das Gold befindet sich am ..."

exitfunc = LetzteWorte()
sys.exit(42)
```

Ausgabe

user@localhost:~$ ./sys1.py
Das Gold befindet sich am ...

Leider verstarb das Programm, bevor es uns den Ort nennen konnte, wo das Gold liegt. Wir können den Rückgabewert des Skriptes in der *bash* bestimmen, in dem wir die Shell-Variable *$?* abfragen:

Ausgabe

user@localhost:~$ echo $?
42

Das folgende Programm benutzt stdin, um einen Filter zu bauen. Mit dessen Hilfe können wir die Ausgabe eines Programmes als Eingabe unseres Programmes benutzen, um zum Beispiel Dateien mit Zeilennummern zu versehen:

```
#!/usr/bin/python
import sys

for n, l in enumerate(sys.stdin):
    print "%d: %s" % (n, l[:-1])
```

Ausgabe

```
user@localhost:~$ ./sys2.py < sys2.py
0: #!/usr/bin/python
1: import sys
2:
3: for n, l in enumerate(sys.stdin):
4:     print "%d: %s" % (n, l[:-1])
```

Alternativ hätten wir das Programm auch mit `cat sys2.py | ./sys2.py` aufrufen können.

Modul tarfile

Das Modul *tarfile* ermöglicht die Behandlung von \.tar(\.gz|\.bz2)?-Dateien unter Python. Ein Beispiel:

```
#!/usr/bin/python
from sys import argv
import tarfile

filename=argv[0]
tarname=filename+".tar.gz"

tar=tarfile.open(tarname, "w:gz")
tar.add(filename)
tar.close()

file=open(filename)
file.seek(0, 2)
tar=open(tarname)
tar.seek(0, 2)
print "Original: " + str(file.tell()) + " Bytes"
print "Compressed: " + str(tar.tell()) + " Bytes"

# Noch ein paar Kommentare, damit die Datei gross genug ist. :-)
# Bei zu kleinen Dateien wirkt sich die Kompression negativ aus.
```

Ausgabe

```
user@localhost:~$ ./tar.py
Original: 464 Bytes
Compressed: 403 Bytes
```

Modul time

Das Modul *time* stellt Funktionen für das Rechnen mit Zeit zur Verfügung. Es sollte nicht mit der Klasse time im Modul *datetime* verwechselt werden. Ein Beispielprogramm:

```
#!/usr/bin/python

from time import clock, strftime

def do_loop(limit):
  start_time=clock()
  for i in range(1,limit):
    for j in range(1,limit):
      pass
  end_time=clock()
  return end_time - start_time

def calibrate():
  limit=1
  elapsed=0.0
  while (elapsed<1.0):
    limit=limit*2
    elapsed=do_loop(limit)
  return limit

print 'Kalibriere Zeitrechnung...'
limit = calibrate()
print 'Rechne (' + str(limit) + ')^2 Schleifen...'

print 'Vorher: ' + strftime('%X')
elapsed=do_loop(limit)
print 'Nachher: ' + strftime('%X')

print "Gemessene Zeit: " + str(elapsed) + "s"
```

Ausgabe

```
user@localhost:~$ ./timing.py
Kalibriere Zeitrechnung...
Rechne (4096)^2 Schleifen...
Vorher: 17:14:57
Nachher: 17:14:58
Gemessene Zeit: 1.16s
```

Modul uu

Das folgende Programm kodiert und dekodiert sich selbst:

```
#!/usr/bin/python
from sys import argv
from uu import encode, decode

infile=argv[0]
tmpfile=infile+".uu"
outfile=tmpfile+".py"

encode(infile, tmpfile)
decode(tmpfile, outfile)

file=open(outfile)
data=file.read()
lines=data.splitlines()
for line in lines:
     print line
```

Die Ausgabe soll hier nicht extra aufgeführt werden, da sie identisch mit dem Programmcode ist. Für die Interessierten hier die dabei erzeugte Datei uudemo.py.uu:

```
begin 755 uudemo.py
M(R$O=7-R+V)I;B]P>71H;VX*9G)O;2!S>7,@:6UP;W)T(&%R9W8*9G)O;2!U
M=2!I;7!O<G0@96YC;V1E+"!D96-O9&4*"FEN9FEL93UA<F=V6S!="G1M<&9I
M;&4]:6YF:6QE*R(N=74B"F]U=&9I;&4]=&UP9FEL92LB+G!Y(@H*96YC;V1E
M*&EN9FEL92P@=&UP9FEL92D*9&5C;V1E*'1M<&9I;&4L(&]U=&9I;&4I"@IF
M:6QE/6]P96XH;W5T9FEL92D*9&%T83UF:6QE+G)E860H*0IL:6YE<SUD871A
M+G-P;&ET;&EN97,H*0IF;W(@;&EN92!I;B!L:6YE<SH*(" @("!P<FEN="!L
$:6YE"@

end
```

Allgemeine Informationen über Module

Das folgende Skript zeigt, wie man mehr über ein Modul herausfindet, von dem man nur den Namen kennt.

```
#!/usr/bin/python

import os
help(os)
print dir(os)
```

Die Funktionen help() und dir() geben nützliche Informationen über die Module aus. Es ist recht praktisch, sie im interaktiven Modus (siehe Kapitel Erste Schritte) einzugeben.

Anmerkungen

[1] In der deutschsprachigen Literatur zu komplexen Zahlen ist das "i" häufig anzutreffen.

XML

WARNUNG: Dieses Kapitel ist vollständig veraltet und sollte ersetzt oder entfernt werden. Es beschreibt nicht die aktuell zumeist verwendeten XML-Bibliotheken ElementTree (http:/ / docs. python. org/ library/ xml. etree. elementtree. html) und lxml (http:/ / lxml. de/), sondern das ältere, langsame und sehr speicherintensive MiniDOM-Paket und die kompliziert zu verwendende SAX-Bibliothek. Diesen gegenüber bieten ElementTree und lxml sehr hohe Geschwindigkeit (http:/ / lxml. de/ performance. html) und Speichereffizienz (http:/ / effbot. org/ zone/ celementtree. htm), leichtere Bedienbarkeit und eine wesentlich höhere Abdeckung von XML-Standards und -Features (z.B. Validierung, XPath oder XSLT). Während es für SAX zumindest noch einige wenige vertretbare Anwendungsfälle gibt, sollte neu geschriebener Code insbesondere nicht mehr MiniDOM verwenden.

Einleitung

XML ist eine Auszeichnungssprache, mit der sich andere Sprachen definieren lassen. Ebenfalls nutzt man XML, um hierarchisch gegliederte Dokumente zu erzeugen. Mehr über XML findet sich in den Wikibooks XML und XSLT. Eine der Sprachen, die mit Hilfe von XML definiert wurden ist XHTML (HTML). Dieses Kapitel deckt Teile der XML-Module ab, die mit Python mitgeliefert werden. Wenn Sie höherwertige Anwendungen mit Schemata und Namespaces benötigen, sollten Sie sich das Modul *python-libxml2* anschauen. Es lässt sich mit den Paketverwaltungstools ihrer Linux-Distribution leicht installieren.

XML-Dokument erzeugen

Das folgende Beispiel erzeugt ein XML-Dokument und gibt es formatiert aus. Wir benutzen hierzu das Modul xml.dom. Das Dokument enthält drei Knoten (*Node* oder *Element* genannt), das eigentliche Dokument, `Test1`-Knoten, einen Knoten namens `Hello`, der ein Attribut trägt und einen Textknoten:

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

import xml.dom

# Dokument erzeugen
implement = xml.dom.getDOMImplementation()
doc = implement.createDocument(None, "Test1", None)

# Child Element "Hello"
elem = doc.createElement("Hello")
elem.setAttribute("output", "yes")
text = doc.createTextNode("Hello, World!")
elem.appendChild(text)

# Child an Dokument anhängen
doc.documentElement.appendChild(elem)

# Ausgeben
```

```
print doc.toprettyxml()
```

Ausgabe

user@localhost:~$ ./xml1.py

```
<?xml version="1.0" ?>
<Test1>
        <Hello output="yes">
                Hello, World!
        </Hello>
</Test1>
```

Mit xml.dom.getDOMImplementation() beschaffen wir uns Informationen über die aktuelle XML-Implementation. Optional können wir mit der Funktion Eigenschaften anfordern, es würde dann eine passende Implementation herausgesucht. Diese Implementation benutzen wir, um ein eigenes Dokument mit dem so genannten *Root-Element* zu erzeugen: createDocument(). Dieser Funktion könnten wir noch den Namespace und den Dokumententyp mitteilen.

Haben wir ein Dokument erzeugt, können wir mit createElement() nach Belieben neue Elemente erstellen. Knoten können Attribute haben, die man mit setAttribute() in das Element einfügt. Wird ein Element erstellt, so wird es nicht sofort in das Dokument eingefügt. appendChild() erledigt dies.

createTextNode() erzeugt einen speziellen Knoten, der nur aus Text besteht. Dieser kann keine weiteren Attribute haben. Das Root-Element holt man sich mit doc.documentElement, an dieses kann man zum Schluss den Elemente-Baum anhängen.

XML-Dokumente möchte man auch auf die Festplatte schreiben. Das folgende Beispiel implementiert eine Adresseneingabe. Lässt man den Namen bei der Eingabe leer, so wird die Eingabe abgebrochen, das XML-Dokument ausgegeben. Bitte beachten Sie, dass die Datei bei jedem Aufruf überschrieben wird. Es sollte sich also nach Möglichkeit keine andere wichtige Datei gleichen Namens im Verzeichnis befinden. Das Beispielprogramm erzeugt einen Baum, der `<Adressen>` beinhaltet. Jede dieser Adressen wird durch einen `<Adresse>`-Knoten repräsentiert, in dem sich `<Name>`- und `<Anschrift>`-Knoten befinden. Als Besonderheit wird noch ein Kommentar in jedes Adressenfeld geschrieben:

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

import xml.dom

# Dokument erzeugen
implement = xml.dom.getDOMImplementation()
doc = implement.createDocument(None, "Adressen", None)

while True:
    # Namen holen, ist er leer dann Abbruch
    name_text = raw_input("Name der Person: ")
    if name_text == "": break

    # Kommentar
    kommentar_text = "Adresse von %s" % (name_text)
    commentNode = doc.createComment(kommentar_text)
```

```
    # neues Adresse-Elemen
    adresseElem = doc.createElement("Adresse")
    adresseElem.appendChild(commentNode)
    # Name anfügen
    nameElem = doc.createElement("Name")
    adresseElem.appendChild(nameElem)
    nameTextElem = doc.createTextNode(name_text)
    nameElem.appendChild(nameTextElem)
    # Anschrift
    anschrift_text = raw_input("Anschrift: ")
    anschriftElem = doc.createElement("Anschrift")
    adresseElem.appendChild(anschriftElem)
    anschriftTextElem = doc.createTextNode(anschrift_text)
    anschriftElem.appendChild(anschriftTextElem)
    # Anhängen an Dokument
    doc.documentElement.appendChild(adresseElem)

# Ausgeben
datei = open("testxml2.xml", "w")
doc.writexml(datei, "\n", "  ")
datei.close()
```

So könnte bei geeigneter Eingabe die Datei testxml2.xml aussehen:

Ausgabe

user@localhost:~$ cat testxml2.xml

```
<?xml version="1.0" ?>
<Adressen>
  <Adresse>
    <!--Adresse von Sir Spamalot-->
    <Name>
      Sir Spamalot
    </Name>
    <Anschrift>
      Spamhouse 123
    </Anschrift>
  </Adresse>
  <Adresse>
    <!--Adresse von Lady of the Lake-->
    <Name>
      Lady of the Lake
    </Name>
    <Anschrift>
      West End 23
    </Anschrift>
  </Adresse>
  <Adresse>
```

```
    <!--Adresse von Brian-->
    <Name>
      Brian
    </Name>
    <Anschrift>
      Im Flying Circus
    </Anschrift>
  </Adresse>
</Adressen>
```

Dieses Beispiel enthält verschachtelte Elemente. Es wird nach Namen und Anschrift gefragt, dazu werden passende Knoten erzeugt und aneinander gehängt. Nach der Eingabe des Namens erzeugt createComment() einen Kommentar, der ebenfalls mit appendChild() angehängt werden kann. Neu ist, dass wir das erzeugte Dokument schreiben. Dazu nutzen wir writexml(), welche eine offene Datei erwartet. Außerdem geben wir noch die Art der Einrückung mit: Zwischen zwei Elementen soll eine neue Zeile eingefügt werden, je zwei Element-Ebenen sollen durch Leerzeichen getrennt werden.

XML lesen

Eine XML-Datei kann man bequem mit den Funktionen von xml.dom.minidom, einer "leichtgewichtigen" Implementation von XML lesen. Hierzu nutzen wir eine rekursive Funktion, um uns alle Knoten auszugeben.

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

import xml.dom.minidom

datei = open("testxml2.xml", "r")
dom = xml.dom.minidom.parse(datei)
datei.close()

def dokument(domina):
    for node in domina.childNodes:
        print "NodeName:", node.nodeName,
        if node.nodeType == node.ELEMENT_NODE:
            print "Typ ELEMENT_NODE"
        elif node.nodeType == node.TEXT_NODE:
            print "Typ TEXT_NODE, Content: ", node.nodeValue.strip()
        elif node.nodeType == node.COMMENT_NODE:
            print "Typ COMMENT_NODE, "
        dokument(node)

dokument(dom)
```

Ausgabe

```
user@localhost:~$ ./xml3.py
NodeName: Adressen Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Adresse Typ ELEMENT_NODE
```

NodeName: #text Typ TEXT_NODE, Content:
NodeName: #comment Typ COMMENT_NODE,
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Name Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content: Sir Spamalot
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Anschrift Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content: Spamhouse 123
NodeName: #text Typ TEXT_NODE, Content:
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Adresse Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content:
NodeName: #comment Typ COMMENT_NODE,
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Name Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content: Lady of the Lake
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Anschrift Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content: West End 23
NodeName: #text Typ TEXT_NODE, Content:
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Adresse Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content:
NodeName: #comment Typ COMMENT_NODE,
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Name Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content: Brian
NodeName: #text Typ TEXT_NODE, Content:
NodeName: Anschrift Typ ELEMENT_NODE
NodeName: #text Typ TEXT_NODE, Content: Im Flying Circus
NodeName: #text Typ TEXT_NODE, Content:
NodeName: #text Typ TEXT_NODE, Content:

Die Überraschung dürfte groß sein. In einer so kleinen Textdatei werden so viele Knoten gefunden. Der Ausspruch "Man sieht den Wald vor lauter Bäumen nicht mehr" dürfte hier passen. Im Programm wird mit xml.dom.minidom.parse() eine offene Datei gelesen. Die Funktion *dokument()* wird mit jedem Kindelement, das mit childNodes ermittelt wird, neu aufgerufen. Kinder können Elemente, wie `<Adresse>` oder Kommentare oder beliebiger Text sein. Beliebigen Text, in der Ausgabe an den Stellen mit `#text` zu erkennen, haben wir alleine dadurch recht oft, da wir den Text mit Einrückungszeichen (Indent) beim Speichern gefüllt haben. Nur sehr wenige `TEXT_NODE`-Zeilen stehen hierbei für sinnvollen Text, alle anderen sind Knoten, die nur wegen der Leerzeichen und Newline-Zeichen (*\n*) hineingeschrieben wurden. Da wir strip() bei jeder Ausgabe benutzen, sehen wir von diesen Zeichen nichts.

Um nur die interessanten Elemente auszugeben, müssen wir die Struktur berücksichtigen. Wir lassen alles außer Acht, was nicht zum gewünschten Elementinhalt passt:

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

```
import xml.dom.minidom

datei = open("testxml2.xml", "r")
dom = xml.dom.minidom.parse(datei)
datei.close()

def liesText(pretext, knoten):
    for k in knoten.childNodes:
        if k.nodeType == k.TEXT_NODE:
            print pretext, k.nodeValue.strip()

def liesAdressen(knoten):
    for num, elem in enumerate(knoten.getElementsByTagName("Adresse")):
        print "%d. Adresse:" % (num + 1)
        for knotenNamen in elem.getElementsByTagName("Name"):
            liesText("Name:", knotenNamen)
        for knotenAnschrift in elem.getElementsByTagName("Anschrift"):
            liesText("Anschrift:", knotenAnschrift)
        print

def dokument(start):
    for elem in start.getElementsByTagName("Adressen"):
        liesAdressen(elem)

dokument(dom)
```

Ausgabe

```
user@localhost:~$ ./xml4.py
1. Adresse:
Name: Sir Spamalot
Anschrift: Spamhouse 123
2. Adresse:
Name: Lady of the Lake
Anschrift: West End 23
3. Adresse:
Name: Brian
Anschrift: Im Flying Circus
```

Diese Ausgabe ist wesentlich nützlicher. Mit getElementsByTagName() können wir für die interessanten Tags alle Kindelemente holen. In der Funktion *dokument()* wird so für den Adressen-Tag, das Root-Element, genau einmal die Funktion *liesAdressen()* aufgerufen. Diese Funktion erzeugt die eigentliche Adressenausgabe. Es wird dabei über alle `Adresse`-Elemente iteriert, da Kindelemente von `Adresse` nur `Name` und `Anschrift` sein können, werden diese in der Funktion ebenfalls verarbeitet. Alle Textelemente werden von *liesText()* bearbeitet.

SAX

SAX ist ein Ereignis-basierendes Protokoll zum Lesen von XML-Dokumenten. Jedes Mal, wenn ein Ereignis auftritt, zum Beispiel ein Element gelesen wird, wird eine Methode aufgerufen, die dieses Ereignis behandelt. Anders als bei DOM können so sehr lange XML-Dokumente gelesen werden, die wesentlich größer sind als der zur Verfügung stehende Speicher. Das folgende Beispiel zeigt, wie man einen Handler erstellt, der auf einige Ereignisse reagieren kann:

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

import xml.sax

class MiniHandler(xml.sax.handler.ContentHandler):
    def startDocument(self):
        print "ANFANG"

    def endDocument(self):
        print "ENDE"

    def startElement(self, name, attrs):
        print "Element", name

    def characters(self, content):
        s = content.strip()
        if s != "":
            print "Textinhalt:", s

handler = MiniHandler()
datei = open("testxml2.xml", "r")
xml.sax.parse(datei, handler)
datei.close()
```

Ausgabe

```
user@localhost:~$ ./xml5.py
ANFANG
Element Adressen
Element Adresse
Element Name
Textinhalt: Sir Spamalot
Element Anschrift
Textinhalt: Spamhouse 123
Element Adresse
Element Name
Textinhalt: Lady of the Lake
Element Anschrift
Textinhalt: West End 23
Element Adresse
Element Name
```

Textinhalt: Brian
Element Anschrift
Textinhalt: Im Flying Circus
ENDE

xml.sax.handler.ContentHandler ist die Basisklasse für eigene Handler. In davon abgeleiteten Klassen werden einige Ereignisse wie startDocument() neu definiert, um so passend zur Anwendung darauf reagieren zu können. In unserem Beispiel geben wir die Meldungen aus. Eine Instanz des MiniHandlers wird, neben der offenen Datei, xml.sax.parse() mitgegeben. Das Dokument wird sukzessive verarbeitet, die im MiniHandler anlaufenden Ereignisse dort bearbeitet. Diese sind selbstverständlich nur ein Auszug aller möglichen Ereignisse.

Nützliche Helfer

Für diese nützlichen Helfer bindet man das Modul *xml.sax.saxutils* ein:

Funktion	Bedeutung
escape(Daten[, Entitäten])	Ersetzt in einem String "*Daten*" alle Vorkommen von Sonderzeichen durch im Dictionary angegebene Entitäten. Zum Beispiel < durch <. Einige Entitäten werden ohne ein erweitertes Entitäten-Verzeichnis ersetzt.
unescape(Daten[, Entitäten])	Wie *escape()*, nur umgekehrt. Für ein Beispiel siehe Kapitel Netzwerk. Sie können ein für *escape()* vorbereitetes Entitäten-Verzeichnis hier nicht wiederverwenden, sondern müssen den Inhalt umkehren.

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

import xml.sax.saxutils as util

buch = {'ä' : '&auml;'}

print util.escape("Anzahl der Männer < Anzahl aller Menschen!", buch)
```

Ausgabe

```
user@localhost:~$ ./xml6.py
Anzahl der M&auml;nner &lt; Anzahl aller Menschen!
```

Hier brauchten wir das Entitätenverzeichnis nur für die Umlaute, das Kleiner-Zeichen wurde per Vorgabe ersetzt.

Zusammenfassung

In diesem Kapitel haben Sie einen Überblick über die Möglichkeiten der Verarbeitung von XML-Dokumenten mit den mitgelieferten Modulen bekommen. Es wurden Techniken wie DOM und SAX vorgestellt. Tiefgreifendere Techniken werden von anderen extern erhältlichen Modulen abgedeckt. Diese liegen außerhalb der Zielsetzung dieses Buches.

Datenbanken

In diesem Kapitel geht es um die Ansteuerung von Datenbakmanagementsystemen, kurz **DBMS**. Sie dienen dazu, Daten dauerhaft zu speichern und mit Hilfe einer eigenen Sprache, zumeist **SQL**, selektiv abzufragen. Im Regal EDV bei Wikibooks finden Sie einige Bücher zum Thema Datenbanken und SQL. Wir verwenden die Begriffe DBMS und Datenbanken in den folgenden Abschnitten synonym.

Wir stellen einige Datenbanksysteme vor und zeigen kurz, wie diese Systeme in Python angesteuert werden. Es geht uns hierbei insbesondere nicht um die Darstellung der Abfragesprache.

SQLite

SQLite ist eine Datenbank, die auf einer einzelnen Datei basiert. Es muss kein Server installiert werden, dafür ist es nur schwer möglich, dass verschiedene Anwendungen gleichzeitig auf eine Datenbank zugreifen. Für Webapplikationen eignet sich dieses System nicht so gut wie andere hier vorgestellte Datenbanksysteme. Dafür sind Datenbanken schnell und ohne lästigen Ballast eingerichtet. Der Zugriff erfolgt mit Hilfe einiger weniger API-Funktionen. Das folgende Beispiel legt eine Datenbank als Datei an, fragt den Nutzer nach Eingaben und speichert diese Eingaben in der Datenbank. **STRG-C** bricht die Eingabe ab.

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

import sqlite3

con = sqlite3.connect('sl1.db')
con.isolation_level = None

con.execute("CREATE TABLE IF NOT EXISTS tiere (name TEXT, farbe TEXT)")

try:
    while True:
        tier = raw_input("(Abbruch mit STRG-C) Sag mir mal ein Tier> ")
        farbe = raw_input("Welche Farbe hat \"%s\"? > " % tier)
        con.execute("INSERT INTO tiere(name, farbe) VALUES(?, ?)",
(tier, farbe))
except:
    print ; print

rows = con.execute("SELECT * FROM tiere")
print "Meine Lieblingstiere:"
for row in rows:
    print row[0], "hat die Farbe", row[1]
```

Ausgabe

```
user@localhost:~$ ./sl1.py
(Abbruch mit STRG-C) Sag mir mal ein Tier> Maus
Welche Farbe hat "Maus"? > mausgrau
(Abbruch mit STRG-C) Sag mir mal ein Tier> Bär
Welche Farbe hat "Bär"? > braun
(Abbruch mit STRG-C) Sag mir mal ein Tier> Delfin
Welche Farbe hat "Delfin"? > blau-grau
(Abbruch mit STRG-C) Sag mir mal ein Tier>STRG-C

Meine Lieblingstiere:
Maus hat die Farbe mausgrau
Bär hat die Farbe braun
```

Delfin hat die Farbe blau-grau

Mit connect() wird die Verbindung zu einer SQLite3-Datei hergestellt. Falls keine Datei gewünscht wird, kann durch den Parameter *:memory:* die Datenbank auch im verfügbaren Speicher angelegt werden. Zurückgegeben wird ein so genanntes "Connection-Object".

SQLite3 ist transaktionsbasiert. Wir müssten bei allen die Datenbank ändernden Zugriffen wie `INSERT` ein `Commit` durchführen, durch die Angabe von isolation_level = None wird die DBMS in den *Autocommit-Modus* gesetzt, Änderungen werden also sofort geschrieben.

execute() führt ein SQL-Statement aus. Ein solches Statement kann Parameter beinhalten, die in SQL als Fragezeichen[1] geschrieben werden. Die Parameter werden als Tupel übergeben. execute() gibt ein so genanntes *Cursor*-Objekt zurück. Bei `SELECT`-Abfragen können wir dieses nutzen, um über alle abgefragten Zeilen zu iterieren.

MySQL

Wir stellen ihnen hier ein Beispiel vor, bei dem auf eine *entfernte* Datenbank zugegriffen wird:

```
#!/usr/bin/python

import MySQLdb

db =
MySQLdb.connect("anderer.host.entf3rnt.de","Tandar","ge431m","meine_datenbank")
cursor = db.cursor()
cursor.execute("SELECT VERSION()")
row = cursor.fetchone()
print "server version:", row[0]
cursor.close()
db.close()
```

Ausgabe

```
user@localhost:~$ ./my1.py
server version: 5.0.32-Debian_7etch8-log
```

Die Verbindung wird hier erzeugt mit connect(). Der entfernte Server, der Benutzername, das Passwort und die eigentliche Datenbank, mit der wir uns verbinden wollen, werden als Parameter übergeben. Zurück erhalten wir einen Datenbankhandle oder eine Exception, wenn etwas schief ging.

Anschließend besorgen wir uns ein Cursor-Objekt, mit dessen Hilfe wir Abfragen formulieren können (execute("SELECT VERSION()")) und eine Ergebniszeile erhalten (fetchone()).

Die Ausgabezeile können wir sodann sofort ausgeben. Zum Schluss wird der Cursor wie auch die Verbindung zur Datenbank jeweils mit close() geschlossen.

Das nun folgende Beispiel ergänzt das obere Beispiel um die Fähigkeit, eine Tabelle anzulegen, Daten hineinzuschreiben und diese anschließend zu selektieren:

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

import MySQLdb

db =
```

```
MySQLdb.connect("anderer.host.entf3rnt.de","Tandar","ge431m","meine_datenbank")
cursor = db.cursor()
cursor.execute("CREATE TABLE IF NOT EXISTS bar(uid int, anmeldedatum
date)")
cursor.execute("INSERT INTO bar values(0, '2009-04-17')")
cursor.execute("INSERT INTO bar values(0, '2009-03-13')")
cursor.execute("INSERT INTO bar values(500, '2009-04-16')")
cursor.close()
db.commit()

db.query("SELECT uid, anmeldedatum FROM bar WHERE uid=0")
result = db.store_result()

nZeilen = result.num_rows()
nSpalten = result.num_fields()

print "Anzahl Zeilen:", nZeilen, "Anzahl Spalten:", nSpalten

for zeile in xrange(nZeilen):
    row = result.fetch_row()
    uid, datum = row[0]
    print uid, datum
db.close()
```

Ausgabe

```
user@localhost:~$ ./my2.py
Anzahl Zeilen: 2 Anzahl Spalten: 2
0 2009-04-17
0 2009-03-13
```

Anders als im oberen Beispiel nutzen wir query(), um eine Anfrage an die Datenbank zu senden. Wir können das Ergebnis dieser Abfrage mit store_result() speichern und erhalten so die Ergebnismenge als Speicherobjekt zurück. Alternativ können wir use_result() verwenden, um die Ergebniszeilen nach und nach zu erhalten.

Mit num_rows() und num_fields() ermitteln wir die Anzahl der Ergebniszeilen und die Anzahl der Felder pro Zeile.

fetch_row() entnimmt der Ergebnismenge eine Zeile und liefert sie als Zeilentupel heraus. Die einzelnen Felder (hier uid und datum) sind selbst ein Tupel im ersten Element der Zeile.

AnyDBM

AnyDBM ist ein Modul, welches sich mit DBM-ähnlichen Datenbanken beschäftigt. Von dieser Sorte gibt es zwei Stück[2], nämlich *DBM* und *GNU-DBM*. Diese unterscheiden sich in ihrem internen Aufbau und der Lizenz, sind aber ansonsten gleich.

DBM-Datenbanken sind eine Art von Wörterbüchern, sie speichern String-Dictionaries ab. Folgendes Beispiel zeigt die grundsätzliche Arbeitsweise mit ihnen:

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

import anydbm
```

```
db = anydbm.open("wertpapiere.gdbm", "c", 0660)
db["Siemens"] = "1000"
db["Apple"] = "2000"
db["Red Hat"] = "3000"
db.close()

db = anydbm.open("wertpapiere.gdbm", "r")
for key, value in db.iteritems():
    print "Von der Aktie", key, "habe ich", value, "Stück"
db.close()
```

Ausgabe

```
user@localhost:~$ ./dbm1.py
Von der Aktie Apple habe ich 2000 Stück
Von der Aktie Red Hat habe ich 3000 Stück
Von der Aktie Siemens habe ich 1000 Stück
```

Man kann nur Strings speichern. Mit open(dateiname, art, dateiflags) wird eine solche Datei angelegt oder gelesen. Die *art* ist `"c"` zum Erzeugen der Datenbank, wenn sie nicht existiert, `"w"` zum Schreiben, `"r"` zum Lesen. *dateiflags* ist ein numerischer Wert, der, in Abhängigkeit von der aktuellen *umask*[3], den Dateimodus spezifiziert.

Die Werte werden wie in einem Dictionary eingetragen, anschließend wird die Datei mit close() wieder geschlossen. Über die Schlüssel-Wertpaare kann man mit der Methode iteritems() iterieren.

Zusammenfassung

In diesem Kapitel haben wir einige Datenbanken kennen gelernt und die typische Arbeitsweise mit ihnen aufgezeigt.

Anmerkungen

[1] Dieses ist sogleich die sichere Variante, sie schützt gut vor SQL-Injection.
[2] Soweit wir wissen...
[3] Siehe die Manual-Seite zu *bash(1)*

Netzwerk

Einfacher Zugriff auf Webressourcen

"Wie ist das Wetter auf Hawaii?", *"Welches Stück wird gerade im Radio gespielt?"* oder *"Wie lautet die aktuelle Linux-Kernelversion?"* sind typische Fragen, deren Antworten sich zumeist mit Hilfe eines Browsers finden lassen. Möchte man jedoch in einer konkreten Anwendung nicht immer den Browser aufrufen, helfen zumeist kleine Programme dabei, die benötigten Antworten automatisch zu finden. Die aktuelle Kernelversion zum Beispiel ist leicht aus einer Webseite herauszuextrahieren. Hierzu sind allerdings Kenntnisse vom aktuellen Aufbau der Seite unerlässlich.

```
#!/usr/bin/python

import urllib2, re


url = urllib2.urlopen('http://www.kernel.org')

html = url.read()
```

```
url.close()


table = re.search("""<table class="kver">.*?<td >(?P<kertext>.*?)</td>\s*<td ><strong>(?P<kerver>.*?)</strong></td>""", html, re.S)
print table.group('kertext'), table.group('kerver')
```

Ausgabe

```
user@localhost:~$ ./url1.py
mainline: 3.7-rc6
```

Die Funktion urllib2.urlopen() öffnet eine angegebene Webseite, die wir mit url.read() vollständig lesen können. In der Variablen html ist nun der gesamte Inhalt der Webseite gespeichert, wir benötigen also den weiteren Netzwerkzugriff nicht mehr und schließen daher die Verbindung mit url.close(). Schaut man sich die Webseite auf http:/ / www. kernel. org (http:/ / www. kernel. org) im Quellcode an, so stellt man fest, dass die für uns interessanten Informationen in einer Tabelle hinterlegt sind, die mit `<table class="kver">` beginnt. Innerhalb der ersten beiden `<td>`-Zeilen befinden sich die Informationen, die wir mit dem regulären Ausdruck herauslösen.

Lesen bei WikiBooks

Etwas komplizierter ist es, auf Wikibooks-Inhalte zuzugreifen. Die MediaWiki-Software verlangt ein Mindestmaß an übertragenen Browser-Informationen und das Protokoll *GET*. Das nun folgende Programm benutzt daher einen Request, dem wir einige Header hinzufügen.

```
#!/usr/bin/python

import urllib2, re
import xml.sax.saxutils as util

header = {'User-agent' : 'PythonUnterLinux', 'Accept-Charset' :
'utf-8'}
request =
urllib2.Request('http://de.wikibooks.org/w/index.php?title=Diskussion:Python_unter_Linux:_Netzwerk&action=edit',
 headers=header)

print "DEBUG: ", request.get_method()
url = urllib2.urlopen(request)
html = url.read()
url.close()

found = re.search("<textarea .*?>(?P<inhalt>.*?)</textarea>", html, re.S)
print util.unescape(found.group('inhalt'))
```

Ausgabe

```
user@localhost:~$ ./url2.py
DEBUG: GET
<!-- Dient zum Testen der hier vorgestellten Skripte -->
TESTINHALT
```

Die *HTTP Request Header* geben den Namen unseres selbstgeschriebenen Browsers mit "PythonUnterLinux" an. `Accept-Charset` ist eine Information darüber, welche Zeichencodierung dieser Browser versteht. Diese Header

werden als Argument unserem Request mit auf den Weg gegeben. Die Verbindung wird wieder mit urlopen() geöffnet, der Rest des Programmes gleicht dem aus dem ersten Beispiel.

Wir öffnen die Diskussionsseite dieser Seite, um die dort enthaltenen Informationen bequem aus einer HTML-Textbox extrahieren zu können. Mit dem Argument action=edit in der URL geben wir an, dass wir die Seite eigentlich zum Schreiben öffnen. Das machen wir, da so die interessanten Informationen leichter zu finden sind. Der Reguläre Ausdruck extrahiert aus dem `<textarea>`-Bereich der Webseite den Inhalt. Dieser Inhalt ist HTML-codiert, weswegen wir den String mit xml.sax.saxutils.unescape() wieder in eine natürlich-lesbare Zeichenkette umwandeln.

Auf WikiBooks schreiben

Um anonym[1] auf WB schreiben zu können, müssen zuerst einige Informationen von der Webseite geholt werden, auf der Sie schreiben möchten. Im HTML-Quellcode der Seite sind sie als `<input...` zu erkennen. Diese Daten müssen beim schreibenden Zugriff mitübergeben werden, es handelt sich in diesem Fall um ein *POST*-Request. Das folgende Programm fügt auf die Diskussionsseite dieses Kapitels einen kleinen Text hinzu, weitere Ausgabe erfolgt nicht.

```
#!/usr/bin/env python


import urllib, urllib2, re
import xml.sax.saxutils as util


# Lies die Seite
header = {'User-agent' : 'PythonUnterLinux', 'Accept-Charset' :
'utf-8'}
request =
urllib2.Request('http://de.wikibooks.org/w/index.php?title=Diskussion:Python_unter_Linux:_Netzwerk&action=edit')
url = urllib2.urlopen(request)
html = url.read()
url.close()


# RegExp vorbereiten
s_str = "<form id=\"editform\" name=\"editform\" method=\"post\" action=\"(?P<actionurl>.*?)\".*?>"
s_str += ".*?<input type=\"text\" name=\"wpAntispam\" id=\"wpAntispam\" value=\"(?P<wpAntispamValue>)\" />"
s_str += ".*?<input type=\'hidden\' value=\"(?P<wpSectionValue>.*?)\" name=\"wpSection\" />"
s_str += ".*?<input type=\'hidden\' value=\"(?P<wpStarttimeValue>.*?)\" name=\"wpStarttime\" />"
s_str += ".*?<input type=\'hidden\' value=\"(?P<wpEdittimeValue>.*?)\" name=\"wpEdittime\" />"
s_str += ".*?<input type=\'hidden' value=\"(?P<wpScrolltopValue>.*?)\" name=\"wpScrolltop\" id=\"wpScrolltop\" />"
s_str += ".*?<textarea name=\"wpTextbox1\" id=\"wpTextbox1\" cols=\"80\" rows=\"25\" .*?>(?P<content>.*?)</textarea>"
s_str += ".*?<input tabindex=\'2\' type=\'text\' value=\"(?P<wpSummaryValue>.*?)\" name=\'wpSummary\' id=\'wpSummary\'
maxlength=\'200\' size=\'60\' />"
s_str += ".*?<input name=\"wpAutoSummary\" type=\"hidden\" value=\"(?P<wpAutoSummaryValue>.*?)\" />"
s_str += ".*?<input type=\'hidden\' value=\"(?P<wpEditTokenValue>.{2})\" name=\"wpEditToken\" />"


# RegExp ausfuehren
found = re.search(s_str, html, re.S)


# Neuen Inhalt aufbauen
```

```
new_content = util.unescape(found.group("content")) + "\n===Neuer
Inhalt===\ntest neues Skript"
summary = "Python unter Linux Browser"

# zu uebermittelnde Daten vorbereiten
data_dict = {"wpAntispam" : found.group("wpSectionValue"),
    "wpSection" : found.group("wpSectionValue"),
    "wpStarttime" : found.group("wpStarttimeValue"),
    "wpEdittime" : found.group("wpEdittimeValue"),
    "wpScrolltop" : found.group("wpScrolltopValue"),
    "wpTextbox1" : new_content,
    "wpSummary" : summary,
    "wpAutoSummary" : found.group("wpAutoSummaryValue"),
    "wpEditToken" : found.group("wpEditTokenValue")}

data = urllib.urlencode(data_dict)

# und abschicken...
request =
urllib2.Request('http://de.wikibooks.org/w/index.php?title=Diskussion:Python_unter_Linux:_Netzwerk&action=edit')
url = urllib2.urlopen(request)
url.close()
```

Der erste Teil des Programmes ist dazu da, die Seite zu lesen. Der neue Inhalt, wie auch die Zusammenfassung der Änderung, werden in den Variablen new_content und summary vorbereitet. Auf der gelesenen Webseite befinden sich einige Daten wie `wpSummary` -Zusammenfassung der Änderung-, `wpAutoSummary` -Konstanter Wert, der zurückgegeben werden muss- und `wpTextbox1`, der den eigentlichen, zu übermittelnden Inhalt der Seite darstellt. Diese Felder müssen extrahiert und, gegebenfalls modifiziert, zurückgeschickt werden. urllib.urlencode() erzeugt aus dem Dictionary einen String, der dem Request mitgegeben wird.

Anmelden mit Cookies

In manchen Foren und Wikis kann oder muss man sich anmelden, um Beiträge zu schreiben. Die Anmeldedaten werden häufig in einem Cookie gespeichert. So auch bei Wikibooks. Das folgende Beispiel zeigt eine Möglichkeit, sich bei Wikibooks anzumelden. Auf eine Testausgabe auf der Diskussionsseite haben wir diesmal verzichtet, den Code dazu kennen Sie schon aus dem vorherigen Beispiel. Es wird lediglich eine Bestätigung über den erfolgreichen Anmeldevorgang ausgegeben:

```
#!/usr/bin/python

import urllib, urllib2, re
import cookielib

USERNAME = "Testaccount007"
PASSWORD = "test123"

header = {'User-agent' : 'PythonUnterLinux', 'Accept-Charset' :
'utf-8'}
data_dict = {"wpName" : USERNAME, "wpPassword" : PASSWORD, "wpRemember"
```

```
 : "1"}
data = urllib.urlencode(data_dict)


cookieMonster = cookielib.CookieJar()
opener =
urllib2.build_opener(urllib2.HTTPCookieProcessor(cookieMonster))
urllib2.install_opener(opener)
request =
urllib2.Request('http://de.wikibooks.org/w/index.php?title=Spezial:Anmelden&action=submitlogin&type=login',
 data, header)
url = urllib2.urlopen(request)
html = url.read()
url.close()


res = re.search("<p>(?P<ret>.*?)</p>", html, re.S)
print res.group('ret')
```

Ausgabe

```
user@localhost:~$ ./url4.py
Du bist jetzt als „Testaccount007“ bei Wikibooks angemeldet.
```

Auf der Anmeldeseite stehen im Formular unter anderem die Eingabezeilen, die mit `wpName` und `wpPassword` im HTML-Code benannt sind. Diese Variablen werden mittels urllib.urlencode() in Daten für den anstehenden *POST*-Request umgewandelt. Bei der Verwaltung von Cookies hilft uns die Klasse cookielib.CookieJar(). Mit ihrer Hilfe bauen wir einen *Opener*, den man sich als eine Art Schlüssel vorstellen kann. Mit opener.open() könnten wir die Webseite schon öffnen, jedoch haben wir es an der Stelle vorgezogen, den Opener global mit urllib2.install_opener() zu installieren.

Zeitserver

Selbstverständlich kann man mit Python nicht nur Browser schreiben, sondern ganz bequem auch Server. Wir implementieren hier einen Server, bei dem sich Clients anmelden müssen und zur Belohnung die Zeit dauerhaft angezeigt bekommen. Clients sind hier einfache Telnet-Sitzungen, die sie nach dem Starten des Servers in einem anderen Fenster öffnen können.

Unser Server wartet auf zwei Verbindungen. Die Clients müssen sich durch die Eingabe von **`HELO`** ausweisen. Sind zwei Clients akzeptiert worden, wird an beide die aktuelle Zeit geschickt.

```
#!/usr/bin/python

import socket, select, time

host = '127.0.0.1'
port = 6000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))

s.listen(2)
```

```
allConnected = False
allClients = []

# Clients verbinden
while not allConnected:
    clientsock, clientaddr = s.accept()
    print "Verbindungswunsch von", clientsock.getpeername(),
    data = clientsock.recv(4096)
    data.strip()
    if data.startswith("HELO"):
        allClients.append(clientsock)
        print "akzeptiert"
    else:
        clientsock.close()
        print "nicht akzeptiert"
    print "Wir haben ", len(allClients), "Clients"
    if len(allClients) == 2:
        allConnected = True

# Daten senden, empfangen
while True:
    time.sleep(2)
    listIn, listOut, listTmp = select.select(allClients, allClients,
[], 1)
    for sock in listIn:
        data = sock.recv(4096)
        print "--\nAnkommende Daten:", len(data), "Zeichen. Inhalt: ",
data
        if len(data) == 0:
            print "Client hat die Verbindung getrennt"
            sock.close()
            allClients.remove(sock)
            listIn.remove(sock)
            listOut.remove(sock)
    for sock in listOut:
        msg = time.strftime("%H:%M:%S\n")
        num = sock.send(msg)
```

Mit socket.socket(socket.AF_INET, socket.SOCK_STREAM) wird ein Socket erzeugt, eine Struktur, die eine Netzwerkverbindung repräsentiert. In diesem Fall soll es eine dauerhafte TCP/IP-Verbindung sein. Der Port, auf dem diese Verbindung lauscht, soll wiederverwertbar sein. Wenn der Server sich beendet, soll der Port wieder genutzt werden können. Dazu dient setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1). Anschließend binden wir den Port an die IP-Adresse und die Portnummer (bind((host, port))). Solange nicht alle Clients verbunden sind, werden Verbindungswünsche akzeptiert (accept()). Sendet der Client etwas, das mit **HELO** beginnt, so wird er aufgenommen, sonst abgelehnt.

Die folgende Schleife wird alle zwei Sekunden ausgeführt. select.select(allClients, allClients, [], 1) wartet höchstens eine Sekunde auf Clients, die zum Senden oder Empfangen bereit sind. Die Funktion gibt drei Listen zurück. Die ersten beiden Listen enthalten dabei diejenigen Sockets, die zum Senden oder zum Empfangen bereit sind. Sendet ein Socket eine leere Zeichenkette, dann bedeutet das, dass er sich beendet hat. Sonst wird an alle Clients die Zeit ausgegeben.

Ausgabe

user@localhost:~$ ./server.py
Verbindungswunsch von ('127.0.0.1', 40510) akzeptiert
Wir haben 1 Clients
Verbindungswunsch von ('127.0.0.1', 40511) akzeptiert
Wir haben 2 Clients

Auf zwei anderen Konsolen (Hier wird **HELO** eingegeben):

Ausgabe

user@localhost:~$ telnet localhost 6000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
HELO
15:46:20
...

Chatserver

Einen Server fürs Chatten zu schreiben ist nur wenig schwerer. Hierbei geht es darum, dass sich einige Clients während einer Chat-Sitzung verbinden, wieder abmelden und Daten von einem Client an alle anderen Clients übertragen werden. Folgendes Beispiel verdeutlicht das:

```
#!/usr/bin/python

import socket, select, time

host = '127.0.0.1'
port = 6000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
allClients = []

# Daten senden, empfangen
while True:
    time.sleep(2)
    # Server socket hinzufuegen
    listIn, listOut, listTmp = select.select(allClients + [s],
allClients, [], 1)
    for sock in listIn:
```

```
        if sock is s:
            clientsock, clientaddr = s.accept()
            print "+ Verbindungswunsch von", clientsock.getpeername(),
"akzeptiert"
            allClients.append(clientsock)
        else:
            data = sock.recv(4096)
            if len(data) == 0 or data.startswith("quit"):
                print "- Client", clientsock.getpeername(), "hat die
Verbindung getrennt"
                sock.close()
                allClients.remove(sock)
                listIn.remove(sock)
                listOut.remove(sock)
            else:
                # nicht an den Sender schreiben
                listOut.remove(sock)
                for sout in listOut:
                    sout.send(data)
```

Der Server läuft anfangs ohne Verbindung. Wenn ein Client sich verbinden möchte, so wird der Server-Socket (s) lesebereit. Voraussetzung dafür ist, dass wir im Aufruf von select() den Server als im Prinzip *empfangsbereit* einstufen.

Ein Client kann sich nun durch Eingabe von **`quit`** beenden. Falls der Client Daten sendet, werden diese Daten an alle anderen Clients ebenfalls verschickt, nicht jedoch an denjenigen, der gesendet hat. Dadurch soll ein weiteres Echo der Eingabe vermieden werden.

Die Ausgabe des Programms ist bei einer Telnet-Sitzung das, was man von einem Chat erwartet: Es werden Daten an alle Clients übertragen. Deswegen verzichten wir hier auf die Darstellung.

Zusammenfassung

Um spezialisierte Browser zu schreiben, benötigt man Wissen über die Webseite und Kenntnisse der benötigten Protokolle. Beides kann man zumeist aus den Quelltexten der Webseiten herauslesen. Zumeist sind nur wenige Zeilen Programmcode nötig, um eine Webseite komplett einzulesen, dafür viele Programmzeilen, die Daten aus dem HTML-Code zu extrahieren. Falls Sie die hier vorgestellten und gegebenenfalls modifizierten Programme auf Wikibooks zu mehr, als nur zu Testzwecken benutzen wollen, fragen Sie bitte die Administratoren um Erlaubnis. Python enthält ebenfalls eine vollständige Anbindung an Netzwerkprotokollen. Hiermit lassen sich Clients und Server schreiben.

Anmerkung

[1] So anonym sind Sie gar nicht. Auf WB werden alle Änderungen, die ein Nutzer oder eine IP macht protokoliert

---

PyGame

PyGame ist eine Gruppe von Modulen, die einen Programmierer bei der Erstellung von Spielen unterstützt. Diese Module beinhalten Zugriff auf genau ein Grafikfenster. PyGame unterstützt den Entwickler mit Grafikprimitiven, einfachem Soundzugriff sowie Sprites und vielem mehr. Typische GUI-Elemente gibt es in diesen Modulen jedoch

nicht. PyGame beinhaltet eine Grafikbibliothek, zu der Wikibooks auch ein Buch hat, nämlich **SDL**. Einige der hier angeführten Beispiele sind diesem Buch entlehnt. Hintergrundinformationen zu den Modulen findet sich auf der PyGame-Webseite (http://www.pygame.org/).

Ein Grafikfenster

```
#!/usr/bin/python
import pygame
import sys

def init():
    WINWIDTH = 640
    WINHEIGHT = 480
    pygame.init()
    screen = pygame.display.set_mode((WINWIDTH, WINHEIGHT))
    screen.fill((200, 200, 200))
    pygame.display.update()

def event_loop():
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
            elif event.type == pygame.KEYDOWN:
                sys.exit()

if __name__ == '__main__':
    init()
    event_loop()
```

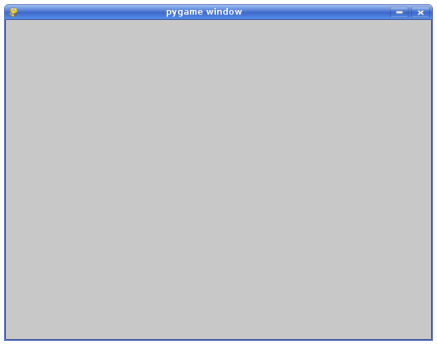


Bild der Anwendung

PyGame muss initialisiert werden, das übernimmt die Funktion pygame.init(), welche vor allen anderen PyGame-Funktionen aufgerufen werden muss. Ein neues Fenster erhalten wir mit der Methode pygame.display.set_mode(), der wir neben der Fenstergröße als Tupel auch noch weitere Flags mitgeben könnten. Weitere Angaben wären zum Beispiel, dass wir den Vollbildmodus (pygame.FULLSCREEN) oder OpenGL-Unterstützung (pygame.OPENGL) wünschen. pygame.display.set_mode() übergibt eine so genannte *Surface*, eine Struktur, die das Grafikfenster repräsentiert.
Den Bildschirm füllen wir mit einer Farbe screen.fill((200, 200, 200)) (grau), die wir als Tupel übergeben. Damit diese Färbung wirksam wird, frischt pygame.display.update() den gesamten Bildschirm auf.

PyGame speichert alle Ereignisse wie Tastendrücke, Mausbewegungen und Joystick-Kommandos in einer Folge von Events. Mit pygame.event.get() holen wir uns das nächste anstehende Ereignis. Im Attribut event.type ist die Art von Ereignis gespeichert, die anliegt. In unserem einfachen Beispiel wird nur nach pygame.QUIT und pygame.KEYDOWN verzweigt, in diesen Fällen beendet sich das Programm. Die nachstehende Tabelle enthält

einen Ausschnitt der möglichen Ereignisse, für eine vollständige Liste ziehen Sie bitte die Online-Dokumentation zu Rate.

Ereignis	Bedeutung
QUIT	Anwender wünscht das Programm zu beenden, beispielsweise durch Drücken des Schließen-Knopfes
KEYDOWN	Taste wurde heruntergedrückt
KEYUP	heruntergedrückte Taste wurde wieder losgelassen
MOUSEMOTION	Die Maus wurde bewegt
MOUSEBUTTONDOWN	Ein Knopf an der Maus wurde gedrückt
USEREVENT	Ein frei definierbares Ereignis passierte. Genau genommen gibt es hier viele weitere Events, die frei definierbar sind, nämlich alle zwischen USEREVENT und NUMEVENTS-1.

Malprogramm

Um die Events einmal real zu benutzen, haben wir ein Malprogramm als Beispiel geschrieben. Mit den Tasten **0** bis **3** steuert man die Farbwahl, drückt man einen der Mausknöpfe im Fenster, so wird dort ein Klecks mit der aktuell gewählten Farbe gezeichnet. Die Taste **Escape** bricht das Programm ab.

```
#!/usr/bin/python
import pygame
import sys

class Malprogramm:
    def __init__(self, width, height):
        self._width = width
        self._height = height
        pygame.init()
        self._screen = pygame.display.set_mode((self._width,
self._height))
        self._screen.fill((200, 200, 200))
        pygame.display.update()
        self._drawColor = (200, 0, 0)

    def malen(self, (x, y)):
        pygame.draw.circle(self._screen, self._drawColor, (x, y), 10)
        pygame.display.update((x-10, y-10, 20, 20))

    def event_loop(self):
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_ESCAPE:
                        sys.exit()
                    elif event.key == pygame.K_0:
                        self._drawColor = (255, 255, 255)
```

```
                    elif event.key == pygame.K_1:
                        self._drawColor = (0, 0, 255)
                    elif event.key == pygame.K_2:
                        self._drawColor = (0, 255, 0)
                    elif event.key == pygame.K_3:
                        self._drawColor = (255, 0, 0)

                elif event.type == pygame.MOUSEBUTTONDOWN:
                    self.malen(event.pos)

if __name__ == '__main__':
    m = Malprogramm(640, 480)
    m.event_loop()
```

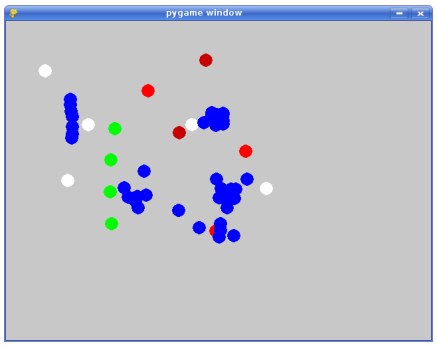


Bild der Anwendung

Die __init__()-Methode der Klasse verhält sich, wie die Funktion init() aus dem ersten Beispiel, bis auf dass sie eine Startfarbe belegt: self._drawColor = (200, 0, 0) (rot). Diese Farbe wird in der Methode *event_loop()* bei entsprechenden Tastendrücken verändert.

In der Methode malen() wird eine bestimmte angegeben Stelle mit einem ausgefüllten Kreis bemalt. pygame.draw.circle() benötigt dazu die Surface, die Farbe als Tupel, den Mittelpunkt als Tupel und den Radius. pygame.display.update() frischt den Bildschirm wieder auf, damit wir diesen Kreis sehen können. In diesem Fall benutzen wir eine Variante der Methode, da wir nicht den gesamten Bildschirm bemalt haben, sondern nur ein Rechteck. Dieses den Kreisfleck umgebene Rechteck übergeben wir der Methode pygame.display.update(), die dadurch wesentlich schneller arbeiten kann, als müsste sie den gesamten Bildschirm auffrischen.

Bei gedrückter Taste ist in event.key die Konstante der gedrückten Taste gespeichert. Einen unvollständigen Überblick über die Informationen, die Sie aus der event-Variablen herauslösen können in Abhängigkeit vom gewählten Ereignistyp gibt die folgende Tabelle:

Ereignistyp	Event-Felder	Bedeutung
KEYDOWN	unicode	Das Unicode-Zeichen dieses Tastendruckes
	key	K_0..K_9 - Zifferntaste K_a..K_z Buchstabentaste und viele mehr
	mod	KMOD_LSHIFT - linke Schift-Taste, KMOD_LCTRL linke STRG-Taste KMOD_RALT recht ALT-Taste und viele mehr
KEYUP	key, mod	wie oben
MOUSEMOTION	pos	Absolute Position innerhalb des Fensters als Tupel
	rel	Veränderung zur letzten Position
	buttons	gedrückte Mausknöpfe
MOUSEBUTTONDOWN, MOUSEBUTTONUP	pos, button	wie oben

Animation

Zu folgendem Beispiel passt der Ausschnitt aus Heinrich Heines Gedicht: *Ein Jüngling liebt ein Mädchen*:

```
Ein Jüngling liebt ein Mädchen,
die hat einen andern erwählt;
der andre liebt eine andre,
und hat sich mit dieser vermählt.

Das Mädchen heiratet aus Ärger
den ersten besten Mann,
der ihr in den Weg gelaufen;
der Jüngling ist übel dran.
```

Wir versuchen einmal eine ähnliche Situation zu programmieren, wobei der Jüngling dem Mädchen hinterherläuft, diese wiederum ihrem Erwählten, der seinerseits einer anderen hinterherläuft. Diese wiederum versucht den Jüngling zu erhaschen. Technisch gesehen lassen wir schlicht einige Punkte sich aufeinander zu bewegen.

Das folgende Programm generiert ein Ereignis (USEREVENT) auf der Basis eines Timers. Der Timer löst das Ereignis nach 200 Millisekunden aus, in der Event-Loop wird entschieden, ob nach weiteren 0,2 Sekunden erneut das gleiche Ereignis erfolgen soll. Nach jedem Zeitintervall werden Linien zwschen Punkten, die sich bewegen, neu gezeichnet:

```
#!/usr/bin/python
import pygame
import math
import sys

class Animation:

    def __init__(self, width, height):
        self._width = width
        self._height = height
        pygame.init()
        self._screen = pygame.display.set_mode((self._width,
self._height))
        self._screen.fill((200, 200, 200))
        pygame.display.update()
        self._punkte = [(0.0, 0.0), (width - 1.0, 0.0), (width - 1.0,
height - 1.0), (0.0, height - 1.0)]
        pygame.time.set_timer(pygame.USEREVENT, 200)

    def malen(self):
        pygame.draw.line(self._screen, (0,0,0), self._punkte[0],
self._punkte[1], 1)
        pygame.draw.line(self._screen, (0,0,0), self._punkte[1],
self._punkte[2], 1)
        pygame.draw.line(self._screen, (0,0,0), self._punkte[2],
self._punkte[3], 1)
```

```
        pygame.draw.line(self._screen, (0,0,0), self._punkte[3],
self._punkte[0], 1)
        pygame.display.update()

    def berechnen(self):
        punkte_tmp = []
        anz_punkte = len(self._punkte)
        dx = 0.0 # vorbelegt, da wir den Wert zurueckgeben
        for i in xrange(4):
            x1, y1 = self._punkte[i]
            x2, y2 = self._punkte[(i+1) % anz_punkte]
            dx = x2 - x1
            dy = y2 - y1
            # Einheitsvektor
            laenge = math.sqrt(dx * dx + dy * dy)
            ex = dx / laenge
            ey = dy / laenge
            x_neu = x1 + 5.0 * ex
            y_neu = y1 + 5.0 * ey
            punkte_tmp.append((x_neu, y_neu))
        self._punkte = punkte_tmp
        return dx

    def event_loop(self):
        while True:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    sys.exit()
                elif event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_ESCAPE:
                        sys.exit()
                elif event.type == pygame.USEREVENT:
                    self.malen()
                    dist = self.berechnen()
                    if dist > 20.0:
                        pygame.time.set_timer(pygame.USEREVENT, 200)

if __name__ == '__main__':
    a = Animation(800, 800)
    a.event_loop()
```

self._punkte ist eine Liste, die Punkte als Tupel enthält. Diese Punkte werden in der Methode berechnen() neu berechnet und es werden Linien zwischen den Punkten neu gezeichnet. Die Punkte sollen sich dabei aufeinander zubewegen. Der Timer wird in __init__() mit dem Aufruf pygame.time.set_timer() gestartet. Die Argumente sind der Ereignistyp und die Zeitspanne in Millisekunden. Die Methode malen() sorgt dafür, daß zwischen allen vier Punkten Linien in schwarz gemalt werden. Tritt ein USEREVENT ein, so werden die Linien zwischen den Punkten gemalt, es werden neue Punkte berechnet und entschieden, ob weitere Punkte berechnet werden müssen. Gegebenenfalls wird der Timer erneut gestartet.

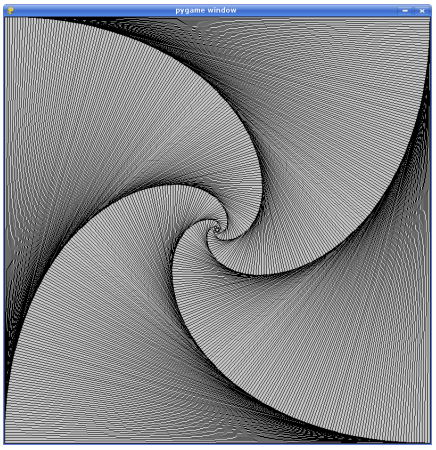


Bild der Animation

🔍 Details

Die Berechnung erfolgt nach folgendem Schema:
Zuerst werden alle Punkte in den Ecken verteilt. Anschließend bewegt sich jeder Punkt ein Stück auf den anderen zu. Der Punkt *self._punkte[0]* bewegt sich in die Richtung des Punktes *self._punkte[1]* und so fort, der letzte bewegt sich wieder auf den ersten Punkt zu. Mathematisch bedeutet das, $\vec{P}_{(t+200ms)} = \vec{P}_t + v * \vec{e}$ zu berechnen, wobei $\vec{e}$ der normierte Richtungsvektor zwischen zwei benachbarten Punkten ist. und $t + 200ms$ den nächsten Zeitschritt bedeutet.

Bilder und Fonts

Das folgende Beispiel demonstriert, wie man Bilder in PyGame lädt und mit Hilfe von Fonts einen Text in ihnen aufbringt. Damit das Programm korrekt funktioniert, muss ein Bild mit dem Namen beispiel.png im aktuellen Verzeichnis liegen.

```
#!/usr/bin/python
import pygame
import sys

WINWIDTH = 640
WINHEIGHT = 480

def init(width, height):
    pygame.init()
    screen = pygame.display.set_mode((width, height))
    screen.fill((250, 250, 250))
    pygame.display.update()
    return screen

def load_pic(filename, width, height):
    surface = pygame.image.load(filename)
    picW, picH = surface.get_size()
    transform = False
    if picW > width:
        picH = 1.0 * width / picW * picH
        picW = width
        transform = True
```

```
    if picH > height:
        picW = 1.0 * height / picH * picW
        picH = height
        transform = True
    if transform:
        w = int(round(picW))
        h = int(round(picH))
        tmp = pygame.transform.scale(surface, (w, h))
        surface = tmp
    return surface

def blit_pic(surface, pic, width, height):
    picW, picH = pic.get_size()
    w = (width - picW) / 2
    h = (height - picH) / 2
    surface.blit(pic, (w, h))
    font = pygame.font.SysFont('curier', 50)
    text = font.render("Hallo, Welt", True, (0, 0, 200))
    picW, picH = text.get_size()
    w = (width - picW) / 2
    h = (height - picH) / 2
    surface.blit(text, (w, h))
    pygame.display.update()

def event_loop():
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()
            elif event.type == pygame.KEYDOWN:
                sys.exit()

if __name__ == '__main__':
    screen = init(WINWIDTH, WINHEIGHT)
    pic = load_pic('beispiel.png', WINWIDTH, WINHEIGHT)
    blit_pic(screen, pic, WINWIDTH, WINHEIGHT)
    event_loop()
```

Bilder anzeigen

Neu an diesem Programm sind die beiden Funktionen load_pic() und blit_pic(). In load_pic() wird ein Bild geladen. Dieses Bild wird repräsentiert durch eine Surface, die sich manipulieren lässt, also beispielsweise drehen, bemalen und skalieren. Die Methode surface.get_size() liefert uns die Größe des Bildes. Das Bild wird so skaliert, dass es auf das Fenster passt.

blit_pic() dient dazu, das Bild auf das Grafikfenster zu zeichnen. Damit es mittig erscheint, wird hier mit Hilfe der Surface-Größe der Rand ausgerechnet.surface.blit(pic, (w, h)) übernimmt das eigentliche Blitten, eine Operation, die eine Surface auf einen andere kopiert. Der Parameter (w, h) ist hierbei der Ursprung des Bildes.

Ebenfalls mittig soll eine Schrift auf das Bild gebracht werden. Hierzu laden wir mit pygame.font.SysFont() eine Schriftart "Curier" in 50 Punkte Größe. Mit font.render() wird dann eine neue Surface erzeugt, die einen konkreten Text unter Anwendung von Anti-Alias mit der gewählten Farbe repräsentiert. Auch diese wird auf den Bildschirm gezeichnet. Nach beiden Blit-Operationen wird das Grafikfenster aufgefrischt.

Musik

Das folgende Programm spielt *WAV* und *OGG/Vorbis*-Dateien, die auf der Kommandozeile übergeben werden ab:

```
#!/usr/bin/python
import pygame
import sys
import os.path

def hinweis():
    print "./sound soundfile"
    sys.exit()

if len(sys.argv) != 2:
    hinweis()

if not os.path.exists(sys.argv[1]):
    hinweis()

pygame.init()
pygame.mixer.music.set_endevent(pygame.USEREVENT)
pygame.mixer.music.load(sys.argv[1])
pygame.mixer.music.play()

while True:
    for event in pygame.event.get():
        if event.type == pygame.USEREVENT:
            sys.exit()
```

Die Funktion pygame.init() initialisiert auch den Mixer. pygame.mixer.music.set_endevent() legt ein Ereignis fest, welches eintrifft, sobald eine Sound-Datei zu ende gespielt ist. Die auf der Kommandozeile übergebene Sounddatei wird mit pygame.mixer.music.load() geladen, anschließend mit pygame.mixer.music.play() abgespielt.

Mit Hilfe einer zusätzlichen Funktion pygame.mixer.set_volume(lautstärke) könnten wir noch die Lautstärke einstellen. Dieser Wert liegt zwischen `0.0` und `1.0`. Da das Abspielen einer Sounddatei das Programm überdauern kann, müssen wir hier eine Event-Loop einsetzen. Das Programm würde sonst beendet werden, bevor die Datei abgespielt würde. Am Ende bekommen wir das angeforderte Ereignis, das Programm kann sich dann zum richtigen Zeitpunkt beenden.

Zusammenfassung

Dieses Kapitel hat einen Überblick über Möglichkeiten von PyGame geboten. Wir können genau ein Grafikfenster öffnen, zeichnen, auf Ereignisse reagieren und selber Ereignisse erzeugen. Darüber hinaus haben wir gezeigt, wie man Bilder lädt und Texte zeichnet. PyGame kann mehr, als wir hier ausgeführt haben... Experimentieren Sie selbst!

Grafische Benutzeroberflächen mit Qt4

In diesem Kapitel geht es um grafische Benutzeroberflächen. Wir geben einen Einblick in die Programmierung mit Qt Version 4, zu der Wikibooks das Buch Qt für C++-Anfänger als zusätzliches Angebot hat.

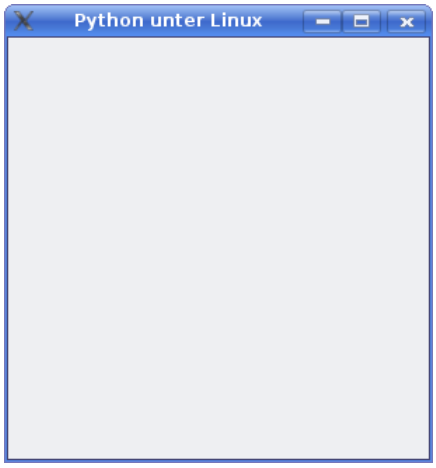
Fenster, öffne Dich!

Die erste Anwendung öffnet ein Fenster und hat einen Fenstertitel:

```
#!/usr/bin/python

import sys
from PyQt4 import QtGui

class Fenster(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setWindowTitle("Python unter Linux")
        self.resize(300, 300)
        self.show()

app = QtGui.QApplication(sys.argv)
f = Fenster()
app.exec_()
```

Wir binden PyQt4.QtGui ein, um auf Elemente der grafischen Benutzeroberfläche zugreifen zu können. Die Fensterklasse wird von QMainWindow abgeleitet, einer Klasse, die schon die wichtigsten Elemente eines Hauptfensters mitbringt. Es ist recht einfach, wie wir später sehen werden, auf Grundlage dieser Klasse Menüs und eine Statusleiste hinzuzufügen. setWindowTitle() legt den Namen für das Fenster fest, resize() eine Anfangsgröße, die wir dynamisch ändern können. Mit show() wird das Objekt angezeigt.

Im Hauptprogramm erzeugen wir ein QApplication()-Objekt und unser Fenster. exec_() startet die Anwendung und wartet, bis das Fenster geschlossen wird.



guiqt1.py

Signale empfangen

Grafische Nutzeroberflächen bestehen aus Elementen, in Qt **Widgets** genannt, die sich um die Interaktion mit Benutzern kümmern. Widgets sind zum Beispiel Textfelder (QLabel), Editoren (QTextEdit) oder Knöpfe (QPushButton). Viele dieser Widgets nehmen Benutzereingaben entgegen, so kann ein Knopf gedrückt werden, ein Menüpunkt aktiviert werden. Qts Widgets senden bei Interaktionen Signale aus, die man verarbeiten kann. Die folgende Anwendung zeigt, wie man Menüs erzeugt und reagiert, wenn sie aktiviert werden:

```
#!/usr/bin/python
import sys
from PyQt4 import QtCore, QtGui

class Fenster(QtGui.QMainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setWindowTitle("Python unter Linux")
        self.makeActions()
        self.makeMenu()
        self.show()

    def makeActions(self):
        self._exitAction = QtGui.QAction("&Ende", None)
        self._helpAction = QtGui.QAction("Hilfe!", None)
        self.connect(self._exitAction, QtCore.SIGNAL('triggered()'),
self.slotClose)
        self.connect(self._helpAction, QtCore.SIGNAL('triggered()'),
self.slotHelp)

    def makeMenu(self):
        menuBar = self.menuBar()
        fileMenu = menuBar.addMenu("&Datei")
        fileMenu.addAction(self._exitAction)
        helpMenu = menuBar.addMenu("&Hilfe")
        helpMenu.addAction(self._helpAction)
```

```
    def slotClose(self):
        self.close()

    def slotHelp(self):
        QtGui.QMessageBox.information(None, "Dies ist die Hilfe", "Hilf
 dir selbst, sonst hilft dir keiner!")

app = QtGui.QApplication(sys.argv)
f = Fenster()
app.exec_()
```

guiqt2.py

Innerhalb von makeActions() definieren wir zwei QAction()-Objekte. Diese Objekte haben einen Namen und optional ein Bild. Wann immer QAction()-Objekte aktiviert werden, sie also das Signal QtCore.SIGNAL('triggered()') senden, soll eine Funktion aufgerufen werden, die wir in der Klasse definiert haben, also beispielsweise slotClose(). Diese Verknüpfung zwischen den beiden Objekten, QAction() auf der einen Seite und dem Fenster()-Objekt auf der anderen Seite übernimmt die Funktion connect().

makeMenu() erzeugt die Menüs. QtGui.QMainWindow enthält selbst schon eine Menüzeile, die man sich mit menuBar() beschafft. An diese Menüzeile fügt man mit addMenu() einen Menüeintrag hinzu. Diese Menüs findet man zuoberst in der Menüzeile einer jeden Anwendung. Jedes Menü kann beliebig viele QAction()-Einträge haben. Da diese einen eigenen Titel haben, werden sie sogleich angezeigt. Die weiteren Methoden der Klasse beinhalten die Ereignisse: close() schließt das Fenster und QMessageBox.information() zeigt einen informativen Text an.

Mehr Widgets

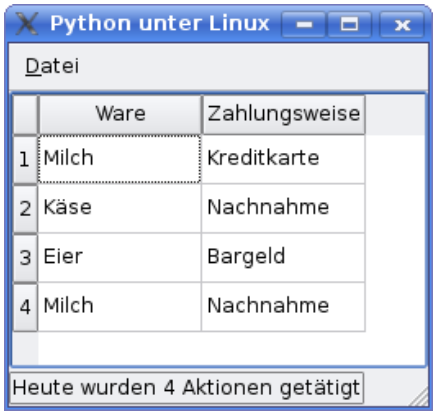
Hauptfenster haben üblicherweise zentrale Widgets. Das sind zumeist selbst geschriebene Widget-Klassen, in denen die hauptsächliche Interaktion stattfindet, wie zum Beispiel die Anzeige einer Webseite im Browser. Das folgende Beispiel implementiert einen Einkaufsberater. Der eigentliche Einkauf findet in einem eigenen Dialog statt, die Zusammenfassung erfolgt in einer Tabelle im Hauptfenster. Zwischen Dialog und Hauptfenster wird eine Nachricht über den Kaufwunsch und die Zahlungsweise ausgetauscht.

```
#!/usr/bin/python
# -*- coding:utf-8 -*-
import sys
from PyQt4 import QtCore, QtGui

class Dialog(QtGui.QDialog):
    """Der Einkaufsdialog"""
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.setWindowTitle("Elektronischer Einkauf")
        vbox = QtGui.QVBoxLayout()
        self._liste = QtGui.QListWidget()
        self._liste.addItem(u"Milch")
        self._liste.addItem(u"Eier")
```

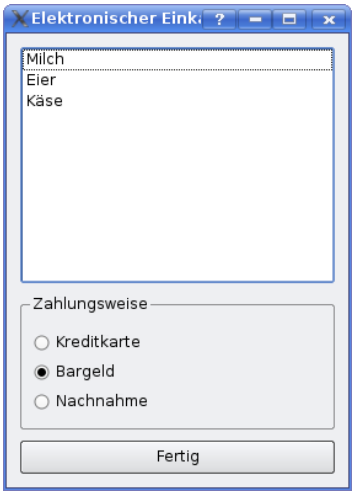
```
        self._liste.addItem(u"Käse")
        vbox.addWidget(self._liste)
        group = QtGui.QGroupBox("Zahlungsweise")
        vboxGroup = QtGui.QVBoxLayout()
        self._r1 = QtGui.QRadioButton("Kreditkarte")
        vboxGroup.addWidget(self._r1)
        self._r2 = QtGui.QRadioButton("Bargeld")
        vboxGroup.addWidget(self._r2)
        self._r3 = QtGui.QRadioButton("Nachnahme")
        vboxGroup.addWidget(self._r3)
        self._r2.setChecked(True)
        group.setLayout(vboxGroup)
        vbox.addWidget(group)
        fertigButton = QtGui.QPushButton("Fertig")
        self.connect(fertigButton, QtCore.SIGNAL('clicked()'),
self.fertig)
        vbox.addWidget(fertigButton)
        self.setLayout(vbox)
        self.show()

    def fertig(self):
        ware = u"%s" % self._liste.currentItem().text()
        label = ""
        for rb in [self._r1, self._r2, self._r3]:
            if rb.isChecked():
                label = "%s" % rb.text()
                break
        self.emit(QtCore.SIGNAL('signalEinkaufFertig'), (ware, label))
        self.accept()


class Fenster(QtGui.QMainWindow):
    """Diese Klasse stellt das Hauptfenster dar, das zentrale Widget
ist eine Tabelle"""
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.setWindowTitle("Python unter Linux")
        self.makeActions()
        self.makeMenu()
        # Statuszeile
        self._label = QtGui.QLabel(u"Einkaufszähler")
        self.statusBar().addWidget(self._label)
        # Tabelle
        self._tableWidget = QtGui.QTableWidget(0, 2)
        self._tableWidget.setHorizontalHeaderLabels(["Ware",
"Zahlungsweise"])
        self.setCentralWidget(self._tableWidget)
```

```
        self.show()

    def makeActions(self):
        self._dialogAction = QtGui.QAction("Einkauf", None)
        self._exitAction = QtGui.QAction("&Ende", None)
        self.connect(self._dialogAction, QtCore.SIGNAL('triggered()'),
self.slotEinkauf)
        self.connect(self._exitAction, QtCore.SIGNAL('triggered()'),
self.slotClose)

    def makeMenu(self):
        menuBar = self.menuBar()
        fileMenu = menuBar.addMenu("&Datei")
        fileMenu.addAction(self._dialogAction)
        fileMenu.addSeparator()
        fileMenu.addAction(self._exitAction)

    def slotEinkauf(self):
        """Ruft den Einkaufsdialog auf"""
        d = Dialog()
        self.connect(d, QtCore.SIGNAL('signalEinkaufFertig'),
self.slotEinkaufFertig)
        d.exec_()

    def slotClose(self):
        """Wird aufgerufen, wenn das Fenster geschlossen wird"""
        ret = QtGui.QMessageBox.question(None, "Ende?", "Wollen Sie
wirklich schon gehen?", QtGui.QMessageBox.Yes, QtGui.QMessageBox.No)
        if ret == QtGui.QMessageBox.Yes:
            self.close()

    def slotEinkaufFertig(self, ware):
        """Dieser Slot fügt eine Tabellenzeile an und stellt in dieser
die gekaufen Waren vor"""
        numRows = self._tableWidget.rowCount()
        # Neue Zeile
        self._tableWidget.insertRow(numRows)
        # Inhalte einfügen
        t1 = QtGui.QTableWidgetItem(ware[0])
        t2 = QtGui.QTableWidgetItem(ware[1])
        self._tableWidget.setItem(numRows, 0, t1)
        self._tableWidget.setItem(numRows, 1, t2)
        # Update Statuszeile
        text = u"Heute wurden %d Aktionen getätigt" % (numRows + 1)
        self._label.setText(text)
```

```
if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    f = Fenster()
    app.exec_()
```



guiqt3.py Anwendung



guiqt3.py Dialogfenster

Der Einkaufsberater besteht aus zwei Elementen, nämlich dem Dialog, einer von QDialog abgeleiteten Klasse, und dem Hauptfenster. Der Dialog wird angezeigt, wenn die Methode slotEinkauf() vom Hauptfenster aufgerufen wurde. Der Dialog besteht aus einigen ineinander geschachtelten Elementen: Innerhalb des vertikalen Layouts (QVBoxLayout) werden ein List-Widget (QListWidget), eine Group-Box (QGroupBox) und ein Knopf (QPushButton) eingefügt. Der Listbox werden mit addItem() einige Beispielwaren hinzugefügt. Die Group-Box enthält ein eigenes vertikales Layout, mit dem sie drei Auswahlschalter (QRadioButton) zur Angabe der Zahlungsweise verwaltet, von denen einer vorselektiert (self._r2.setChecked(True)) ist.

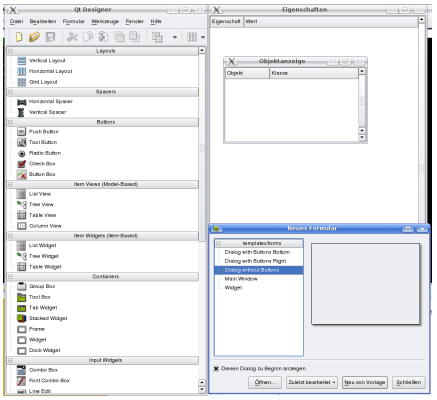
Wird der Knopf ("Fertig") gedrückt, soll die Methode Dialog.fertig() aufgerufen werden. In dieser Methode werden die Auswahlknöpfe abgefragt, genau einer von ihnen ist immer aktiviert. Ebenfalls wird der im List-Widget aktivierte Text abgefragt. Sobald beide Informationen vorliegen, wird ein Signal abgeschickt. Dieses besorgt die Methode emit(), die das Signal wie auch ein Tupel mit Informationen erwartet. Anschließend wird der Dialog geschlossen.

Das Hauptfenster enthält Menüs, eine Statuszeile, in der sich eine Textfeld (QLabel) befindet wie auch als zentrales Element eine Tabelle. Wird per Menü die Methode slotEinkauf() aufgerufen, erstellt sie den beschriebenen Dialog, verknüft das von diesem ausgehende Signal mit dem Slot slotEinkaufFertig() und ruft dessen exec_()-Methode auf.

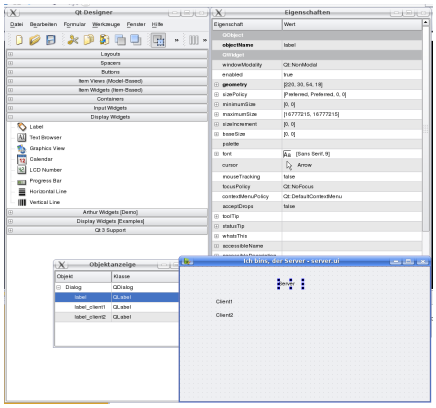
slotEinkaufFertig() enthält im Parameter ware das Tupel mit den aktuellen Einkaufsinformationen. Diese Informationen sollen in die Tabelle geschrieben werden, wobei diese zuerst um eine Zeile wachsen muss. Dann werden pro Spalte ein QTableWidgetItem erstellt, diese werden per setItem(Reihe, Spalte, Item) in die Tabelle eingefügt. Zu guter Letzt wird das Label der Statuszeile auf den neusten Stand gebracht.

Design-Server

Der Einkaufsführer brachte es an den Tag: Das händische schreiben von Widgets und das passende Plazieren derselbigen auf einem Fenster macht viel Mühe. Viel leichter geht es, wenn man sich die Fenster und deren Inhalte zusammenklickt. Das Programm hierzu heißt *Designer* (designer-qt4). Mit seiner Hilfe plaziert man auf Fenstern die Widgets, die man benötigt und layoutet sie auf grafische Weise.



Designer



Unser Dialogfenster

Das folgende Programm benötigt nur drei Labels in einem Dialog-Fenster, von denen zwei label_client1 und label_client2 heißen. Auf diese beiden Labels wollen wir im Quelltext verweisen. Schauen Sie sich hierzu bitte das Bild *Unser Dialogfenster* an. Speichern Sie die Designer-Datei unter dem Namen server.ui ab. Sie enthält in XML-Notation alle Informationen über das Dialog-Fenster.

Um den Dialog für Python benutzbar zu machen, ist noch ein Zwischenschritt nötig:

Ausgabe

```
user@localhost:~$ pyuic4 -o server_ui.py server.ui
(keine Ausgabe)
```

Sie erhalten eine Datei mit dem Namen server_ui.py. Diese können Sie direkt in den Python-Code importieren.

Das Programm, welches wir nun besprechen wollen, ist ein Server, der Verbindungsinformationen bereit hält. Man kann sich an ihm anmelden, wobei er zwei Clients akzeptiert. Diesen sendet er einen Willkommensgruß und schreibt auf den Dialog einen Verbindungshinweis. Wird ein Client beendet, so registriert der Server das ist bereit, eine weitere Verbindung zu akzeptieren. Der Server tut also nichts als zu merken, ob gerade ein Client verbunden ist oder nicht.

```
#!/usr/bin/python

import sys
from server_ui import Ui_Dialog
from PyQt4 import QtCore, QtGui, QtNetwork

class Dialog(QtGui.QDialog, Ui_Dialog):
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.setupUi(self)

    def connect(self, num):
```

```
        if num == 1:
            self.label_client1.setText ("Client1 connected")
        else:
            self.label_client2.setText ("Client2 connected")

    def disconnect(self, num):
        if num == 1:
            self.label_client1.setText ("Client1 disconnected")
        else:
            self.label_client2.setText ("Client2 disconnected")

class Server(QtNetwork.QTcpServer):
    def __init__(self, numClients):
        QtNetwork.QTcpServer.__init__(self)
        self._numClientsWanted = numClients
        self._numClients = 0
        self.address =
QtNetwork.QHostAddress(QtNetwork.QHostAddress.LocalHost)
        # socketInfo: [(socket, connected, num), ...]
        self.socketInfo = []
        for i in xrange(numClients):
            self.socketInfo.append((None, False, i+1))
        ret = self.listen(self.address, 6000)
        self.connect(self, QtCore.SIGNAL("newConnection()"),
self.connection)

    def connection(self):
        if self.hasPendingConnections():
            print "connection available",
            socket = self.nextPendingConnection()
            if self._numClients < self._numClientsWanted:
                self.connect(socket, QtCore.SIGNAL("disconnected()"),
self.dis)
                for i, (p, c, n) in enumerate(self.socketInfo):
                    if c == False:
                        self.socketInfo[i] = (socket, True, n)
                        print "accepted"
                        msg = "You are accepted. Client %d" % n
                        socket.write(msg)
                        self.emit(QtCore.SIGNAL("connected"), n)
                        break
                self._numClients += 1
            else:
                print "rejected"
                socket.close()

    def dis(self):
```

```
        print "socket disconnected"
        self._numClients = 0
        for i, (p, c, n) in enumerate(self.socketInfo):
            if c:
                if p.state() ==
QtNetwork.QAbstractSocket.UnconnectedState:
                    p.close()
                    self.socketInfo[i] = (None, False, n)
                    self.emit(QtCore.SIGNAL("disconnected"), n)
                else:
                    self._numClients += 1

app = QtGui.QApplication(sys.argv)
window = Dialog()
server = Server(2)
app.connect(server, QtCore.SIGNAL("connected"), window.connect)
app.connect(server, QtCore.SIGNAL("disconnected"), window.disconnect)
window.show()
sys.exit(app.exec_())
```

Die Designer-Datei wird mit from server_ui import Ui_Dialog eingebunden. Der Name ist in der Python-Datei server_ui.py bekannt. Ein Dialog auf der Basis wird erstellt, in dem er vom QDialog und dem Ui_Dialog abgeleitet wird: class Dialog(QtGui.QDialog, Ui_Dialog). Die Methode setupUi(self) initialisiert diesen Dialog anschließend. Der Rest der Klasse dient dazu, die Verbindungsinformationen anzuzeigen.

Der Server wird von der Basisklasse QTcpServer abgeleitet. Der Server soll maximal 2 Verbingungen gleichzeitig haben, und auf dem eigenen Host (Local host, QHostAddress(QtNetwork.QHostAddress.LocalHost)) laufen, seine Portadresse lautet 6000. socketInfo ist eine Liste, die die Clients speichert, wobei jeder Client durch einen TCP-Socket, einen Hinweis darauf, ob die Verbindung aktiv ist und eine eindeutige Nummer gekennzeichnet ist. Mit listen(self.address, 6000) wird auf Verbindungswünsche der Clients gewartet. Treffen diese ein, wird die Methode connection() aufgerufen. Falls wir den Verbindungswunsch akzeptieren, wird mit nextPendingConnection() der Socket des Clients geholt und an passender Stelle in die Struktur socketInfo eingefügt. Dem Client wird eine Nachricht geschickt und dem Dialog wird angezeigt, dass sich ein Client verbunden hat.

Falls sich ein Client verabschiedet, beispielsweise, weil die DSL-Leitung unterbrochen wurde oder der Nutzer das Client-Programm auf andere Weise beendete, wird die Methode dis() aufgerufen. Sie findet heraus, welcher Client sich abgemeldet hat und gibt entsprechend Nachricht darüber aus.

Um das Programm zu testen, benutzen Sie einfach das Programm *Telnet*. Starten Sie hierzu den Server und rufen auf einer anderen Konsole folgendes auf:

Ausgabe

```
user@localhost:~$ telnet localhost 6000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
You are accepted. Client 1
```

Sobald Sie sich verbunden haben, wird dieses auch im Dialogfenster angezeigt.

Zusammenfassung

In diesem Kapitel haben Sie die Grundzüge der Programmierung mit dem Toolkit Qt Version 4 kennen gelernt. Wir haben einige Widgets besprochen, das Signal-Slot-Konzept kennen gelernt und Server-Programmierung als Thema für Fortgeschrittene Benutzer besprochen. Das Kapitel wurde mit einer Einführung in den Qt-Designer abgerundet.

Grafische Benutzeroberflächen mit wxPython

Sie haben nun einen weiteren Abschnitt erreicht, der sich mit grafischen Benutzeroberflächen in Python beschäftigt. Diesmal mit der plattformunabhängigen, auf wxWidgets aufbauenden Variante wxPython (http://de.wikipedia.org/wiki/ WxPython). Genauere Details zu den besonderen Aspekten wxPythons finden sich in den jeweiligen Abschnitten.

Vorbereitung und Installation

Da wxPython nicht zum Standardrepertoire einer Pythoninstallation gehört, muss es erst installiert werden - und zwar überall da, wo die Anwendung später ausgeführt werden soll. Es gibt Installationspakete für alle möglichen Betriebssysteme. In den meisten Linux-Distributionen findet sich wxPython im Paketmanager. Falls das bei Ihnen nicht der Fall ist, können Sie Pythons setuptools (http:/ / pypi. python. org/ pypi/ setuptools) oder ein Installationspaket (http:/ / www. wxpython. org/ download. php#binaries|entsprechendes) verwenden. Die erste Variante sollte die einfachste sein, funktionierte bei mir im Test aber (ohne größere Schritte zur Aufklärung des Problemes) leider nicht:

Ausgabe

```
user@localhost:~$ sudo easy_install wxpython
Searching for wxpython
Reading http://pypi.python.org/simple/wxpython/
Reading http://wxPython.org/
Reading http://wxPython.org/download.php
Best match: wxPython src-2.8.9.1
Downloading http://downloads.sourceforge.net/wxpython/wxPython-src-2.8.9.1.tar.bz2
Processing wxPython-src-2.8.9.1.tar.bz2
error: Couldn't find a setup script in /tmp/easy_install-PkAhhW/wxPython-src-2.8.9.1.tar.bz2
```

Lassen Sie sich davon aber nicht beirren. Vermutlich brauchen Sie nur in Ihrem Paketmanager nach *wxPython* zu suchen und alles andere geht von selbst.

Plopp!

...oder welches Geräusch auch immer Sie mit dem Öffnen eines Fensters assoziieren: Mit wxPython erreichen Sie diesen Punkt jedenfalls einfach und schnell:

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

import wx

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = wx.Frame(None, title="Hallo Welt!")
    frame.Show()
```

```
    app.MainLoop()
```

Kurz zur Erläuterung: In **Zeile 4** importieren wir das Paket wx, welches alle Komponenten von wxPython beinhaltet. In älteren Beispielen könnten sich auch noch die Varianten "`import wxPython as wx`" oder "`from wxPython import *`" finden. **Verwenden Sie diese nicht!** Sie sind veraltet. Sternchen-Importe - wie in der zweiten Variante - sind außerdem generell nicht gern gesehen.

In **Zeile 7** wird das Applikations-Objekt erstellt. Ohne dieses Objekt geht erstmal garnichts. Bei der Erstellung wird das grafische System, der Message- bzw. Eventhandler und alles weitere wichtige initialisiert. Die Nutzung von wx.PySimpleApp() macht das ganze am unkompliziertesten.

Zeile 8 erzeugt ein Frame-Objekt. Dieser stellt in unserem Beispiel das Hauptfenster dar. Eine Applikation kann aus mehreren Frames bestehen, aber nur einen Haupt-Frame haben. Allen weiteren muss dann als ersten Parameter den ihnen übergeordneten Frame übergeben werden. An dieser Stelle wird hier None übergeben, da ein Hauptframe keinen *Parent* hat. Bisher gibt es fuer diesen Parameter auch keinen Standard-Wert, so das mindestens None angegeben werden muss.

Die folgende **Zeile 9** sorgt dafür, dass der Frame auch angezeigt wird. Der Frame kann zwar auch dynamisch im laufenden Betrieb erzeugt und variiert werden, aber die initialen Elemente sollten erzeugt werden, bevor .Show() ausgeführt wird.

Abschließend starten wir die *MainLoop*, also die Haupt-Eventschleife von wxWidgets. Diese Schleife läuft so lange, bis das Hauptfenster (genauer alle Threads) in irgendeiner Weise geschlossen wird.

Bravo! Nun haben Sie das erste Fenster erstellt und auch schon einiges grundlegendes ueber wxPython gelernt. Das war auch gar nicht so schwer, oder?

Einen Schritt weiter

Komplexere Anwendungen erben üblicherweise von Hauptelementen wie wx.Frame ab. Beispielsweise so:

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

import wx


class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Hallo Welt!")
        self.panel = wx.Panel(self)
        self.label = wx.StaticText(
            self.panel,
            label = "Python unter Linux",
            pos = (5, 5),
        )


if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

Je nach Anforderung wird beispielsweise auch von wx.Panel abgeerbt. Im Laufe dieses Tutorials werden wir ein entsprechendes Beispiel behandeln.

Windows

In *wx* stammen alle sichtbaren Elemente von wx.Window ab. Somit wird jedes entsprechende Element von einfachem Text bis hin zur Menüleiste auch *window* genannt. Dabei unterscheiden wir zum einen zwischen **Containerobjekten**, die weitere Windows enthalten können, zum Beispiel wx.Frame, wx.Panel und zum anderen **Controls**: wx.StaticText, wx.TextCtrl, wx.Button. Diese Unterscheidung ist zwar etwas oberflächlich, aber im Moment völlig ausreichend für uns.

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

import wx


class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Hallo Welt!")
        self.panel = wx.Panel(self)
        self.label = wx.StaticText(
            self.panel,
            label = "Python unter Linux",
            pos = (5, 5),
        )


if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

Die Markierung zeigt den derzeitigen Hauptteil des Programmes. Falls Sie **Zeile 9** nicht ganz verstehen, schauen Sie noch einmal zurück in das Kapitel OOP (http:/ / de. wikibooks. org/ wiki/ Python_unter_Linux:_Rund_um_OOP#Vererbung).

Das Hauptpanel des Frames wird in **Zeile 10** angelegt und dann mit einem einfachen Text versehen. Die Parameter für wx.StaticText bekommen der Übersicht halber jeweils eine eigene Zeile und sollten selbsterklärend sein. Nähere Informationen finden Sie im entsprechenden Eintrag der API (http:/ / www. wxpython. org/ docs/ api/ wx. StaticText-class.html).

Hier sieht man gut den grundlegenden Aufbau eines Fensters. Das Panel dient als Eltern-Element fuer alle weiteren Controls. Es erscheint manchmal nicht notwendig, sollte aber immer genutzt werden. Manche Systeme können Controls nicht korrekt oder gar nicht darstellen, wenn sie direkt auf dem Frame angebracht werden.

Events

Alle modernen grafischen Anwendungen arbeiten Eventbasiert. Ihre Elemente senden Signale an den sogenannten Eventhandler und empfangen sie von ihm. Der Löwenanteil der Funktionalität wird so realisiert. Hier werden wir nun darauf eingehen, wie man Events in wxPython benutzt.

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

import wx


class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, title="Hallo Welt!")
        self.panel = wx.Panel(self)
        self.label = wx.StaticText(
            self.panel,
            label = "Python unter Linux",
            pos = (5, 5),
            size = (self.Size.width - 10, -1),
        )

        self.input = wx.TextCtrl(
            self.panel,
            pos = (5, 30),
            size = (100, -1),
        )

        self.btn_text = wx.Button(
            self.panel,
            label = "Text ändern",
            pos = (110, 30),
            size = (95, -1),
        )
        self.btn_text.Bind(wx.EVT_BUTTON, self.on_change_text)

        self.btn_title = wx.Button(
            self.panel,
            label = "Titel ändern",
            pos = (210, 30),
            size = (95, -1),
        )
        self.btn_title.Bind(wx.EVT_BUTTON, self.on_change_title)


    def on_change_text(self, evt):
        self.label.Label = self.input.Value
```

```
    def on_change_title(self, evt):
        self.Title = self.input.Value


if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

Das Beispiel zeigt eine einfache Anwendung von Buttons. Mit Hilfe der Bind-Methode werden der Event-Binder wx.EVT_BUTTON und eine Methode an einen Knopf gebunden. Diese Methode wird dann aufgerufen, sobald der Knopf gedrueckt wird.

Alle Event-Binder fangen mit *EVT_* an und sind komplett groß geschrieben. Im von mir genutzten wxPython 2.8.9.1 gibt es 233 solcher Objekte mit meist selbsterklärendem Namen. Einige davon werden wir hier im Laufe des Programmaufbaus noch kennenlernen.

🔍 **Details**

Wie funktioniert ein Event?

Objekte oder Methoden können jederzeit Events auslösen. Dabei erzeugen sie Instanzen
von entsprechenden Event-Klassen und schicken diese dann an den EventHandler.
Bei diesem sind auch die Bindungen der Events gespeichert, so das er die entsprechenden
Methoden mit der Instanz des ausgelösten Events aufrufen kann.

Grafische Benutzeroberflächen mit GTK+

In diesem Kapitel geht es um die Programmierung von grafischen Benutzeroberflächen mit GTK+. GTK+ wird in Anwendungen wie GIMP benutzt und bildet die Basis für die Desktop-Umgebung GNOME. Falls Sie mehr Informationen zur Programmierung in GTK+ in anderen Sprachen suchen, werden Sie im Wikibuch GTK mit Builder fündig. In diesem Kapitel geben wir ihnen einen Einblick in die Programmierung mit dem Modul *python-gtk2*, welches Sie sich mit ihrem Paketmanager leicht installieren können.

Das erste Fenster

Das folgende Programm öffnet ein Fenster und stellt Text in einem Label dar. Anschließend wartet das Programm darauf, beendet zu werden:

```
#!/usr/bin/python

import gtk

class HalloWelt(object):

    def __init__(self):
        """ Initialisiert das Fenster """
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_title("Mein erstes Fenster")
        self.window.set_default_size(300, 100)
```

```
        self.window.connect("delete_event", self.event_delete)
        self.window.connect("destroy", self.destroy)
        label = gtk.Label("Hallo, Welt!")
        self.window.add(label)
        label.show()
        self.window.show()

    def event_delete(self, widget, event, data=None):
        """ reagiert auf 'delete_event' """
        return False

    def destroy(self, data=None):
        """ reagiert auf 'destroy' """
        gtk.main_quit()

    def main(self):
        """ Nachrichtenschleife """
        gtk.main()


if __name__ == "__main__":
    hallo = HalloWelt()
    hallo.main()
```

Die Klasse HalloWelt enthält vier Methoden.

Innerhalb von __init__() wird mit gtk.Window() ein neues so genanntes *Top-Level*-Fenster erzeugt. Es gibt verschiedene Arten von Fenstern, auf die wir nicht eingehen. Dieses Top-Level-Fenster ist genau das, was man sich unter einem Programm-Fenster vorstellt. set_title() legt einen Fenstertitel fest und set_default_size() eine Anfangsgröße. Die Größe des Fensters ist zur Laufzeit änderbar.

Die nächsten beiden Funktionen verknüpfen Ereignisse mit so genannten Callback-Methoden. Diese werden aufgerufen, wenn das Ereignis "`delete_event`" oder "`destroy`" auftritt. Das `delete_event` wird immer dann aufgerufen, wenn der Anwender versucht, das Fenster zu schließen. `destroy` erfolgt dann, wenn das Fenster echt geschlossen wird. Dieses Signal kann man im Gegensatz zum `delete_event` nicht abweisen.

Nun fehlt uns noch ein Label, das wir mit gtk.Label() erzeugen und mit window.add() dem Fenster hinzufügen. Das Label wie auch das Fenster werden sichtbar gemacht. Hierzu dient die Funktion show().

Die Methode event_delete() ist verknüpft mit dem `delete_event`. An dieser Stelle könnte das Programm fragen, ob der Anwender das Programm wirklich beenden möchte. Gibt die Methode False zurück, wird `destroy` als neues Signal gesendet. Gibt man True zurück, wird das Ereignis abgewiesen.

Die Methode destroy() hat nun die Aufgabe, das Fenster zu zerstören, in diesem Fall soll die gesamte Anwendung beendet werden.

Jede Anwendung, die grafische Nutzeroberflächen benutzt, verfügt über eine Hauptschleife, in der Nachrichten an ihre Empfänger weitergeleitet werden. Solche Nachrichten können Anwenderwünsche wie das Beenden der Anwendung sein, Mausbewegungen oder das Drücken von Knöpfen. Diese Hauptschleife wird mit gtk.main() bereitgestellt.

Layout

Benutzt man mehrere grafische Elemente, wie zum Beispiel mehrere Labels, kann man mit Hilfe von Layout-Funktionen bestimmen, wie diese Elemente angeordnet werden. So lässt sich ein Hauptfenster mit einem Tabellenlayout versehen oder in waagerechte und senkrechte Boxen unterteilen.

Tabellenlayout

Das einfachste Layout ist die Anordnung von Elementen in Form einer Tabelle. Folgendes Beispiel verdeutlicht das:

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

import gtk

class Tabelle1(object):

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_title("Mein erstes Fenster")
        self.window.set_default_size(300, 100)
        self.window.connect("delete_event", self.event_delete)
        self.window.connect("destroy", self.destroy)
        table = gtk.Table(3, 2, False)
        self.window.add(table)
        label = gtk.Label("Zelle 1-1")
        table.attach(label, 0, 1, 0, 1)
        label.show()
        label = gtk.Label("Zelle 1-2")
        table.attach(label, 1, 2, 0, 1)
        label.show()
        label = gtk.Label("Eine etwas größere Zelle")
        table.attach(label, 0, 1, 1, 2)
        label.show()
        label = gtk.Label("Zelle 2-2")
        table.attach(label, 1, 2, 1, 2)
        label.show()
        label = gtk.Label("<u>die letzte Zeile geht über die gesamte
Tabellenbreite</u>")
        label.set_use_markup(True)
        table.attach(label, 0, 2, 2, 3)
        label.show()
        table.show()
        self.window.show()


    def event_delete(self, widget, event, data=None):
        return False
```

```
    def destroy(self, data=None):
        gtk.main_quit()


    def main(self):
        gtk.main()


if __name__ == "__main__":
    tab = Tabelle1()
    tab.main()
```

Innerhalb von __init__() wird eine Tabelle erzeugt. In die Tabellenzellen werden jeweils einige Labels untergebracht. Eines der Labels benutzt HTML-Markup, um seinen Text darzustellen und geht über die gesamte Tabellenbreite.

Eine Tabelle wird mit gtk.Table(Zeilen, Spalten, homogen). Zeilen und Spalten sind selbst erklärend. Ist der Wert von *homogen* True, dann haben alle Tabellenelemente die gleiche Größe. Obwohl man die Tabelle selbst nicht sieht, muss man sie mit window.add(table) dem Fenster hinzufügen und mit show() sichtbar machen.

Ein Element wird der Tabelle mit der Methode attach(Element, links, rechts, oben, unten) hinzugefügt. Dabei beziehen sich die Koordinaten auf die Tabellenränder, nicht auf die Zeilen/Spalten-Nummerierung. Der linke Rand einer 3X2-Tabelle ist 0, der rechte Rand ist 2. Oben ist 0, unten ist 3. Es werden also die (gedachten) Gitterlinien durchnummeriert. Auf diese Weise kann man auch Elemente einfügen, die sich über mehrere Zeilen und Spalten erstrecken, wie im letzten Label-Beispiel.

Der Text in einem Label kann mit HTML-Markup versehen werden, in diesem Fall wird er unterstrichen dargestellt. Um Markup zu aktivieren, benutzt man die Methode set_use_markup().

Boxenlayout

Es gibt horizontale und vertikale Boxen. Horizontale Boxen legen Widgets nebeneinander, vertikale übereinander. Im folgenden Beispiel benutzen wir statt Labels gtk.Button()-Objekte (Schaltknöpfe), diese haben den Vorteil, dass man ihre Ausdehnung sieht. Das Hauptlayout besteht aus einer horizontalen Box, in die eine vertikale Box und ein Schaltknopf gelegt werden:

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

import gtk

class Box1(object):

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_title("Mein erstes Fenster")
        self.window.set_default_size(300, 100)
        self.window.connect("delete_event", self.event_delete)
        self.window.connect("destroy", self.destroy)
        # horizontale Box
```

```
        hbox = gtk.HBox(True, 30)
        # Vertikale Box
        vbox = gtk.VBox(True, 1)
        button = gtk.Button("vbox 0")
        button.show()
        vbox.pack_start(button)
        button = gtk.Button("vbox 1")
        button.show()
        vbox.pack_start(button)
        button = gtk.Button("vbox 2")
        button.show()
        vbox.pack_start(button)
        button = gtk.Button("vbox 3 in der letzten Reihe")
        button.show()
        vbox.pack_start(button)
        # Ende vertikale box
        hbox.pack_start(vbox)
        button = gtk.Button("Noch ein Knopf")
        button.show()
        hbox.pack_start(button)
        # Alles zusammenfügen
        self.window.add(hbox)
        # Anzeigen
        vbox.show()
        hbox.show()
        self.window.show()

    def event_delete(self, widget, event, data=None):
        return False

    def destroy(self, data=None):
        gtk.main_quit()

    def main(self):
        gtk.main()

if __name__ == "__main__":
    box = Box1()
    box.main()
```

Boxen werden erzeugt mit gtk.HBox(homogen, abstand) und gtk.VBox(homogen, abstand). Der Parameter *homogen* bestimmt, dass alle enthaltenen Widgets dieselbe Breite (HBox) oder dieselbe Höhe (VBox) haben. *abstand* ist ein Wert in Pixeln, der den Abstand zweier benachbarter Widgets vorgibt.

Widgets werden mit pack_start(Widget) von links nach rechts und von oben nach unten hinzugefügt.Um in umgekehrter Reihenfolge hinzuzufügen, kann man die Funktion pack_end(Widget) benutzen. Beide Funktionen haben mehr Parameter, als hier dargestellt, nämlich pack_start(Widget, expandieren, füllen, platz). Mit ihnen kann man bestimmen, ob ein Widget sich über den gesamten Platz ausdehnen darf (*expandieren=True*) und wenn ja, ob

der zusätzliche Platz dem Widget gehört (*füllen=True*). Mit *platz* wird zusätzlicher Rand um das hinzugefügte Widget herum belegt.

Boxen müssen ebenso wie andere Widgets sichtbar gemacht werden.

Wir ermutigen Sie an dieser Stelle, die verschiedenen Möglichkeiten der Parameter auszuprobieren, um ein Gefühl für ihre Bedeutung zu bekommen.

Grafische Elemente

Button

Schaltknöpfe dienen dazu, beim Drücken ein Signal zu erzeugen. Dieses Signal kann mit einer so genannten Callback-Funktion aufgefangen und bearbeitet werden, wie folgendes Beispiel zeigt. Auf Knopfdruck wird der Text eines Labels verändert:

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

import gtk

class Button1(object):

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_title("Von Buttons und Labeln")
        self.window.set_default_size(300, 100)
        self.window.connect("delete_event", self.event_delete)
        self.window.connect("destroy", self.destroy)
        # vertikale Box
        vbox = gtk.VBox(True, 30)
        button = gtk.Button("Drück mich")
        button.connect("clicked", self.button_clicked)
        button.show()
        vbox.pack_start(button)
        self.label = gtk.Label("")
        self.label.show()
        vbox.pack_start(self.label)
        self.window.add(vbox)
        vbox.show()
        self.window.show()

    def button_clicked(self, data=None):
        self.label.set_text("Danke! Sie haben ein einfaches\nLabel sehr
 glücklich gemacht.")

    def event_delete(self, widget, event, data=None):
        return False

    def destroy(self, data=None):
```

```
        gtk.main_quit()

    def main(self):
        gtk.main()

if __name__ == "__main__":
    b = Button1()
    b.main()
```

Callback-Funktionen wie button_clicked(data), oder in diesem Fall Methoden, haben in GTK+ immer eine ähnliche Gestalt. Mit dem optionalen Parameter data können Daten übergeben werden, die von connect(signal, callback, data) bereitgestellt werden.

Dialog, Textfeld, Rahmen

Das folgende Beispiel enthält zwei Textzeilen, in die der Anwender seinen Namen und seine E-Mail eingeben kann. Diese Daten werden in einem Anzeigedialog aufgeführt. Die Widgets sind innerhalb eines Rahmens mit einem Titelfeld untergebracht.

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

import gtk

class Entry1(object):

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_title("Entry und Frame")
        self.window.set_default_size(300, 100)
        self.window.connect("delete_event", self.event_delete)
        self.window.connect("destroy", self.destroy)

        frame = gtk.Frame("Möchten Sie SPAM bekommen?")
        self.window.add(frame)
        # vertikale Box
        vbox = gtk.VBox(True, 10)
        frame.add(vbox)
        # obere hbox
        hbox1 = gtk.HBox(True, 0)
        vbox.pack_start(hbox1)
        label = gtk.Label("Name")
        label.show()
        hbox1.pack_start(label)
        self.entry1 = gtk.Entry()
        self.entry1.show()
        hbox1.pack_start(self.entry1)
        hbox1.show()
        # untere hbox
```

```
        hbox2 = gtk.HBox(True, 0)
        vbox.pack_start(hbox2)
        label = gtk.Label("E-Mail")
        label.show()
        hbox2.pack_start(label)
        self.entry2 = gtk.Entry()
        self.entry2.show()
        hbox2.pack_start(self.entry2)
        hbox2.show()
        # Knopf
        button = gtk.Button("Im SPAM-Verteiler eintragen")
        button.connect("clicked", self.button_clicked)
        button.show()
        vbox.pack_start(button)
        # fertig vertikale Box
        vbox.show()
        frame.show()
        self.window.show()

    def button_clicked(self, data=None):
        name = self.entry1.get_text()
        email = self.entry2.get_text()
        text = "%s, wir schicken ihnen ab sofort regelmäßig SPAM an die
 Adresse %s!" % (name, email)
        dlg = gtk.MessageDialog(flags=gtk.DIALOG_MODAL,
buttons=gtk.BUTTONS_OK, message_format=text)
        dlg.set_title("Danke für ihre Adresse")
        dlg.run()
        dlg.destroy()

    def event_delete(self, widget, event, data=None):
        return False

    def destroy(self, data=None):
        gtk.main_quit()

    def main(self):
        gtk.main()

if __name__ == "__main__":
    e = Entry1()
    e.main()
```

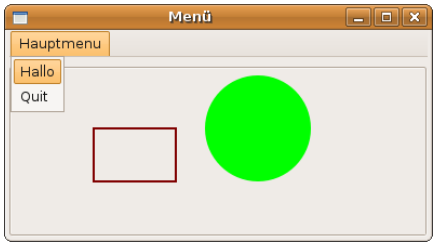
Rahmen werden mit einem optionalen Titelfeld durch die Funktion gtk.Frame(titel) erzeugt. Dieser Rahmen kann selbst ein Layout aufnehmen, in diesem Fall eine vertikale Box. Innerhalb dieser Box werden Label und Texteingaben eingefügt. Texteingaben sind einzeilige Textfelder, die mit gtk.Entry(länge) erzeugt werden. *länge* ist optional und meint die maximale Anzahl an Zeichen, die das Feld aufnehmen kann. Zum Schluss wird ein Knopf eingefügt, der mit der Callback-Methode button_clicked() verknüpft wird.

Innerhalb von button_clicked() werden die Textzeilen mit get_text() ausgelesen und in einem Text zusammengefügt. Anschließend wird ein Dialogobjekt erstellt. Dieser Dialog ist modal, lässt bis zum Ende keine weiteren Eingaben zu. Er enthält lediglich den Knopf "OK" und stellt den Text wie auch einen Titel dar.

Der Dialog wird mit run() gestartet. Wenn der Anwender auf "OK" klickt, wird die run()-Methode beendet und der Dialog mit destroy() geschlossen.

Menü, Zeichnen mit Cairo

Im folgenden Beispiel erzeugen wir ein Menü und eine Malfläche, in die wir mit Hilfe der Vektorgrafik-Bibliothek *Cairo* einige Grafikprimitive wie Kreise, Rechtecke oder Text zeichnen. Hierzu benötigen wir zwei Klassen, nämlich Malfeld, die Klasse, die den Zeichenbereich bereitstellt und Menu1, die das Menü erzeugt und das Malfeld innerhalb eines Frames aufnimmt. Aktiviert der Nutzer das Menü, wird entweder etwas Text gezeichnet oder das Programm verlassen.



Vektorgrafik mit Cairo

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

import gtk
import math

class Malfeld(gtk.DrawingArea):

    def __init__(self):
        gtk.DrawingArea.__init__(self)
        self.connect("expose_event", self.event_expose)
        self.drawText = False

    def event_expose(self, widget, event):
        grafik = widget.window.cairo_create()
        width, height = widget.window.get_size()
        width = width / 5
        height = height / 3
        self.draw(grafik, width, height)
        if self.drawText:
            self.draw_text(grafik, *self.drawText)
            self.drawText = False
        return False

    def draw(self, context, w, h):
        context.set_source_rgb(0.5, 0.0, 0.0)
        context.rectangle(w, h, w, h)
        context.stroke()
        context.set_source_rgb(0.0, 1.0, 0.0)
        context.arc(3.0 * w, h, min(w, h), 0, 2.0 * math.pi)
```

```
        context.fill()

    def draw_text(self, context, x, y, text):
        context.set_source_rgb(0.0, 0.0, 1.0)
        context.set_font_size(30)
        context.move_to(x, y + 30)
        context.show_text(text)

    def set_draw_text(self, x, y, text):
        self.drawText = (x, y, text)
        self.queue_draw()


class Menu1(object):

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.set_title("Menü")
        self.window.set_default_size(300, 100)
        self.window.connect("delete_event", self.event_delete)
        self.window.connect("destroy", self.destroy)
        # Menu
        menu = gtk.Menu()
        # erstes Item
        menuItem = gtk.MenuItem("Hallo")
        menuItem.show()
        menuItem.connect("activate", self.menu_clicked)
        menu.append(menuItem)
        # Quit-Item
        menuItem = gtk.MenuItem("Quit")
        menuItem.show()
        menuItem.connect("activate", lambda w: gtk.main_quit())
        menu.append(menuItem)
        # Hauptmenu wird in der Menuzeile angezeigt
        mainMenu = gtk.MenuItem("Hauptmenu")
        mainMenu.set_submenu(menu)
        mainMenu.show()
        # Menuzeile
        menuBar = gtk.MenuBar()
        menuBar.append(mainMenu)
        menuBar.show()
        # Malfeld
        frame = gtk.Frame("Malfeld")
        self.malen = Malfeld()
        self.malen.show()
        frame.add(self.malen)
        frame.show()
```

```
        # VBox
        vbox = gtk.VBox()
        vbox.pack_start(menuBar, False, False, 0)
        vbox.pack_start(frame, True, True, 0)
        vbox.show()
        self.window.add(vbox)
        self.window.show()

    def menu_clicked(self, widget, data=None):
        self.malen.set_draw_text(10, 10, "Menü wurde geklickt")

    def event_delete(self, widget, event, data=None):
        return False

    def destroy(self, data=None):
        gtk.main_quit()

    def main(self):
        gtk.main()


if __name__ == "__main__":
    m = Menu1()
    m.main()
```

Die Klasse Malfeld wird von der GTK+-Klasse DrawingArea abgeleitet, einer Klasse, die selbst eine Zeichenfläche implementiert, jedoch nur wenig Unterstützung für Grafikprimitive mitbringt. Diese Klasse stellt Ereignisse bereit, die darauf hinweisen, wann neu gezeichnet werden soll. "`expose_event`" ist ein solches Ereignis. Es tritt dann ein, wenn das Fenster verdeckt oder seine Größe verändert wurde. Dieses Ereignis verknüpfen wir mit der Methode event_expose().

event_expose(widget, event) übergibt das neu zu zeichnende Widget und das betreffende Ereignis. Innerhalb der Funktion wird mit cairo_create() die Unterstützung für Vektorgrafiken aktiviert. Der Rückgabewert ist ein Grafikkontext, also eine Klasse, die alles Nötige zum Zeichnen mitbringt. Es kann übrigens nur innerhalb von event_expose() gezeichnet werden. Der Grafikkontext kann nicht für spätere Zwecke gespeichert werden. Mit get_size() holen wir uns die Größe des Zeichenbereiches und berechnen Orte vor, wo wir Dinge zeichnen wollen. Anschließend werden einige Grafiken gemalt und bei Bedarf Text gezeichnet. Der Bedarf ergibt sich aus dem Aufruf der Methode set_draw_text(), die beim Aktivieren eines Menüs aufgerufen wird.

Innerhalb von draw() wird mit set_source_rgb(rot, grün, blau) eine Farbe vorgegeben. Die Farbwerte dürfen zwischen `0.0` und `1.0` liegen. Ein Rechteck wird mit rectangle(x, y, Breite, Höhe) vorbereitet, mit stroke() wird das Rechteck in seinen Umrissen gezeichnet. Füllen kann man das zuletzt angegebene Objekt mit fill(). Einen Kreis zeichnet man mit der Methode arc(x, y, Radius, Winkel_Von, Winkel_Bis).

draw_text() zeichnet Text an einer angegebenen Position. Hierzu wird die Schriftgröße festgelegt, anschließend wird ein Grafikcursor an eine Stelle im Fenster gesetzt (move_to(x, y)). show_text(Text) zeichnet dann den eigentlichen Text in der zuletzt festgelegten Farbe.

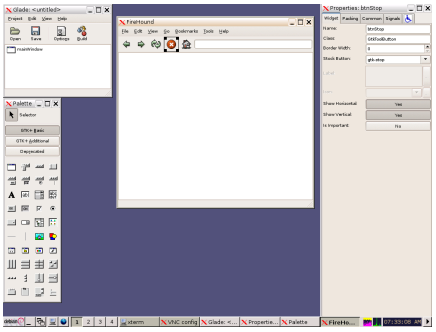
Die Klasse Menu1 erzeugt das Hauptfenster. Es werden einige Menüitems mit MenuItem(Text) erzeugt und mit entsprechenden Callback-Funktionen verknüpft. Diese Items werden zu einem Menüeintrag zusammengefasst und dem Menü, das mit Menu() erzeugt wird, hinzugefügt. Das angezeigte Menü ist wiederum ein Menuitem und hat den

Namen mainMenu. Dieses wird der MenuBar() hinzugefügt. Dies ist schon das, was im oberen Bereich des Fensters angezeigt werden soll.

Es werden also zuerst Items erzeugt, dann werden diese Items in einem Menü zusammengefasst und wiederum in einem Item eingetragen. Diese Items werden einer Menüzeile hinzugefügt. Auf diese Weise erzeugt man Menüs und Untermenüs. Das Verfahren gleicht jenem der Layouts. Dabei werden ebenfalls Elemente in Layoutboxen gepackt, diese werden wiederum in Boxen verpackt und irgendwann dem Fenster oder dem Frame hinzugefügt.

Benutzeroberflächen - schnell und einfach

Benutzeroberflächen können unter GTK+ mit grafischen Werkzeugen erstellt werden. Ein bekanntes Werkzeug ist *Glade*[1], dessen Bedienung wir kurz ansprechen wollen. Sie benötigen für das Beispiel das Modul python-glade2 sowie - selbstverständlich - Glade Version 2[2].

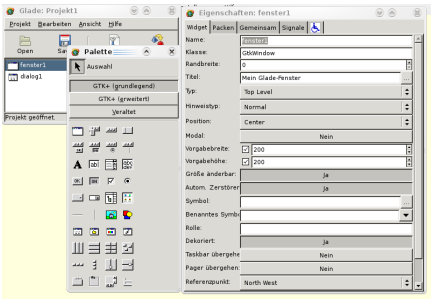


Glade hilft beim Erstellen von grafischen Benutzeroberfächen

Eine Benutzeroberfläche erstellt man in Glade fast vollständig visuell. Es wird eine XML-Datei gespeichert, die im Hauptprogramm geöffnet wird. Dort werden auch die in Glade erstellten Callback-Funktionen registriert, anschließend kann die Benutzeroberfläche genutzt werden.

Benutzeroberfläche erstellen

Für das folgende Beispiel starten Sie *Glade* und erzeugen ein Hauptfenster und einen Dialog. Wenn Sie das Beispiel praktisch nachvollziehen wollen, müssen Sie darauf achten, dass die Namen der GUI-Elemente (ID) mit denen im Quellcode übereinstimmen.



Benutzeroberfläche in Glade erstellen

Hauptfenster

In das Hauptfenster kommt zuerst eine vertikale Box mit drei Abschnitten. Anschließend wird ein Menü, ein Knopf und ein Label (als Statuszeile) eingefügt. Im Eigenschaften-Dialog wird im Reiter "Signale" ein Signal hinzugefügt. Wir verbinden dazu das Signal `destroy` (nicht zu verwechseln mit `destroy_signal`) mit der Callbackfunktion on_fenster1_destroy. Ebenfalls wird das `"clicked"`-Signal vom Knopf mit der Callback-Funktion on_button1_clicked verknüpft.

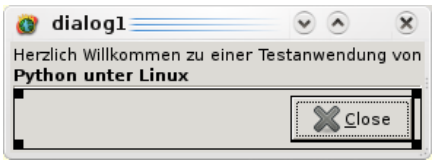
Das Hauptfenster

Das Menü wird etwas aufgeräumt, übrig bleiben `Datei->Quit` und `Hilfe->Info`. Die `"activate"`-Signale werden mit on_quit1_activate beziehungsweise on_info1_activate verknüpft.

Damit das folgende Beispiel funktioniert, muss das Statuszeilen-Label die ID "Label2" tragen und das Fenster die ID "fenster1" haben.

Dialog

Es wird ein Dialog mit der ID "dialog1" erstellt. Der Dialog soll über eine Standard-Knopfanordnung mit einem Schließen-Knopf verfügen. Das clicked-Signal dieses Knopfes wird mit on_closebutton1_clicked verknüpft. Wir fügen ein Label mit etwas Beispieltext hinzu.



Der Dialog

Abschluss

Dann sind wir fertig und können das Projekt speichern, wobei wir nur die entstehende projekt1.glade-Datei benötigen. Im Speichern-Dialog können wir darauf verzichten, dass alle sonstigen Programmdateien erstellt werden. Diese sind für Python nur überflüssiger Ballast.

Quellcode

Das folgende Programm lädt die in Glade erstellte Benutzeroberfläche und stellt sie dar. In der Statuszeile wird verfolgt, wie oft der zentrale Knopf gedrückt wird. Mit den Menüs kann man das Programm verlassen oder den erstellten Dialog aufrufen.

```
#!/usr/bin/python

import gtk
import gtk.glade

class HelloGlade(object):
    def __init__(self):
        self.gladeDatei = "projekt1.glade"
        self.widgets = gtk.glade.XML(self.gladeDatei, "fenster1")
        events = {"on_fenster1_destroy"    : gtk.main_quit,
            "on_quit1_activate"  : gtk.main_quit,
            "on_info1_activate"  : self.info_dlg,
            "on_button1_clicked" : self.statusbar_info}
        self.widgets.signal_autoconnect(events)
        self.numClicked = 0

    def info_dlg(self, widget):
        self.dlg = gtk.glade.XML(self.gladeDatei, "dialog1")
        events = {"on_closebutton1_clicked" :  self.dlg_close}
        self.dlg.signal_autoconnect(events)

    def dlg_close(self, widget):
        d = self.dlg.get_widget("dialog1")
        d.destroy()
```

```
    def statusbar_info(self, widget):
        label = self.widgets.get_widget("label2")
        self.numClicked += 1
        text = "Es wurde %d mal geklickt" % self.numClicked
        label.set_text(text)


if __name__ == "__main__":
    h = HelloGlade()
    gtk.main()
```

Die GUI-Datei wird mit gtk.glade.XML(datei, id) geladen. *id* ist hierbei optional, kann also entfallen. Sonst bezeichnet dieser Parameter den Namen des GUI-elementes, das geladen werden soll.

In einem Dictionary legen wir alle das Fenster betreffenden Callback-Funktionen mit ihren zugehörigen Aktionen an. Wird beispielsweise on_info1_activate aufgerufen, führt das in unserem Beispiel dazu, dass info_dlg() ebenfalls aufgerufen wird. Mit signal_autoconnect(events) verknüpfen wir die Ereignisse mit unseren entsprechenden Klassenmethoden.

Wenn das `Hilfe-Info`-Menü betätigt wird, dann wird in info_dlg() der betreffende Dialog gestartet. Auch hierfür holen wir uns die Beschreibung des Dialogs aus der Glade-Datei und verknüpfen die passenden Ereignisse. Soll mit einem Klick auf den Schließen-Knopf der Dialog beendet werden, wird dlg_close() aufgerufen. Das eigentliche Widget dieses Dialogs wird mit get_widget(id) ermittelt, anschließend wird ihm das `destroy`-Signal mit destroy() geschickt.

Drückt der Anwender auf den zentralen Knopf in der Mitte vom Hauptfenster, wird die Methode statusbar_info() aufgerufen. Es wird auch hier das Label-Widget mit get_widget(id) geholt, anschließend wird der Text mit set_text() verändert.

Zusammenfassung

Anwendungen mit grafischen Benutzeroberflächen lassen sich mit PyGTK leicht schreiben. In diesem Kapitel haben wir die Grundlagen des Layouts kennen gelernt, einige wenige Widgets vorgestellt und gezeigt, wie sich GUIs mit Glade erstellen lassen.

Anmerkungen

[1] Weitere Designer-Tools finden Sie unter (http://wiki.python.org/moin/GuiProgramming) aufgelistet.
[2] Zur neuen Version 3 liegen uns noch keine Python-Erfahrungen vor.

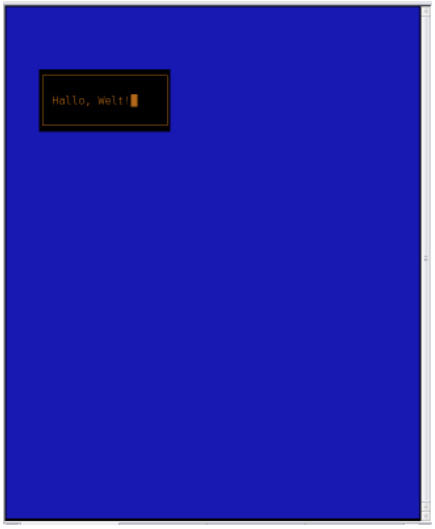
Textorientierte Benutzeroberflächen mit Curses

In diesem Kapitel geht es um rein textorientierte Benutzeroberflächen, wie Sie sie möglicherweise von rein textorientierten Programmen wie *mutt*, *lynx* oder dem Midnight-Commander her kennen. Es werden Teile des Curses-Moduls vorgestellt. Zu Curses gibt es ein eigenes hervorragendes Wikibook, nämlich Ncurses. Dieser Abschnitt benutzt einige der dort vorgestellten Ideen.

Hallo, Welt!

Unser erstes Beispiel erzeugt ein Fenster, in dem sich der historisch bekannte Spruch *Hallo, Welt!* zeigt. Bitte beachten Sie, dass dieses Beispiel Farben voraussetzt. Es wird nicht geprüft, ob ihr Terminal Farben darstellen kann.

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

import curses
#start
stdscr = curses.initscr()
# Keine Anzeige gedrückter Tasten
curses.noecho()
# Kein line-buffer
curses.cbreak()
# Escape-Sequenzen aktivieren
stdscr.keypad(1)

# Farben
curses.start_color()
curses.init_pair(1, curses.COLOR_GREEN, curses.COLOR_BLUE)
curses.init_pair(2, curses.COLOR_YELLOW, curses.COLOR_BLACK)

# Fenster und Hintergrundfarben
stdscr.bkgd(curses.color_pair(1))
stdscr.refresh()

win = curses.newwin(5, 20, 5, 5)
win.bkgd(curses.color_pair(2))
win.box()
win.addstr(2, 2, "Hallo, Welt!")
win.refresh()

# Warten auf Tastendruck
c = stdscr.getch()

# Ende
curses.nocbreak()
stdscr.keypad(0)
curses.echo()
curses.endwin()
```



Hallo Welt

Die beiden wichtigen Initialisierungsfunktionen in diesem Beispiel sind curses.initscr(), mit der die Curses-Umgebung eingerichtet wird und curses.start_color(), mit der das Programm die Möglichkeit erhält, Farbe darzustellen. Farben werden als Paar erzeugt, mit curses.init_pair(2, curses.COLOR_YELLOW, curses.COLOR_BLACK) wird ein Paar mit der Nummer `2` erzeugt, welches gelb auf schwarzem Grund darstellen kann. stdscr.bkgd(curses.color_pair(1)) referenziert ein solches Paar und legt den Hintergrund für das Terminal fest. Es wird nichts angezeigt, ohne dass ein Refresh (stdscr.refresh()) erfolgt.

Ein Fenster kann mit curses.newwin(Höhe, Breite, Y0, X0) erzeugt werden. Die Parameter sind ungewöhnlich, stehen hier Y-Werte vor den betreffenden X-Werten. Das ist eine Besonderheit der Terminal-Bibliotheken *curses* und *ncurses* und war schon immer so.

Einem so erzeugten Fenster kann ebenfalls ein Hintergrund zugewiesen werden, zur Verschönerung dient auch die Funktion box(). Sie zeichnet einen Rahmen rund um das Fenster. addstr(Y, X, Text) schreibt sodann einen Text auf das Fenster. Auch hier benötigt man wieder einen Refresh, um etwas sehen zu können. Die Ereignisbehandlung ist anders als in bisherigen Nutzeroberflächen rein von der Tastatur gesteuert. Mausereignisse werden ebenfalls zuerst von getch() entgegengenommen.

Nach dem der Nutzer eine Taste drückte, wird das Programm freundlich beendet. Hier ist insbesondere curses.endwin() zu nennen, das Gegenstück zu curses.initscr().

Einige der im Quellcode genutzten Funktionen haben wir Ihnen bei der Erläuterung unterschlagen. Das wollen wir gleich nachholen.

Kurzreferenz Curses

Hier werden ausgewählte Funktionen und Konstanten vorgestellt. Die Liste ist selbstverständlich nicht vollständig. Eine Darstellung der Art `[attrib]` bedeutet, dass der Wert optional ist und damit weggelassen werden darf.

Curses Funktionen

Hier eine Auflistung einiger gebräuchlicher curses-Funktionen:

Funktion	Bedeutung
cbreak() nocbreak()	Im CBreak-Modus werden Tasten einzeln angenommen, es wird nicht auf die Bestätigung durch *Return* gewartet.
attrib = color_pair(Farbnummer) init_pair(nummer, vordergrund, hintergrund)	*color_pair()*: Es wird auf eine vorher mit *Farbnummer* vordefinierte Vorder- und Hintergrundfarbe verwiesen. Die Attribute werden zurückgegeben. *init_pair()*: Erzeugt ein Farbpaar aus Vordergrundfarbe und Hintergrundfarbe und weist diese Kombination der Nummer zu. *nummer* kann ein Wert zwischen 1 und *COLOR_PAIRS*-1 sein.
doupdate() noutrefresh() refresh()	Diese Gruppe von Funktionen kümmert sich um den Refresh. *refresh()* ist ein *noutrefresh()* gefolgt von *doupdate()*. *noutrefresh()* updatet den virtuellen Bildschirm, markiert also den Bereich als einen, den es zu refreshen gilt, während *doupdate()* den physikalischen Bildschirm, also die echte Anzeige, auffrischt.
echo() noecho()	*echo()* bewirkt, dass Tastendrücke angezeigt werden. Dieses lässt sich mit *noecho()* ausschalten.

mouse = getmouse() (alle, aktuell) = mousemask(neu) vorher= mouseinterval(neu)	*getmouse()*: Bei einem Mausereignis kann mit dieser Funktion der Mauszustand als Tupel (*ID*, *X*, *Y*, unbenutzt, *Knopf*) bereitgestellt werden. *mousemask()*: erwartet eine Maske, bestehend aus den zu behandelnden Mausereignissen. Es wird ein Tupel aus allen verfügbaren Mausereignissen als Maske und den zuletzt aktiven zurückgeliefert. *mouseinterval()*: Setzt ein neues Intervall in Millisekunden, welches zwischen zwei Maustastendrücken vergehen darf, damit das Ereignis noch als *Doppelklick* erkannt wird. Die Funktion gibt das zuletzt aktive Intervall zurück.
bool = has_colors()	*True*, wenn das Teminal Farben darstellen kann. Sonst *False*.

Window-Funktionen

Einige Funktionen, die nur mit einem Window-Argument Sinn ergeben:

Funktion	Bedeutung
addch([y, x,] Z[,attrib]) addstr([y,x,] str[,attrib])	*addch()*:Zeichnet an der (optionalen) Stelle (y, x) das einzelne Zeichen *Z*. Dieses Zeichen kann mit Attributen versehen werden. *addstr()*: Selbiges für Strings
bkgd(Z[,attr])	Legt die Hintergrundfarbe / Zeichen / Attribute fest.
box([vertZ, horZ])	benutzt *vertZ* und *horZ* (beide optional) um einen Rahmen um ein Fenster zu Zeichnen. Ohne diese Parameter wird ein Linienrand gezeichnet.
clear()	Löscht beim nächsten *refresh()* das komplette Fenster
clrtoeol()	Löscht von der momentanen Cursorposition bis zum Ende der Zeile
keypad(value)	Ist *value* == 1, dann werden spezielle Tasten wie Funktionstasten von Curses als solche interpretiert. Mit *0* schaltet man dieses Verhalten wieder ab.

Konstanten

- Attribute, mit denen das Verhalten von Zeichen und geändert werden kann:

Konstante	Bedeutung
A_BLINK	Text wird blinkend dargestellt
A_BOLD	Text wird in Fettdruck dargestellt
A_NORMAL	Text wird wieder normal dargestellt.
A_UNDERLINE	Text wird unterstrichen angezeigt

- Tastaturkonstanten:

Konstante	Bedeutung
KEY_UP, KEY_LEFT, KEY_RIGHT, KEY_DOWN	Cursortasten
KEY_F1...KEY_Fn	Funktionstasten
KEY_BACKSPACE	Backspace-Taste
KEY_MOUSE	Ein Mausereignis trat ein

- Mauskonstanten:

Konstante	Bedeutung
BUTTON1_PRESSED .. BUTTON4_PRESSED	Knopf wurde gedrückt
BUTTON1_RELEASED .. BUTTON4_RELEASED	Knopf wurde losgelassen
BUTTON1_CLICKED .. BUTTON4_CLICKED	Knopf gedrück und wieder losgelassen, innerhalt eines mit *mouseinterval()* einstellbaren Intervalles.
BUTTON1_DOUBLE_CLICKED .. BUTTON4_DOUBLE_CLICKED,	Doppelklick
BUTTON1_TRIPLE_CLICKED, .. BUTTON4_TRIPLE_CLICKED,	3 klicks hintereinander
BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT	Es wurde die *Shift*-, *Steuerungs*, oder die *Alt*-Taste beim Klicken gedrückt. Man beachte, dass einige Terminals selbst darauf reagieren, das Programm diese Ereignisse also nicht mitbekommt.

- Farbkonstanten: COLOR_BLACK, COLOR_BLUE, COLOR_CYAN, COLOR_GREEN, COLOR_MAGENTA, COLOR_RED, COLOR_WHITE, COLOR_YELLOW

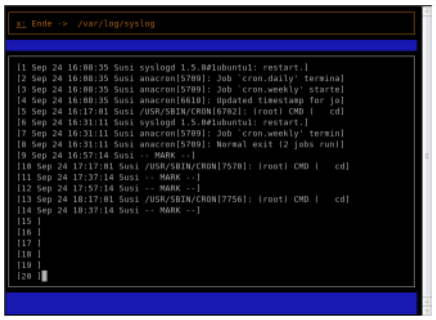
Mehr Fenster

Im folgenden Beispiel geht es um das Lesen von Log-Dateien (Sie müssen Leserecht darauf haben, sonst funktioniert es nicht) und Menüs. Es werden auf Wunsch zwei Fenster erzeugt, eines stellt ein Menü dar, im anderen wird wahlweise /var/log/syslog und /var/log/messages dargestellt. Scrollen kann man in den Dateien mit den Cursor-Tasten. Beendet wird mit der Taste **x**.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import curses
import linecache

def init_curses():
    stdscr = curses.initscr()
    curses.noecho()
    curses.cbreak()
    stdscr.keypad(1)
    curses.start_color()
    curses.init_pair(1, curses.COLOR_GREEN, curses.COLOR_BLUE)
    curses.init_pair(2, curses.COLOR_YELLOW, curses.COLOR_BLACK)
    stdscr.bkgd(curses.color_pair(1))
    stdscr.refresh()
    return stdscr

def show_menu(win):
    win.clear()
    win.bkgd(curses.color_pair(2))
    win.box()
    win.addstr(1, 2, "F1:", curses.A_UNDERLINE)
    win.addstr(1, 6, "messages")
    win.addstr(1, 20, "F2:", curses.A_UNDERLINE)
```

```
    win.addstr(1, 24, "syslog")
    win.addstr(1, 38, "x:", curses.A_UNDERLINE)
    win.addstr(1, 42, "Exit")
    win.refresh()

def read_file(menu_win, filename):
    menu_win.clear()
    menu_win.box()
    menu_win.addstr(1, 2, "x:", curses.A_UNDERLINE)
    menu_win.addstr(1, 5, "Ende ->")
    menu_win.addstr(1, 14, filename)
    menu_win.refresh()
    file_win = curses.newwin(22, 80, 4, 0)
    file_win.box()
    line_start = 1
    line_max = 20
    while True:
      file_win.clear()
      file_win.box()
      for i in xrange(line_start, line_max + line_start):
          line = linecache.getline(filename, i)
          s = ''
          if len(line) > 60:
              s = '[%d %s]' % (i, line[:60])
          else:
              s = '[%d %s]' % (i, line[:-1])
          file_win.addstr(i - line_start + 1, 2, s)
      file_win.refresh()
      c = stdscr.getch()
      if c == ord('x'):
          break
      elif c == curses.KEY_UP:
          if line_start > 1:
              line_start -= 1
      elif c == curses.KEY_DOWN:
          line_start += 1

stdscr = init_curses()
mwin = curses.newwin(3, 80, 0, 0)

while True:
  stdscr.clear()
  stdscr.refresh()
  show_menu(mwin)
  c = stdscr.getch()
  if c == ord('x'):
    break
```

```
  elif c == curses.KEY_F1:
    read_file(mwin, '/var/log/messages')
    show_menu(mwin)
  elif c == curses.KEY_F2:
    read_file(mwin, '/var/log/syslog')
    show_menu(mwin)

curses.nocbreak()
stdscr.keypad(0)
curses.echo()
curses.endwin()
```



Logdateien

Es wird ein neues Modul mit dem Namen *linecache* eingebunden. Dieses Modul enthält die Funktion linecache.getline(filename, i). Hiermit kann eine bestimmte Zeile aus der Datei ausgegeben werden. Wir nutzen sie in read_file().

In read_file() wird eine Datei gelesen. Zuerst wird das Menüfenster aktualisiert, dann wird ein neues Fenster erzeugt und eine Datei zeilenweise eingelesen. Es wird darauf geachtet, dass maximal 60 Zeichen einer Zeile dargestellt werden. Anschließend wird eine eigene Warteschleife durchlaufen, die die Tastatureingaben empfängt. Bei der Eingabe von **x** wird die Schleife abgebrochen und zum Hauptprogramm zurückverzweigt. Falls eine Cursortaste gedrückt wurde, so wird der Bereich der Logdatei, der beim nächsten Durchgang gelesen wird, neu berechnet. Hier ist nicht wichtig, wie lang die Datei ist, denn linecache.getline() liefert uns einen leeren String zurück, wenn wir versuchen, über das Dateiende hinaus zu lesen.

Im Hauptprogramm wird ein neues Fenster stdscr erzeugt. Von dort aus wird die Hauptschleife abgearbeitet. Bei **F1** wird /var/log/messages und bei **F2** /var/log/syslog gelesen. Mit **x** wird das Programm abgebrochen. Bei jedem Durchgang wird das Menü aktualisiert und das alte Log-Fenster, welches vielleicht noch auf dem Bildschirm erscheint, gelöscht.

Große Fenster

Im letzten Beispiel haben wir gesehen, dass es Mühe kostet, einen Text in einem Fenster darzustellen, wenn die Zeilenlänge oder die Textlänge größer ist als die Fensterausmaße. Es gibt einen Weg darum herum. Wir können mit Curses sehr große Fenster erzeugen, und nur einen Teil auf den Bildschirm bringen. Solche großen Fenster heißen *Pad* und sind nicht an die Ausmaße des darunter liegenden Fensters gebunden. Das folgende Programm nutzt solch ein Pad, es ist 10000 Zeilen hoch und 3000 Spalten breit. Es wird diesmal die Datei /var/log/messages gelesen, navigieren können Sie mit allen vier Cursortasten, bei der Eingabe von **x** wird das Programm beendet.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import curses
import linecache

def init_curses():
    stdscr = curses.initscr()
    curses.noecho()
    curses.cbreak()
```

```
    stdscr.keypad(1)
    curses.start_color()
    curses.init_pair(1, curses.COLOR_BLACK, curses.COLOR_BLUE)
    curses.init_pair(2, curses.COLOR_GREEN, curses.COLOR_BLUE)
    stdscr.bkgd(curses.color_pair(1))
    stdscr.box()
    stdscr.refresh()
    return stdscr

stdscr = init_curses()
# Aktuelle Screengröße ermitteln
maxy, maxx = stdscr.getmaxyx()

# Großes Pad erzeugen
pad = curses.newpad(10000, 3000)
for i in xrange(10000):
    line = linecache.getline('/var/log/messages', i)
    s = '[%5d] %s' % (i, line)
    pad.addstr(i, 1, s)

# Wir merken uns, wo wir sind
line_start = 1
col_start = 1

while True:
  # Teile des Pads aufs Display bringen
  pad.refresh(line_start, col_start, 1, 1, maxy-2, maxx-2)
  c = stdscr.getch()
  if c == ord('x'):
    break
  elif c == curses.KEY_DOWN:
    line_start += 1
  elif c == curses.KEY_UP:
    if line_start > 1: line_start -= 1
  elif c == curses.KEY_LEFT:
    if col_start > 1: col_start -= 1
  elif c == curses.KEY_RIGHT:
    col_start += 1

# ende
curses.nocbreak()
stdscr.keypad(0)
curses.echo()
curses.endwin()
```

Um das innere Pad an das äußere Fenster anzupassen, fragen wir mit stdscr.getmaxyx() die Ausmaße des Fensters ab. Unser Pad wird dann im Inneren liegen, umgeben von einem Rand. Das eigentliche Pad wird mit curses.newpad(10000, 3000) erzeugt. Es ist 10000 Zeilen hoch und 3000 Zeilen lang. Das sollte für diese Datei

reichen. Mit der schon bekannten Methode wird das Fenster mit dem Dateiinhalt gefüllt. Neu ist noch, dass wir lediglich einen Ausschnitt aus dem Pad auf den Bildschirm bringen. Dies erledigt pad.refresh(PadStartZeile, PadStartSpalte, WinY, WinX, WinBreite, WinHöhe). Hierbei sind "Win*"-Argumente die Ausmaße des umgebenden Fensters. "PadStartZeile" und "PadStartSpalte" hingegen sind derjenige Ursprung, der im inneren des Pads liegt und in der linken oberen Ecke des Fensters zu sehen sein soll. Beim Druck auf eine Cursor-Taste werden diese Werte entsprechend modifiziert, dann wird das Pad neu gezeichnet. Es wird also innerhalb des Pads gescrollt.

Mausereignisse

Auf Mausereignisse sind wir bisher noch nicht eingegangen. Grund genug, das hier in einem kleinen Abschnitt nachzuholen. Mausereignisse werden in der Hauptschleife ebenso bearbeitet wie Tasten, die spezielle Taste ist curses.KEY_MOUSE.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import curses
import linecache

def init_curses():
    stdscr = curses.initscr()
    curses.noecho()
    curses.cbreak()
    stdscr.keypad(1)
    curses.start_color()
    curses.init_pair(1, curses.COLOR_BLACK, curses.COLOR_BLUE)
    stdscr.bkgd(curses.color_pair(1))
    stdscr.box()
    stdscr.refresh()
    return stdscr

stdscr = init_curses()
# Maus initialisieren
avail, oldmask = curses.mousemask(curses.BUTTON1_PRESSED)
curses.mousemask(avail)

while True:
  c = stdscr.getch()
  if c == ord('x'):
    break
  elif c == curses.KEY_MOUSE:
    id, x, y, z, button = curses.getmouse()
    s = "Mouse-Ereignis bei (%d ; %d ; %d), ID= %d, button = %d" % (x,
y, z, id, button)
    stdscr.addstr(1, 1, s)
    stdscr.clrtoeol()
    stdscr.refresh()
```

```
# ende
curses.nocbreak()
stdscr.keypad(0)
curses.echo()
curses.endwin()
```

Damit auf Mausereignisse überhaupt reagiert werden kann, muss eine Maus-Maske erstellt werden. Die Funktion curses.mousemask() erwartet eine solche Maske und liefert ein Tupel aus allen verfügbaren Mausereignissen und der letzten Maske zurück. Wir wollen wirklich alle Ereignisse behandelt wissen und nutzen einen zweiten Aufruf dieser Funktion, um das mitzuteilen.

curses.getmouse() liefert uns ein Tupel mit den benötigten Mausinformationen. Diese werden auf dem Bildschirm dargestellt.

Textbox

Curses enthält einen rudimentären Editor, den man nutzen kann, um innerhalb eines Fensters Text zu schreiben. Er wird eingebunden durch curses.textpad.Textbox(Fenster) und beinhaltet eine Methode edit(), um Text entgegenzunehmen. Diese Methode kann mit einem Validator zum Filtern von Tastendrücken versehen werden. Das folgende Programm demonstriert, wie man diesen Editor einbindet und den eingegebenen Text in einem anderen Fenster darstellt. Beendet wird der Editor mit **STRG+G**. Dann wird im anderen Fenster der eingegebene Text dargestellt, das Programm kann anschließend mit einem Tastendruck beendet werden.

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

import curses
import curses.textpad

# Initialisieren
stdscr = curses.initscr()
curses.noecho()
curses.cbreak()
stdscr.keypad(1)
curses.start_color()
curses.init_pair(1, curses.COLOR_GREEN, curses.COLOR_BLUE)
curses.init_pair(2, curses.COLOR_YELLOW, curses.COLOR_BLACK)
stdscr.bkgd(curses.color_pair(1))
stdscr.refresh()

# Edit-Fenster
win1 = curses.newwin(20, 40, 10, 5)
win1.bkgd(curses.color_pair(2))

# Darstellungsfenster
win2 = curses.newwin(20, 40, 10, 50)
win2.bkgd(curses.color_pair(2))
win2.refresh()

# Textbox
```

```
textbox = curses.textpad.Textbox(win1)
text = textbox.edit()

# Text übernehmen
win2.addstr(0, 0, text)
win2.refresh()

# Ende
c = stdscr.getch()
curses.nocbreak()
stdscr.keypad(0)
curses.echo()
curses.endwin()
```

Zusammenfassung

Sie haben in diesem Kapitel einen groben Überblick über einige ausgewählte Fähigkeiten von *Curses* erhalten. *Curses* selbst kann nur sehr einfache Dinge, wie Rechtecke als Fenster verwalten und auf Tastendrücke reagieren. Dafür sind mit *Curses* geschriebene Oberflächen schneller als grafische, was aber auch kein Kunststück ist. Als Ausblick können wir Ihnen das Modul UrWid (http://excess.org/urwid/) nennen, welches ein recht vollständiges TUI-Toolkit darstellt und auf *Curses* basiert.

Weiterführende Informationen

Bitte bis auf begründbare Ausnahmen nicht mehr als **2** Links pro Thema. Dies soll keine Linksammlung werden, sondern das Buch sinnvoll ergänzen. Vorhandene Wikibücher sollen am Anfang der Kapitel oder in der Projektbeschreibung stehen. Bessere Verweise sollen weniger aussagekräftige Links ablösen.

Einführung, erste Schritte

(Allgemeine Einführungen)

- http://diveintopython.net/toc/index.html (http://diveintopython.net/toc/index.html) englisch, beschreibt Version 2.3, Beispiele

Module

(Mitgelieferte Module)

- http://docs.python.org/2/py-modindex.html (http://docs.python.org/2/py-modindex.html) Liste von Modulen, englisch, Beispiele

Dateien, I/O

- http://www.johnny-lin.com/cdat_tips/tips_fileio/bin_array.html (http://www.johnny-lin.com/cdat_tips/tips_fileio/bin_array.html) englisch, einlesen eines Arrays

Datenbanken

- http://wiki.python.org/moin/MySQL (http://wiki.python.org/moin/MySQL) - Linksammlung, englisch
- http://wiki.python.org/moin/PostgreSQL (http://wiki.python.org/moin/PostgreSQL) - Linksammlung, englisch

PyGame

- http://www.spieleprogrammierer.de/wiki/Pygame-Tutorial Tutorial, deutsch

Qt

- http://zetcode.com/tutorials/pyqt4/ (http://zetcode.com/tutorials/pyqt4/) Tutorial, englisch, Beispiele
- http://www.commandprompt.com/community/pyqt/ (http://www.commandprompt.com/community/pyqt/) Tutorial, englisch, Beispiele

wxPython

- http://www.zetcode.com/wxpython/ (http://www.zetcode.com/wxpython/) Tutorial, englisch
- http://www.wxpython.org/docs/api/ (http://www.wxpython.org/docs/api/) wxPython API, englisch

PyGTK

- http://www.pygtk.org/ (http://www.pygtk.org/) Tutorial und Referenzdokumentation, englisch

Curses

- http://docs.python.org/2.7/howto/curses.html (http://docs.python.org/2.7/howto/curses.html) HowTo, englisch

Sonstiges

- http://www.python-forum.de/ (http://www.python-forum.de/) Python-Forum, deutsch

---

Quelle(n) und Bearbeiter des/der Artikel(s)

Python unter Linux: ALLES *Quelle*: https://de.wikibooks.org/w/index.php?oldid=639673 *Bearbeiter*: Dirk Huenniger, Tandar

Quelle(n), Lizenz(en) und Autor(en) des Bildes

Image:Nuvola_apps_terminal.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:Nuvola_apps_terminal.png *Lizenz*: GNU Lesser General Public License *Bearbeiter*: Alno, Alphax

Image:Searchtool.svg *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:Searchtool.svg *Lizenz*: GNU Lesser General Public License *Bearbeiter*: David Vignoni, Ysangkok

Image:Attention green.svg *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:Attention_green.svg *Lizenz*: Public Domain *Bearbeiter*: As per Attention_yellow.svg: (apparently users Ttog and D0ktorz derivative work: FT2 (talk)

Image:A_small_cup_of_coffee.JPG *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:A_small_cup_of_coffee.JPG *Lizenz*: Creative Commons Attribution-Sharealike 2.0 *Bearbeiter*: Julius Schorzman

Datei:PythonLinuxPygame1.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxPygame1.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxPygame2.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxPygame2.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxPygame3.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxPygame3.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxPygame4.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxPygame4.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxQt4-Bsp1.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxQt4-Bsp1.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxQt4-Bsp2.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxQt4-Bsp2.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxQt4-Bsp3-1.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxQt4-Bsp3-1.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxQt4-Bsp13-2.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxQt4-Bsp13-2.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxGuiqt4-des4-1.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxGuiqt4-des4-1.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxGuiqt4-des4-2.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxGuiqt4-des4-2.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxPyGtk-cairo1.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxPyGtk-cairo1.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

File:Glade on Debian.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:Glade_on_Debian.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Crazycomputers of the English Wikipedia

File:PythonUnterLinuxPyGtk-glade1.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonUnterLinuxPyGtk-glade1.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

File:PythonUnterLinuxPyGtk-glade1-fenster.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonUnterLinuxPyGtk-glade1-fenster.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

File:PythonUnterLinuxPyGtk-glade1-dialog.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonUnterLinuxPyGtk-glade1-dialog.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxCurses-hallowelt.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxCurses-hallowelt.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Datei:PythonLinuxCurses-logdateien.png *Quelle*: https://de.wikibooks.org/w/index.php?title=Datei:PythonLinuxCurses-logdateien.png *Lizenz*: Public Domain *Bearbeiter*: Tandar

Lizenz