Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

>> Download (Zip, ca. 4,8 MB)

Dieses Buch bietet sowohl eine **umfassende Einführung** in die Sprache Python *(aktuell zur Version 2.5)* als auch **viele weiterführende Kapitel zu fortgeschrittenen Themen** wie GUI-Entwicklung, Web-Programmierung mit Django oder Netzwerkkommunikation. Nach der Lektüre sind Sie in der Lage, Python professionell einzusetzen. Die Einführung erfolgt systematisch vom ersten einfachen Programm bis hin zu komplexen objektorientierten Programmen. Das Buch ist konsequent praxisorientiert und zum Lernen und Nachschlagen hervorragend geeignet.

Nutzen Sie die HTML-Version zum Reinschnuppern oder als immer verfügbare Ergänzung zu Ihrem Buch.

Auf unserer Katalogseite steht Ihnen ab sofort zusätzlich eine **Volltextsuche im Buch** zur Verfügung.

Die gedruckte Version des Buches erhalten Sie in unserem Online-Shop - versandkostenfrei innerhalb Deutschlands und Österreichs.
Zum Online-Shop

Inhaltsverzeichnis

top

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Der Anfang ist die Hälfte des Ganzen.« – Aristoteles

1 Einleitung

Bevor Sie in die wunderbare Welt von Python eintauchen, möchten wir Ihnen dieses Buch kurz vorstellen. Dabei werden Sie grundlegende Informationen darüber erhalten, wie das Buch aufgebaut ist und was Sie bei der Lektüre beachten sollten. Außerdem umreißen wir die Ziele und Konzepte des Buches, damit Sie im Vorfeld wissen, was Sie erwartet.

top

1.1 Warum haben wir dieses Buch geschrieben?

Wir, Peter Kaiser und Johannes Ernesti, sind vor einigen Jahren mehr oder weniger durch Zufall auf die Programmiersprache Python aufmerksam geworden und bis heute dabei geblieben. Die Einfachheit, Flexibilität und Eleganz von Python hat uns fasziniert. Mit Python lässt sich eine Idee in sehr kurzer Zeit zu einem ersten lauffähigen Programm fortentwickeln. Zudem braucht sich der Programmierer keine Gedanken über die Lauffähigkeit seines Codes auf verschiedenen Betriebssystemen zu machen, da Python-Code unmodifiziert unter allen wichtigen Betriebssystemen läuft. Kurzum: Die Programmiersprache Python vereinfacht den Programmieralltag erheblich und erlaubt es dem Programmierer, kurze, elegante und produktive Programme für komplexe Aufgaben zu schreiben. Aus diesen Gründen nutzen wir für unsere eigenen Projekte mittlerweile fast ausschließlich Python.

Allerdings hatte unsere erste Begegnung mit Python auch ihre Schattenseiten. Zwar gibt es viele Bücher zum Thema Python und auch im Internet findet sich sehr viel Dokumentationsmaterial, doch diese sind entweder sehr technisch oder nur zum Einstieg in die Sprache Python gedacht. Die Fülle an Tutorials macht es einem Einsteiger einfach, in die Python-Welt »hineinzuschnüffeln« und die ersten Schritte zu wagen. Es ist mit guten Büchern sogar möglich, innerhalb weniger Tage ein fundiertes Grundwissen aufzubauen, mit dem sich durchaus arbeiten lässt. Das Problem

tritt jedoch erst beim Übergang zur fortgeschrittenen Programmierung auf, weil man nun mit den einführenden Tutorien nicht mehr voran kommt, trotzdem aber noch nicht dazu in der Lage ist, die zumeist sehr technische Dokumentation von Python zur Weiterbildung zu nutzen.

Unserer Ansicht nach fehlt ein Leitfaden, der einen breiten Überblick über die Möglichkeiten von Python bietet, ohne sich dabei in allzu technischen Details zu verlieren. Vielmehr sollten das Problem und der Lösungsansatz im Vordergrund stehen. Einen solchen Leitfaden möchten wir Ihnen mit diesem Buch präsentieren.

Dieses Buch bietet Ihnen neben einer umfassenden Einführung in die Sprache Python viele weiterführende Kapitel, die Sie letztendlich dazu in die Lage versetzen, Python professionell einzusetzen. Außerdem gibt Ihnen das Buch stets Anhaltspunkte und Begriffe an die Hand, mit denen Sie eine weiterführende Recherche, beispielsweise in der Python-Dokumentation, durchführen können.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Readability counts.«
– Tim Peters in »The Zen of Python«

2 Überblick über Python

Bevor es an die Programmierung mit Python geht, möchten wir Ihnen Python in diesem Kapitel zunächst einmal vorstellen. Dazu beschäftigen wir uns erst mit der Geschichte von Python und besprechen danach die grundlegenden Konzepte, auf denen Python aufbaut. In den beiden letzten Abschnitten dieses Kapitels werden wir einen Überblick über Einsatzmöglichkeiten und -gebiete von Python geben.

Betrachten Sie dieses Kapitel also als narrative Einführung in die Thematik, die den darauf folgenden fachlichen Einstieg vorbereitet.

top

2.1 Geschichte und Entstehung

Python wurde Anfang der 90er-Jahre von dem Holländer *Guido van Rossum* am *Centrum voor Wiskunde en Informatica* (CWI) in Amsterdam entwickelt. Ursprünglich war Python als Scriptsprache für das verteilte Betriebssystem Amoeba gedacht.

Vor der Entwicklung von Python hatte van Rossum an der Entwicklung der Programmiersprache *ABC* mitgewirkt, die mit dem Ziel entworfen wurde, möglichst einfach zu sein, sodass sie problemlos einem interessierten Laien ohne Programmiererfahrung beigebracht werden kann. Die Erfahrung aus positiver und negativer Kritik an ABC nutzte van Rossum für die Entwicklung von Python. Er schuf somit eine Programmiersprache, die mächtig und zugleich einfach und leicht zu erlernen sein sollte. Inzwischen liegt Python in der Version 2.5 vor. Die Version 3.0 wird Mitte 2008 folgen.

Der Name Python lehnt sich nicht etwa an die Schlangenart an, sondern ist eine Hommage an die britische Komikertruppe Monty Python. Ebenfalls auf Monty Python geht die Bezeichnung BDFL (für »Benevolent Dictator for Life«) zurück, die die Python-Community für Guido van Rossum erfand. Inzwischen arbeitet Guido van Rossum für Google, wo er die Hälfte seiner Arbeitszeit damit verbringt, Python weiterzuentwickeln.

Seit 2001 existiert die nicht-kommerzielle *Python Software Foundation*, die die Rechte am Python-Code besitzt und Lobbyarbeit für Python betreibt. So organisiert die Python Software Foundation beispielsweise die PyCon-Konferenz, die jährlich in den USA stattfindet. Auch in Europa finden regelmäßig größere und kleinere Python-Konferenzen statt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Python is more concerned with making it easy to write good programs than difficult to write bad ones.« – Steve Holden auf comp.lang.python

3 Die Arbeit mit Python

Kommen wir nun zum etwas technischeren Teil der Einleitung, in dem das notwendige Vorwissen für die folgenden Kapitel vermittelt wird. Dabei geht es zunächst um das Einrichten der Entwicklungsplattform und um eine grundlegende Einführung in das Erstellen und Ausführen eines Python-Programms.

top

3.1 Die Verwendung von Python

Die jeweils aktuelle Version von Python können Sie von der offiziellen Python-Website unter http://www.python.org als Installationsdatei für Ihr Betriebssystem herunterladen und installieren. Alternativ können Sie Python 2.5.1 von der CD installieren, die diesem Buch beiliegt.

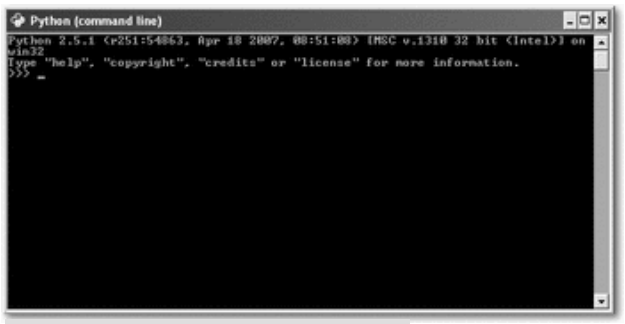
Auf die eigentliche Installation soll hier nicht näher eingegangen werden, da sich diese an die in Ihrem Betriebssystem üblichen Vorgänge anlehnt und wir davon ausgehen, dass Sie wissen, wie man auf Ihrem System Software installiert.

Grundsätzlich werden, wenn man einmal von Python selbst absieht, zwei wichtige Komponenten installiert: der interaktive Modus und IDLE.

Im sogenannten *interaktiven Modus*, auch Python Shell genannt, können einzelne Programmzeilen eingegeben und die Ergebnisse direkt betrachtet werden. Der interaktive Modus ist damit besonders zum Lernen der Sprache Python interessant und wird deshalb in diesem Buch häufig verwendet.

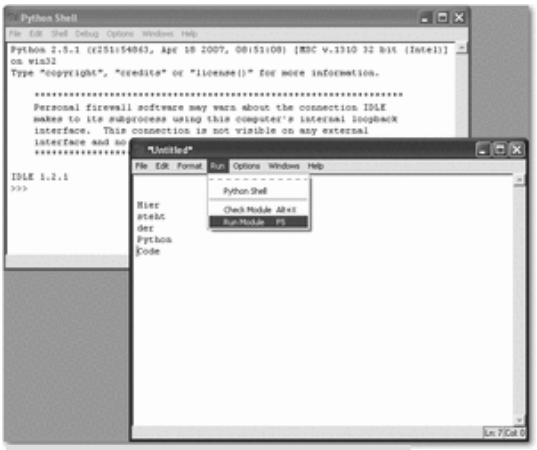
Bei *IDLE* (Integrated DeveLopment Environment) handelt es sich um eine rudimentäre Python-Entwicklungsumgebung mit grafischer Benutzeroberfläche. Beim Starten von IDLE wird zunächst nur ein Fenster geöffnet, das eine Python Shell beinhaltet. Zudem kann in IDLE über den Menüpunkt File • New

Window eine neue Python-Programmdatei erstellt und editiert werden. Nachdem die Programmdatei gespeichert wurde, kann sie über den Menüpunkt Run • Run Module in der Python Shell von IDLE ausgeführt werden. Abgesehen davon bietet IDLE dem Programmierer einige Komfortfunktionen wie beispielsweise das farbige Hervorheben bestimmter Code-Elemente (»Syntax Highlighter«) oder eine automatische Vervollständigung von Code.



Hier klicken, um das Bild zu vergrößern

Abbildung 3.1 Python im interaktiven Modus (Python Shell)



Hier klicken, um das Bild zu vergrößern

Abbildung 3.2 Die Entwicklungsumgebung IDLE

Wenn Sie mit IDLE nicht zufrieden sind, finden Sie eine Übersicht über die wichtigsten Python-Entwicklungsumgebungen im Anhang dieses Buchs. Zudem befindet sich auf der offiziellen Python-Website unter http://wiki.python.org/moin/PythonEditors eine umfassende Auflistung aller Entwicklungsumgebungen und Editoren für Python.

Die folgenden Abschnitte geben eine kurze Einführung darüber, wie Sie den interaktiven Modus und IDLE auf Ihrem System starten und verwenden können. In Abschnitt 3.2 werden wir dann darauf eingehen, wie eine Python-Programmdatei erstellt und ausgeführt wird.

top

3.1.1 Windows ▼▲

Sie finden die Windows-Installationsdatei von Python 2.5.1 auf der dem Buch beigelegten CD-ROM.

Nach der Installation von Python unter Windows finden Sie im Wesentlichen zwei neue Einträge im Startmenü: »Python (command line)« und »IDLE (Python GUI)«. Ersteres startet den interaktiven Modus von Python in der Kommandozeile (»schwarzes Fenster«) und Letzteres die grafische Entwicklungsumgebung IDLE.

top

3.1.2 Linux ▼▲

Sie finden den Quellcode von Python 2.5.1 auf der dem Buch beigelegten CD–ROM.

Beachten Sie, dass Python bei vielen Linux-Distributionen bereits im Lieferumfang enthalten ist oder sich mit dem jeweiligen Paketmanager der Distribution bequem nachinstallieren lässt. Sollten Sie eine Distribution ohne Paketmanager einsetzen oder sollte Python nicht verfügbar sein, müssen Sie den Quellcode von Python selbst kompilieren und installieren. Dazu können Sie den Anweisungen der im Quelltext enthaltenen Readme-Datei folgen.

Nach der Installation können Sie den interaktiven Modus bzw. IDLE aus einer Shell heraus mit den Befehlen `python` bzw. `idle` starten.

top

3.1.3 Mac OS X ▲

Sie finden die Mac OS X-Installationsdatei von Python 2.5.1 auf der dem Buch beigelegten CD-ROM.

Nach der Installation von Python können Sie den interaktiven Modus und IDLE, ähnlich wie bei Linux, aus einer Terminal-Sitzung heraus mit den Befehlen `python` bzw. `idle` starten.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Hmm, wo ist denn die Any-Key-Taste? Naja, ich bestell mir erst einmal ein Bier!« – Homer Simpson

4 Der interaktive Modus

Startet man den Python-Interpreter ohne Argumente, gelangt man in den sogenannten **interaktiven Modus**. Dieser Modus bietet dem Programmierer die Möglichkeit, Kommandos direkt an den Interpreter zu senden, ohne zuvor ein Programm erstellen zu müssen. Der interaktive Modus wird häufig genutzt, um schnell etwas auszuprobieren oder zu testen. Zum Schreiben wirklicher Programme ist er allerdings nicht geeignet. Dennoch möchten wir hier mit dem interaktiven Modus beginnen, da er einen schnellen und unkomplizierten Einstieg in die Sprache Python ermöglicht.

Dieser Abschnitt soll Sie mit einigen Grundlagen vertraut machen, die zum Verständnis der folgenden Kapitel wichtig sind. Am besten setzen Sie die Beispiele dieses Kapitels am Rechner parallel zu Ihrer Lektüre um.

Zur Begrüßung gibt der Interpreter einige Zeilen aus, die Sie in ähnlicher Form jetzt auch vor sich haben müssten:

```
Python 2.5.1 (r251:54863, Apr 19 2007, 11:03:39)
[GCC 4.1.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Nach der Eingabeaufforderung (>>>) kann beliebiger Python-Code eingegeben werden. Bei Zeilen, die nicht mit >>> beginnen, handelt es sich um Ausgaben des Interpreters.

Zur Bedienung des interaktiven Modus ist noch zu sagen, dass er über eine *History-Funktion* verfügt. Das heißt, dass Sie über die ↑ und ↓ -Tasten alte Eingaben bequem wieder hervorholen können und nicht erneut eingeben müssen.

Wir beginnen mit der Einführung von konstanten Werten. Dabei unterscheiden wir zunächst einmal drei Typen von Werten: ganze Zahlen, Gleitkommazahlen und Zeichenketten. Es gibt dabei

bestimmte Regeln, nach denen man einen Zahlenwert oder eine Zeichenkette zu schreiben hat, damit diese vom Interpreter erkannt werden. Eine solche Schreibweise nennt man *Literal*.

top

4.1 Ganze Zahlen

Als erstes und einfachstes Beispiel erzeugen wir im interpretierten Modus eine ganze Zahl. Der Interpreter antwortet darauf, indem er ihren Wert ausgibt:

```
>>> -9
-9
>>> 1139
1139
>>> +12
12
```

Das Literal für eine ganze Zahl besteht dabei aus den Ziffern 1 bis 9. Zudem kann ein positives oder negatives Vorzeichen vorangestellt werden. Eine Zahl ohne Vorzeichen wird stets als positiv angenommen.

Es ist möglich, mehrere ganze Zahlen, durch *Operatoren* wie +, -, * oder / zu einem *Term* zu verbinden. In diesem Fall antwortet der Interpreter mit dem Wert des Terms:

```
>>> 5 + 9
14
```

Wie Sie sehen, lässt sich Python ganz intuitiv als eine Art Taschenrechner verwenden. Das nächste Beispiel ist etwas komplexer und beinhaltet gleich mehrere miteinander verknüpfte Rechenoperationen:

```
>>> ((21 - 3) * 9 + 8) / 4
42
```

Hier zeigt sich, dass der Interpreter die gewohnten mathematischen Rechengesetze anwendet und das erwartete Ergebnis ausgibt.

Eine Besonderheit gibt es allerdings schon: Dividiert man beispielsweise 6 durch 4, gibt der Interpreter, möglicherweise entgegen Ihrer Erwartung, eine 1 aus:

```
>>> 6 / 4
1
```

Dieses Verhalten hängt damit zusammen, dass wir uns im Zahlenraum der ganzen Zahlen bewegen. Das Divisionsergebnis von *1,5* würde diesen Raum sprengen. Deshalb wird der Nachkommaanteil verworfen (*Integer-Division*). Die Integer-Division ist keineswegs ein Fehlverhalten des Interpreters, sondern eine nützliche und wichtige Operation. Wie Sie ein nicht gerundetes Ergebnis erhalten können, sehen Sie im nächsten Abschnitt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr

Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Willst du dich am Ganzen erquicken, so muss du das Ganze im Kleinsten erblicken.« – Johann Wolfgang von Goethe

5 Grundlegendes zu Python-Programmen

In diesem Kapitel werden wir einige Grundlagen zur Programmierung in Python erarbeiten. Insbesondere werden Sie Ihr erstes »richtiges« Python-Programm erstellen, das nicht mehr nur im interaktiven Modus läuft.

top

5.1 Grundstruktur eines Python-Programms

Das Wort *Syntax* kommt aus dem Griechischen und bedeutet »Satzbau«. Unter der Syntax einer Programmiersprache ist die vollständige Beschreibung erlaubter und verbotener Konstruktionen zu verstehen. Die Syntax wird durch eine Art Grammatik festgelegt, an die sich der Programmierer zu halten hat. Tut er es nicht, so verursacht er den allseits bekannten *Syntax Error*.

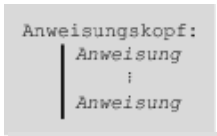
Um Ihnen ein Gefühl für die Sprache Python zu vermitteln, möchten wir zunächst einen Überblick über ihre Syntax geben. Dazu ist zu sagen, dass Python dem Programmierer sehr genaue Vorgaben macht, wie er seinen Quellcode zu strukturieren hat. Obwohl erfahrene Programmierer darin eine Einschränkung sehen mögen, kommt diese Eigenschaft gerade Programmierneulingen zugute, denn unstrukturierter und unübersichtlicher Code ist eine der größten Fehlerquellen in der Programmierung.

Grundsätzlich besteht ein Python-Programm aus einzelnen *Anweisungen*, die im einfachsten Fall genau eine Zeile im Quelltext einnehmen. Folgende Anweisung gibt beispielsweise einen Text auf dem Bildschirm aus:

```
print "Hallo Welt"
```

Einige Anweisungen lassen sich in einen *Anweisungskopf* und

einen *Anweisungskörper* unterteilen, wobei der Körper weitere Anweisungen enthalten kann:



Hier klicken, um das Bild zu vergrößern

Abbildung 5.1 Struktur einer mehrzeiligen Anweisung

Das könnte in einem konkreten Python-Programm etwa so aussehen:

```
if x > 10:
    print "Der Interpreter leistet gute Arbeit"
    print "Zweite Zeile!"
```

Die Zugehörigkeit des Körpers zum Kopf wird in Python durch einen Doppelpunkt am Ende des Anweisungskopfes und durch eine tiefere Einrückung des Anweisungskörpers festgelegt. Die Einrückung kann sowohl über Tabulatoren als auch über Leerzeichen erfolgen, wobei man gut beraten ist, beides nicht zu vermischen. Wir empfehlen eine Einrückungstiefe von jeweils vier Leerzeichen.

Python unterscheidet sich hier von vielen gängigen Programmiersprachen, in denen die Zuordnung von Anweisungskopf und Anweisungskörper durch geschweifte Klammern oder Schlüsselwörter wie »Begin« und »End« erreicht wird.

Achtung!

Ein Programm, in dem sowohl Leerzeichen als auch Tabulatoren verwendet wurden, kann vom Python-Interpreter anstandslos übersetzt werden, da jeder Tabulator intern durch acht Leerzeichen ersetzt wird. Dies kann aber zu schwer auffindbaren Fehlern führen, denn viele Editoren verwenden standardmäßig eine Tabulatorweite von vier Leerzeichen. Dadurch scheinen bestimmte Quellcodeabschnitte gleich weit eingerückt, obwohl sie es de facto nicht sind.

Bitte stellen Sie Ihren Editor so ein, dass jeder Tabulator automatisch durch Leerzeichen ersetzt wird, oder verwenden Sie ausschließlich Leerzeichen zur Einrückung Ihres Codes.

Möglicherweise fragen Sie sich jetzt, wie solche Anweisungen, die über mehrere Zeilen gehen, mit dem interaktiven Modus vereinbar sind, in dem ja immer nur eine Zeile bearbeitet werden kann. Nun, generell werden wir, wenn ein Codebeispiel mehrere Zeilen lang ist, nicht den interaktiven Modus verwenden. Dennoch ist die Frage berechtigt. Die Antwort: Es wird ganz intuitiv zeilenweise eingegeben. Wenn der Interpreter erkennt, dass eine Anweisung noch nicht vollendet ist, ändert er den Eingabeprompt von >>> in .... Geben wir einmal unser obiges Beispiel in den interaktiven Modus ein:

```
>>> x = 123
>>> if x > 10:
...     print "Der Interpreter leistet gute Arbeit"
...     print "Zweite Zeile!"
...
>>>
```

Beachten Sie, dass Sie, auch wenn eine Zeile mit ... beginnt, die aktuelle Einrückungstiefe berücksichtigen müssen. Das Ende des Anweisungskörpers kann der Interpreter nicht automatisch

erkennen, da er beliebig viele Anweisungen enthalten kann. Deswegen muss ein Anweisungskörper im interaktiven Modus durch Drücken der ↵ -Taste beendet werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»To iterate is human, to recurse divine« – L. Peter Deutsch

6 Kontrollstrukturen

Unter einer *Kontrollstruktur* versteht man ein Konstrukt, mit dessen Hilfe sich der Ablauf eines Programms steuern lässt. Dabei unterscheidet man in Python zwei Arten von Kontrollstrukturen: *Schleifen* und *Fallunterscheidungen*. Schleifen dienen dazu, einen Codeblock mehrmals auszuführen. Fallunterscheidungen hingegen knüpfen einen Codeblock an eine Bedingung, sodass er nur ausgeführt wird, wenn die Bedingung erfüllt ist. Wie und in welchem Umfang diese zwei Typen unterstützt werden, ist von Programmiersprache zu Programmiersprache verschieden. Python kennt jeweils zwei Unterarten, die wir hier behandeln werden.

Auch wenn das in den kommenden Beispielen noch nicht gezeigt wird, können Kontrollstrukturen beliebig ineinander verschachtelt werden. Die Einrückungstiefe wächst dabei kontinuierlich.

top

6.1 Fallunterscheidungen ▼

In Python gibt es zwei Arten von Fallunterscheidungen: die klassische `if`-*Anweisung* und die sogenannte *conditional expression* als erweiterte Möglichkeit der bedingten Ausführung von Code. Wir werden im Folgenden beide Arten der Fallunterscheidung detailliert besprechen und mit Beispielen erläutern. Dabei werden wir mit der `if`-Anweisung beginnen.

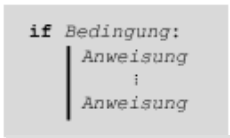
top

6.1.1 If, elif, else ▼▲

Die einfachste Möglichkeit der Fallunterscheidung ist die `if`-Anweisung. Eine `if`-Anweisung besteht aus einem Anweisungskopf, der eine Bedingung enthält, und aus einem Codeblock als Anweisungskörper (siehe Abbildung 6.1).

Der Codeblock wird nur ausgeführt, wenn sich die Bedingung als

wahr herausstellt. Die Bedingung einer `if`-Anweisung muss dabei ein Ausdruck sein, der nach seiner Auswertung einen Wahrheitswert (`True` oder `False`) ergibt. Typischerweise werden hier die logischen Ausdrücke angewendet, die in Abschnitt 4.5 eingeführt wurden.



Hier klicken, um das Bild zu vergrößern

Abbildung 6.1 Struktur einer if-Anweisung

Als Beispiel betrachten wir eine `if`-Anweisung, die einen entsprechenden Text nur dann ausgibt, wenn die Variable `x` den Wert `1` hat:

```
if x == 1:
    print "x hat den Wert 1"
```

Beachten Sie, dass für dieses und die folgenden Beispiele eine Variable `x` bereits existieren muss. Sollte dies nicht der Fall sein, so bekommen Sie einen `NameError`.

Selbstverständlich können auch andere vergleichende Operatoren oder ein komplexerer logischer Ausdruck verwendet werden und mehr als eine Anweisung im Körper stehen:

```
if (x <= 1) and (x * x > 20):
    print "x ist kleiner ..."
    print "...oder gleich 1"
```

In vielen Fällen ist es mit einer einzelnen `if`-Anweisung nicht getan, und man benötigt eine ganze Kette von Fallunterscheidungen. Wir möchten im nächsten Beispiel zwei unterschiedliche Strings ausgeben, je nachdem, ob `x == 1` oder `x == 2` gilt. Dazu wäre nach Ihrem bisherigen Kenntnisstand folgender Code notwendig:

```
if x == 1:
    print "x hat den Wert 1"
if x == 2:
    print "x hat den Wert 2"
```

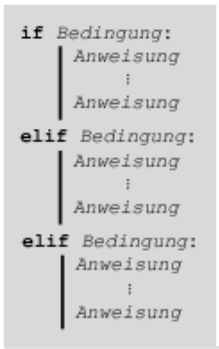
Dies ist aus Sicht des Interpreters eine ineffiziente Art, das Ziel zu erreichen, denn beide Bedingungen werden in jedem Fall ausgewertet und überprüft. Jedoch bräuchte die zweite Fallunterscheidung nicht mehr in Betracht gezogen zu werden, wenn die Bedingung der ersten bereits `True` ergeben hat. Die Variable `x` kann unter keinen Umständen sowohl den Wert `1` als auch `2` haben. Um solche Fälle aus Sicht des Interpreters performanter und aus Sicht des Programmierers übersichtlicher zu machen, kann eine `if`-Anweisung um einen oder mehrere sogenannte `elif`-Zweige (»*elif*« ist ein Kürzel für »*else if*«) erweitert werden.

Die Bedingung eines solchen Zweiges wird nur evaluiert, wenn alle vorherigen `if`- bzw. `elif`-Bedingungen `False` ergaben.

Das obige Beispiel kann mithilfe von `elif` folgendermaßen verfasst werden:

```
if x == 1:
    print "x hat den Wert 1"
elif x == 2:
    print "x hat den Wert 2"
```

Eine if-Anweisung kann um beliebig viele elif-Zweige erweitert werden:



Hier klicken, um das Bild zu vergrößern

Abbildung 6.2 Struktur einer if-Anweisung mit elif-Zweigen

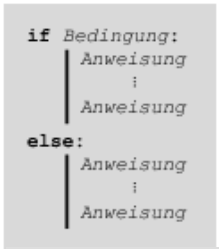
Im Quelltext könnte dies folgendermaßen aussehen:

```
if x == 1:
    print "x hat den Wert 1"
elif x == 2:
    print "x hat den Wert 2"
elif x == 3:
    print "x hat den Wert 3"
```

Als letzte Erweiterung der if-Anweisung ist es möglich, alle bisher unbehandelten Fälle auf einmal abzufangen. So möchten wir beispielsweise nicht nur einen entsprechenden String ausgeben, wenn x == 1 bzw. x == 2 gilt, sondern zusätzlich in allen anderen Fällen, also zum Beispiel x == 35, eine Fehlermeldung. Dazu kann eine if-Anweisung um einen sogenannten else-Zweig erweitert werden. Ist dieser vorhanden, so muss er an das Ende der if-Anweisung geschrieben werden (siehe Abbildung 6.3):

Konkret im Quelltext kann dies so aussehen:

```
if x == 1:
    print "x hat den Wert 1"
elif x == 2:
    print "x hat den Wert 2"
else:
    print "Fehler: Der Wert von x ist weder 1 noch 2"
```



Hier klicken, um das Bild zu vergrößern

Abbildung 6.3 Struktur einer if-Anweisung mit else-Zweig

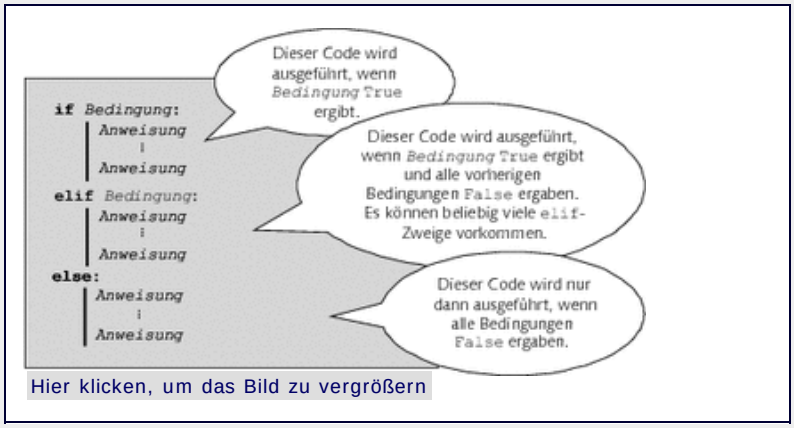
Der dem else-Zweig untergeordnete Codeblock wird nur dann ausgeführt, wenn alle vorherigen Bedingungen nicht erfüllt waren. Zu einer if-Anweisung darf maximal ein else-Zweig gehören. Im Beispiel wurde else in Kombination mit elif verwendet, was möglich, aber nicht zwingend ist.

Hinweis

Sollten Sie bereits eine Programmiersprache wie C oder Java beherrschen, so wird Sie interessieren, dass in Python kein

Pendant zur switch/case-Kontrollstruktur dieser Sprachen existiert. Das Verhalten dieser Kontrollstruktur kann trotzdem durch eine Kaskade von if/elif/else Zweigen nachgebildet werden.

Abschließend soll die folgende Grafik den Aufbau einer if-Anweisung noch einmal zusammenfassend und übersichtlich darstellen:



Hier klicken, um das Bild zu vergrößern

Abbildung 6.4 Aufbau einer if-Anweisung

top

6.1.2 Conditional expressions

Betrachten Sie, in Anlehnung an den vorherigen Abschnitt, einmal folgenden Code:

```
if x == 1:
    var = 20
else:
    var = 30
```

Es ist festzustellen, dass wir für einen geringfügigen Unterschied in der Zuweisung satte vier Zeilen Code benötigt haben, und es drängt sich die Frage auf, ob wir hier nicht mit Kanonen auf Spatzen schießen. Wir werden Ihnen jetzt zeigen, dass dieser Code mithilfe einer sogenannten *conditional expression* (dt. *bedingter Ausdruck*) in eine Zeile passt.

Ein solcher bedingter Ausdruck kann abhängig von einer Bedingung zwei verschiedene Werte annehmen. So ist es zum Beispiel möglich, var in derselben Zuweisung je nach Wert von x entweder auf 20 oder auf 30 zu setzen:

```
var = (20 if x == 1 else 30)
```

Die Klammern umschließen in diesem Fall den bedingten Ausdruck. Sie sind nicht notwendig, erhöhen aber die Übersicht. Der Aufbau einer *conditional expression* orientiert sich an der englischen Sprache und lautet folgendermaßen:

A **if** *Bedingung* **else** *B*

Sie nimmt dabei entweder den Wert *A* an, wenn die Bedingung erfüllt ist, oder andernfalls den Wert *B*. Sie könnten sich also vorstellen, dass die *conditional expression* nach dem Gleichheitszeichen entweder durch *A* oder *B*, also durch 20 oder 30 ersetzt wird. Nach der Auswertung des bedingten Ausdrucks ergibt sich also wieder eine gültige Zuweisung.

Diese Form, eine Anweisung an eine Bedingung zu knüpfen, kann selbstverständlich nicht nur auf Zuweisungen angewandt werden. Im folgenden Beispiel wird mit derselben print-Anweisung je nach Wert von x ein anderer String ausgegeben:

```
print ("x hat den Wert 1" if x == 1 else "x ist ungleich 1")
```

Beachten Sie, dass es sich bei *Bedingung* um einen logischen sowie bei *A* und *B* um einen beliebigen arithmetischen Ausdruck handeln kann. Eine *conditional expression* kann folglich auch so aussehen:

```
xyz = (a ** 2 if (a > 10 and b < 5) else b ** 2)
```

Dabei ist zu beachten, dass sich die Auswertungsreihenfolge der bedingten Ausdrücke von den normalen Auswertungsregeln von Python-Code unterscheidet. Es wird immer zunächst die Bedingung ausgewertet und erst dann, je nach Ergebnis, entweder der linke oder der rechte Teil des Ausdrucks. Eine solche springende Auswertungsreihenfolge wird *lazy evaluation* genannt.

Die hier vorgestellten *conditional expressions* können in der Praxis dazu verwendet werden, umständlichen und langen Code sehr elegant zu verkürzen. Allerdings geht all das stark auf Kosten der Lesbarkeit und Übersichtlichkeit. Wir werden deshalb in diesem Buch nur in Ausnahmefällen davon Gebrauch machen. Es steht Ihnen allerdings frei, *conditional expressions* in Ihren eigenen Projekten nach Herzenslust zu verwenden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»For every complex problem there is an answer that is clear, simple, and wrong.« – H. L. Mencken

7 Das Laufzeitmodell

Dieses Kapitel wird Ihnen vermitteln, wie Python Variablen zur Laufzeit verwaltet und welche Besonderheiten sich dadurch für den Programmierer ergeben.

Variablen sind Platzhalter für Werte wie Zahlen, Mengen oder sonstige Strukturen. Für die Programmierung ist der Begriff *Speicherstelle* eher zutreffend, da hier Variablen vor allem den Zweck erfüllen, Daten für ihre Weiterverwendung zwischenzuspeichern. Wie Sie bereits wissen, kann in Python eine neue Variable mit dem Namen a wie folgt angelegt werden:

```
>>> a = 1337
```

Anschließend kann der Platzhalter a wie der Zahlenwert 1337 benutzt werden:

```
>>> 2674 / a
2
```

Um zu verstehen, was intern passiert, wenn wir eine neue Variable erzeugen, müssen zwei Begriffe voneinander abgegrenzt werden: *Referenz* und *Instanz*. Eine *Instanz* ist ein konkretes Datenobjekt im Speicher, das nach der Vorlage eines bestimmten Datentyps erzeugt wurde – zum Beispiel die spezielle Zahl 1337 aus der Kategorie der Ganzzahlen.

Im Folgenden betrachten wir der Einfachheit halber nur Ganzzahlen und Strings – das Prinzip gilt aber für beliebige Datenobjekte.

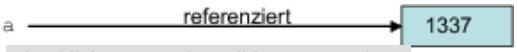
Im einfachsten Fall lässt sich eine Instanz einer Ganzzahl folgendermaßen anlegen:

```
>>> 12345
12345
```

Für uns als Programmierer ist diese Instanz allerdings wenig praktisch, da sie zwar nach ihrer Erzeugung ausgegeben wird, dann aber nicht mehr zugänglich ist und wir so ihren Wert nicht weiterverwenden können.

An dieser Stelle kommen die *Referenzen* ins Spiel. »Referenz« bedeutet so viel wie »Verweis«. Erst durch Referenzen wird es möglich, mit den Instanzen zu arbeiten, weil Referenzen uns den Zugriff auf diese ermöglichen. Die einfachste Form einer Referenz in Python ist ein *symbolischer Name* wie im obigen Beispiel a. Mit dem Zuweisungsoperator = kann man eine Referenz auf eine Instanz erzeugen, wobei die Referenz links und die Instanz rechts vom Operator stehen.

Damit können wir unser Beispiel wie folgt beschreiben: Wir erzeugen eine neue Instanz einer Ganzzahl mit dem Wert 1337. Außerdem legen wir eine Referenz `variable` auf diese Instanz an. Dies lässt sich auch grafisch verdeutlichen:



Hier klicken, um das Bild zu vergrößern

Abbildung 7.1 Schema der Referenz-Instanz-Beziehung

Es ist auch möglich, bereits referenzierte Instanzen mit weiteren Referenzen zu versehen:

```
>>> referenz1 = 1337
>>> referenz2 = referenz1
```

Grafisch veranschaulicht sieht das Ergebnis so aus:

Hier klicken, um das Bild zu vergrößern

Abbildung 7.2 Zwei Referenzen auf dieselbe Instanz

Besonders wichtig ist hierbei, dass es nach wie vor nur eine Instanz mit dem Wert 1337 im Speicher gibt, obwohl wir mit zwei verschiedenen Namen `referenz1` und `referenz2` darauf zugreifen können. Durch die Zuweisung `referenz2 = referenz1` wurde also nicht die Instanz 1337 kopiert, sondern nur ein weiteres Mal referenziert.

Bitte beachten Sie, dass Referenzen auf dieselbe Instanz untereinander unabhängig sind und sich der Wert, auf den die anderen Referenzen verweisen, nicht ändert, wenn wir einer von ihnen eine neue Instanz zuweisen:

```
>>> referenz1 = 1337
>>> referenz2 = referenz1
>>> referenz1
1337
>>> referenz2
1337
>>> referenz1 = 2674
>>> referenz1
2674
>>> referenz2
1337
```

Bis zu den ersten beiden Ausgaben haben wir die in Abbildung 7.2 veranschaulichte Situation: Die beiden Referenzen `referenz1` und `referenz2` verweisen auf dieselbe Instanz 1337. Anschließend erzeugen wir eine neue Instanz 2674 und weisen sie `referenz1` zu. Die Ausgabe zeigt, dass `referenz2` nach wie vor auf 1337 zeigt und nicht verändert wurde. Die Situation nach der dritten Zuweisung sieht also so aus:

Hier klicken, um das Bild zu vergrößern

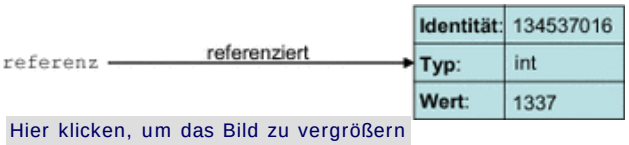
Abbildung 7.3 Die beiden Referenzen sind voneinander unabhängig.

Da Sie nun wissen, was Referenzen und Instanzen sind und wie sie im Programm verwendet werden, beschäftigen wir uns nun mit den Eigenschaften von Instanzen im Detail.

top

7.1 Die Struktur von Instanzen

Jede Instanz in Python umfasst drei Merkmale: ihren *Datentyp*, ihren *Wert* und ihre *Identität*. Unser Eingangsbeispiel könnte man sich folgendermaßen dreigeteilt vorstellen:



Hier klicken, um das Bild zu vergrößern

Abbildung 7.4 Eine Instanz mit ihren drei Eigenschaften

Datentyp

Der Datentyp dient bei der Erzeugung der Instanz als Bauplan und legt fest, welche Werte die Instanz annehmen darf. So erlaubt der Datentyp `int` beispielsweise das Speichern einer ganzen Zahl. Strings lassen sich mit dem Datentyp `str` verwalten. Im folgenden Beispiel wird gezeigt, wie sich die Datentypen verschiedener Instanzen mithilfe von `type` herausfinden lassen: [Bei `type` handelt es sich um eine sogenannte Funktion. Was genau das bedeutet, ist an dieser Stelle noch nicht wichtig. Wir werden uns in Kapitel 10 eingehend mit Funktionen beschäftigen und dort auch auf `type` zurückkommen.]

```
>>> type(1337)
<type 'int'>
>>> type("Hallo Welt")
<type 'str'>
>>> v1 = 2674
>>> type(v1)
<type 'int'>
```

Die Funktion `type` ist unter anderem dann nützlich, wenn wir überprüfen wollen, ob zwei Instanzen den gleichen Typ besitzen oder ob eine Instanz einen bestimmten Typ hat:

```
>>> v1 = 1337
>>> type(v1) == type(2674)
True
>>> type(v1) == int
True
```

Hierbei ist zu beachten, dass sich ein Typ nur auf Instanzen bezieht und rein gar nichts mit den verknüpften Referenzen zu tun hat. Eine Referenz hat keinen Typ und kann Instanzen beliebiger Typen referenzieren. Folgendes ist durchaus möglich:

```
>>> zuerst_ein_string = "Ich bin ein String"
>>> type(zuerst_ein_string)
<type 'str'>
>>> zuerst_ein_string = 1789
>>> type(zuerst_ein_string)
<type 'int'>
```

Es ist also falsch, zu sagen »`zuerst_ein_string` hat den Typ `str`«. Korrekt ist: »`zuerst ein string` referenziert momentan

eine Instanz des Typs str«.

Wert

Was den Wert der Instanz konkret ausmacht, hängt von ihrem Typ ab. Dies können beispielsweise Zahlen, Zeichenketten oder Daten anderer Typen sein, die Sie später noch kennenlernen werden. In den obigen Beispielen waren es 1337, 2674, 1798, "Hallo Welt" und "Ich bin ein String".

Mit dem Operator == kann man Instanzen bezüglich ihres Wertes vergleichen:

```
>>> v1 = 1337
>>> v2 = 1337
>>> v1 == v2
True
>>> v1 == 2674
False
```

Mithilfe unseres grafischen Modells kann man sich die Arbeitsweise des Operators == gut veranschaulichen:

Hier klicken, um das Bild zu vergrößern

Abbildung 7.5 Wertevergleich zweier Instanzen (in diesem Fall False)

Der Wertevergleich ist nur dann sinnvoll, wenn er sich auf strukturell ähnliche Datentypen bezieht, wie zum Beispiel Ganzzahlen und Gleitkommazahlen:

```
>>> gleitkommazahl = 1987.0
>>> type(gleitkommazahl)
<type 'float'>
>>> ganzzahl = 1987
>>> type(ganzzahl)
<type 'int'>
>>> gleitkommazahl == ganzzahl
True
```

Obwohl gleitkommazahl und ganzzahl verschiedene Typen haben, liefert der Vergleich mit == den Wahrheitswert True.

Zahlen und Zeichenketten haben strukturell wenig gemeinsam, da es sich bei Zahlen um einzelne Werte handelt, während bei Zeichenketten mehrere Buchstaben zu einer Einheit zusammengefasst werden. Aus diesem Grund liefert der Operator == für den Vergleich zwischen Strings und Zahlen immer False, auch wenn die Werte für einen Menschen gleich aussehen:

```
>>> string = "1234"
>>> string == 1234
False
```

Ob der Operator == für zwei bestimmte Typen definiert ist, hängt von den Datentypen selbst ab. Ist er nicht vorhanden, wird die Indentität der Instanzen zum Vergleich herangezogen, was im folgenden Absatz erläutert wird.

Identität

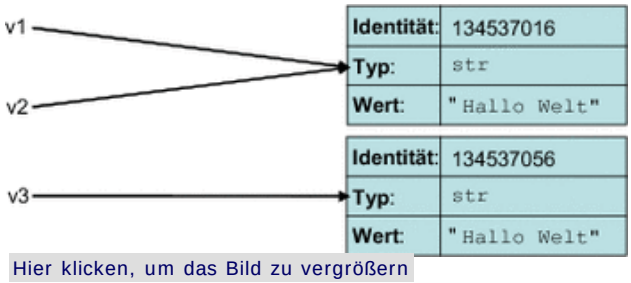
Die *Identität* einer Instanz dient dazu, sie von allen anderen Instanzen zu unterscheiden. Sie ist mit dem individuellen Fingerabdruck eines Menschen vergleichbar, da sie für jede Instanz programmweit eindeutig ist und sich nicht ändern kann. Eine Identität ist eine Ganzzahl und lässt sich mithilfe der Funktion id ermitteln:

```
>>> id(1337)
```

```
134537016
>>> v1 = "Hallo Welt"
>>> id(v1)
3082572528
```

Identitäten werden immer dann wichtig, wenn man prüfen möchte, ob es sich um eine ganz bestimmte Instanz handelt und nicht nur um eine mit dem gleichen Typ und Wert:

```
>>> v1 = "Hallo Welt"
>>> v2 = v1
>>> v3 = "Hallo Welt"
>>> type(v1) == type(v3)
True
>>> v1 == v3
True
>>> id(v1) == id(v3)
False
>>> id(v1) == id(v2)
True
```



Hier klicken, um das Bild zu vergrößern

Abbildung 7.6 Drei Referenzen, zwei Instanzen

In diesem Beispiel hat Python zwei verschiedene Instanzen mit dem Typ `str` und dem Wert `"Hallo Welt"` angelegt, wobei `v1` und `v2` auf dieselbe Instanz verweisen. Grafisch veranschaulicht bedeutet dies Folgendes (siehe Abbildung 7.6).

Der Vergleich auf Identitätengleichheit hat in Python eine so große Bedeutung, dass für diesen Zweck ein eigener Operator definiert wurde: `is`.

Der Ausdruck `id(referenz1) == id(referenz2)` bedeutet das Gleiche wie `referenz1 is referenz2`. Dies kann man sich so vorstellen:

Hier klicken, um das Bild zu vergrößern

Abbildung 7.7 Identitätenvergleich zweier Instanzen

Der in Abbildung 7.7 gezeigte Vergleich würde den Wahrheitswert `False` ergeben, da sich die Identitäten der beiden Instanzen unterscheiden.

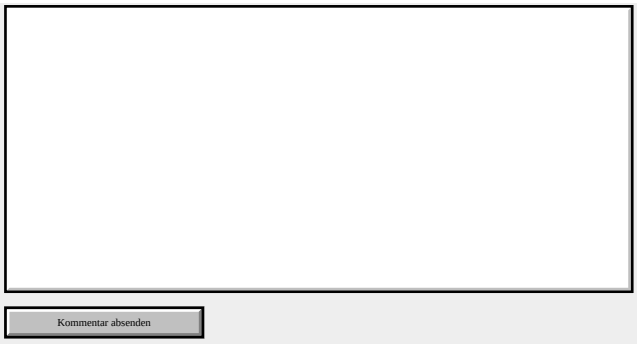
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Alles ist Zahl.« – Pythagoras

8 Basisdatentypen

Im vorherigen Kapitel haben wir unter anderem besprochen, was ein Datentyp ist. Hier möchten wir näher beleuchten, welche Datentypen es gibt und wie sie verwendet werden können. Bislang wurden nur einfache Datentypen erwähnt, die beispielsweise eine Zahl oder einen Wahrheitswert aufnehmen können. Darüber hinaus existieren auch sehr komplexe Datentypen, die eine Liste oder Zuordnung verschiedenster Daten speichern und Operationen anbieten, um diese komfortabel zu verarbeiten.

Python definiert dabei eine Reihe von sogenannten *Basisdatentypen*. Das sind »eingebaute« Typen, die dem Programmierer zu jeder Zeit zur Verfügung stehen. Dabei wird allgemein zwischen *numerischen Datentypen*, *sequenziellen Datentypen*, *assoziativen Datentypen* und *Mengen* unterschieden.

Bevor wir uns mit den Datentypen selbst befassen, werden Sie im folgenden Abschnitt umfassend in die Thematik der Operatoren eingeführt.

top

8.1 Operatoren

Den Begriff des Operators kennen Sie aus der Mathematik, wo er ein Formelzeichen beschreibt, das für eine bestimmte

Rechenoperation steht. In Python können Sie Operatoren beispielsweise verwenden, um zwei numerische Werte zu einem arithmetischen Ausdruck zu verbinden:

```
>>> 1 + 2
3
```

Die Werte, auf denen ein Operator angewendet wird, also in diesem Fall 1 und 2, werden *Operanden* genannt. Auch für andere Datentypen gibt es Operatoren. So kann + etwa auch zwei Strings zusammenfügen:

```
>>> "A" + "B"
'AB'
```

In Python hängt die Bedeutung eines Operators also davon ab, auf welchen Datentyp er angewendet wird. Wir werden uns in diesem Abschnitt auf die Operatoren +, -, * und < beschränken, da diese ausreichen, um das dahinter liegende Prinzip zu erklären. In den folgenden Beispielen kommen immer wieder die drei Referenzen a, b und c vor, die in den Beispielen selbst nicht angelegt werden. Um die Beispiele ausführen zu können, müssen die Referenzen natürlich existieren und beispielsweise je eine ganze Zahl referenzieren.

Betrachten Sie einmal folgende Ausdrücke:

```
(a * b) + c
a * (b + c)
```

Beide sind in ihrer Bedeutung eindeutig, da durch die Klammern angezeigt wird, welcher Teil des Ausdrucks zuerst ausgewertet werden soll. Doch schon bei etwas komplexeren Ausdrücken fällt auf, dass es unpraktikabel ist, die Eindeutigkeit eines Ausdruckes allein durch Klammern erwirken zu wollen. Betrachten wir also einmal den obigen Ausdruck ohne Klammern:

```
a * b + c
```

Nun ist nicht mehr ersichtlich, welcher Teil des Ausdrucks zuerst ausgewertet werden soll. Doch eine Regelung ist hier unerlässlich, denn je nach Auswertungsreihenfolge kommen unterschiedliche Ergebnisse heraus. Um dieses Problem zu lösen, haben Operatoren in Python, wie in der Mathematik auch, eine *Bindigkeit*. Diese ist so definiert, dass * stärker bindet als +, es gilt also »Punktrechnung vor Strichrechnung«. Es gibt in Python eine sogenannte *Operatorrangfolge*, die definiert, welcher Operator wie stark bindet, und somit einem klammernlosen Ausdruck eine eindeutige Auswertungsreihenfolge und damit einen eindeutigen Wert vermittelt.

Sie finden die Operatorrangfolge in Form einer Tabelle im Anhang dieses Buchs.

Damit wäre die Auswertung eines Ausdrucks, der aus Operatoren verschiedener Bindigkeit besteht, gesichert. Doch wie sieht es aus, wenn der gleiche Operator mehrmals im Ausdruck vorkommt? Einen Unterschied in der Bindigkeit kann es dann ja nicht mehr geben. Betrachten Sie dazu folgende Ausdrücke:

```
a + b + c
a - b - c
```

In beiden Fällen ist die Auswertungsreihenfolge weder durch Klammern noch durch die Operatorrangfolge eindeutig geklärt. Sie sehen, dass dies für die Auswertung des ersten Ausdrucks zwar kein Problem darstellt, doch spätestens beim zweiten Ausdruck ist eine Regelung vonnöten, da je nach Auswertungsreihenfolge zwei

verschiedene Ergebnisse möglich sind. In einem solchen Fall gilt in Python die Regelung, dass Ausdrücke oder Teilausdrücke, die nur aus Operatoren gleicher Bindigkeit bestehen, *von links nach rechts* ausgewertet werden.

Wir haben bisher nur über Operatoren gesprochen, die als Ergebnis wieder einen Wert vom Typ der Operanden liefern. So ist das Ergebnis einer Addition zweier ganzer Zahlen stets wieder eine ganze Zahl. Dies ist jedoch nicht für jeden Operator der Fall. Sie kennen bereits die Vergleichsoperatoren, die, unabhängig vom Datentyp der Operanden, einen Wahrheitswert ergeben. Denken Sie also einmal über die Auswertungsreihenfolge dieses Ausdrucks nach:

```
a < b < c
```

Theoretisch wäre es möglich, und es wird in einigen Programmiersprachen auch so gemacht, nach dem oben besprochenen Schema zu verfahren: Die Vergleichskette soll von links nach rechts ausgewertet werden. In diesem Fall würde zuerst `a < b` ausgewertet und `True` ergeben. Im nächsten Vergleich wäre dann `True < c`. Eine solche Form der Auswertung ist zwar möglich, hat jedoch keinen praktischen Nutzen, denn was soll `True < c` genau bedeuten?

In Python werden solche Operatoren gesondert behandelt. Der Ausdruck `a < b < c` wird so ausgewertet, dass er äquivalent zu

```
a < b and b < c
```

ist. Das entspricht der mathematischen Sichtweise, denn der Ausdruck bedeutet tatsächlich »liegt b zwischen a und c?«. Als zweites, etwas komplexeres Beispiel wird der Ausdruck

```
a < b <= c != d > e
```

ausgewertet zu:

```
a < b and b <= c and c != d and d > e
```

Dieses Verhalten trifft auf folgende Operatoren zu: `<`, `<=`, `>`, `>=`, `==`, `!=`, `<>`, `is`, `is not`, `in` und `not in`.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»I have always wished that my computer would be as easy to use as my telephone. My wish has come true. I no longer know how to use my telephone.« – Bjarne Stroustrup

9 Benutzerinteraktion und Dateizugriff

Nachdem wir Sie in die grundlegenden Sprachelemente von Python eingeführt haben, wartet hier das erste praxisorientierte Kapitel auf Sie. Bisher können Sie Instanzen diverser Datentypen erstellen und mit ihnen arbeiten. Darüber hinaus wissen Sie bereits, wie der Programmfluss durch Kontrollstrukturen beeinflusst werden kann. Es ist an der Zeit, all dieses Wissen sinnvoll zu verwenden und Sie in die Lage zu versetzen, komplexere Programme zu schreiben. Dieses Kapitel widmet sich zunächst einmal der Interaktion des Programms mit dem Benutzer, also demjenigen, der Ihr Programm auf seinem Rechner startet. Dazu unterscheiden wir zwei Teilbereiche: Textausgaben auf dem Bildschirm und Eingaben von der Tastatur. Zudem soll in diesem Kapitel das Lesen und Schreiben von Dateien behandelt werden. Wir bleiben, wie bisher, bei einer Konsolenanwendung.

top

9.1 Bildschirmausgaben

Wie Sie bereits wissen, wird ein Text auf dem Bildschirm mithilfe des Schlüsselwortes `print` ausgegeben. Etwa so:

```
print "Text"
```

`print` ist keineswegs auf die Ausgabe von Strings beschränkt. Vielmehr kann jedes beliebige Objekt in einer `print`-Anweisung verwendet werden, so zum Beispiel ein Dictionary:

```
print {"bla" : 19, "blubb": [1,2,3,4]}
```

Dies würde folgende Ausgabe zur Folge haben:

```
{'bla': 19, 'blubb': [1, 2, 3, 4]}
```

Des Weiteren kann print nicht nur einen einzelnen Wert ausgeben, sondern beliebig viele. Sie werden, durch Kommata getrennt, hinter das Schlüsselwort geschrieben. Bei der Ausgabe erscheint für jedes Komma ein zusätzliches Leerzeichen an der entsprechenden Stelle:

```
print "Die magischen Zahlen sind:", 15, "und", 9
```

In diesem Fall würde Folgendes auf dem Bildschirm erscheinen:

```
Die magischen Zahlen sind: 15 und 9
```

Da print eine Ausgabe standardmäßig damit beendet, in die nächste Zeile zu springen, lässt es sich ohne Argumente zur Ausgabe einer leeren Zeile verwenden:

print

Um den Wechsel in eine neue Zeile zu verhindern, muss ein Komma am Ende der Parameterliste geschrieben werden:

```
print "abc",
print "def"
```

Die Ausgabe der beiden Anweisungen ist abcdef. Der Unterschied wird deutlich, wenn das Komma weggelassen wird:

```
print "abc"
print "def"
```

Die Ausgabe lautet jetzt:

```
abc
def
```

Eine Kleinigkeit noch, bevor es weitergeht: Das Schlüsselwort print erlaubt es, Ausgaben in eine beliebige Datei umzuleiten. Dazu werden zwei schließende spitze Klammern hinter das Schlüsselwort geschrieben, gefolgt von einem Dateiobjekt und der normalen Parameterliste:

```
print >> dateiobj, "Hallo Welt"
```

Was genau ein Dateiobjekt ist und wie man damit umgeht, wird in Abschnitt 9.3 beschrieben.

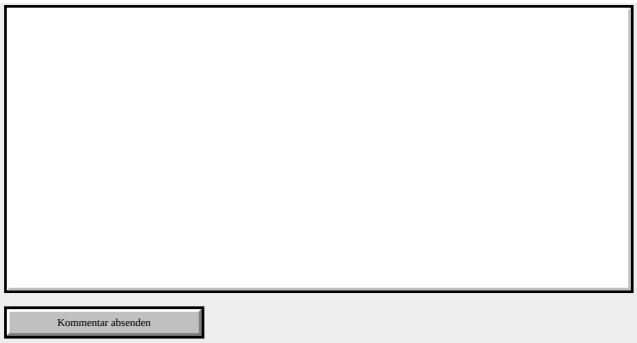
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Um Rekursion zu verstehen, muss man zunächst einmal Rekursion verstehen.« – Unbekannter Autor.

10 Funktionen

Wenn Sie mit dem Wissen, das wir Ihnen bisher über die Programmiersprache Python vermittelt haben, ein größeres Programm schreiben wollten, so wäre dies womöglich zum Scheitern verurteilt, da die Les- und Wartbarkeit unserer bisherigen Beispielquelltexte mit zunehmender Größe rapide abnehmen würde. Es ist daher ein erstrebenswertes Ziel, den Quelltext so übersichtlich und aufgeräumt zu gestalten, dass man sich selbst nach langen Programmierpausen problemlos wieder hineinlesen kann.

Ein zweites, viel gravierenderes Problem stellen Redundanzen im Code dar. In größeren Quelltexten gibt es eine Menge Operationen, die an unterschiedlichen Stellen genau so oder in ähnlicher Form immer wieder durchgeführt werden müssen. Aus Mangel an Alternativen würden Sie diese immer wieder genau da implementieren, wo sie gebraucht werden würden. Sie können sich sicherlich vorstellen, dass ein solcher Quelltext kein Paradebeispiel für sauberen Code darstellen würde.

Python ist, wie viele andere Programmiersprachen auch, eine *funktionale Sprache.* [Beachten Sie, dass es einen Unterschied zwischen *funktionalen* und *rein funktionalen* Programmiersprachen gibt. Als Vertreter rein funktionaler Programmiersprachen kann beispielsweise Haskell angesehen werden.] Das bedeutet, dass Ihnen ein Hilfsmittel zur Seite gestellt wird, mit dem Sie Ihr Programm in Unterprogramme unterteilen können. Ein solches Unterprogramm wird *Funktion* genannt. Dadurch wird das Problem der dramatisch abnehmenden Übersichtlichkeit gelöst, denn Funktionen ermöglichen es Ihnen, gewisse Teile des Quellcodes zu kapseln, zu gruppieren oder von anderen Teilen abzugrenzen. Des Weiteren kann eine Funktion an beliebigen Stellen des Quellcodes beliebig oft aufgerufen werden, was es dem Programmierer in der

Regel ermöglicht, Quellcode ohne Codedopplungen zu schreiben.

Damit eine Funktion korrekt arbeiten kann, müssen bei ihrem Aufruf möglicherweise Informationen übertragen werden. So sollte eine Funktion, die beispielsweise die Fakultät einer ganzen Zahl berechnet, wissen, von welcher Zahl die Fakultät zu berechnen ist. Dazu können beim Aufruf sogenannte *Parameter* übergeben werden. Zudem sollte eine Funktion dem aufrufenden, übergeordneten Programm das Ergebnis der Berechnung mitteilen können. Dazu verfügt jede Funktion über einen sogenannten *Rückgabewert*.

Sie haben, möglicherweise ohne das zu bemerken, bereits mit Funktionen gearbeitet: bei der Verwendung von `len` und `range` zum Beispiel. Im Folgenden möchten wir die Handhabung einer bestehenden Funktion am Beispiel von `range` erläutern.

Die eingebaute Funktion `range` wurde in Abschnitt 6.2.4 zum Steuern einer `for`-Schleife eingesetzt. Dort wurde sie in ihrer Bedeutung jedoch sehr reduziert dargestellt, denn eigentlich erzeugt `range` eine Liste mit einer begrenzten Anzahl von fortlaufenden, numerischen Elementen. Sie kann also durchaus ohne korrespondierende `for`-Schleife verwendet werden:

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

Im obigen Beispiel wurde `range` aufgerufen, man nennt dies den *Funktionsaufruf*. Dazu wird hinter den Namen der Funktion ein (möglicherweise leeres) Klammernpaar geschrieben. Innerhalb dieser Klammern stehen, durch Kommata getrennt, die Parameter der Funktion. Wie viele es sind und welche Art von Parametern eine Funktion erwartet, hängt von ihrer Definition ab und ist sehr verschieden. In diesem Fall benötigt `range` drei Parameter, um ausreichend Informationen zu erlangen. Die Gesamtheit der Parameter wird *Funktionsschnittstelle* genannt. Konkrete, über eine Schnittstelle übergebene Instanzen werden *Argumente* genannt. Ein Parameter hingegen bezeichnet einen Platzhalter für Argumente.

Nachdem die Funktion abgearbeitet wurde, wird ihr Ergebnis, im Falle von `range` die erzeugte Liste, als Rückgabewert zurückgegeben. Sie können sich bildlich vorstellen, dass der Funktionsaufruf, wie er im Quelltext steht, durch den Rückgabewert ersetzt wird. Im obigen Beispiel wurde der Rückgabewert automatisch vom Interpreter ausgegeben. Es ist aber auch möglich, das Ergebnis zu referenzieren und weiterzuverwenden:

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

So viel vorerst zur Verwendung von vordefinierten Funktionen. Python erlaubt es Ihnen, eigene Funktionen zu schreiben, die nach demselben Schema verwendet werden können, wie es hier beschrieben wurde. Im nächsten Abschnitt werden wir uns ausführlich damit befassen, wie eine eigene Funktion erstellt werden kann.

top

10.1 Schreiben einer Funktion

Bevor wir uns an konkreten Quelltext wagen, möchten wir rekapitulieren, was eine Funktion ausmacht, was also bei der Definition einer Funktion anzugeben wäre:

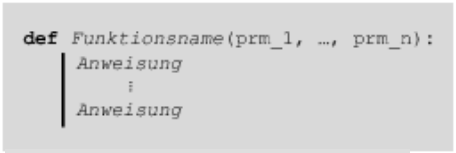
- Eine Funktion muss einen Namen haben, über den sie in anderen Teilen des Programms aufgerufen werden kann. Die Zusammensetzung des Funktionsnamens erfolgt nach

denselben Regeln wie die Namensgebung einer Referenz.

- Eine Funktion muss eine Schnittstelle haben, über die Informationen vom aufrufenden Programmteil in den Kontext der Funktion übertragen werden können. Eine Schnittstelle kann aus beliebig vielen (unter Umständen auch keinen) Parametern bestehen. Funktionsintern wird jedem dieser Parameter ein Name gegeben. Sie lassen sich dann wie Referenzen im Funktionskörper verwenden.

- Eine Funktion muss einen Wert zurückgeben. Jede Funktion gibt automatisch None zurück, wenn der Rückgabewert nicht ausdrücklich angegeben wurde.

Zur Definition einer Funktion wird in Python das Schlüsselwort def verwendet. Syntaktisch sieht die Definition folgendermaßen aus:



Hier klicken, um das Bild zu vergrößern

Abbildung 10.1 Definition einer Funktion

Nach dem Schlüsselwort def steht der gewählte Funktionsname. Hinter diesem werden in einem Klammernpaar die Namen aller Parameter aufgelistet. Nach der Definition der Schnittstelle folgen ein Doppelpunkt und, eine Stufe weiter eingerückt, der Funktionskörper. Bei dem Funktionskörper handelt es sich um einen beliebigen Codeblock, in dem die Parameternamen als Referenzen verwendet werden dürfen. Im Funktionskörper dürfen auch wieder Funktionen aufgerufen werden.

Betrachten wir einmal die konkrete Implementierung einer Funktion, die die Fakultät einer ganzen Zahl berechnet und das Ergebnis auf dem Bildschirm ausgibt:

```
def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    print ergebnis
```

Anhand dieses Beispiels können Sie sehr gut nachvollziehen, wie der Parameter zahl im Funktionskörper verarbeitet wird. Nachdem die Berechnung vollzogen ist, wird ergebnis mittels print ausgegeben. Beachten Sie, dass die Referenz zahl nur innerhalb des Funktionskörpers definiert ist und nichts mit anderen Referenzen außerhalb der Funktion zu tun hat.

Wenn Sie das obige Beispiel jetzt speichern und ausführen, werden Sie feststellen, dass zwar keine Fehlermeldung angezeigt wird, aber auch sonst nichts passiert. Nun, das liegt daran, dass wir bisher nur eine Funktion definiert haben. Um sie konkret im Einsatz zu sehen, müssen wir sie mindestens einmal aufrufen. Folgendes Programm liest in einer Schleife Zahlen vom Benutzer ein und berechnet deren Fakultät:

```
def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    print ergebnis

while True:
    eingabe = int(raw_input("Geben Sie eine Zahl ein: "))
    fak(eingabe)
```

Sie sehen, dass der Quellcode sehr schön in zwei Komponenten aufgeteilt wurde: zum einen in die Funktionsdefinition oben und zum anderen in das auszuführende Hauptprogramm unten. Das

Hauptprogramm besteht aus einer Endlosschleife, in der im Wesentlichen die Funktion fak mit der eingegebenen Zahl als Parameter aufgerufen wird.

Betrachten Sie noch einmal die beiden Komponenten des Programms. Es wäre erstrebenswert, das Programm so zu ändern, dass sich das Hauptprogramm allein um die Interaktion mit dem Benutzer und das Anstoßen der Berechnung kümmert, während das Unterprogramm fak die Berechnung tatsächlich durchführt. Das Ziel dieses Ansatzes ist es vor allem, dass die Funktion fak auch in anderen Programmteilen zur Berechnung einer weiteren Fakultät aufgerufen werden kann. Dazu ist es unerlässlich, dass fak sich ausschließlich um die Berechnung kümmert. Es passt nicht wirklich in dieses Konzept, dass fak das Ergebnis der Berechnung selbst ausgibt.

Idealerweise sollte unsere Funktion fak die Berechnung abschließen und das Ergebnis an das Hauptprogramm zurückgeben, sodass die Ausgabe dort erfolgen kann. Dies kann durch das Schlüsselwort return erreicht werden, das die Ausführung der Funktion sofort beendet und einen eventuell angegebenen Rückgabewert zurückgibt.

```
def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    return ergebnis

while True:
    eingabe = int(raw_input("Geben Sie eine Zahl ein: "))
    print fak(eingabe)
```

Das Beenden einer Funktion mit return kann zu jeder Zeit im Funktionsablauf geschehen. Folgende Version der Funktion prüft vor der Berechnung, ob es sich bei dem übergebenen Parameter um eine negative Zahl handelt. Ist das der Fall, so wird die Abhandlung der Funktion sofort abgebrochen:

```
def fak(zahl):
    if zahl < 0:
        return None
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    return ergebnis

while True:
    eingabe = int(raw_input("Geben Sie eine Zahl ein: "))
    ergebnis = fak(eingabe)
    if ergebnis is None:
        print "Fehler bei der Berechnung"
    else:
        print ergebnis
```

In der zweiten Zeile des Funktionskörpers wurde mit return None explizit der Wert None zurückgegeben. Das ist nicht unbedingt nötig; folgender Code wäre äquivalent:

```
if zahl < 0:
    return
```

Vom Programmablauf her ist es egal, ob Sie None explizit oder implizit zurückgeben. Aus Gründen der Lesbarkeit ist return None in diesem Fall trotzdem sinnvoll, denn es handelt sich um einen ausdrücklich gewünschten Rückgabewert. Er ist Teil der Funktionslogik und nicht bloß ein Nebenprodukt, das beim Funktionsabbruch entsteht.

Die Funktion fak, wie sie in diesem Beispiel zu sehen ist, kann zu jeder Zeit zur Berechnung einer Fakultät aufgerufen werden, unabhängig davon, in welchem Kontext diese Fakultät benötigt wird.

Selbstverständlich können Sie in Ihrem Quelltext mehrere eigene

Funktionen definieren und aufrufen. Das folgende Beispiel soll bei Eingabe einer negativen Zahl keine Fehlermeldung, sondern die Fakultät des Betrages dieser Zahl ausgeben:

```
def betrag(zahl):
    if zahl < 0:
        return -zahl
    else:
        return zahl

def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    return ergebnis

while True:
    eingabe = int(raw_input("Geben Sie eine Zahl ein: "))
    print fak(betrag(eingabe))
```

Für die Berechnung des Betrags einer Zahl gibt es in Python auch die Built-in Function abs. Diese werden wir noch in diesem Kapitel besprechen.

Ein Begriff soll noch eingeführt werden, bevor wir uns den Funktionsparametern widmen. Eine Funktion kann über ihren Namen nicht nur aufgerufen, sondern auch wie eine Instanz behandelt werden. So ist es beispielsweise möglich, den Typ einer Funktion abzufragen. Die folgenden Beispiele nehmen an, dass die Funktion fak im interaktiven Modus verfügbar ist:

```
>>> type(fak)
<type 'function'>
>>> p = fak
>>> p(5)
120
>>> fak(5)
120
```

Der Name der Funktion, in diesem Fall fak, wird aufgrund dieser Eigenschaften auch *Funktionsobjekt* genannt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Divide et impera!« – Julius Caesar

11 Modularisierung

Unter Modularisierung versteht man die Aufteilung des Quelltextes in einzelne Teile, sogenannte *Module*. Grundsätzlich gibt es zwei Arten von Modulen:

Zum einen kann jedes Python-Programm sogenannte *Bibliotheken* (engl. *Libraries*) einbinden. Eine Bibliothek dient häufig einem ganz bestimmten Zweck, wie etwa der Arbeit mit Dateien eines bestimmten Dateiformats, und stellt üblicherweise Datentypen oder Funktionen bereit, die nach dem Einbinden verwendet werden können. Es ist möglich, eigene Bibliotheken zu schreiben oder eine Bibliothek eines Drittanbieters zu installieren. Ein gutes Argument für Python ist die umfangreiche *Standardbibliothek*, die im Lieferumfang enthalten ist. Sie bietet eine hohe Grundfunktionalität, die in jeder Python-Umgebung verfügbar ist.

Die zweite Möglichkeit zur Modularisierung sind lokale Module. Darunter versteht man die Kapselung einzelner Programmteile – auch hier üblicherweise Datentypen oder Funktionen – in eigene Programmdateien. Diese Dateien können wie Bibliotheken eingebunden werden, sind aber in keinem anderen Python-Programm verfügbar. Diese Form der Modularisierung hilft bei der Programmierung ungemein, da sie dem Programmierer die Möglichkeit gibt, sehr langen Programmcode überschaubar auf verschiedene Programmdateien aufzuteilen.

top

11.1 Einbinden externer Programmbibliotheken

Eine Bibliothek, sei es ein Teil der Standardbibliothek oder eine selbst geschriebene, kann mithilfe der `import`-Anweisung eingebunden werden. Wir werden in den Beispielen hauptsächlich das Modul `math` der Standardbibliothek verwenden. Das ist ein Modul, das mathematische Funktionen wie `sin` oder `cos` sowie mathematische Konstanten wie `pi` bereitstellt. Um sich diese

Funktionalität in einem Programm zunutze machen zu können, ist folgende import-Anweisung nötig:

```
import math
```

Eine import-Anweisung besteht aus dem Schlüsselwort import, gefolgt von einem Modulnamen. Es können mehrere Module gleichzeitig eingebunden werden, indem diese, durch Kommata getrennt, hinter das Schlüsselwort geschrieben werden:

```
import math, random
```

Dies ist äquivalent zu:

```
import math
import random
```

Obwohl eine import-Anweisung prinzipiell überall im Quellcode stehen kann, ist es der Übersichtlichkeit halber sinnvoll, alle Module zu Beginn des Quelltextes einzubinden.

Nachdem eine Bibliothek eingebunden wurde, wird für diese ein neuer Namensraum mit dem Namen der Bibliothek erstellt. Über diesen Namensraum sind alle Funktionen, Datentypen und Konstanten der Bibliothek im Programm nutzbar. Mit einem Namensraum kann wie mit einer Instanz umgegangen werden, und die Funktionen der Bibliothek können wie Methoden des Namensraums verwendet werden. So bindet folgendes Beispielprogramm die Bibliothek math ein und berechnet den Sinus von der Kreiszahl π :

```
import math
print math.sin(math.pi)
```

Es ist möglich, den Namen des Namensraums durch eine import/as-Anweisung festzulegen:

```
import math as mathematik
print mathematik.sin(mathematik.pi)
```

Beachten Sie, dass dieser Name keine zusätzliche Option ist, sondern das Modul math nun ausschließlich über den Namensraum mathematik erreichbar ist.

Des Weiteren kann die import-Anweisung so verwendet werden, dass kein eigener Namensraum für die eingebundene Bibliothek erzeugt wird, sondern alle Elemente dieser Bibliothek im globalen Namensraum des Programms zur Verfügung stehen:

```
from math import *
print sin(pi)
```

Wenn die import-Anweisung in dieser Weise verwendet wird, sollten Sie beachten, dass keine Referenzen, Funktionen oder Instanzen des einzubindenden Moduls in den aktuellen Namensraum importiert werden, wenn sie mit einem Unterstrich beginnen. Diese Elemente eines Moduls werden als privat und damit als modulintern angesehen.

Hinweis

Der Sinn von Namensräumen ist es, thematisch abgegrenzte Bereiche, also zum Beispiel eine Bibliothek, zu kapseln und über einen gemeinsamen Namen anzusprechen. Wenn Sie den kompletten Inhalt einer Bibliothek in den globalen Namensraum

eines Programms einbinden, kann es vorkommen, dass die Bibliothek mit eventuell vorhandenen Referenzen interferiert. In einem solchen Fall werden die bereits bestehenden Referenzen kommentarlos überschrieben, wie das folgende Beispiel zeigt:

```
>>> pi = 1234
>>> pi
1234
>>> from math import *
>>> pi
3.1415926535897931
```

Aus diesem Grund ist es immer sinnvoll, eine Bibliothek, wenn sie vollständig eingebunden wird, in einem eigenen Namensraum zu kapseln und damit die Anzahl der im globalen Namensraum eingebundenen Elemente möglichst gering zu halten.

Im Hinweiskasten wurde gesagt, dass man die Anzahl der in den globalen Namensraum importierten Objekte möglichst gering halten sollte. Aus diesem Grund ist die oben geschriebene Form der `from/import`-Anweisung nicht gerade praktikabel. Es ist aber möglich, anstatt des Sterns eine Liste von zu importierenden Elementen der Bibliothek anzugeben:

```
from math import sin, pi
print sin(pi)
```

In diesem Fall werden ausschließlich die Funktion `sin` und die Konstante `pi` in den globalen Namensraum importiert. Auch hier ist es möglich, durch ein dem Namen nachgestelltes as einen eigenen Namen festzulegen:

```
from math import sin as hallo, pi as welt
print hallo(welt)
```

So viel zum Einbinden externer Bibliotheken. Sie werden die Standardbibliothek von Python im dritten Teil dieses Buches noch ausführlich kennenlernen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Abstraction is selective ignorance« – Andrew Koenig

12 Objektorientierung

In diesem Kapitel wird endlich die Katze aus dem Sack gelassen: Sie werden in das wichtigste und umfassendste Konzept von Python eingeführt, die *Objektorientierung*. Der Begriff Objektorientierung beschreibt ein Programmierparadigma, das die Wiederverwendbarkeit von Quellcode steigert und es außerdem erleichtert, die Konsistenz von Datenobjekten zu sichern. Diese Vorteile werden dadurch erreicht, dass man Datenstrukturen und die dazugehörigen Operationen zu einem sogenannten *Objekt* zusammenfasst und den Zugriff auf diese Strukturen nur über bestimmte Schnittstellen erlaubt.

Diese Vorgehensweise werden wir an einem Beispiel veranschaulichen, indem wir zuerst auf dem bisherigen Weg eine Lösung erarbeiten und diese ein zweites Mal, diesmal aber objektorientiert, implementieren.

Stellen wir uns einmal vor, wir würden für eine Bank ein System für die Verwaltung von Konten entwickeln, das das Anlegen neuer Konten, Überweisungen sowie Ein- und Auszahlungen ermöglicht. Ein möglicher Ansatz sähe so aus, dass wir für jedes Bankkonto ein Dictionary anlegen, in dem dann alle Informationen über den Kunden und seinen Finanzstatus gespeichert sind. Um die gewünschten Operationen zu unterstützen, würden wir Funktionen definieren. Ein Dictionary für ein stark vereinfachtes Konto könnte folgendermaßen aussehen:

```
konto = {
    "Inhaber" : "Hans Meier",
    "Kontonummer" : 567123,
    "Kontostand" : 12350.0,
    "MaxTagesumsatz" : 1500,
    "UmsatzHeute" : 10.0
```

```
    }
```

Wir gehen modellhaft davon aus, dass jedes Konto einen "Inhaber" hat, der durch einen String mit seinem Namen identifiziert wird. Das Konto hat eine ganzzahlige "Kontonummer", um es von allen anderen Konten zu unterscheiden. Mit der Gleitkommazahl, die mit dem Schlüssel "Kontostand" verknüpft ist, wird das aktuelle Guthaben in Euro gespeichert. Die Schlüssel "MaxTagesumsatz" und "UmsatzHeute" dienen dazu, den Tagesumsatz eines jeden Kunden zu seinem eigenen Schutz auf ein bestimmtes Limit zu begrenzen. "MaxTagesumsatz" gibt dabei an, wie viel Geld pro Tag maximal von dem bzw. auf das Konto bewegt werden darf. Mit "UmsatzHeute" »merkt« sich das System, wie viel am heutigen Tag schon umgesetzt worden ist. Zu Beginn eines neuen Tages wird dieser Wert wieder auf null gesetzt. Die von uns betrachteten Konten sollen prinzipiell nicht überzogen werden können, der Kontostand bleibt also immer positiv.

Ausgehend von dieser Datenstruktur wollen wir nun die geforderten Operationen als Funktionen definieren. Als Erstes brauchen wir eine Funktion, die ein neues Konto nach bestimmten Vorgaben erzeugt:

```
def neues_konto(inhaber, kontonummer, kontostand,
                max_tagesumsatz=1500):
    return {
        "Inhaber" : inhaber,
        "Kontonummer" : kontonummer,
        "Kontostand" : kontostand,
        "MaxTagesumsatz" : max_tagesumsatz,
        "UmsatzHeute" : 0
        }
```

Da diese einfache Funktion selbsterklärend ist, wenden wir uns gleich den Überweisungen zu.

An einem Geldtransfer sind immer ein Sender (das Quellkonto) und ein Empfänger (das Zielkonto) beteiligt. Außerdem muss zum Durchführen der Überweisung der gewünschte Geldbetrag bekannt sein. Die Funktion wird also drei Parameter erwarten: `quelle`, `ziel` und `betr`. Nach unseren Voraussetzungen ist eine Überweisung nur dann möglich, wenn auf dem Quellkonto genug Geld vorhanden ist (es darf nicht überzogen werden) und die Tagesumsätze der beiden Konten ihr Limit nicht überschreiten. Die Überweisungsfunktion soll einen Wahrheitswert zurückgeben, der angibt, ob die Überweisung ausgeführt werden konnte oder nicht. Damit ließe sie sich folgendermaßen implementieren:

```
def geldtransfer(quelle, ziel, betr):
    # Hier erfolgt der Test, ob der Transfer möglich ist
    if(quelle["Kontostand"] < betr or
       quelle["UmsatzHeute"] + betr >
quelle["MaxTagesumsatz"] or
       ziel["UmsatzHeute"] + betr >
ziel["MaxTagesumsatz"]):

        return False # Transfer unmöglich
    else:
        # Alles OK - Auf geht's

        quelle["Kontostand"] -= betr
        quelle["UmsatzHeute"] += betr
        ziel["Kontostand"] += betr
        ziel["UmsatzHeute"] += betr
        return True
```

Die Funktion überprüft zuerst, ob der Transfer durchführbar ist, und beendet den Funktionsaufruf frühzeitig mit dem Rückgabewert `False`, falls dies nicht der Fall ist. Wenn genug Geld auf dem Quellkonto vorhanden ist und kein Tagesumsatzlimit überschritten wird, aktualisiert die Funktion Kontostände und Tagesumsätze entsprechend der Überweisung und gibt `True` zurück.

Die letzten Operationen für unsere Modellkonten sind das Ein- beziehungsweise Auszahlen am Geldautomaten oder Bankschalter.

Beide Funktionen benötigen als Parameter das betreffende Konto und den jeweiligen Geldbetrag als Parameter. Da die Funktionen sehr einfach sind, möchten wir uns nicht weiter mit Erklärungen aufhalten, sondern direkt den Quellcode präsentieren:

```
def einzahlen(konto, betrag):
    if konto["UmsatzHeute"] + betrag >
konto["MaxTagesumsatz"]:
        return False # Tageslimit überschritten
    else:
        konto["Kontostand"] += betrag
        konto["UmsatzHeute"] += betrag
        return True

def auszahlen(konto, betrag):
    if konto["UmsatzHeute"] + betrag >
konto["MaxTagesumsatz"]:
        return False # Tageslimit überschritten
    else:
        konto["Kontostand"] -= betrag
        konto["UmsatzHeute"] += betrag
        return True
```

Auch diese Funktionen geben abhängig von ihrem Erfolg einen Wahrheitswert zurück.

Um einen Überblick über den aktuellen Status unserer Konten zu erhalten, wollten wir eine einfache Ausgabefunktion definieren:

```
def zeige_konto(konto):
    print "Konto von %s" % konto["Inhaber"]
    print "Aktueller Kontostand: %.2f Euro" %
konto["Kontostand"]
    print "(Heute schon %.2f von %d umgesetzt)" % (
        konto["UmsatzHeute"], konto["MaxTagesumsatz"])
```

Mit diesen Definitionen könnten wir beispielsweise folgende Bankoperationen simulieren:

```
>>> k1 = neues_konto("Heinz Meier", 567123, 12350.0)
>>> k2 = neues_konto("Erwin Schmidt", 396754, 15000.0)
>>> geldtransfer(k1, k2, 160)
True
>>> geldtransfer(k2, k1, 1000)
True
>>> geldtransfer(k2, k1, 500)
False
>>> einzahlen(k2, 500)
False
>>> zeige_konto(k1)
Konto von Heinz Meier
Aktueller Kontostand: 13190.00 Euro
(Heute schon 1160.00 von 1500 umgesetzt)
>>> zeige_konto(k2)
Konto von Erwin Schmidt
Aktueller Kontostand: 14160.00 Euro
(Heute schon 1160.00 von 1500 umgesetzt)
```

Zuerst eröffnet Heinz Meier ein neues Konto k1 mit der Kontonummer 567123 mit dem Startguthaben von 12350 Euro. Erwin Schmidt zahlt 15000 Euro auf sein neues Konto k2 mit der Kontonummer 396754 ein. Beide haben den standardmäßigen maximalen Tagesumsatz von 1500 Euro gewählt. Nun treten die beiden in geschäftlichen Kontakt miteinander, wobei Herr Schmid einen DVD-Recorder von Herrn Meier für 160 Euro kauft, der per Überweisung bezahlt wird. Am selben Tag erwirbt Herr Meier Herrn Schmidts gebrauchten Spitzenlaptop, der für 1000 Euro den Besitzer wechselt. Als Herr Meier in den Abendstunden stark an der Heimkinoanlage von Herrn Schmid interessiert ist und ihm dafür 500 Euro überweisen möchte, wird er enttäuscht, denn die Überweisung schlägt fehl. Völlig verdattert zieht Herr Schmidt den voreiligen Schluss, er habe zu wenig Geld auf seinem Konto. Deshalb möchte er den Betrag auf sein Konto einzahlen und anschließend erneut überweisen. Als aber auch die Einzahlung abgelehnt wird, wendet er sich an einen Bankangestellten. Dieser lässt sich die Informationen der beteiligten Konten anzeigen. Dabei sieht er, dass die gewünschte Überweisung das Tageslimit von Herrn Schmidts Konto überschreitet und deshalb nicht ausgeführt werden kann.

Wie Sie sehen, arbeitet unsere Banksimulation wie erwartet und

ermöglicht uns eine relativ einfache Handhabung von Kontodaten. Sie weist aber einige unschöne Eigenheiten auf, wir im Folgenden besprechen werden.

In dem Beispiel sind die Datenstruktur und die Funktionen für ihre Verarbeitung getrennt definiert, was dazu führt, dass das Konto-Dictionary bei jedem Funktionsaufruf als Parameter übergeben werden muss. Man kann sich aber auf den Standpunkt stellen, dass ein Konto nur mit den dazugehörigen Verwaltungsfunktionen sinnvoll benutzt werden kann und auch umgekehrt die Verwaltungsfunktionen eines Kontos nur in Zusammenhang mit dem Konto nützlich sind. Außerdem könnte ein findiger Bankangestellter, der diese Funktionsbibliothek verwendet, ein darauf aufbauendes Programm so formulieren, dass er seinen Kontostand ein wenig aufbessert: Er kann einfach die Werte des Dictionarys direkt verändern, da er nicht an die vorgesehenen Funktionen gebunden ist. Diese direkte Möglichkeit, Daten zu verändern, kann auch die Funktionsweise des Programms beeinflussen, wenn den Eigenschaften des Kontos Werte von nicht sinnvollen Datentypen zugewiesen werden. Beispielsweise könnte dem Kontostand direkt eine Liste zugewiesen werden, was spätestens bei der nächsten Überweisung zu einem TypeError führen würde:

```
>>> k1 = neues_konto("Heinz Meier", 567123, 12350.0)
>>> k2 = neues_konto("Erwin Schmidt", 396754, 15000.0)
>>> k1["Kontostand"] = [3, "Hehe, das gibt einen tollen
Fehler"]
>>> geldtransfer(k1, k2, 160)
Traceback (most recent call last):
  […]
TypeError: unsupported operand type(s) for -=: 'list' and
'int'
```

Wir wünschen uns also eine Möglichkeit, die eigentlichen Daten, also im Beispiel das Konto, mit den Verarbeitungsfunktionen zu einer Einheit zu koppeln und diese Verbindung vor direkten Zugriffen auf die enthaltenen Daten zu schützen, um ihre Konsistenz zu sichern.

Genau diese Wünsche befriedigt die Objektorientierung, indem sie Daten und Verarbeitungsfunktionen zu sogenannten *Objekten* zusammenfasst. Dabei werden die Daten eines solchen Objekts *Attribute* und die Verarbeitungsfunktionen *Methoden* genannt. Attribute und Methoden werden unter dem Begriff *Member* einer Klasse zusammengefasst. Schematisch ließe sich das Objekt eines Kontos also folgendermaßen darstellen:

Konto	
Attribute	**Methoden**
Inhaber	neues_konto()
Kontostand	geldtransfer()
MaxTagesumsatz	einzahlen()
UmsatzHeute	auszahlen()
	zeige_konto()

Tabelle 12.1 Schema eines Konto-Objekts

Die Begriffe »Attribut« und »Methode« sind Ihnen bereits aus früheren Kapiteln von den Basisdatentypen bekannt, denn jede Instanz eines Basisdatentyps stellt – auch wenn Sie es zu dem Zeitpunkt vielleicht noch nicht wussten – ein Objekt dar. Sie wissen auch schon, dass auf die Attribute und Methoden eines Objekts zugegriffen wird, indem man die Referenz auf das Objekt und das dazugehörige Member durch einen Punkt getrennt aufschreibt.

Angenommen, k1 und k2 seien Konto-Objekte, wie sie das obige Schema zeigt, mit den Daten von Herrn Meier und Herrn Schmidt, dann könnte man das letzte Beispiel folgendermaßen formulieren (der Code ist so natürlich noch nicht lauffähig, da die Definition

für die Konto-Objekte fehlt):

```
>>> k1.geldtransfer(k2, 160)
True
>>> k2.geldtransfer(k1, 1000)
True
>>> k2.geldtransfer(k1, 500)
False
>>> k2.einzahlen(500)
False
>>> k1.zeige_konto()
Konto von Heinz Meier
Aktueller Kontostand: 13190.00 Euro
(Heute schon 1160.00 von 1500 umgesetzt)
>>> k2.zeige_konto()
Konto von Erwin Schmidt
Aktueller Kontostand: 14160.00 Euro
(Heute schon 1160.00 von 1500 umgesetzt)
```

Die Methoden `geldtransfer` und `zeige_konto` haben nun beim Aufruf einen Parameter weniger, da das Konto, auf das sie sich jeweils beziehen, nun am Anfang des Aufrufs steht. Da Sie seit der Einführung der Basisdatentypen bereits mit dem Umgang mit Objekten vertraut sind, wird für Sie in diesem Kapitel nur die Technik wirklich neu sein, wie Sie Ihre eigenen Objekte mithilfe von Klassen definieren können.

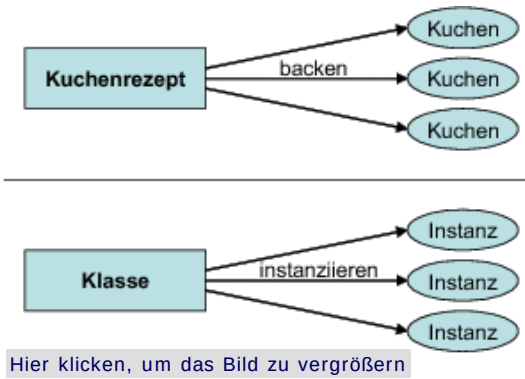
top

12.1 Klassen

Objekte werden über sogenannte *Klassen* definiert. Eine Klasse ist dabei einfach eine formale Beschreibung, wie bestimmte Objekte auszusehen haben, also welche Attribute und Methoden sie besitzen.

Mit einer Klasse allein kann man noch nicht sinnvoll arbeiten, da sie wirklich nur die Beschreibung von Objekten darstellt, selbst aber kein Objekt ist. Man kann das Verhältnis von Klasse und Objekt mit dem von Backrezept und Kuchen vergleichen: Das Rezept definiert die Zutaten und den Herstellungsprozess eines Kuchens und damit auch seine Eigenschaften. Trotzdem reicht ein Rezept allein nicht aus, um die Verwandten zu einer leckeren Torte am Sonntagnachmittag einzuladen. Erst beim Backen wird aus der abstrakten Beschreibung ein fertiger Kuchen.

Ein anderer Name für ein Objekt ist *Instanz*. Das objektorientierte Backen wird daher *Instanziieren* genannt. So, wie es zu einem Rezept mehrere Kuchen geben kann, so können auch mehrere Instanzen von einer Klasse erzeugt werden:



Hier klicken, um das Bild zu vergrößern

Abbildung 12.1 Analogie von Rezept/Kuchen und Klasse/Objekt

Zur Definition einer neuen Klasse in Python dient das Schlüsselwort `class`, dem der Name der neuen Klasse folgt. Die einfachste Klasse hat weder Methoden noch Attribute und wird folgendermaßen definiert:

```
class Konto(object):
    pass
```

(object)

Lassen sie sich an dieser Stelle nicht von dem hinter dem Klassennamen irritieren. Schreiben Sie es einfach immer wie oben gezeigt in Ihre Klassendefinitionen, bis Sie die Hintergründe dafür in Abschnitt 12.2, »Vererbung«, erfahren.

Wie bereits gesagt wurde, lässt sich mit einer Klasse allein nicht arbeiten, weil sie nur eine abstrakte Beschreibung ist. Deshalb wollen wir nun eine Instanz der noch leeren Beispielklasse `Konto` erzeugen. Um eine Klasse zu instanziieren, ruft man die Klasse wie eine Funktion ohne Parameter auf, indem man dem Klassennamen ein rundes Klammernpaar nachstellt. Der Rückgabewert dieses Aufrufs ist eine neue Instanz der Klasse:

```
>>> Konto()
<__main__.Konto instance at 0x00BA75A8>
```

Die schwer lesbare Ausgabe soll uns mitteilen, dass der Rückgabewert von `Konto()` eine Instanz der Klasse `Konto` im Hauptnamensraum `__main__` ist und im Speicher unter der Adresse `0x00BA75A8` abgelegt wurde – uns reicht als Information aus, dass eine neue Instanz der Klasse `Konto` erzeugt worden ist.

Nun ist dieses `Konto`-Objekt weit davon entfernt, unseren Anforderungen vom Anfang des Kapitels zu genügen, und ist somit bis jetzt der bisherigen Dictionary-Implementation unterlegen. Wir werden vor der Erzeugung von neuen Konten erst die Definition von Methoden behandeln.

top

12.1.1 Definieren von Methoden ▼▲

Im Prinzip unterscheidet sich eine Methode nur durch zwei Aspekte von einer normalen Funktion: Erstens wird sie innerhalb eines von `class` eingeleiteten Blocks definiert, und zweitens erhält sie als ersten Parameter immer eine Referenz auf die Instanz, über die sie aufgerufen wird. Dieser erste Parameter muss nur bei der Definition explizit hingeschrieben werden und wird beim Aufruf der Methode automatisch mit der entsprechenden Instanz verknüpft. Da sich die Referenz auf das Objekt selbst bezieht, gibt man dem ersten Parameter den Namen `self` (dt. *selbst*). Methoden besitzen genau wie Funktionen einen eigenen Namensraum, können auf globale Variablen zugreifen und Werte per `return` an die aufrufende Ebene zurückgeben.

Damit können wir unsere Kontoklasse um die noch fehlenden Methoden ergänzen, wobei wir zunächst nur die Methodenköpfe ohne den enthaltenen Code aufschreiben, da wir noch nicht wissen, wie man mit Attributen eigener Klassen umgeht:

```
class Konto(object):
    def geldtransfer(self, ziel, betrag):
        pass

    def einzahlen(self, betrag):
        pass

    def auszahlen(self, betrag):
        pass

    def zeige_konto(self):
        pass
```

Beachten Sie den `self`-Parameter am Anfang jeder Methode, für den automatisch eine Referenz auf die Instanz übergeben wird, die beim Aufruf auf der linken Seite des Punktes steht:

```
>>> k = Konto()
>>> k.einzahlen(500)
```

Hier wird an die Methode `einzahlen` eine Referenz auf das Konto `k` übergeben, auf das dann innerhalb von `einzahlen` über den

Parameter self zugegriffen werden kann.

Im nächsten Abschnitt werden Sie dann lernen, wie Sie auch die Erzeugung neuer Objekte nach Ihren Vorstellungen anpassen können und wie man neue Attribute anlegt.

12.1.2 Konstruktor, Destruktor und die Erzeugung von Attributen ▼▲

Der Lebenszyklus jeder Instanz sieht gleich aus: Sie wird erzeugt, benutzt und anschließend wieder beseitigt. Da es eines der Hauptziele der Objektorientierung war, die Daten eines Objekts vor direktem Zugriff von außen zu schützen, können wir einem Objekt nicht beim Erzeugen seinen Anfangswert direkt zuweisen. Stattdessen geschieht diese Zuweisung mittels einer speziellen Methode, die automatisch beim Instanziieren eines Objekts aufgerufen wird. Man nennt diese Methode auch *Konstruktor* (engl. *construct* = »errichten«) einer Klasse. Pythons Konstruktoren haben alle den Namen __init__ und werden genau wie jede andere Methode definiert:

```
class Beispielklasse(object):
    def __init__(self):
         print "Hier spricht der Konstruktor"
```

Wenn wir jetzt wie gehabt eine Instanz der Klasse Beispielklasse erzeugen, wird implizit die __init__-Methode aufgerufen, und der Text »Hier spricht der Konstruktor« erscheint auf dem Bildschirm:

```
>>> Beispielklasse()
Hier spricht der Konstruktor
<__main__.Konto instance at 0x00BA3670>
```

Konstruktoren können sinnvollerweise keine Rückgabewerte haben, da sie nicht direkt aufgerufen werden und beim Erstellen einer neuen Instanz schon eine Referenz auf diese zurückgegeben wird.

Dem Konstruktor steht der sogenannte *Destruktor* (engl. *destruct* = »zerstören«) gegenüber, der immer dann aufgerufen wird, wenn eine Instanz von der Garbage Collection aus dem Speicher entfernt wird. Ein Destruktor ist eine bis auf self parameterlose Methode, die auf den Namen __del__ hört:

```
class Beispielklasse(object):
    def __init__(self):
        print "Hier spricht der Konstruktor"

    def __del__(self):
        print "Und hier kommt der Destruktor"
```

Das folgende Beispiel zeigt, dass der Destruktor beim Entfernen der Instanz mit dem del-Statement aufgerufen wird:

```
>>> obj = Beispielklasse()
Hier spricht der Konstruktor
>>> del obj
Und hier kommt der Destruktor
```

Dieses Verhalten und der Umstand, dass der Destruktor sehr ähnlich heißt wie das del-Statement, führen oft zu der falschen Annahme, dass der Destruktor bei jedem del-Statement aufgerufen würde. Dies ist aber nur dann der Fall, wenn die letzte Referenz auf ein Objekt mit del entfernt wurde, da erst dann die Garbage Collection aktiv wird, wie es das folgende Beispiel zeigt:

```
>>> v1 = Beispielklasse()
Hier spricht der Konstruktor
>>> v2 = v1
```

```
>>> del v1
>>> del v2
Und hier kommt der Destruktor
```

Wie Sie sehen, wurde `__del__` einmalig nach dem zweiten `del`-Statement aufgerufen und nicht zweimal. Dies wird auch dann noch einmal klar, wenn man sich vor Augen hält, dass ein Objekt zum Entfernen erst einmal erzeugt werden muss: Für einen Konstruktor-Aufruf gibt es *genau einen* Destruktor-Aufruf desselben Objekts.

Im Gegensatz zu Konstruktoren werden Destruktoren relativ selten benutzt, was daran liegt, das Python schon von sich aus einen Großteil der »Drecksarbeit« erledigt und man sich in der Regel nicht um das Aufräumen im Speicher kümmern muss. Destruktoren werden aber häufig benötigt, um beispielsweise bestehende Netzwerkverbindungen sauber zu trennen, den Programmablauf zu dokumentieren oder Fehler zu finden.

Neue Attribute anlegen

Da es die Hauptaufgabe eines Konstruktors ist, einen konsistenten Initialzustand einer Instanz herzustellen und sie damit in einen benutzbaren Zustand zu versetzen, sollten alle Attribute einer Klasse auch dort definiert werden. [Es gibt sehr wenige Sonderfälle, in denen diese Regel eine unpraktische Einschränkung ist. Deshalb muss man nicht zwingend alle Attribute in der `__init__`-Methode definieren. Sie sollten aber im Regelfall, soweit es möglich ist, alle Attribute Ihrer Klassen im Konstruktor erstellen.] Die Definition neuer Attribute erfolgt durch eine einfache Wertezuweisung, wie Sie sie von normalen Variablen her kennen. Damit können wir die Funktion `neues_konto` durch den Konstruktor der Klasse `Konto` ersetzen, der dann wie folgt implementiert werden kann. Für den Parameter `self` wird dabei beim Aufruf automatisch eine Referenz auf die neu erzeugte `Konto`-Instanz übergeben:

```
class Konto(object):
    def __init__(self, inhaber, kontonummer, kontostand,
                       max_tagesumsatz=1500):
        self.Inhaber = inhaber
        self.Kontonummer = kontonummer
        self.Kontostand = kontostand
        self.MaxTagesumsatz = max_tagesumsatz
        self.UmsatzHeute = 0

    # hier kommen die restlichen Methoden hin
```

Da `self` eine Referenz auf die zu erstellende Instanz enthält, können wir über sie die neuen Attribute anlegen, wie in dem Beispiel gezeigt wird. Auf dieser Basis können auch die anderen Funktionen der nicht objektorientierten Variante auf die Kontoklasse übertragen werden. Wir werden uns hier aus Platzgründen auf die Methode `geldtransfer` beschränken. Es sollte dann kein Problem mehr für Sie darstellen, auch die anderen Methoden zu implementieren.

```
class Konto(object):
    # hier kommt der Konstruktor hin

    def geldtransfer(self, ziel, betrag):
        # Hier erfolgt der Test, ob der Transfer möglich
ist
        if(self.Kontostand < betrag or
           self.UmsatzHeute + betrag > self.MaxTagesumsatz
or
           ziel.UmsatzHeute + betrag >
ziel.MaxTagesumsatz):
            return False # Transfer unmöglich
        else:
            # Alles OK - Auf geht's

            self.Kontostand -= betrag
            self.UmsatzHeute += betrag
            ziel.Kontostand += betrag
            ziel.UmsatzHeute += betrag
            return True

    # hier wären die restlichen Methoden
```

Bis zu dieser Stelle haben wir unser erstes großes Ziel erreicht, die Kontodaten und die dazugehörigen Verarbeitungsfunktionen zu einer Einheit zu verbinden. Allerdings ist es immer noch möglich, außerhalb der Klasse auf die Attribute direkt zuzugreifen und diese zu verändern, und folgender Code würde Hotzenplotz immer noch unrechtmäßig bereichern:

```
>>> k = Konto("Hotzenplotz", 321987, 10000.0)
>>> k.Kontostand = 500000.0
>>> k.Kontostand
500000.0
```

Auch die Zuweisung von Werten ungültiger Datentypen wird noch nicht verhindert. Erst mithilfe der privaten Member, die im nächsten Abschnitt beschrieben werden, erreichen wir eine Lösung, die auch die Konsistenz unserer Objekte sichert.

top

12.1.3 Private Member ▼▲

Attribute und Methoden von Klassen, die von außen nicht sichtbar sein sollen, weil sie bei falscher Verwendung die Konsistenz von Objekten beeinträchtigen, können so gekennzeichnet werden, dass nur die Klasse selbst darauf zugreifen kann. Die Manipulation der Objekte erfolgt ausschließlich über die von außen sichtbaren und dadurch dafür vorgesehenen Methoden und Attribute. Die für die Verwendung von außen bestimmten Methoden und Attribute werden auch als *Schnittstelle* der Klasse (engl. *Interface*) bezeichnet.

Für das Benutzerprogramm, das eine Klasse einsetzt, ist nur die Definition der Schnittstelle von Bedeutung. Was hinter den Kulissen, also im Innern der Objekte wirklich passiert, ist dabei vollkommen unerheblich, solange sich die Klasse nach außen hin gemäß der Schnittstelle verhält.

Unsere Kontoklasse könnte also beispielsweise bei jeder größeren Bareinzahlung automatisch eine Benachrichtigung an die Bankdirektion verschicken, dass höchstwahrscheinlich nicht rechtmäßig erworbenes Geld eingezahlt wurde. Das würde uns als Benutzer der Klasse so lange nicht interessieren, wie die Methode `einzahlen` auch den Kontostand korrekt anpassen und abhängig vom Erfolg der Einzahlung `True` oder `False` zurückgeben würde.

Um definierte Schnittstellen zu implementieren, müssen wir eine Möglichkeit haben, Member explizit als öffentlich, also als Teil der Schnittstelle, oder als privat, also als Implementationsdetail, zu deklarieren.

Im Gegensatz zu vielen anderen Programmiersprachen, die dieses Konzept mit eigenen Schlüsselwörtern implementieren, legt in Python der Name eines Attributs fest, ob es von außen explizit verwendet werden soll oder nicht. Dabei gibt es drei Kategorien: [Wenn man es ganz genau nimmt, sind auch diese Member nicht wirklich gegen Zugriffe von außen geschützt: Sie werden intern von Python durch Namen des Schemas *_Klassenname_Attributname* ersetzt, und deshalb führen Versuche, von außen auf die ursprünglichen Namen zuzugreifen, zu Fehlern. Über den geänderten Namen kann aber weiterhin von überall aus auf die Attribute zugegriffen werden.]

Namensschema	Bezeichnung	Bedeutung
`name`	*Public* (Öffentlich)	Normale Member ohne führende Unterstriche sind sowohl innerhalb einer Klasse also auch von außen les- und schreibbar.
		Auf Member, deren Namen mit einem Unterstrich beginnt, kann zwar sowohl von innen als auch von außen lesend und

_name	*Protected* (Geschützt)	schreibend zugegriffen werden, aber der Entwickler einer Klasse teilt den anderen Programmierern dadurch mit, dass dieses Member nicht direkt benutzt werden sollte.
__name	*Private* (Privat)	Namen mit zwei führenden Unterstrichen sind für wirklich private Member gedacht, die von außen nicht sichtbar sind und deshalb nur über Methoden der Klasse verändert und ausgelesen werden können.2

Tabelle 12.2 Namensschemata für öffentliche, private und geschützte Member

Protected Members sind weiterhin nach außen sichtbar und voll veränderbar. Sie sind nur nach einer Konvention geschützt, die es allen Programmierern empfiehlt, solche Attribute von außen nicht zu benutzen. Es handelt sich hierbei um eine Schnittstellendefinition, die nicht durch eine technische Lese- bzw. Schreibsperre erreicht wird, sondern auf einer Konvention zwischen allen Python-Programmierern beruht: Member, die mit einem Unterstrich beginnen, sollen von außen nicht benutzt werden. Wer es trotzdem tut, sollte sich darüber im Klaren sein, dass dies zu nicht beabsichtigtem Verhalten führen kann. Der Vorteil einer solchen Privatisierung durch eine Abmachung besteht gegenüber der technischen Sperre darin, dass immer noch auf die Member zugegriffen werden kann, wenn dies unbedingt erforderlich sein sollte. Dies erleichtert beispielsweise das Entwickeln von Debuggern zur Fehlersuche in Programmen oder Analysetools enorm.

Wenn Sie einem Membernamen zwei Unterstriche voranstellen, so verändern sich die Zugriffsbestimmungen auf technischer Ebene – er wird zu einem *Private Member*. In unserem Kontobeispiel soll insbesondere der Kontostand nicht mehr von außen direkt verändert werden können, sondern nur über die dazu vorgesehenen Methoden. Deshalb benennen wir das Attribut `Kontostand` um in `__Kontostand` um, womit es nach außen hin geschützt wird. Da auch die anderen Attribute nur noch über die Verarbeitungsroutinen mit neuen Werten versehen werden sollen, werden sie ebenfalls als private deklariert:

```
class Konto(object):
     def __init__(self, inhaber, kontonummer, kontostand,
                        max_tagesumsatz=1500):
         self.__Inhaber = inhaber
         self.__Kontonummer = kontonummer
         self.__Kontostand = kontostand
         self.__MaxTagesumsatz = max_tagesumsatz
         self.__UmsatzHeute = 0

     # hier wären die restlichen Methoden
```

Nun führen alle Zugriffe von außen auf diese Member zu einem `AttributeError`:

```
>>> k = Konto("Hotzenplotz", 321987, 10000.0)
>>> k.__Kontostand
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    k.__Kontostand
AttributeError: Konto instance has no attribute
'__Kontostand'
```

Es aber so, dass wir gar nichts dagegen haben, dass jemand den Kontostand ausliest, der Kontostand soll nur nicht von außen direkt verändert werden können. Abhilfe schaffen sogenannte *Getter-Methoden*, deren einfache Aufgabe es ist, die Werte privater Attribute zurückzugeben. Das folgende Beispiel definiert eine Methode `kontostand`, die den Wert des privaten Attributs `__Kontostand` zurückgibt. Das ist möglich, weil `kontostand` als Methode von `Konto` auf dessen Attribute, egal ob privat oder

nicht, zugreifen darf:

```
class Konto(object):
    # hier wäre der Konstruktor

    def kontostand(self):
        return self.__Kontostand

    # hier wären die restlichen Methoden
```

Durch diese einfache Maßnahme ist nun unser Ziel erreicht, dass der Kontostand zwar gegen unzulässige Schreibzugriffe geschützt ist, aber trotzdem noch von außen gelesen werden kann. Folgendes Beispiel verdeutlicht noch einmal das Ergebnis:

```
>>> k = Konto("Hotzenplotz", 321987, 10000.0)
>>> k.kontostand()
10000.0
>>> k.__Kontostand = 99999999.0
>>> k.kontostand()
10000.0
```

Zwar führt der Versuch, den Kontostand von außen zu erhöhen, zu keinem Fehler, aber der Rückgabewert von kontostand nach der vermeintlichen Zuweisung zeigt, dass sich der Wert des Attributs nicht verändert hat.

Das Konzept der Getter-Methoden zum Auslesen von versteckten Attributen wird durch sogenannte *Setter-Methoden* ergänzt, die die genauen Gegenspieler der Getter sind. Mit ihnen lässt sich eine Schnittstelle definieren, die Werte von außen zu manipulieren, wobei die Setter-Methode dafür Sorge tragen sollte, dass keine ungültigen Werte gesetzt werden. Würde Herr Schmidt aufgrund seiner Probleme beim Bezahlen sein Tageslimit für die Zukunft erhöhen wollen, so müsste ein Bankangestellter das private Attribut `__MaxTagesumsatz` verändern können, was mit der aktuellen `Konto`-Klasse nicht möglich ist. Zu diesem Zwecke könnte man eine Setter-Methode `setMaxTagesumsatz` definieren, die als einzigen Parameter neben `self` den gewünschten neuen Tagesumsatz `neues_limit` erhält. Bevor nun das neue Tageslimit gesetzt werden kann, wird der übergebene Wert auf Gültigkeit geprüft – ein Tageslimit muss eine positive Ganz- oder Gleitkommazahl und größer als `0` sein:

```
class Konto(object):
    # hier wäre der Konstruktor

    # Getter-Methode für das Tageslimit
    def maxTagesumsatz(self):
        return self.__MaxTagesumsatz

    # Setter-Methode für das Tageslimit
    def setMaxTagesumsatz(self, neues_limit):
        if(type(neues_limit) in (float, int) and
           neues_limit > 0):
            self.__MaxTagesumsatz = neues_limit
            return True
        else:
            return False

    # hier wären die restlichen Methoden
```

Das Methoden-Paar `maxTagesumsatz` und `setMaxTagesumsatz` ermöglicht nun den komfortablen und trotzdem sicheren Zugriff auf den maximalen Tagesumsatz, indem sichergestellt wird, dass nur gültige Werte gespeichert werden. Die Setter-Methode prüft, ob der Datentyp von `neues_limit` entweder `float` oder `int` ist und ob sein Wert im gültigen Bereich liegt, und setzt abhängig vom Ausgang dieser Prüfung das Attribut `__MaxTagesumsatz` auf den neuen Wert oder eben nicht. Anhand des Rückgabewertes der Funktion kann der Bankangestellte dann sehen, ob er einen Fehler bei der Übergabe gemacht hat. [Eine elegantere Methode, die aufrufende Ebene auf solche Fehler hinzuweisen, lernen Sie in Abschnitt 13.1, »Exception Handling«, kennen. Sie könnten dann beispielsweise bei ungültigen Werten einen `ValueError` produzieren.]

top

12.1.4 Versteckte Setter und Getter

Das im letzten Abschnitt angesprochene Konzept, mithilfe von Setter- und Getter-Methoden das Lesen und Schreiben von Attributen anzupassen, hat den oft als negativ empfundenen Nebeneffekt, dass man beim Benutzen von Attributen auf Methoden zurückgreifen muss. Viel schöner wäre es, wenn man von außen weiterhin Attribute »sehen« und benutzen könnte, die Klasse aber intern die Werte auf Gültigkeit prüfen und so die Konsistenz der Objekte sichern könnte. Schauen Sie sich einmal die beiden gleichwertigen, ohne die dazugehörigen Definitionen natürlich noch nicht funktionierenden Beispiele an:

```
>>> k = Konto("Hotzenplotz", 321987, 10000.0)
>>> k.kontostand()
10000.0
>>> k.setMaxTagesumsatz(2000)
```

Dieses Beispiel nutzt den bekannten Getter/Setter-Ansatz und liest sich schlechter als das folgende Beispiel, weil syntaktisch die Zugriffe auf Attribute durch Methoden verschleiert werden:

```
>>> k = Konto("Hotzenplotz", 321987, 10000.0)
>>> k.Kontostand
10000.0
>>> k.MaxTagesumsatz = 2000
```

In Python wird dieser Wunsch durch die Möglichkeit befriedigt, beim Lesen und Schreiben von Attributen implizit Methoden aufzurufen, die sich um den Ablauf kümmern. Solche sogenannten *Managed Attributes* (dt. *verwaltete Attribute*) werden durch Instanzen des Datentyps `property` unterstützt. Der Konstruktor von `property` erwartet vier optionale Parameter:

property([fget[, fset[, fdel[, doc]]]])

Der Parameter *fget* erwartet eine Referenz auf eine Getter-Methode für das neue Attribut, und *fset* eine Referenz auf die dazugehörige Setter-Methode. Mit dem Parameter *fdel* kann zusätzlich eine Methode angegeben werden, die dann ausgeführt werden soll, wenn das Attribut per `del` gelöscht wird. Mit dem Parameter *doc* kann das *Managed Attribute* mit einem sogenannten Docstring versehen werden. Was ein Docstring ist, können Sie in Abschnitt 13.3 nachlesen und wird an dieser Stelle nicht weiter behandelt.

Wir werden beispielhaft das Attribut `MaxTagesumsatz` als `property` implementieren. Alle `property`-Attribute einer Klasse werden außerhalb jeder Methode direkt auf der ersten Einrückebene innerhalb des `class`-Blocks definiert, indem man dem gewünschten Namen des Attributs den Rückgabewert von `property` zuweist. Im Falle unseres Kontos würde `MaxTagesumsatz` auf folgende Weise zum *Managed Attribute*:

```
class Konto(object):
    # hier wäre der Konstruktor

    # Getter-Methode für das Tageslimit
    def maxTagesumsatz(self):
        print "Getter wurde gerufen"
        return self.__MaxTagesumsatz

    # Setter-Methode für das Tageslimit
    def setMaxTagesumsatz(self, neues_limit):
        if(type(neues_limit) in (float, int) and
           neues_limit > 0):
            print "Setter wurde mit %s aufgerufen" %
neues_limit
            self.__MaxTagesumsatz = neues_limit
        else:
            print "Fehlerhafter Setter-Parameter:",
neues_limit

    # folgende Zeile erzeugt das Property-Attribut
    MaxTagesumsatz = property(maxTagesumsatz,
setMaxTagesumsatz)

    # hier wären die restlichen Methoden
```

Die print-Anweisungen dienen nur dazu, dass wir in unserem Beispiel gleich sehen können, dass die Methoden auch wirklich aufgerufen werden. Außerdem wurden die Rückgabewerte von setMaxTagesumsatz entfernt, da diese die aufrufende Ebene nicht mehr erreichen können und somit sinnlos geworden sind. [Um Fehler zu signalisieren, sollte der Setter Exceptions werfen. Wie das geht, lernen Sie später.]

Nun können wir das neue Attribut wie ein gewöhnliches benutzen, und trotzdem haben wir durch die impliziten Methodenaufrufe volle Kontrolle über seine Werte:

```
>>> k = Konto("Hotzenplotz", 321987, 10000.0)
>>> k.MaxTagesumsatz
Getter wurde aufgerufen
1500
>>> k.MaxTagesumsatz = 9999.0
Setter wurde mit 9999.0 aufgerufen
>>> k.MaxTagesumsatz
Getter wurde aufgerufen
9999.0
>>> k.MaxTagesumsatz = ("Fehlerhafter Wert", "Hehe")
Fehlerhafter Setter-Parameter: ('Fehlerhafter Wert',
'Hehe')
>>> k.MaxTagesumsatz
Getter wurde aufgerufen
9999.0
```

Das Beispiel demonstriert die Funktion des property-Attributs, und durch die Ausgaben lässt sich sehr schön verfolgen, wann die Setter bzw. Getter aufgerufen werden.

12.1.5 Statische Member

Bisher war es so, dass die Klasse den Bauplan für ihre Instanzen definierte und nur benutzt wurde, um Instanzen zu erzeugen. Während des Programmlaufs drehte sich die eigentliche Arbeit nur um die Instanzen, während die Klassen selbst in den Hintergrund traten. Insbesondere hatte jedes Objekt seine eigenen Attribute und seine eigenen Methoden, die von denen der anderen Objekte unabhängig waren. Das ist auch sinnvoll, denn schließlich hat jedes Konto seine eigene Kontonummer, und diese soll auch unabhängig von allen anderen Konten gespeichert werden.

Diese Art von Membern wird *nicht-statisch* genannt, weil sie für jedes Objekt einer Klasse dynamisch neu erstellt werden. Demgegenüber stehen die sogenannten *statischen Membern*, die sich alle Instanzen einer Klasse teilen.

Angenommen, wir wollten zählen, wie viele Konten unsere Bank gerade besitzt, dann könnten wir dies erreichen, indem wir die Instanzen der Klasse Konto zählen. Eine Möglichkeit wäre, einen globalen Zähler bei jedem Konstruktoraufruf von Konto um eins zu erhöhen und bei jedem Aufruf von __del__ wieder und eins zu verringern. Dieser Ansatz würde allerdings das Kapselungsprinzip verletzen, da wir direkt von einer tieferen Ebene auf globale Daten zugreifen würden. Da dies die Gefahr unerwünschter Seiteneffekte bietet, ist es als schlechter Stil verpönt. Eine wesentlich elegantere Lösung bestünde darin, der Klasse Konto einen internen Zähler ihrer eigenen Instanzen als statisches Attribut zu geben. Dieser würde dann bei den entsprechenden Konstruktor- und Destruktoraufrufen herauf- bzw. heruntergezählt.

Statische Attribute werden im Gegensatz zu nicht-statischen Attributen außerhalb des Konstruktors definiert, indem sie wie property-Attribute direkt in dem class-Block durch Zuweisung mit einem Anfangswert versehen werden. Es hat sich eingebürgert, dass dies in der Regel direkt unterhalb der class-Anweisung noch vor der Konstruktordefinition erfolgt. Im Falle unseres Instanzenzählers – wir nennen ihn Anzahl – sieht das wie folgt aus:

```
class Konto(object):
    Anzahl = 0 # Zu Beginn ist die Instanzanzahl 0
```

```
    # hier wäre der Konstruktor
    # hier wären die restlichen Methoden
```

Damit besitzt die Klasse `Konto` ein statisches Attribut `Anzahl`, das sich alle ihre Instanzen teilen. Damit `Anzahl` auch wirklich die Instanzen zählt, passen wir den Konstruktor an und erstellen einen Destruktor. Der Zugriff auf statische Member erfolgt etwas anders als der auf nicht-statische, da beim Verändern der Werte statt des `self` eine Referenz auf die Klasse (in diesem Fall `Konto`) vor dem Punkt stehen muss. Weil sich statische Attribute immer auf die jeweiligen Klassen beziehen – der Zugriff mithilfe des Klassennamens macht es noch einmal deutlich –, werden statische Member auch *Klassen-Member* (engl. *class members*) genannt.

```
class Konto(object):
    Anzahl = 0 # Zu Beginn ist die Instanzanzahl 0

    def __init__(self, inhaber, kontonummer, kontostand,
                 max_tagesumsatz=1500):
        self.__Inhaber = inhaber
        self.__Kontonummer = kontonummer
        self.__Kontostand = kontostand
        self.__MaxTagesumsatz = max_tagesumsatz
        self.__UmsatzHeute = 0
        Konto.Anzahl += 1 # Instanzzähler erhöhen

    def __del__(self):
        Konto.Anzahl -= 1

    # hier wären die restlichen Methoden
```

Zur Demonstration der Funktion des statischen Members folgt jetzt ein kleines Beispiel:

```
>>> k1 = Konto("Florian Kroll", 3111987, 50000.0)
>>> k2 = Konto("Lucas Hövelmann", 25031988, 43000.0)
>>> k3 = Konto("Sebastian Sentner", 6091987, 44000.0)
>>> Konto.Anzahl
3
>>> k1.Anzahl
3
>>> del k2
>>> Konto.Anzahl
2
>>> del k1
>>> k3.Anzahl
1
>>> del k1
>>> del k3
>>> Konto.Anzahl
0
```

Erst werden drei neue `Konto`-Instanzen erzeugt, und wie die Ausgabe zeigt, enthält das statische Attribut `Anzahl` die korrekte Anzahl. Dann werden die Referenzen nacheinander wieder freigegeben, was zur Folge hat, dass die Instanzen von der Garbage Collection entsorgt werden. Die Werte von `Anzahl` spiegeln dies wider. Außerdem zeigt der Zugriff auf `Anzahl` über die Klasse `Konto` direkt als `Konto.Anzahl` und indirekt über die Instanzen `k1` und `k2` als `k1.Anzahl` bzw `k2.Anzahl`, dass der Wert wirklich von allen Instanzen geteilt wird.

Wie der Zugriff mit `Konto.Anzahl` verdeutlicht, ist es auch dann möglich, auf statische Member einer Klasse zuzugreifen, wenn es gar keine Instanzen der Klasse gibt.

Neben statischen Attributen gibt es in Python auch *statische Methoden*, die allerdings kaum genutzt werden und eine untergeordnete Rolle spielen. Da sich statische Methoden nicht auf einzelne Instanzen beziehen, erwarten sie keinen `self`-Parameter, was aber auch dazu führt, dass sie keinen Zugriff auf die Attribute und Methoden der Instanzen haben. Ihre Definition erfolgt ähnlich wie die von `property`-Attributen, nur dass anstelle von `property` die Built-in Function `staticmethod` verwendet wird:

```
class Konto(object):
    Anzahl = 0 # Zu Beginn ist die Instanzanzahl 0

    def zeigeAnzahl():
        print "Die Instanzanzahl ist", Konto.Anzahl

    zeigeAnzahl = staticmethod(zeigeAnzahl)
```

```
    # Die restlichen Member wären hier
```

Statische Methoden können auch aufgerufen werden, wenn es noch gar keine Instanz der Klasse gibt:

```
>>> Konto.zeigeAnzahl()
Die Instanzanzahl ist 0
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Die Grenzen meiner Sprache sind die Grenzen meiner Welt« – Ludwig Wittgenstein

13 Weitere Spracheigenschaften

Zu diesem Zeitpunkt sollten Sie bereits relativ gut in Python programmieren können. In diesem Kapitel werden wir einige weitere Spracheigenschaften von Python behandeln. Wichtig ist, dass dieses Kapitel kein Sammelbecken für »den uninteressanten Rest« darstellt, sondern dass viele der hier vorgestellten Techniken sehr elegant und wichtig sind. Betrachten Sie dieses Kapitel also als essentielle Ergänzung zum bisher Gelernten.

top

13.1 Exception Handling

Stellen Sie sich einmal ein Programm vor, das über eine vergleichsweise tiefe Aufrufhierarchie verfügt, das heißt, dass Funktionen weitere Unterfunktionen aufrufen, die ihrerseits wieder Funktionen aufrufen. Es ist häufig so, dass die übergeordneten Funktionen nicht korrekt weiterarbeiten können, wenn in einer

ihrer Unterfunktionen ein Fehler aufgetreten ist. Es ist also notwendig, die Information, dass ein Fehler aufgetreten ist, durch die Aufrufhierarchie nach oben zu schleusen, damit jede übergeordnete Funktion auf den Fehler reagieren und sich daran anpassen kann.

Bislang konnten wir Fehler, die innerhalb einer Funktion aufgetreten sind, allein anhand des Rückgabewertes der Funktion kenntlich machen. Es wäre mit viel Aufwand verbunden, einen solchen Rückgabewert durch die Funktionshierarchie nach oben durchzureichen, zumal es sich hierbei um Ausnahmen handelt. Wir würden also sehr viel Code dafür aufwenden, um sehr seltene Fälle zu behandeln.

Für genau solche Fälle unterstützt Python ein Programmierkonzept, das *Exceptionhandling* (dt. *Ausnahmebehandlung*) genannt wird. Im Fehlerfall würde unsere Unterfunktion dann eine sogenannte Exception erzeugen und, bildlich gesprochen, nach oben werfen. Die Ausführung der Funktion ist damit beendet. Jede übergeordnete Funktion hat jetzt drei Möglichkeiten:

- Sie fängt die Exception ab, führt den Code aus, der für den Fehlerfall vorgesehen ist, und fährt dann normal fort. In einem solchen Fall bemerken weitere übergeordnete Funktionen die Exception nicht.
- Sie fängt die Exception ab, führt den Code aus, der für den Fehlerfall vorgesehen ist, und wirft die Exception weiter nach oben. In einem solchen Fall ist auch die Ausführung dieser Funktion sofort beendet, und die übergeordnete Funktion steht vor der Wahl, die Exception abzufangen oder nicht.
- Sie lässt die Exception passieren, ohne sie abzufangen. In diesem Fall ist die Ausführung der Funktion sofort beendet, und die übergeordnete Funktion steht vor der Wahl, die Exception abzufangen oder nicht.

Bisher haben wir bei einer solchen Ausgabe

```
>>> abc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'abc' is not defined
```

ganz allgemein von einem »Fehler« oder einer »Fehlermeldung« gesprochen. Dies ist nicht ganz korrekt: Im Folgenden möchten wir diese Ausgabe als *Traceback* bezeichnen. Welche Informationen ein Traceback enthält und wie diese verwendet werden können, wurde bereits in Abschnitt 5.4, »Der Fehlerfall«, behandelt. Ein Traceback wird immer dann angezeigt, wenn eine Exception bis nach ganz oben durchgereicht wurde, ohne abgefangen zu werden, doch was genau ist eine Exception?

Eine *Exception* ist eine Klasse, die Attribute und Methoden zur Klassifizierung und Bearbeitung des Fehlers enthält. Einige dieser Informationen werden im Traceback angezeigt, so etwa die Beschreibung des Fehlers (»`name 'abc' is not defined`«). Eine Exception kann im Programm selbst abgefangen und behandelt werden, ohne dass der Benutzer etwas davon mitbekommt. Näheres zum Abfangen einer Exception erfahren Sie im weiteren Verlauf dieses Kapitels. Sollte eine Exception nicht abgefangen werden, so wird sie in Form eines Tracebacks ausgegeben und der Programmablauf wird beendet.

top

13.1.1 Eingebaute Exceptions ▼▲

In Python existieren eine Reihe von eingebauten Exceptions, zum Beispiel die bereits bekannten Exceptions `SyntaxError`, `NameError` oder `TypeError`. Solche Exceptions werden von Funktionen der Standardbibliothek oder vom Interpreter selbst

geworfen. Diese Exceptions sind eingebaut, das bedeutet, dass sie zu jeder Zeit im Quelltext verwendet werden können:

```
>>> NameError
<type 'exceptions.NameError'>
>>> SyntaxError
<type 'exceptions.SyntaxError'>
```

Die eingebauten Exceptions sind hierarchisch organisiert, das heißt, sie erben von gemeinsamen Basisklassen. Sie sind deswegen in ihrem Attribut- und Methodenumfang weitestgehend identisch. Die Vererbungshierarchie sieht so aus wie in Abbildung 13.1.

Hier klicken, um das Bild zu vergrößern

Abbildung 13.1 Vererbungshierarchie der eingebauten Exceptions

Nachfolgend soll die Verwendung der Exception `BaseException` besprochen werden, da sie die Basisklasse aller Exceptions darstellt. Sie stellt eine gewisse Grundfunktionalität bereit, die folglich für alle Exception-Typen vorhanden ist. Im Anhang finden Sie eine Beschreibung aller eingebauten Exceptions.

BaseException

Die Klasse `BaseException` verfügt im Wesentlichen über zwei Attribute namens `message` und `args`, in denen nähere Informationen zur Exception stehen können. Das Tupel `args` enthält alle bei der Instanziierung von `BaseException` übergebenen Argumente. Wenn nur eines übergeben wurde, wird dieses zusätzlich in `message` geschrieben, sodass direkt auf dieses Argument zugegriffen werden kann:

```
>>> e = BaseException("Hallo Welt")
>>> e.message
'Hallo Welt'
>>> e.args
('Hallo Welt',)
```

Werden bei der Instanziierung mehrere Argumente übergeben, so werden diese in das Tupel `args` geschrieben. Das Attribut `message` ist in diesem Fall ein leerer String:

```
>>> e = BaseException(1,2,3,4,5,6)
>>> e.message
''
>>> e.args
(1, 2, 3, 4, 5, 6)
```

Die Klasse `BaseException` erlaubt es, durch Slicing auf die gespeicherten Argumente zuzugreifen:

```
>>> e = BaseException(1,2,3,4,5,6)
>>> e[2:4]
(3, 4)
```

top

13.1.2 Werfen einer Exception

Bisher haben wir nur Exceptions betrachtet, die in einem Fehlerfall vom Python-Interpreter geworfen wurden. Es ist jedoch auch möglich, mithilfe der raise-Anweisung selbst eine Exception zu werfen:

```
>>> raise SyntaxError("Hallo Welt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: Hallo Welt
```

Dazu wird das Schlüsselwort raise, gefolgt von einer Instanz, geschrieben. Diese darf nur Instanz einer selbst erstellten Klasse oder eines vordefinierten Exception-Typs sein. Es ist zwar auch möglich, einen String zu werfen, davon wird jedoch ausdrücklich abgeraten, da dies als »deprecated« eingestuft wurde und in späteren Python-Versionen nicht mehr möglich sein wird:

```
>>> raise "Hallo Welt"
__main__:1: DeprecationWarning: raising a string exception
is deprecated
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Hallo Welt
```

Das Werfen anderer Datentypen ist nicht möglich:

```
>>> raise 123
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exceptions must be classes, instances, or
strings
(deprecated), not int
```

Die raise-Anweisung kann in zwei Varianten verwendet werden. Die erste Möglichkeit haben Sie eben kennengelernt, also das Schlüsselwort raise, gefolgt von einer Instanz einer Klasse.

Durch die zweite Verwendungsmöglichkeit von raise spart man sich die explizite Instanziierung der Klasse. Dazu folgen auf das Schlüsselwort raise drei durch Kommata getrennte Angaben:

- Die erste Angabe muss der Datentyp der Exception sein, die geworfen werden soll. Da die beiden folgenden Angaben optional sind, kann eine raise-Anweisung auch so aussehen:

  ```
  >>> raise SyntaxError
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  SyntaxError: None
  ```

  In diesem Fall wird automatisch eine Instanz der Klasse SyntaxError erzeugt und geworfen. Der Konstruktor der Klasse bekommt dabei keine Argumente übergeben, sodass hier beispielsweise keine Fehlermeldung erscheint (SyntaxError: None).

- Als zweite, optionale Angabe der raise-Anweisung kann ein Argument für den Konstruktor der Klasse übergeben werden. Im Falle einer Standard-Exception entspricht dieses der Fehlermeldung.

  ```
  >>> raise SyntaxError, "Dies ist ein Fehler"
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  SyntaxError: Dies ist ein Fehler
  >>> raise SyntaxError, [1,2,3,4]
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  SyntaxError: [1, 2, 3, 4]
  ```

- Die dritte, ebenfalls optionale Angabe der raise-Anweisung kann ein sogenanntes Traceback-Objekt sein. Ein solches Objekt hält den aktuellen Programmstatus fest und erlaubt es somit, eine Exception aus einem bestimmten Kontext heraus zu werfen, obwohl sich der Programmfluss selbst nicht in diesem Kontext befindet. Näheres zum Traceback-Objekt erfahren Sie im Zusammenhang mit dem Modul traceback in Abschnitt 21.6.

- Im folgenden Abschnitt möchten wir besprechen, wie Exceptions im Programm abgefangen werden können, sodass sie nicht in einem Traceback enden, sondern zur Ausnahmebehandlung eingesetzt werden können. Beachten Sie, dass wir sowohl in diesem als auch im nächsten Kapitel bei den eingebauten Exceptions bleiben. Selbstdefinierte Exceptions werden das Thema von Abschnitt 13.1.4 sein.

top

13.1.3 Abfangen einer Exception

Es wurde bereits gesagt, dass eine Exception innerhalb des Programms abgefangen und behandelt werden kann. Stellen Sie sich dazu einmal vor, wir wollten eine Funktion schreiben, die es uns erlaubt, auf ein Element einer Liste lst mit dem Index n zuzugreifen. Die Funktion muss intern prüfen, ob ein n-tes Element in lst existiert, und, wenn ja, dieses zurückgeben. Sollte kein solches Element existieren, soll die Funktion None zurückgeben.

Nach Ihrem bisherigem Kenntnisstand würde die Funktion folgendermaßen aussehen:

```
def get(lst, n):
    if 0 <= n < len(lst):
        return lst[n]
    else:
        return None
```

Eine zweite – und in vielen Fällen elegantere – Variante wäre es, einfach auf das n-te Element der Liste lst zuzugreifen und eine eventuell geworfene IndexError-Exception abzufangen. Wenn diese Exception abgefangen wurde, gibt die Funktion None zurück:

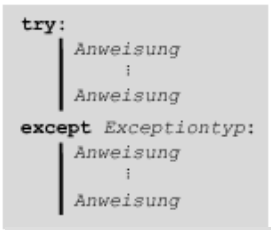
```
def get(lst, n):
    try:
        return lst[n]
    except IndexError:
        return None
```

Zum Abfangen einer Exception wird eine sogenannte try/except-Anweisung verwendet. Eine solche Anweisung besteht zunächst einmal aus zwei Teilen:

- Der try-Block wird durch das Schlüsselwort try eingeleitet, gefolgt von einem Doppelpunkt und, um eine Ebene weiter eingerückt, einem beliebigen Codeblock. Dieser Codeblock wird zunächst ausgeführt. Wenn in diesem Codeblock eine Exception auftritt, wird seine Ausführung sofort beendet und der except-Zweig der Anweisung ausgeführt.

- Der except-Zweig wird durch das Schlüsselwort except eingeleitet, gefolgt von einer optionalen Liste von Exception-Typen, einem Doppelpunkt und, um eine Ebene weiter eingerückt, einem beliebigen Codeblock. Dieser Codeblock wird nur dann ausgeführt, wenn innerhalb des try-Blocks eine der aufgelisteten Exceptions geworfen wurde.

Eine grundlegende try/except-Anweisung hat also folgende Struktur:



Hier klicken, um das Bild zu vergrößern

Abbildung 13.2 Struktur einer try/except-Anweisung

Kommen wir zurück zu unserer Beispielfunktion get. Es wäre durchaus möglich, dass bei einem Funktionsaufruf für n fälschlicherweise keine ganze Zahl, sondern zum Beispiel ein String übergeben wird. In einem solchen Fall wird kein IndexError, sondern ein TypeError geworfen, welcher von der try/except-Anweisung bislang nicht abgefangen wird:

```
>>> get([1,2,3], "s")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in get
TypeError: list indices must be integers
```

Die Funktion soll nun dahingehend erweitert werden, dass auch ein TypeError abgefangen und dann ebenfalls None zurückgegeben wird. Dazu haben wir im Wesentlichen drei Möglichkeiten. Die erste wäre es, die Liste der abzufangenden Exception-Typen im vorhandenen except-Zweig um den TypeError zu erweitern. Beachten Sie dabei, dass zwei oder mehr Exception-Typen im Kopf eines except-Zweiges als Tupel angegeben werden müssen.

```
try:
    return lst[n]
except (IndexError, TypeError):
    return None
```

Dies ist recht einfach und führt im gewählten Beispiel zu dem gewünschten Resultat. Stellen Sie sich jedoch einmal vor, Sie wollten je nach Exception-Typ verschiedenen Code ausführen. Um ein solches Verhalten erreichen zu können, kann eine try/except-Anweisung über beliebig viele except-Zweige verfügen.

```
try:
    return lst[n]
except IndexError:
    return None
except TypeError:
    return None
```

Die dritte – weniger elegante – Möglichkeit wäre es, alle Exceptions auf einmal abzufangen. Dazu wird einfach ein except-Zweig ohne Angabe eines Exception-Typs geschrieben:

```
try:
    return lst[n]
except:
    return None
```

Hinweis

Beachten Sie unbedingt, dass es nur in wenigen Fällen sinnvoll ist, alle möglichen Exceptions auf einmal abzufangen. Durch diese Art Exception Handlings kann es vorkommen, dass unabsichtlich auch Exceptions abgefangen werden, die nichts mit dem obigen Code zu tun haben. Das betrifft unter anderem die KeyInterrupt-Exception, die bei einem Programmabbruch per Tastenkombination geworfen wird.

Eine Exception ist nichts anderes als eine Instanz einer Klasse. Von daher stellt sich die Frage, ob und wie man innerhalb eines except-Zweiges Zugriff auf die geworfene Instanz erlangt. Das ist möglich, indem nach dem Tupel der abzufangenden Exception-Typen und durch ein Komma von diesem getrennt ein Bezeichner geschrieben wird. Unter diesem Namen kann nun innerhalb des Codeblocks auf die geworfene Exception zugegriffen werden. Dies könnte folgendermaßen aussehen:

```
try:
    print [1,2,3][10]
except (IndexError, TypeError), e:
    print "Fehlermeldung:", e.message
```

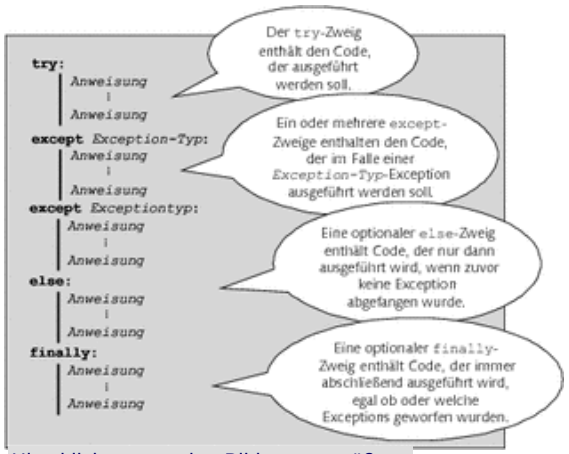
Die Ausgabe des obigen Beispiels lautet:

```
Fehlermeldung: list index out of range
```

Zusätzlich kann eine try/except-Anweisung über einen else- und einen finally-Zweig verfügen, die jeweils nur ein einziges Mal pro Anweisung vorkommen dürfen. Der dem else-Zweig zugehörige Codeblock wird ausgeführt, wenn keine Exception aufgetreten ist, und der dem finally-Zweig zugehörige Codeblock wird in jedem Fall nach Behandlung aller Exceptions und nach dem Ausführen des else-Zweigs ausgeführt, egal, ob oder welche Exceptions vorher aufgetreten sind. Dieser finally-Zweig eignet sich daher besonders für Dinge, die in jedem Fall erledigt werden müssen, wie beispielsweise das Schließen eines Dateiobjekts.

Beachten Sie, dass sowohl der else- als auch der finally-Zweig ans Ende der try/except-Anweisung geschrieben werden müssen. Wenn beide Zweige vorkommen, muss der else-Zweig vor dem finally-Zweig stehen.

Abbildung 13.3 zeigt eine vollständige try/except-Anweisung.



Hier klicken, um das Bild zu vergrößern

Abbildung 13.3 Eine vollständige try/except-Anweisung

Abschließend noch einige Bemerkungen dazu, wie eine try/except-Anweisung ausgeführt wird. Zunächst wird der dem try-Zweig zugehörige Code ausgeführt. Sollte innerhalb dieses Codes eine Exception geworfen werden, so wird der dem entsprechenden except-Zweig zugehörige Code ausgeführt. Ist kein passender except-Zweig vorhanden, so wird die Exception nicht abgefangen und endet, wenn sie auch anderswo nicht abgefangen wird, als Traceback auf dem Bildschirm.

Sollte im try-Zweig keine Exception geworfen werden, so wird keiner der except-Zweige ausgeführt, sondern zunächst der else- und dann der finally-Zweig, wobei beide Zweige optional sind.

Beachten Sie, dass der finally-Zweig in jedem Fall, also auch wenn Exceptions aufgetreten sind, zum Schluss ausgeführt wird.

Exceptions, die innerhalb eines except-, else- oder finally-Zweiges geworfen werden, werden so behandelt, als würfe die gesamte try/except-Anweisung diese Exception. Exceptions, die in diesen Zweigen geworfen werden, können also nicht von nachfolgenden except-Zweigen der gleichen Anweisung wieder abgefangen werden. Es ist jedoch möglich, try/except-Anweisungen zu verschachteln:

```
try:
    try:
        raise TypeError
    except IndexError:
        print "Ein IndexError ist aufgetreten"
except TypeError:
    print "Ein TypeError ist aufgetreten"
```

Im try-Zweig der inneren try/except-Anweisung wird ein TypeError geworfen, der von der Anweisung selbst nicht abgefangen wird. Die Exception wandert dann, bildlich gesprochen, eine Ebene höher und durchläuft die nächste try/except-Anweisung. In dieser wird der geworfene TypeError abgefangen und eine entsprechende Meldung ausgegeben. Die Ausgabe des Beispiels lautet also: Ein TypeError ist aufgetreten, es wird kein Traceback angezeigt.

top

13.1.4 Eigene Exceptions

Beim Werfen und Abfangen von Exceptions sind Sie nicht auf den eingebauten Satz von Exception-Typen beschränkt, vielmehr können Sie selbst beliebige neue Typen erstellen. Theoretisch brauchen Sie dabei auch keine besonderen Regeln einhalten, denn eine Exception kann eine beliebige selbst definierte Klasse sein.

Im Programmieralltag werden Sie jedoch vermutlich kaum eine selbst definierte Exception von Grund auf neu schreiben, vielmehr möchten Sie auf das Verhalten der vorhandenen Exception-Typen zurückgreifen und nur Platz für weitere Informationen schaffen oder gar überhaupt keine Änderungen vornehmen.

In solchen Fällen erstellen Sie eine eigene Klasse, die von einer der eingebauten Exceptions erbt. Dazu bietet sich die Basisklasse Exception an. Die folgende Beispielfunktion soll zwei ganze Zahlen dividieren und im Falle einer Division durch Null keinen ZeroDivisionError, sondern einen eigenen Exception-Typ mit weiteren Informationen werfen. Dazu definieren wir zunächst eine von Exception abgeleitete Klasse und fügen ein Attribut für den Zähler der Division hinzu:

```
class DivisionByZeroError(Exception):
    def __init__(self, z):
        self.zaehler = z
```

Dann definieren wir die Funktion. Sie erwartet zwei Parameter und gibt das Ergebnis ihrer Division zurück. Wenn der Nenner null ist, wird die soeben erstellte Klasse DivisionByZeroError geworfen:

```
def division(z, n):
    if n == 0:
        raise DivisionByZeroError(z)
    return z / n
```

Die dem Konstruktor der Klasse übergebenen zusätzlichen Informationen werden im Traceback nicht angezeigt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 4, in division
__main__.DivisionByZeroError
```

Sie kommen erst zum Tragen, wenn die Exception abgefangen und bearbeitet wird:

```
try:
    division(12, 0)
except DivisionByZeroError, e:
    print "Nulldivision: %d / 0" % e.zaehler
```

Dieser Code fängt die entstandene Exception ab und gibt daraufhin eine Fehlermeldung aus. Aufgrund der zusätzlichen Informationen, die die Klasse durch das Attribut `zaehler` bereitstellt, lässt sich die vorangegangene Berechnung rekonstruieren. Die Ausgabe des Beispiels lautet:

```
Nulldivision: 12 / 0
```

Damit eine solche selbst definierte Exception mit weiterführenden Informationen auch eine Fehlermeldung enthalten kann, muss sie die Magic-Function `__str__` implementieren:

```
class DivisionByZeroError(Exception):
    def __init__(self, z):
        self.zaehler = z

    def __str__(self):
        return "Division durch Null"
```

Ein Traceback, der durch diese Exception verursacht wird, würde folgendermaßen aussehen:

```
>>> raise DivisionByZeroError(12, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.DivisionByZeroError: Division durch Null
```

Das Exception Handling hilft ungemein beim Schreiben von strukturiertem und lesbarem Code, sodass Sie diese Techniken verinnerlichen sollten. Wir werden auch im Laufe dieses Buches immer wieder Exceptions verwenden.

top

13.1.5 Erneutes Werfen einer Exception ▲

In vielen Fällen, gerade bei einer tiefen Funktionshierarchie, ist es sinnvoll, eine Exception abzufangen, die für diesen Fall vorgesehene Fehlerbehandlung zu starten und die Exception danach erneut zu werfen. Dazu folgendes Beispiel:

```
def funktion3():
    raise TypeError

def funktion2():
    funktion3()

def funktion1():
    funktion2()

funktion1()
```

Im Beispiel wird die Funktion `funktion1` aufgerufen, die ihrerseits `funktion2` aufruft, in der die Funktion `funktion3` aufgerufen wird. Es handelt sich also um insgesamt drei verschachtelte Funktionsaufrufe. Im Innersten dieser Funktionsaufrufe, in `funktion3`, wird eine `TypeError`-Exception geworfen. Diese Exception wird nicht abgefangen, deshalb sieht der dazugehörige Traceback so aus:

```
Traceback (most recent call last):
```

```
  File "test.py", line 10, in <module>
    funktion1()
  File "test.py", line 8, in funktion1
    return funktion2()
  File "test.py", line 5, in funktion2
    return funktion3()
  File "test.py", line 2, in funktion3
    raise TypeError
TypeError
```

Der Traceback beschreibt erwartungsgemäß die Funktionshierarchie zum Zeitpunkt der `raise`-Anweisung. Diese Liste wird auch *Call Stack* genannt.

Der Gedanke, der hinter dem Exception-Prinzip steht, ist der, dass sich eine Exception in der Aufrufhierarchie nach oben arbeitet und an jeder Station abgefangen werden kann. In unserem Beispiel soll die Funktion `funktion1` die `TypeError`-Exception abfangen, damit sie eine spezielle, auf den `TypeError` zugeschnittene Fehlerbehandlung durchführen kann. So könnte dann beispielsweise ein Dateiobjekt geschlossen werden. Nachdem `funktion1` ihre funktionsinterne Fehlerbehandlung durchgeführt hat, soll die Exception weiter nach oben gereicht werden. Dazu wird sie erneut geworfen, wie im folgenden Beispiel:

```
def funktion3():
    raise TypeError

def funktion2():
    funktion3()

def funktion1():
    try:
        funktion2()
    except TypeError:
        # Fehlerbehandlung
        raise TypeError

funktion1()
```

Im Gegensatz zum vorherigen Beispiel sieht der nun auftretende Traceback so aus:

```
Traceback (most recent call last):
  File "test.py", line 14, in <module>
    funktion1()
  File "test.py", line 12, in funktion1
    raise TypeError
TypeError
```

Man sieht, dass dieser Traceback Informationen über den Kontext der zweiten `raise`-Anweisung enthält. Diese sind aber gar nicht von Belang, sondern eher ein Nebenprodukt der Fehlerbehandlung innerhalb der Funktion `funktion1`. Optimal wäre es, wenn trotz des temporären Abfangens der Exception in `funktion1` der resultierende Traceback den Kontext der ursprünglichen `raise`-Anweisung beschreiben würde. Um das zu erreichen, wird eine `raise`-Anweisung ohne Angabe eines Exception-Typs geschrieben:

```
def funktion3():
    raise TypeError

def funktion2():
    funktion3()

def funktion1():
    try:
        funktion2()
    except TypeError, e:
        # Fehlerbehandlung
        raise

funktion1()
```

Der in diesem Beispiel ausgegebene Traceback sieht folgendermaßen aus:

```
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    funktion1()
  File "test.py", line 11, in funktion1
```

```
    funktion2()
  File "test.py", line 7, in funktion2
    funktion3()
  File "test.py", line 4, in funktion3
    raise TypeError
TypeError
```

Sie sehen, dass es sich dabei um den Stacktrace der Stelle handelt, an der die Exception ursprünglich geworfen wurde. Der Traceback enthält damit die gewünschten Informationen über die Stelle, an der der Fehler tatsächlich aufgetreten ist.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Jede mathematische Formel in einem Buch halbiert die Verkaufszahlen dieses Buches« – Stephen Hawking

14 Mathematik

Herzlich willkommen zum dritten Teil dieses Buches. Hier möchten wir uns intensiv mit der Standardbibliothek von Python auseinandersetzen und alle wichtigen Module besprechen. Außerdem werden wir die eine oder andere Drittanbieterbibliothek behandeln. Wir beginnen mit den Modulen der Standardbibliothek, mit deren Hilfe sich im weitesten Sinne mathematische Berechnungen durchführen lassen.

top

14.1 Mathematische Funktionen – math, cmath

Das Modul `math` ist Teil der Standardbibliothek und stellt mathematische Funktionen und Konstanten bereit. Beachten Sie, dass `math` den komplexen Zahlenraum – und damit den Datentyp `complex` – vollständig ignoriert. Das heißt vor allem, dass eine in `math` enthaltene Funktion niemals einen komplexen Parameter akzeptiert oder ein komplexes Ergebnis zurückgibt. So wird die Berechnung der Quadratwurzel von `-1` unter Verwendung der Bibliothek `math` beispielsweise stets eine Exception werfen.

Sollte ein komplexes Ergebnis ausdrücklich gewünscht sein, so kann anstelle von `math` das Modul `cmath` verwendet werden, in dem die Funktionen von `math` enthalten sind, die eine sinnvolle Erweiterung auf den komplexen Zahlen haben.

Im Folgenden werden alle Funktionen von `math` aufgelistet und besprochen. Sollte ein Äquivalent in `cmath` existieren, finden Sie eine entsprechende Anmerkung vor.

Bevor Sie die nachfolgenden Beispiele im interaktiven Modus verwenden können, müssen Sie das Modul `math` einbinden:

```
>>> import math
```

Mathematische Konstanten

math.pi

Die Kreiszahl Pi (π).

```
>>> math.pi
3.1415926535897931
```

Die Konstante ist auch in cmath vorhanden.

math.e

Die Eulersche Zahl e.

```
>>> math.e
2.7182818284590451
```

Die Konstante ist auch in cmath vorhanden.

Zahlentheoretische Funktionen

math.ceil(x)

Die Funktion ceil (für engl. *ceiling*, dt. *Zimmerdecke*) gibt die kleinste ganze Zahl zurück, die größer oder gleich *x* ist. Der Parameter *x* muss eine Instanz eines numerischen Datentyps sein. Der Rückgabewert ist eine Gleitkommazahl.

```
>>> math.ceil(3.5)
4.0
>>> math.ceil(2)
2.0
```

math.fabs(x)

Gibt den Betrag von *x* zurück. Im Gegensatz zur Built-in Function abs ist der Rückgabewert von fabs immer eine Gleitkommazahl.

```
>>> math.fabs(-7)
7.0
>>> math.fabs(-7.5)
7.5
```

math.floor(x)

Die Funktion floor (dt. *Fußboden*) gibt die größte ganze Zahl zurück, die kleiner oder gleich *x* ist. Die Funktion ist damit das Gegenstück zu ceil. Das Ergebnis wird immer als Gleitkommazahl zurückgegeben.

```
>>> math.floor(1.9)
1.0
>>> math.floor(-2.3)
-3.0
```

math.fmod(x, y)

Berechnet *x* Modulo *y*. Beachten Sie, dass diese Funktion nicht immer dasselbe Ergebnis berechnet wie *x* % *y*. So gibt fmod das Ergebnis beispielsweise mit dem Vorzeichen von *x* zurück, während *x*%*y* das Ergebnis mit dem Vorzeichen von *y* zurückgibt. Generell gilt, dass fmod bei Modulo-Operationen mit Gleitkommazahlen und der Modulo-Operator % bei Operationen mit ganzen Zahlen bevorzugt werden sollte.

```
>>> math.fmod(7.5, 3.5)
0.5
```

math.frexp(x)

Extrahiert Mantisse und Exponent der übergebenen Zahl *x*. Das Ergebnis ist ein Tupel der Form `(m, e)`, wobei `m` für die Mantisse und `e` für den Exponenten steht. Mantisse und Exponent sind dabei im Kontext der Formel

$x = m \cdot 2^e$

zu sehen.

```
>>> math.frexp(2.5)
(0.625, 2)
>>> math.frexp(-7.0e12)
(-0.79580786405131221, 43)
```

math.ldexp(m, e)

Diese Funktion ist das Gegenstück zu `frexp`. Sie berechnet $m \cdot 2^e$ und gibt das Ergebnis als Gleitkommazahl zurück.

```
>>> math.ldexp(0.625, 2)
2.5
```

math.modf(x)

Gibt den Nachkomma- und den Vorkommaanteil von *x* als Gleitkommazahlen in einem Tupel zurück.

```
>>> math.modf(10.5)
(0.5, 10.0)
```

Exponential- und Logarithmusfunktionen

math.exp(x)

Berechnet e^x, wobei *e* für die Eulersche Zahl steht. Das Ergebnis ist immer eine Gleitkommazahl.

```
>>> math.exp(1.0)
2.7182818284590451
>>> math.exp(10)
22026.465794806718
```

Die Funktion ist auch in `cmath` vorhanden.

math.log(x[, base])

Berechnet den Logarithmus von *x* zur Basis *base*. Wenn *base* nicht angegeben wurde, wird der Logarithmus Naturalis, also der Logarithmus zur Basis e berechnet.

```
>>> math.log(1)
0.0
>>> math.log(32, 2)
5.0
```

Die Funktion ist auch in `cmath` vorhanden.

math.log10(x)

Berechnet den dekadischen Logarithmus von *x*, also den Logarithmus von *x* zur Basis `10`. Der Aufruf dieser Funktion ist damit äquivalent zu `math.log(x, 10)`.

Die Funktion ist auch in cmath vorhanden.

math.pow(x, y)

Berechnet x^y . Es können für *x*, insbesondere aber auch für *y*, negative Zahlen oder Gleitkommazahlen übergeben werden. Beachten Sie dabei aber, dass *x* und *y* nicht beide negativ sein dürfen, da das Ergebnis sonst eine komplexe Zahl wäre. Diese Funktion ist äquivalent zur Built-in Function pow.

```
>>> pow(2, 3)
8
>>> pow(100, 0.5)
10.0
```

math.sqrt(x)

Berechnet die Quadratwurzel von *x*, wobei *x* größer oder gleich 0 sein muss. Das Ergebnis ist immer eine Gleitkommazahl.

```
>>> math.sqrt(100)
10.0
```

Die Funktion ist auch in cmath vorhanden.

Trigonometrische Funktionen

math.acos(x)

Berechnet den Arkuskosinus von *x*. Der Arkuskosinus ist die Umkehrfunktion des Kosinus. Der Parameter *x* muss eine Gleitkommazahl im Zahlenraum von -1 bis 1 sein. Der Rückgabewert von acos ist ebenfalls eine Gleitkommazahl und wird im Bogenmaß angegeben.

```
>>> math.acos(0.5)
1.0471975511965979
```

Die Funktion ist auch in cmath vorhanden.

math.asin(x)

Berechnet den Arkussinus von *x*. Der Arkussinus ist die Umkehrfunktion des Sinus. Der Parameter *x* muss eine Gleitkommazahl im Zahlenraum von -1 bis 1 sein. Der Rückgabewert von asin ist ebenfalls eine Gleitkommazahl und wird im Bogenmaß angegeben.

```
>>> math.asin(0.5)
0.52359877559829893
```

Die Funktion ist auch in cmath vorhanden.

math.atan(x)

Berechnet den Arkustangens von *x*. Der Arkustangens ist die Umkehrfunktion des Tangens. Der Rückgabewert von atan ist eine Gleitkommazahl, wird im Bogenmaß angegeben und liegt im Bereich von – π / 2 bis + π / 2.

```
>>> math.atan(0.5)
0.46364760900080609
```

Die Funktion ist auch in cmath vorhanden.

math.atan2(y, x)

Berechnet atan(*y* / *x*). Im Gegensatz zur atan-Funktion können die Vorzeichen der Parameter *x* und *y* beachtet und somit der Quadrant des Ergebnisses berechnet werden. Mithilfe der Funktion atan2 lassen sich beispielsweise kartesische Koordinaten in Polarkoordinaten umrechnen.

```
>>> math.atan2(1, 1)
0.78539816339744828
>>> math.atan2(-1, -1)
-2.3561944901923448
```

math.cos(x)

Berechnet den Kosinus von *x*. Der Parameter *x* muss im Bogenmaß angegeben werden.

```
>>> math.cos(math.pi)
-1.0
```

Die Funktion ist auch in cmath vorhanden.

math.hypot(x, y)

Berechnet die Euklidische Norm des Vektors (*x*,*y*). Die Euklidische Norm eines Vektors entspricht der Länge des Vektors und ist definiert als:

Der Funktionsname hypot kommt daher, dass das Ergebnis der Berechnung gleichbedeutend ist mit der Länge der Hypotenuse eines rechtwinkligen Dreiecks mit den Kathetenlängen *x* und *y*.

```
>>> math.hypot(5, 7)
8.6023252670426267
```

math.sin(x)

Berechnet den Sinus von *x*. Der Parameter *x* muss im Bogenmaß angegeben werden.

```
>>> math.sin(math.pi/2)
1.0
```

Die Funktion ist auch in cmath vorhanden.

math.tan(x)

Berechnet den Tangens von *x*. Der Parameter *x* muss im Bogenmaß angegeben werden.

```
>>> math.sin(math.pi/2)
1.0
```

Die Funktion ist auch in cmath vorhanden.

Winkelfunktionen

math.degrees(x)

Rechnet den Winkel *x* vom Bogenmaß in Grad um. Das Ergebnis ist immer eine Gleitkommazahl und wird nach der Formal 360 *x* / 2 π berechnet.

```
>>> math.degrees(math.pi/2)
90.0
```

math.radians(x)

Rechnet den Winkel *x* von Grad ins Bogenmaß um. Das Ergebnis ist immer eine Gleitkommazahl und wird nach der Formel 2 π · *x* / 360 berechnet.

```
>>> math.radians(180.0)
3.1415926535897931
```

Hyperbolische Funktionen

math.cosh(x)

Berechnet den Kosinus Hyperbolicus von *x*. Das Ergebnis ist eine Gleitkommazahl.

```
>>> math.cosh(1.0)
1.5430806348152437
```

Die Funktion ist auch in cmath vorhanden.

math.sinh(x)

Berechnet den Sinus Hyperbolicus von *x*. Das Ergebnis ist eine Gleitkommazahl.

```
>>> math.sinh(1.0)
1.1752011936438014
```

Die Funktion ist auch in cmath vorhanden.

math.tanh(x)

Berechnet den Tangens Hyperbolicus von *x*. Das Ergebnis ist eine Gleitkommazahl.

```
>>> math.tanh(1.0)
0.76159415595576485
```

Die Funktion ist auch in cmath vorhanden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Some people, when confronted with a problem, think ›I know, I'll use regular expressions.‹ Now they have two problems.« – Jamie W. Zawinski

15 Strings

In diesem Kapitel möchten wir einige Module vorstellen, die komfortable Funktionalität bereitstellen, die im engen Zusammenhang mit Strings steht.

top

15.1 Arbeiten mit Zeichenketten – string

Ursprünglich sollte das Modul `string` Funktionen enthalten, die zur Manipulation von Strings gedacht waren. Diese wurden jedoch im Laufe der Entwicklung von Python zu Methoden des Datentyps `str`, und das Modul `string` wurde damit weitestgehend obsolet. Dennoch enthält das Modul einige wichtige Konstanten und, was weitaus interessanter ist, eine Template Engine, die wir in Abschnitt 15.1.1 behandeln werden. Doch kommen wir zunächst zu den in `string` definierten Konstanten.

Konstanten

Die im Modul `string` enthaltenen Konstanten sind allesamt Strings, die Zeichen eines bestimmten Typs enthalten. Auf diese Weise lässt sich einfach testen, ob ein eingelesenes Zeichen dem gewünschten Typ angehört oder ob die Eingabe ungültig war. Besonders interessant wird dies im Hinblick auf Lokalisierungen, denn es gibt Zeichen, die nur in einigen wenigen Sprachen zu den Buchstaben oder Satzzeichen gehören, wie beispielsweise die deutschen Umlaute oder das spanische umgekehrte Fragezeichen.

Beachten Sie, dass sich Konstanten, die auf einer Lokalisierung basieren, automatisch ändern, wenn die Lokalisierung des Betriebssystems geändert wurde.

Wert	Bedeutung
`ascii_letters`	Enthält alle Buchstaben des ASCII-Standards, also alle Buchstaben des englischen Alphabets, jeweils in ihrer großen und kleinen Form. `'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`

ascii_lowercase	Wie ascii_letters, jedoch sind nur Kleinbuchstaben enthalten. `'abcdefghijklmnopqrstuvwxyz'`
ascii_uppercase	Wie ascii_letters, jedoch sind nur Großbuchstaben enthalten. `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`
digits	Enthält alle Ziffern von 0 bis 9. `'0123456789'`
hexdigits	Enthält alle hexadezimalen Ziffern von 0 bis F bzw. f. `'0123456789abcdefABCDEF'`
letters	Enthält alle Buchstaben der aktuellen Lokalisierung.
lowercase	Enthält alle Kleinbuchstaben der aktuellen Lokalisierung.
octdigits	Enthält alle oktalen Ziffern von 0 bis 7. `'01234567'`
punctuation	Enthält alle Satzzeichen der aktuellen Lokalisierung. `'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'`
printable	Enthält alle druckbaren Zeichen. Das ist eine Kombination aus digits, letters, punctuation und whitespace.
uppercase	Enthält alle Großbuchstaben der aktuellen Lokalisierung.
whitespace	Enthält alle Whitespace-Zeichen wie beispielsweise ein Tabulator- oder ein Leerzeichen. `'\t\n\x0b\x0c\r '`

Tabelle 15.1 Konstanten des Moduls »string«

Eingangs wurde bereits erwähnt, für welchen Zweck die aufgelisteten Konstanten gedacht sind. Im folgenden Beispiel soll ihre Verwendung demonstriert werden:

```
import string

c = raw_input("Ein englischer Kleinbuchstabe bitte: ")
if len(c) != 1:
    print "Ein Buchstabe, kein Roman"
elif c not in string.ascii_lowercase:
    print "Falsche Eingabe"
```

Nachdem eine Eingabe vom Benutzer gefordert wurde, wird sie auf ihre Länge geprüft, und anschließend wird getestet, ob der Buchstabe in string.ascii_lowercase enthalten ist.

top

15.1.1 Ein einfaches Template-System

Abgesehen von den soeben besprochenen Konstanten enthält das Modul string die Klasse Template, die ein simples Template-System darstellt. Ein solches Template-System ermöglicht es, einen String mit Platzhaltern zu versehen und diese später durch dynamisch erstellte Werte aufzufüllen.

Im Grunde genommen kann Pythons %-Syntax auch als Template-System angesehen werden:

```
>>> "%s ist %s" % ("Python", "gut")
'Python ist gut'
```

Diese Syntax ist an C angelehnt und sehr mächtig. Das Ziel der Template-Klasse dagegen ist es, ein möglichst einfaches Template-System zu implementieren, das zwar in seinem Funktionsumfang und seiner Flexibilität stark eingeschränkt, dafür aber sehr einfach zu benutzen ist.

Bei der Instanziierung der Template-Klasse wird der String mit den Platzhaltern übergeben:

```
>>> from string import Template
>>> t = Template("$was ist $wie")
```

Ein Platzhalter wird im String durch ein Dollar-Zeichen und einen darauf folgenden Bezeichner gekennzeichnet. In diesem Fall wurden die Platzhalter $was und $wie eingebettet.

Durch Aufruf der Methode substitute einer Template-Instanz können die Platzhalter durch Werte ersetzt werden:

```
>>> t.substitute(was="Python", wie="gut")
'Python ist gut'
>>> t.substitute(wie="mittel", was="Java")
'Java ist mittel'
```

Als Parameter der Methode substitute können beliebig viele *keyword arguments* übergeben werden, die jeweils dem Namen des Platzhalters entsprechen müssen. Zudem ist es möglich, der Methode ein Dictionary zu übergeben, das die Namen der Platzhalter und die einzutragenden Werte als Schlüssel/Wert-Paare enthält.

Die Instanzen, durch die die Platzhalter ersetzt werden, müssen keineswegs ausschließlich Strings sein, wie folgendes Beispiel zeigt:

```
>>> t.substitute(was=[1,2,3], wie=23)
'[1, 2, 3] ist 23'
```

Neben substitute existiert eine weitere Methode namens safe_substitute, die über die gleiche Schnittstelle wie substitute verfügt. Der Unterschied zwischen den beiden Methoden ist der, dass substitute eine Exception wirft, wenn bei einem Aufruf ein Platzhalter nicht ersetzt wird, während safe_substitute einen solchen Umstand schlicht ignoriert.

```
>>> t.substitute(was="Python")
Traceback (most recent call last):
  […]
KeyError: 'wie'
>>> t.safe_substitute(was="Python")
'Python ist $wie'
```

Abschließend noch zwei Bemerkungen zu den Platzhaltern im String. Betrachten Sie einmal folgendes Beispiel:

```
>>> t = Template("Alles Gute zum $gebten Geburtstag")
```

Der Platzhalter heißt eigentlich nur $geb, wird aber direkt von weiterem Text im String gefolgt, sodass der Name unglücklicherweise zu $gebten wird. Für solche Fälle ist es möglich, den Bezeichner des Platzhalters in geschweifte Klammern zu fassen:

```
>>> t = Template("Alles Gute zum ${geb}ten Geburtstag")
>>> t.substitute(geb=21)
'Alles Gute zum 21ten Geburtstag'
```

Problematisch ist außerdem, dass das Dollar-Zeichen sozusagen belegt ist, sodass Sie es nicht einfach so innerhalb des Strings verwenden können. Dieses Problem können Sie folgendermaßen umschiffen:

```
>>> t = Template("Python kostet $preis$$")
>>> t.substitute(preis=0)
'Python kostet 0$'
```

Ein doppeltes Dollar-Zeichen wird im fertigen String durch ein einzelnes Dollar-Zeichen ersetzt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Zehn Minuten!« – Edmund Stoiber

16 Datum und Zeit

In diesem Kapitel werden Sie die Python-Module kennenlernen, mit deren Hilfe Sie komfortabel mit Zeit- und Datumsangaben arbeiten können.

Python stellt dafür zwei Module zur Verfügung: `time` und `datetime`.

Das erste Modul, `time`, orientiert sich an den Funktionen, die von der zugrunde liegenden C-Bibliothek implementiert werden. Mit `datetime` werden Klassen zur Verfügung gestellt, mit denen sich in der Regel einfacher und angenehmer als mit Einzelfunktionen arbeiten lässt.

Wir werden im Folgenden beide Module und ihre Funktionen beleuchten.

top

16.1 Elementare Zeitfunktionen – time

Bevor wir uns mit den Funktionen des `time`-Moduls beschäftigen, müssen wir einige Begriffe einführen, die für das Verständnis, wie Zeitangaben verwaltet werden, erforderlich sind.

Das `time`-Modul setzt direkt auf den Zeitfunktionen der C-Bibliothek des Betriebssystems auf und speichert deshalb alle Zeitangaben als sogenannten *Unix-Timestamp*. Unix-Timestamps sind Zahlen, die einen Zeitpunkt dadurch identifizieren, dass sie die seit Beginn der sogenannten *Unix-Epoche* (auch nur *Epoch* genannt) vergangene Zeit in Sekunden angeben. Die Unix-Epoche begann am 01.01.1970 um 00:00 Uhr.

Ein Unix-Timestamp mit dem Wert 1190132696.0 markiert beispielsweise den 18.09.2007 um 18:24 Uhr und 56 Sekunden, da seit dem Beginn der Unix-Epoche bis zu diesem Zeitpunkt genau 1190132696.0 Sekunden vergangen sind.

Bei dem Umgang mit Zeitstempeln muss man zwei verschiedene Angaben unterscheiden: die *Lokalzeit* und die sogenannte *koordinierte Weltzeit*.

Die Lokalzeit ist abhängig von dem Standort der jeweiligen Uhr und bezieht sich darauf, was die Uhren an diesem Standort anzeigen müssen, um richtig zu gehen. Als koordinierte Weltzeit wird die Lokalzeit auf dem Null-Meridian verstanden, der unter anderem durch Großbritannien verläuft. Die koordinierte Weltzeit wird mit *UTC* für *Coordinated Universal Time* abgekürzt [Nein, die Abkürzung UTC für *Coordinated Universal Time* ist nicht fehlerhaft, sondern rührt daher, dass man einen Kompromiss zwischen der englischen Variante »Coordinated Universal Time« und der französischen Bezeichnung »Temps Universel Coordonné« finden wollte.] . Alle Lokalzeiten lassen sich relativ zur UTC angeben, indem man die Abweichung in Stunden nennt. Beispielsweise hat Mitteleuropa die Lokalzeit UTC+1, was bedeutet, dass unsere Uhren im Vergleich zu denen in England um eine Stunde vorgehen.

Die tatsächliche Lokalzeit kann noch von einem weiteren Faktor beeinflusst werden, der Sommer- bzw. Winterzeit. Diese auch mit *DST* für *Daylight Saving Time* (dt. *Sommerzeit*) abgekürzte Verschiebung ist von den gesetzlichen Regelungen der jeweiligen Region abhängig und hat in der Regel je nach Jahreszeit einen anderen Wert. Das `time`-Modul findet für den Programmierer heraus, welcher DST-Wert auf der gerade benutzten Plattform an dem aktuellen Standort der richtige ist, sodass wir uns darum nicht zu kümmern brauchen.

Neben der schon angesprochenen Zeitdarstellung durch Unix-Timestamps gibt es ein weiteres Format, das durch einen eigenen Datentyp namens `time_struct` implementiert wird. Instanzen des Typs `time_struct` haben neun Attribute, die wahlweise über einen Index oder ihren Namen angesprochen werden können. Die folgende Tabelle zeigt den genauen Aufbau des Datentyps: [Diese Begrenzung kommt durch den Wertebereich für die Unix-Timestamps zustande. Und ja, alle Programme, die auf Unix-Zeitstempel setzen, werden im Jahr 2038 ein Problem bekommen…]

Index	Attributname	Bedeutung und Wertebereich
0	tm_year	Die Jahreszahl des Zeitstempels Werte:2 1970-2038
1	tm_mon	Nummer des Monats Werte: 1-12
2	tm_mday	Nummer des Tags im Monat Werte: 1-31
3	tm_hour	Stunde der Uhrzeit des Zeitstempels Werte: 0-23
4	tm_min	Minute der Uhrzeit des Zeitstempels Werte: 0-59
5	tm_sec	Sekunde der Uhrzeit des aktuellen Zeitstempels Werte:3 0-61
6	tm_wday	Nummer des Wochentages Werte: 0-6 (0 entspricht Montag)
7	tm_yday	Nummer des Tages im Jahr Werte: 0-366
		Gibt an, ob der Zeitstempel durch die Sommerzeit angepasst wurde.

8	tm_isdst	Werte: 0 für »Nein«, 1 für »Ja« und –1 für »Unbekannt«

Tabelle 16.1 Aufbau des Datentyps struct_time

Allen Funktionen, die time_struct-Instanzen als Parameter erwarten, kann alternativ auch ein Tupel mit neun Elementen übergeben werden, das für die entsprechenden Indizes die gewünschten Werte enthält. [Es ist tatsächlich der Bereich von 0 bis 61, um sogenannte Schaltsekunden zu kompensieren. Schaltsekunden dienen dazu, die Ungenauigkeiten der Erdrotation bei Zeitangaben auszugleichen. Sie werden sich in der Regel nicht darum kümmern müssen.]

Nun gehen wir zu der Besprechung der Modulfunktionen und -attribute über.

Attribute

time.accept2dyear

Dieses Attribut enthält einen Wahrheitswert, der angibt, ob Jahreszahlen mit nur zwei Ziffern anstatt vier Ziffern angegeben werden können.

time.altzone

Speichert die Verschiebung der Lokalzeit von der UTC in Sekunden, wobei eine eventuell vorhandene Sommerzeit auch berücksichtigt wird. Liegt die aktuelle Zeitzone östlich vom Null-Meridian, ist der Wert von time.altzone positiv, liegt die lokale Zeitzone westlich davon, negativ.

Dieses Attribut sollte nur dann benutzt werden, wenn time.daylight nicht den Wert 0 hat.

time.daylight

Hat einen Wert, der von 0 verschieden ist, wenn es in der lokalen Zeitzone eine Sommerzeit gibt. Ist für den lokalen Standort keine Sommerzeit definiert, hat time.daylight den Wert 0. Die durch die Sommerzeit entstehende Verschiebung lässt sich mit time.altzone ermitteln.

time.struct_time

Referenz auf den in der Einleitung besprochenen Datentyp struct_time.

Sie können mit time.struct_time direkt Instanzen dieses Typs erzeugen, indem Sie dem Konstruktor eine Sequenz mit neun Elementen übergeben:

```
>>> t = time.struct_time((2007, 9, 18, 18, 24, 56, 0, 0,
0))
>>> t.tm_year
2007
```

time.timezone

Speichert die Verschiebung der Lokalzeit relativ zur UTC in Sekunden, wobei eine eventuell vorhandene Sommerzeit nicht berücksichtigt wird.

time.tzname

Enthält ein Tupel mit zwei Strings. Der erste String ist der Name der lokalen Zeitzone und der zweite der der lokalen Zeitzone mit Sommerzeit. Wenn die Lokalzeit keine Sommerzeit kennt, sollte das zweite Element des Tupels nicht verwendet werden.

```
>>> time.tzname
('Westeurop\xe4ische Normalzeit', 'Westeurop\xe4ische
Sommerzeit')
```

Funktionen

time.asctime([t])

Wandelt eine time.struct_time-Instanz oder ein Tupel mit neun Elementen in einen 24-Zeichen-String um. Die Form des resultierenden Strings zeigt das folgende Beispiel:

```
>>> time.asctime((1987, 7, 26, 10, 40, 0, 0, 0, 0))
'Mon Jul 26 10:40:00 1987'
```

Wird der optionale Parameter *t* nicht übergeben, gibt time.asctime einen 24–Zeichen-String für den aktuellen Zeitpunkt der Lokalzeit zurück.

time.clock()

Gibt die aktuelle Prozessorzeit zurück. Was dies konkret bedeutet, hängt von der gleichnamigen C-Funktion ab, die zu diesem Zweck aufgerufen wird.

Unter Unix gibt time.clock die Prozessorzeit zurück, die der Python-Prozess schon benutzt hat. Unter Windows ist es der zeitliche Abstand zum ersten Aufruf der Funktion.

Wenn Sie die Laufzeit Ihrer Programme analysieren wollen, ist time.clock in jedem Fall die richtige Wahl:

```
>>> start = time.clock()
>>> rechenintensive_funktion()
>>> ende = time.clock()
>>> print "Die Funktion lief %1.2f Sekunden" % (ende -
start)
Die Funktion lief 7.46 Sekunden
```

time.ctime([secs])

Wandelt den als Parameter übergebenen Unix-Timestamp in einen 24-Zeichen-String wie time.asctime um. Wird der optionale Parameter nicht übergeben oder hat er den Wert None, wird der aktuelle Zeitpunkt verwendet.

time.gmtime([secs])

Wandelt einen Unix-Timestamp in ein time.struct_time-Objekt um. Dabei wird immer die koordinierte Weltzeit benutzt, und das tm_isdst-Attribut des resultierenden Objekts hat immer den Wert 0.

Wird der Parameter *secs* nicht übergeben oder hat er den Wert None, wird der aktuelle Zeitstempel, wie er von time.time zurückgegeben wird, benutzt.

```
>>> time.gmtime()
(2007, 9, 18, 21, 40, 38, 1, 261, 0)
```

Das obige Beispiel wurde also nach UTC am 18.09.2007 um 21:40 Uhr ausgeführt.

time.localtime([secs])

Genau wie time.gmtime, aber wandelt den übergebenen Timestamp in eine Angabe der lokalen Zeitzone um.

time.mktime(t)

Wandelt eine time.struct_time-Instanz in einen Unix-Timestamp der Lokalzeit um. Der Rückgabewert ist eine Gleitkommazahl.

Die Funktionen time.localtime und time.mktime sind jeweils Umkehrfunktionen voneinander:

```
>>> t1 = time.localtime()
>>> t2 = time.localtime(time.mktime(t1))
>>> t1 == t2
True
```

time.sleep(secs)

Unterbricht die Programmausführung für die übergebene Zeitspanne. Der Parameter *secs* muss dabei eine Gleitkommazahl sein, die die Dauer der Unterbrechung in Sekunden angibt.

time.strftime(format[, t])

Wandelt die time.struct_time-Instanz *t* oder ein neunelementiges Tupel *t* in einen String um. Dabei wird mit dem ersten Parameter namens *format* ein String übergeben, der das gewünschte Format des Ausgabestrings enthält.

Ähnlich wie der Formatierungsoperator für Strings enthält der Format-String eine Reihe von Platzhaltern, die im Ergebnis durch die entsprechenden Werte ersetzt werden. Jeder Platzhalter besteht aus einem Prozentzeichen und einem Identifikationsbuchstaben. Die folgende Tabelle zeigt alle unterstützten Platzhalter:

Platzhalter	Bedeutung
%a	Lokale Abkürzung für den Namen des Wochentags
%A	Der komplette Name des Wochenstags in der lokalen Sprache
%b	Lokale Abkürzung for den Namen des Monats
%B	Der vollständige Name des Monats in der lokalen Sprache
%c	Das Format für eine angemessene Datums- und Zeitdarstellung auf der lokalen Plattform
%d	Nummer des Tages im aktuellen Monat. Ergibt einen String der Länge 2 im Bereich [01,31].
%H	Stunde im 24-Stunden-Format. Das Ergebnis hat immer zwei Ziffern und liegt im Bereich [00,23].
%I	Stunde im 12-Stunden-Format. Das Ergebnis hat immer zwei Ziffern und liegt im Bereich [01,12].
%j	Nummer des Tages im Jahr. Das Ergebnis hat immer drei Ziffern und liegt im Bereich [001, 366].
%m	Nummer des Monats bestehend aus zwei Ziffern im Bereich [01,12]
%M	Minute als Zahl mit zwei Ziffern. Liegt immer im Bereich [00,59].
%p	Die lokale Entsprechung für AM bzw. PM
%S	Sekunde als Zahl mit zwei Ziffern. Liegt immer im Bereich [00,61].
%U	Nummer der aktuellen Woche im Jahr, wobei der Sonntag als erster Tag der Woche betrachtet wird. Das Ergebis hat immer zwei Ziffern und liegt im Bereich [01,53]. Der Zeitraum am Anfang eines Jahres vor dem ersten Sonntag wird als 0. Woche gewertet.
%w	Nummer das aktuellen Tages in der Woche. Sonntag wird als 0. Tag betrachtet. Das Ergebnis

	liegt im Bereich [0,6].
%W	Wie %U, nur dass statt des Sonntags der Montag als erster Tag der Woche betrachtet wird.
%x	Datumsformat der lokalen Plattform
%X	Zeitformat der lokalen Plattform
%y	Jahr ohne Jahrhundertangabe. Das Ergebnis besteht immer aus zwei Ziffern und liegt im Bereich [00,99].
%Y	Komplette Jahreszahl mit Jahrhundertangabe
%Z	Name der lokalen Zeitzone oder ein leerer String, wenn keine lokale Zeitzone festgelegt wurde.
%%	Ergibt ein Prozentzeichen »%« im Resultatstring.

Tabelle 16.2 Übersicht über alle Platzhalter der time.strftime-Funktion

Mit dem folgenden Ausdruck erzeugen Sie beispielsweise eine Ausgabe des aktuellen Zeitpunkts in einem für Deutschland üblichen Format zu erhalten:

```
>>> time.strftime("%d.%m.%Y um %H:%M:%S Uhr")
'19.09.2007 um 00:21:17 Uhr'
```

time.strptime(string[, format])

Mit `time.strptime` können Sie einen Zeit-String wieder in eine `time.struct_time`-Instanz umwandeln. Der Parameter *format* gibt dabei das Format an, in dem der String die Zeit enthält. Den Aufbau solcher Format-Strings ist der gleiche wie bei `time.strftime`.

```
>>> zeit_string = '19.09.2007 um 00:21:17 Uhr'
>>> time.strptime(zeit_string, "%d.%m.%Y um %H:%M:%S Uhr")
(2007, 9, 19, 0, 21, 17, 2, 262, -1)
```

Wird der optionale Parameter *format* nicht angegeben, wird der Standardwert "%a %b %d %H:%M:%S %Y" verwendet. Dies entspricht dem Ausgabeformat von `time.ctime`.

time.time()

Gibt den aktuellen Unix-Zeitstempel in UTC als Gleitkommazahl zurück.

Beachten Sie hierbei, dass nicht alle Systeme eine höhere Auflösung als eine Sekunde unterstützen und der Nachkommateil somit nicht unbedingt verlässlich ist.

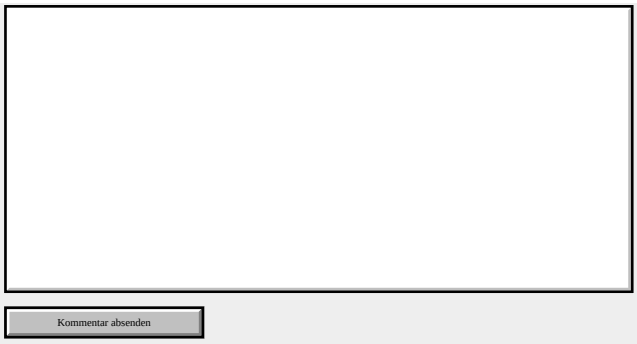
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Aber ich kann dir nur die Tür zeigen, durchgehen musst du ganz allein. Tank, das Sprungprogramm!« – Morpheus in Matrix

17 Schnittstelle zum Betriebssystem

Um Ihre Programme mit dem Betriebssystem interagieren zu lassen, auf dem sie ausgeführt werden, benötigen Sie Zugriff auf dessen Funktionen. Ein Problem dabei ist, dass sich die verschiedenen Betriebssysteme teilweise sehr stark in ihrem Funktionsumfang und in der Art unterscheiden, wie die vorhandenen Operationen zu benutzen sind. Python wurde aber von Grund auf als plattformübergreifende Sprache konzipiert. Um auch Programme, die auf Funktionen des Betriebssystems zurückgreifen müssen, auf möglichst vielen Plattformen ohne Änderungen ausführen zu können, hat man eine Schnittstelle geschaffen, die einheitlichen Zugriff auf Betriebssystemfunktionen bietet. Im Klartext bedeutet dies, dass Sie durch die Benutzung dieser einheitlichen Schnittstelle Programme schreiben können, die plattformunabhängig bleiben, selbst wenn sie auf Betriebssystemfunktionen zurückgreifen.

Die Schnittstelle wird durch das Modul os implementiert, mit dem wir uns im weiteren Verlauf des Kapitels beschäftigen werden.

top

17.1 Funktionen des Betriebssystems – os

Mit dem os-Modul können Sie auf mehrere Klassen von Operationen zugreifen. Da die gebotenen Funktionen sehr umfangreich sind und zu einem großen Teil nur selten gebraucht werden, beschränken wir uns hier auf eine Teilmenge, die sich in folgende Kategorien einteilen lässt:

- Zugriff auf den Prozess, in dem unser Python-Programm läuft, und auf andere Prozesse
- Zugriff auf das Dateisystem
- Informationen über das Betriebssystem

Außerdem stellt das Submodul os.path nützliche Operationen für die Manipulation und Verarbeitung von Pfadnamen bereit.

Das Modul os hat eine eigene Exception-Klasse namens os.error. Immer wenn Sie Fehler innerhalb dieses Moduls abfangen möchten, können Sie dies über os.error tun. Ein alternativer Name für die Fehlerklasse ist OSError.

Wir werden nun eine Auswahl von Funktionen der drei Kategorien besprechen.

top

17.1.1 Zugriff auf den eigenen Prozess und andere Prozesse

os.environ

Diese Konstante enthält ein Dictionary, das die Umgebungsvariablen speichert, die für unser Programm vom Betriebssystem bereitgestellt wurden. Beispielsweise lässt sich auf vielen Plattformen mit os.environ['HOME'] der Pfad des Ordners für die Dateien des aktiven Benutzers ermitteln. Die folgenden Beispiele zeigen den Wert von os.environ['HOME'] auf einem Windows- und einem Linux-Rechner:

```
>>> print os.environ['HOME']
C:\Dokumente und Einstellungen\revelation

>>> print os.environ['HOME']
/home/revelation
```

Sie können die Werte des os.environ-Dictionarys auch verändern, was allerdings auf bestimmten Plattformen zu Problemen führen kann und deshalb mit Vorsicht zu genießen ist.

os.getpid()

Jeder laufende Prozess hat eine eindeutige Identifikationsnummer, die sich mit os.getpid() ermitteln lässt:

```
>>> os.getpid()
1360
```

Diese Funktion ist nur unter Windows- und Unix-Systemen verfügbar.

os.system(cmd)

Mit os.system können Sie beliebige Kommandos des Betriebssystems ausführen, so als ob Sie es in einer echten Konsole tun würden. Beispielsweise lassen wir uns mit folgendem

Beispiel einen neuen Ordner mit dem Namen *test_ordner* über das »mkdir«-Kommando anlegen:

```
>>> os.system("mkdir test_ordner")
0
```

Der Rückgabewert von `os.system` ist der Statuscode, mit dem das aufgerufene Programm beendet wurde, in diesem Fall `0`.

Ein Problem der `os.system`-Funktion ist, dass die Ausgabe des aufgerufenen Programms nicht ohne Weiteres ermittelt werden kann. Für solche Zwecke eignet sich die folgende `os.popen`-Funktion.

os.popen(command[, mode[, bufsize]])

Mit der Funktion `os.popen` können beliebige Befehle wie auf einer Kommandozeile des Betriebssystems ausgeführt werden. Die Funktion gibt ein Dateiobjekt zurück, mit dem auf die Ausgabe des ausgeführten Programms zugegriffen werden kann. Mit dem Parameter `mode` wird wie bei der Built-in Function open angegeben, ob das Dateiobjekt lesend (`"r"`) oder schreibend (`"w"`) geöffnet werden soll. Bei schreibendem Zugriff können auch Daten an das laufende Programm übergeben werden.

Im folgenden Beispiel nutzen wir das Windows-Kommando »dir«, um eine Liste der Dateien und Ordner unter *C:* zu erzeugen:

```
>>> ausgabe = os.popen("dir /B C:\\")
>>> dateien = [zeile.strip() for zeile in ausgabe]
>>> dateien
['AUTOEXEC.BAT', 'CONFIG.SYS', 'Dokumente und
Einstellungen',
'Programme', 'Python25', 'WINDOWS']
```

Die genaue Bedeutung von `mode` und `bufsize` können Sie in Abschnitt 9.3, »Dateien«, nachlesen.

top

17.1.2 Zugriff auf das Dateisystem ▲

Mit den nachfolgend beschriebenen Funktionen können Sie sich wie mit einer Shell durch das Dateisystem bewegen, Informationen zu Dateien und Ordnern ermitteln, diese umbenennen, löschen oder erstellen.

Sie werden oft einen sogenannten *Pfad* (engl. *path*) als Parameter an die beschriebenen Funktionen übergeben können. Dabei unterscheiden wir zwischen absoluten und relativen Pfaden, wobei Letztere sich auf das aktuelle Arbeitsverzeichnis beziehen.

Sofern nichts anderes angemerkt ist, werden Pfade als `str`- oder `unicode`-Instanzen übergeben.

os.access(path, mode)

Mit `os.access` kann überprüft werden, welche Rechte das laufende Python-Programm für den Pfad *path* hat. Der Parameter *mode* gibt dabei eine Bitmaske an, die die zu überprüfenden Rechte enthält.

Folgende Werte können einzeln oder mithilfe des bitweisen ODERs zusammengefasst übergeben werden:

Konstante	Bedeutung
`os.F_OK`	Prüft, ob der Pfad überhaupt exisitert.
`os.R_OK`	Prüft, ob der Pfad gelesen werden darf.
`os.W_OK`	Prüft, ob der Pfad geschrieben werden darf.
`os.X_OK`	Prüft, ob der Pfad ausführbar ist.

Tabelle 17.1 Wert für den mode-Parameter von os.access

Der Rückgabewert von `os.access` ist `True`, wenn alle für *mode* übergebenen Werte auf den Pfad zutreffen, und `False`, wenn mindestens ein Zugriffsrecht für das Programm nicht gilt.

os.chdir(path)

Setzt das aktuelle Arbeitsverzeichnis auf den mit *path* übergebenen Pfad.

os.getcwd()

Gibt einen String zurück, der den Pfad des aktuellen Arbeitsverzeichnisses (*Current Working Directory*) enthält.

os.getcwdu()

Wie `os.getcwd`, gibt aber eine `unicode`-Instanz zurück.

os.chmod(path, mode)

Setzt die Zugriffsrechte der Datei oder des Ordners unter dem übergebenen Pfad. *mode* ist dabei eine dreistellige Oktalzahl, bei der jede Ziffer die Zugriffsrechte für eine Benutzerklasse angibt. Die erste Ziffer steht für den *Besitzer* der Datei, die zweite für seine *Gruppe* und die dritte für alle *anderen Benutzer*.

Dabei sind die einzelnen Ziffern Summen aus den folgenden drei Werten:

Wert	Beschreibung
1	Ausführen
2	Schreiben
4	Lesen

Tabelle 17.2 Zugriffsflags für os.chmod

Wenn Sie nun beispielsweise den nachstehenden `os.chmod`-Aufruf durchführen, erteilen Sie dem Besitzer vollen Lese- und Schreibzugriff:

```
>>> os.chmod("eine_datei", 0640)
```

Ausführen kann er die Datei aber trotzdem nicht. Die restlichen Benutzer seiner Gruppe dürfen die Datei auslesen, aber nicht verändern, und für alle anderen bleibt aufgrund der fehlenden Leseberechtigung auch der Inhalt der Datei verborgen.

Beachten Sie die führende `0` bei den Zugriffsrechten, die das Literal einer Oktalzahl einleitet.

Diese Funktion ist nur unter Windows- und Unix-Systemen verfügbar.

os.listdir(path)

Gibt eine Liste zurück, die alle Dateien und Unterordner des Ordners angibt, der mit *path* übergeben wurde. Diese Liste enthält nicht die speziellen Einträge für das Verzeichnis selbst (`"."`) und das nächsthöhere Verzeichnis (`".."`).

Die Elemente der Liste haben den gleichen Typ wie der übergebene *path*-Parameter, also entweder `str` oder `unicode`.

os.mkdir(path[, mode])

Legt einen neuen Ordner in dem mit *path* übergebenen Pfad an. Der optionale Parameter *mode* gibt dabei eine Bitmaske an, die die Zugriffsrechte für den neuen Ordner festlegt. Standardmäßig wird für *mode* die Oktalzahl `0777` verwendet (siehe zu *mode* auch `os.chmod`).

Ist der angegebene Ordner bereits vorhanden, wird eine `os.error`-Exception geworfen.

Beachten Sie, dass `os.mkdir` nur dann den neuen Ordner erstellen kann, wenn alle übergeordneten Verzeichnisse bereits existieren:

```
>>> os.mkdir(r"C:\Diesen\Pfad\gibt\es\so\noch\nicht")
[…]
WindowsError: [Error 3] Das System kann den angegebenen
Pfad nicht
finden: 'C:\\Diesen\\Pfad\\gibt\\es\\so\\noch\\nicht'
```

Wenn Sie bei Bedarf die Erzeugung der kompletten Ordnerstruktur wünschen, verwenden Sie `os.makedirs`.

os.makedirs(path[, mode])

Wie `os.mkdir`; erzeugt aber im Gegensatz dazu die komplette Verzeichnisstruktur inklusive aller übergeordneten Verzeichnisse. Damit funktioniert auch folgendes Beispiel:

```
>>> os.makedirs(r"C:\Diesen\Pfad\gibt\es\so\noch\nicht")
>>>
```

Wenn der übergebene Ordner schon existiert, wird eine `os.error`-Exception geworfen.

os.remove(path)

Entfernt die mit *path* angegebene Datei aus dem Dateisystem. Wird statt eines Pfads zu einer Datei ein Pfad zu einem Ordner übergeben, wirft `os.remove` eine `os.error`-Exception (siehe dazu `os.rmdir`).

Beachten Sie bitte, dass es unter Windows-Systemen nicht möglich ist, eine Datei zu löschen, die gerade benutzt wird. In diesem Fall wird ebenfalls eine Exception geworfen.

os.removedirs(path)

Löscht eine ganze Ordnerstruktur. Dabei löscht es von der tiefsten bis zur höchsten Ebene nacheinander alle Ordner, sofern diese leer sind. Kann der tiefste Ordner nicht gelöscht werden, wird eine `os.error`-Exception geworfen. Fehler, die beim Entfernen der Elternverzeichnisse auftreten, werden ignoriert.

Wenn Sie beispielsweise

```
>>> os.removedirs(r"C:\Irgend\ein\Beispielpfad")
```

schreiben, wird zuerst versucht, den Ordner *C:\Irgend\ein\Beispielpfad* zu löschen. Wenn dies erfolgreich war, wird *C:\Irgend\ein* entfernt und bei Erfolg anschließend *C:\Irgend*.

os.rename(src, dst)

Benennt die mit *src* angegebene Datei oder den Ordner in *dst* um. Wenn unter dem Pfad *dst* bereits eine Datei oder ein Ordner existiert, wird `os.error` geworfen.

Achtung

Auf Unix-Systemen wird eine bereits unter dem Pfad *dst*

erreichbare Datei ohne Meldung überschrieben, wenn Sie `os.rename` aufrufen. Bei bereits existierenden Ordnern wird aber weiterhin eine Exception erzeugt.

Die Methode `os.rename` funktioniert nur dann, wenn bereits alle übergeordneten Verzeichnisse von *dst* existieren. Wenn Sie die Erzeugung der nötigen Verzeichnisstruktur wünschen, benutzen Sie stattdessen `os.renames`.

os.renames(src, dst)

Wie `os.rename`, legt aber bei Bedarf die Verzeichnisstruktur des Zielpfads an. Außerdem wird nach dem Benennungsvorgang versucht, den *src*-Pfad mittels `os.removedirs` von leeren Ordnern zu reinigen.

os.rmdir(path)

Entfernt den übergebenen Ordner aus dem Dateisystem oder wirft `os.error`, wenn der Ordner nicht existiert.

os.tmpfile()

Legt eine temporäre Datei an und gibt ein geöffnetes Dateiobjekt für sie zurück. Nach dem Schließen der Datei wird sie automatisch gelöscht.

Diese Funktion ist äußerst praktisch, wenn Sie kurz Daten auf die Festplatte auslagern möchten.

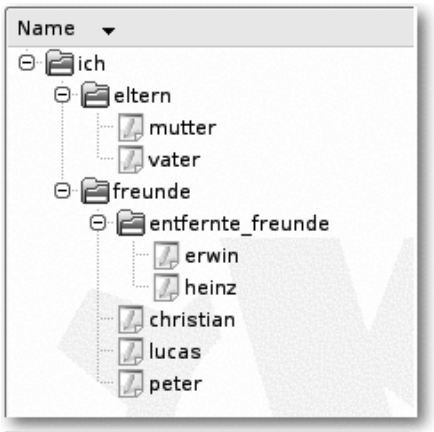
os.walk(top[, topdown=True[, onerror=None]])

Eine sehr komfortable Möglichkeit, einen Verzeichnisbaum komplett zu durchlaufen, stellt die Funktion `os.walk` bereit. Der Parameter *top* gibt dabei die Wurzel des zu durchlaufenden Teilbaums an. Die Iteration geht dabei so vonstatten, dass `os.walk` für den Ordner *top* und für jeden seiner Unterordner ein Tupel mit drei Elementen zurückgibt. Ein solches Tupel kann beispielsweise folgendermaßen aussehen:

```
('ein\\pfad', ['ordner1'], ['datei1', 'datei2'])
```

Das erste Element ist dabei der Pfad zu dem Unterordner inklusive des Pfads relativ zu *top*, das zweite Element enthält eine Liste mit allen Ordnern, die der aktuelle Unterordner selbst enthält, und das letzte Element speichert alle Dateien des Unterordners.

Um dies genau zu verstehen, betrachten wir einen Beispielverzeichnisbaum:



Hier klicken, um das Bild zu vergrößern

Abbildung 17.1 Beispielverzeichnisbaum

Wir nehmen an, dass unser aktuelles Arbeitsverzeichnis der Ordner ist, der dem Ordner *ich* direkt übergeordnet ist.

Dann könnten wir uns einmal die Ausgabe von `os.walk` für das Verzeichnis *ich* ansehen:

```
>>> for t in os.walk("ich"):
    print t

('ich', ['freunde', 'eltern'], [])
('ich\\freunde', ['entfernte_freunde'], ['peter',
'christian', 'lucas'])
('ich\\freunde\\entfernte_freunde', [], ['heinz',
'erwin'])
('ich\\eltern', [], ['vater', 'mutter'])
```

Wie Sie sehen, wird für jeden Ordner ein Tupel erzeugt, das die beschriebenen Informationen enthält. Die doppelten Backslashs "\\" rühren daher, dass das Beispiel auf einem Windows-Rechner agiusgeführt wurde und Backslashs innerhalb von String-Literalen als Escape-Sequenz geschrieben werden müssen.

Sie können die in dem Tupel gespeicherten Listen auch bei Bedarf anpassen, um beispielsweise die Reihenfolge zu verändern, in der die Unterverzeichnisse des aktuellen Verzeichnisses besucht werden sollen, oder wenn Sie Änderungen wie das Hinzufügen oder Löschen von Dateien und Ordnern vorgenommen haben.

Mit dem optionalen Parameter *topdown*, dessen Standardwert `True` ist, kann man festlegen, wo mit dem Durchlaufen begonnen werden soll. Bei der Standardeinstellung wird in dem Verzeichnis begonnen, das im Verzeichnisbaum der Wurzel am nächsten steht, im Beispiel *ich*. Wird *topdown* auf `False` gesetzt, geht `os.walk` genau umgekehrt vor und beginnt mit dem am tiefsten verschachtelten Ordner. In unserem Beispielbaum ist das *ich/freunde/entfernte_freunde*:

```
>>> for t in os.walk("ich", False):
    print t

('ich\\freunde\\entfernte_freunde', [], ['heinz',
'erwin'])
('ich\\freunde', ['entfernte_freunde'], ['peter',
'christian', 'lucas'])
('ich\\eltern', [], ['vater', 'mutter'])
('ich', ['freunde', 'eltern'], [])
```

Zu guter Letzt kann mit dem letzten Parameter namens *onerror* festgelegt werden, wie die Funktion sich verhalten soll, wenn ein Fehler beim Ermitteln des Inhalts eines Verzeichnisses auftritt. Wenn Sie *onerror* nicht auf dem Standardwert `None`, der keine Operation vorsieht, belassen wollen, müssen Sie eine Referenz auf eine Funktion, die einen Parameter erwartet, übergeben. Im Fehlerfall wird dann diese Funktion mit einer `os.error`-Instanz, die den Fehler beschreibt, als Parameter aufgerufen.

Wichtig

Wenn Sie mit einem Betriebssystem arbeiten, das symbolische Links auf Verzeichnisse unterstützt, werden diese beim Durchlaufen der Struktur nicht mit berücksichtigt. Dieses Verhalten ist deshalb sinnvoll, weil sonst schwierig zu vermeidende Endlosschleifen entstehen können.

Achtung

Wenn Sie wie in unserem Beispiel einen relativen Pfadnamen angeben, dürfen Sie das aktuelle Arbeitsverzeichnis nicht während des Durchlaufens mittels `os.walk` verändern.

Wenn Sie es dennoch tun, kann dies zu nicht definiertem Verhalten führen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Don't interrupt me while I'm interrupting« – Winston S. Churchill

18 Parallele Programmierung

Dieses Kapitel wird Sie in die Programmierung mit sogenannten *Threads* einführen, die es ermöglichen, mehrere Aufgaben gleichzeitig auszuführen. Bevor wir allerdings mit den technischen Details und Beispielprogrammen beginnen können, müssen einige Begriffe eingeführt werden, und Sie müssen die prinzipielle Arbeitsweise moderner Betriebssysteme verstehen.

top

18.1 Prozesse, Multitasking und Threads

Im Folgenden werden die Begriffe *Programm* und *Prozess* synonym für ein laufendes Programm verwendet.

Wir sind als Benutzer moderner Computer gewohnt, dass ein Rechner mehrere Programme gleichzeitig ausführen kann. Beispielsweise schreiben wir eine E-Mail, während im Hintergrund das letzte Urlaubsvideo in ein anderes Format umgewandelt wird und eine MP3-Software unseren Lieblingssong aus den Computerlautsprechern ertönen lässt. Abbildung 18.1 zeigt eine typische Arbeitssitzung, wobei jeder Kasten für ein laufendes Programm steht. Die Länge der Kästen entlang der Zeitachse zeigt an, wie lange der jeweilige Prozess läuft.

Hier klicken, um das Bild zu vergrößern

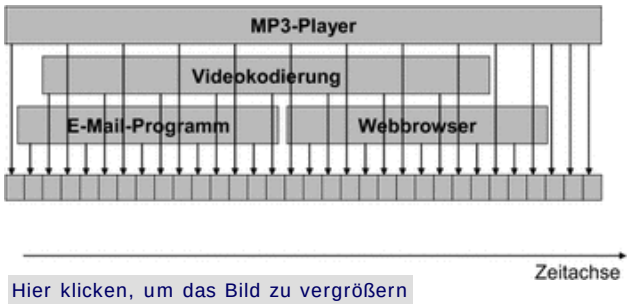
Abbildung 18.1 Mehrere Prozesse laufen gleichzeitig ab

Faktisch kann ein Computer aber nur genau eine einzige Aufgabe zu einem bestimmten Zeitpunkt übernehmen und nicht mehrere gleichzeitig. Selbst bei modernen Prozessoren mit mehr als einem Kern oder bei Rechnern mit vielen Prozessoren ist die Anzahl der gleichzeitig ausführbaren Programme durch die Anzahl der Kerne bzw. Prozessoren beschränkt. Wie ist es also möglich, dass das einleitend beschriebene Szenario auch auf einem Computer mit nur einem Prozessor, der nur einen einzigen Kern besitzt, funktioniert?

Der dahinter stehende Trick ist im Grunde sehr einfach, denn man versteckt die Limitierung der Maschine geschickt vor dem Benutzer, indem man ihm vorgaukelt, es würden mehrere Programme simultan laufen. Dies wird dadurch erreicht, dass man jedem Programm ganz kurz die Kontrolle über den Prozessor zuteilt, es also laufen lässt. Nach Ablauf der sogenannten *Zeitscheibe* wird dem Programm die Kontrolle wieder entzogen, wobei sein aktueller Zustand gespeichert wird. Nun kann dem nächsten Programm eine Zeitscheibe zugeteilt werden. In der Zeit, in der ein Programm darauf wartet, eine Zeitscheibe zugeteilt zu bekommen, wird es als *schlafend* bezeichnet.

Man kann sich die Arbeit eines Computers so vorstellen, dass in rasender Geschwindigkeit alle laufenden Programme geweckt, für eine kurze Zeit ausgeführt und dann wieder schlafen gelegt werden. Durch die hohe Geschwindigkeit des Umschaltens zwischen den Prozessen wird dies vom Benutzer nicht wahrgenommen. Die Verwaltung der Prozesse und ihrer Zeitscheiben wird von den modernen Betriebssystemen übernommen, die deshalb auch *Multitasking-Systeme* (dt. *Mehrprozessbetriebssysteme*) genannt werden.

Die korrekte Darstellung unseres anfänglichen Beispiels müsste also eher wie in Abbildung 18.2 gezeigt aussehen. Dabei symbolisiert jedes kleine Kästchen innerhalb des Blocks »Reale Prozessorbelegung« eine Zeitscheibe:



Hier klicken, um das Bild zu vergrößern

Abbildung 18.2 Die Prozesse wechseln sich ab und laufen nicht gleichzeitig

Innerhalb eines Prozesses selbst kann aber weiterhin nur eine Aufgabe zur selben Zeit verarbeitet werden, da das Programm linear abgearbeitet wird. In vielen Situationen ist es aber erforderlich, dass ein Programm mehrere Operationen zeitgleich durchführt. Beispielsweise sollte die Benutzeroberfläche während einer aufwendigen Berechnung nicht blockieren, sondern den aktuellen Status anzeigen, und der Benutzer sollte die Berechnung gegebenenfalls abbrechen können. Ein anderes Beispiel ist ein Webserver, der während der Verarbeitung einer Client-Anfrage auch noch für weitere Zugriffe verfügbar sein muss.

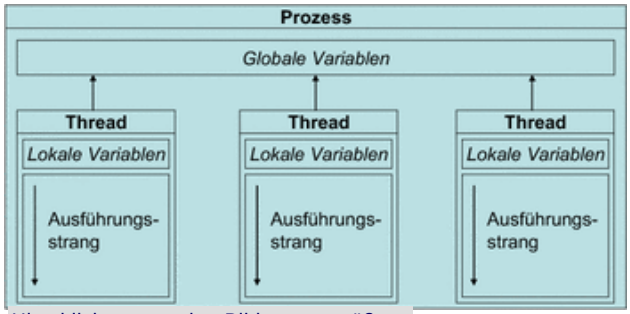
Es ist zwar möglich, die Beschränkung auf nur eine Operation zur selben Zeit durch die Erzeugung weiterer Prozesse zu umgehen, aber um Daten zwischen verschiedenen Prozessen auszutauschen, muss relativ viel Aufwand getrieben werden, weil jeder Prozess seine eigenen Variablen hat, die von den anderen Prozessen abgeschirmt sind.

Eine befriedigende Lösung für das Problem liefern sogenannte

Threads. Ein Thread (dt. *Faden*) ist ein Ausführungsstrang innerhalb eines Prozesses. Standardmäßig besitzt jeder Prozess genau einen Thread, der eben die Ausführung des Prozesses organisiert.

Nun kann ein Prozess aber auch mehrere Threads starten, die dann durch das Betriebssystem wie Prozesse scheinbar gleichzeitig ausgeführt werden. Der Vorteil von Threads gegenüber Prozessen besteht darin, dass sich die Threads eines Prozesses denselben Speicherbereich für globale Variablen teilen. Wenn also in einem Thread eine globale Variable verändert wird, ist der neue Wert auch sofort für alle anderen Threads des Prozesses sichtbar. [Um Fehler zu vermeiden, müssen solche Zugriffe in mehreren Threads speziell mit sogenannten *Critical Sections* abgesichert werden. Wir werden diese Thematik im Laufe des Kapitels noch ausführlicher behandeln.] Demgegenüber hat jeder Thread seine eigenen lokalen Variablen. Außerdem ist die Verwaltung von Threads für das Betriebssystem weniger aufwendig als die Verwaltung von Prozessen. Deshalb werden Threads auch *Leichtgewichtprozesse* genannt.

Die Threads in einem Prozess kann man sich folgendermaßen vorstellen:



Hier klicken, um das Bild zu vergrößern

Abbildung 18.3 Ein Prozess mit drei Threads

Nach dieser theoretischen Einführung werden wir uns der Programmierung mit Threads in Python zuwenden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Gauß wusste alles« – Ulrich Kaiser

19 Datenspeicherung

In den folgenden Abschnitten werden wir uns mit der permanenten Speicherung von Daten in den verschiedensten Formaten befassen. Das schließt unter anderem komprimierte Archive, XML-Dateien und Datenbanken mit ein.

top

19.1 Komprimierte Dateien lesen und schreiben – gzip

Mit dem Modul gzip der Standardbibliothek können Sie auf sehr einfache Weise Dateien verarbeiten, die mit der zlib-Bibliothek [Die zlib ist eine quelloffene Kompressionsbibliothek, die unter anderem vom Unix-Programm gzip verwendet wird. Nähere Informationen können Sie der Homepage unter **http://www.zlib.net** entnehmen.] erstellt wurden. Außerdem können Sie damit selbst zlib-komprimierte Dateien erzeugen.

Das Modul stellt eine Funktion namens open bereit, die sich in ihrer Verwendung an die Built-in Function open anlehnt:

gzip.open(filename[, mode[, compresslevel])

Die Funktion gzip.open gibt ein Objekt zurück, das wie ein ganz normales Dateiobjekt verwendet werden kann.

Die Parameter *filename* und *mode* sind gleichbedeutend mit denen der Built-in Function open.

Mit dem letzten Parameter, *compresslevel*, kann angegeben

werden, wie stark die Daten beim Schreiben in die Datei komprimiert werden sollen. Erlaubt sind Ganzzahlen von 0 bis 9, wobei 0 für die schlechteste und 9 für die beste Kompressionsstufe steht. Je höher die Kompressionsstufe ist, desto mehr Rechenzeit ist auch für das Komprimieren der Daten erforderlich. Wird der Parameter *compress level* nicht angegeben, verwendet gzip standadmäßig die beste Kompression.

```
>>> import gzip
>>> f = gzip.open("testdatei.gz", "wb")
>>> f.write("Hallo Welt")
>>> f.close()
>>> g = gzip.open("testdatei.gz")
>>> g.read()
'Hallo Welt'
```

In dem Beispiel schreiben wir einen einfachen String in die Datei *testdatei.gz* und lesen ihn anschließend wieder aus.

Andere Module für den Zugriff auf komprimierte Daten

Es existieren in der Standardbibliothek von Python noch weitere Module, die den Zugriff auf komprimierte Daten erlauben. Aus Platzgründen muss hier auf eine ausführliche Besprechung verzichtet werden.

Die folgende Tabelle gibt einen Überblick über alle Module, die komprimierte Daten verwalten können:

Modul	Beschreibung
zlib	Eine Low-Level-Bibliothek, die direkten Zugriff auf die Funktionen der zlib ermöglicht. Mit ihr ist es unter anderem möglich, Strings zu komprimieren oder zu entpacken. Das Modul gzip greift intern auf das Modul zlib zurück.
gzip	Beschreibung siehe oben.
bz2	Bietet komfortablen Zugriff auf Daten, die mit dem bzip2-Algorithmus komprimiert wurden, und ermöglicht es, neue komprimierte Dateien zu erzeugen. Auch bz2 implementiert ein Dateiobjekt, das genauso zu handhaben ist wie die Objekte, die die Built-in Function open zurückgibt. In der Regel ist die Kompression von bzip2 der von zlib in puncto Kompressionsrate überlegen.
zipfile	Ermöglicht den Zugriff auf ZIP-Archive, wie sie beispielsweise von dem bekannten Programm *WinZip* erstellt werden. Auch die Manipulation und Erzeugung neuer Archive ist möglich. Das Modul zipfile ist sehr umfangreich und mächtig und in jedem Fall einen näheren Blick wert.
tarfile	Implementiert Funktionen und Klassen, um die in der Unix-Welt weit verbreiteten tar-Archive zu lesen oder zu schreiben.

Tabelle 19.1 Übersicht über Pythons Kompressionsmodule

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

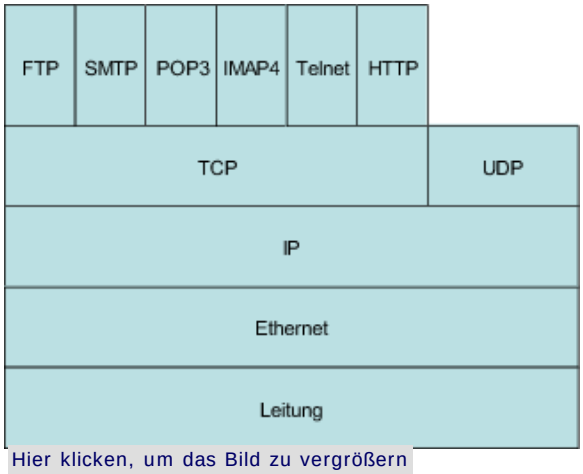
Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Alle reden von Kommunikation, aber die wenigsten haben sich etwas mitzuteilen.« – Hans Magnus Enzensberger

20 Netzwerkkommunikation

Nachdem wir uns ausführlich mit der Speicherung von Daten in Dateien verschiedener Formate oder Datenbanken beschäftigt haben, folgt nun ein Kapitel, das sich mit einer weiteren interessanten Programmierdisziplin beschäftigt: mit der Netzwerkprogrammierung.

Grundsätzlich lässt sich das Themenfeld der Netzwerkkommunikation in mehrere sogenannte *Protokollebenen* (engl. *layer*) aufteilen. Abbildung 20.1 zeigt eine stark vereinfachte Version des *OSI-Schichtenmodells*, das die Hierarchie der verschiedenen Protokollebenen veranschaulicht.



Hier klicken, um das Bild zu vergrößern

Abbildung 20.1 Netzwerkprotokolle

Das rudimentärste Protokoll steht in der Grafik ganz unten. Dabei handelt es sich um die blanke Leitung, über die die Daten in Form von elektrischen Impulsen übermittelt werden. Darauf aufbauend existieren etwas abstraktere Protokolle wie Ethernet und IP. Doch der für Anwendungsprogrammierer eigentlich interessante Teil fängt erst oberhalb des IP-Protokolls an, nämlich bei den Transportprotokollen TCP und UDP. Beide Protokolle werden wir ausführlich im Zusammenhang mit Sockets im nächsten Abschnitt besprechen.

Die Protokolle, die auf TCP aufbauen, sind am weitesten abstrahiert und deshalb für uns ebenfalls sehr interessant. In diesem Buch werden wir folgende Protokolle ausführlich behandeln:

Protokoll	Beschreibung	Modul	Abschnitt
TCP	Grundlegendes verbindungsorientiertes Netzwerkprotokoll	`socket`	20.1.3
UDP	Grundlegendes verbindungsloses Netzwerkprotokoll	`socket`	20.1.2
FTP	Dateiübertragung	`ftplib`	20.4
SMTP	Versenden von E-Mails	`smtplib`	20.5.1
POP3	Abholen von E-Mails	`poplib`	20.5.2
IMAP4	Abholen von E-Mails	`imaplib`	20.5.3
Telnet	Terminalemulation	`telnetlib`	20.6
HTTP	Übertragen von Textdateien, beispielsweise Webseiten	`urllib`	20.2

Tabelle 20.1 Netzwerkprotokolle

Beachten Sie, dass es auch abstrakte Protokolle gibt, die auf UDP aufbauen, beispielsweise NFS (*Network File System*). Wir werden in diesem Buch aber ausschließlich auf TCP basierende Protokolle behandeln.

Wir werden im ersten Unterkapitel zunächst eine ganz grundlegende Einführung in das systemnahe Modul `socket` bringen. Es lohnt sich absolut, einen Blick in dieses Modul zu riskieren, denn es bietet viele Möglichkeiten der Netzwerkprogrammierung, die bei den anderen, abstrakteren Modulen verloren gehen. Außerdem lernen Sie den Komfort, den die abstrakten Schnittstellen bieten, erst wirklich zu schätzen, wenn Sie das `socket`-Modul kennengelernt haben.

Nachdem wir uns mit der Socket API beschäftigt haben, folgen

einige spezielle Module, die beispielsweise mit bestimmten Protokollen wie HTTP oder FTP umgehen können.

20.1 Socket API

Das Modul socket der Standardbibliothek bietet grundlegende Funktionalität zur Netzwerkkommunikation. Das Modul bildet dabei die standardisierte *Socket API* ab, die so oder in ganz ähnlicher Form auch für viele andere Programmiersprachen implementiert ist.

Die Idee, die hinter der Socket API steht, ist die, dass das Programm, das Daten über die Netzwerkschnittstelle senden oder empfangen möchte, dies beim Betriebssystem anmeldet und von diesem einen sogenannten *Socket* (dt. *Steckdose*) bekommt. Über diesen Socket kann das Programm jetzt eine Netzwerkverbindung zu einem anderen Socket aufbauen. Dabei spielt es keine Rolle, ob sich der Zielsocket auf demselben Rechner, einem Rechner im lokalen Netzwerk oder einem Rechner im Internet befindet.

Zunächst ein paar Worte dazu, wie ein Rechner in der komplexen Welt eines Netzwerks adressiert werden kann. Jeder Rechner besitzt in einem Netzwerk, auch dem Internet, eine eindeutige sogenannte *IP-Adresse*, über die er angesprochen werden kann. Eine IP-Adresse ist ein String der folgenden Struktur:

```
"192.168.1.23"
```

Dabei repräsentiert jeder der vier Zahlenwerte ein Byte und kann somit zwischen 0 und 255 liegen. In diesem Fall handelt es sich um eine IP-Adresse eines lokalen Netzwerks, was an der Anfangssequenz 192.168 zu erkennen ist.

Damit ist es jedoch noch nicht getan, denn auf einem einzelnen Rechner könnten mehrere Programme laufen, die gleichzeitig Daten über die Netzwerkschnittstelle senden und empfangen möchten. Aus diesem Grund wird eine Netzwerkverbindung zusätzlich an einen sogenannten *Port* gebunden. Der Port ermöglicht es, ein bestimmtes Programm anzusprechen, das auf einem Rechner mit einer bestimmten IP-Adresse läuft.

Bei einem Port handelt es sich um eine 16-Bit-Zahl – grundsätzlich sind also 65.535 verschiedene Ports verfügbar. Allerdings sind viele dieser Ports für Protokolle und Anwendungen registriert und sollten nicht verwendet werden. Beispielsweise sind für HTTP- und FTP-Server die Ports 80 bzw. 21 registriert. Grundsätzlich können Sie Ports ab 49152 bedenkenlos verwenden.

Beachten Sie, dass beispielsweise eine Firewall oder ein Router bestimmte Ports blockieren kann. Sollten Sie also auf Ihrem Rechner einen Server betreiben wollen, zu dem sich Clients über einen bestimmten Port verbinden können, müssen Sie diesen Port gegebenenfalls mit der entsprechenden Software freischalten.

20.1.1 Client/Server-Systeme

Die beiden Kommunikationspartner einer Netzwerkkommunikation haben in der Regel verschiedene Aufgaben. So existiert zum einen ein *Server* (dt. *Diener*), der bestimmte Dienstleistungen anbietet, und zum anderen ein *Client* (dt. *Kunde*), der diese Dienstleistungen in Anspruch nimmt.

Ein Server ist unter einer bekannten Adresse im Netzwerk erreichbar und operiert passiv, das heißt, er wartet auf eingehende Verbindungen. Sobald eine Verbindungsanfrage eines Clients eintrifft, wird, sofern der Server die Anfrage akzeptiert, ein neuer Socket erzeugt, über den die Kommunikation mit diesem speziellen Client läuft. Wir werden uns zunächst mit sogenannten *seriellen Servern* befassen, das sind Server, bei denen die

Kommunikation mit dem vorherigen Client abgeschlossen sein muss, bevor eine neue Verbindung akzeptiert werden kann. Dem gegenüber stehen die Konzepte der *parallelen Server* und der *multiplexenden Server*, auf die wir auch noch zu sprechen kommen werden.

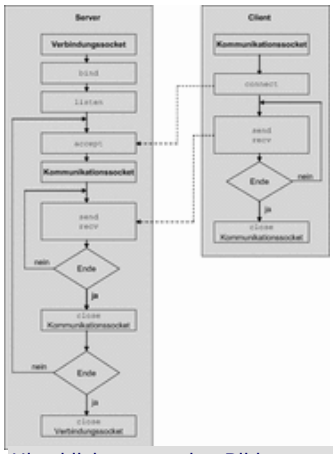
Der Client stellt den aktiven Kommunikationspartner dar. Das heißt, er sendet eine Verbindungsanfrage an den Server und nimmt dann aktiv dessen Dienstleistungen in Anspruch.

Die Stadien, in denen sich ein serieller Server und ein Client vor, während und nach der Kommunikation befinden, sollen durch das Flussdiagramm aus Abbildung 20.2 verdeutlicht werden. Sie können es als eine Art Bauplan für einen seriellen Server und den dazu gehörigen Client auffassen.

Zunächst wird im Serverprogramm der sogenannte *Verbindungssocket* erzeugt. Das ist ein Socket, der ausschließlich dazu gedacht ist, auf eingehende Verbindungen zu horchen und diese gegebenenfalls zu akzeptieren. Über den Verbindungssocket läuft keine Kommunikation. Durch Aufruf der Methoden `bind` und `listen` wird der Verbindungssocket an eine Netzwerkadresse gebunden und dazu instruiert, nach einkommenden Verbindungsanfragen zu lauschen.

Nachdem eine Verbindungsanfrage eingetroffen ist und mittels `accept` akzeptiert wurde, wird ein neuer Socket, der sogenannte *Kommunikationssocket* erzeugt. Über einen solchen Kommunikationssocket wird die vollständige Kommunikation zwischen Server und Client über Methoden wie `send` oder `recv` abgewickelt. Beachten Sie, dass ein Kommunikationssocket immer nur für einen verbundenen Client zuständig ist.

Sobald die Kommunikation beendet ist, wird das Kommunikationsobjekt geschlossen und eventuell eine weitere Verbindung eingegangen. Beachten Sie, dass Verbindungsanfragen, die nicht sofort akzeptiert werden, keineswegs verloren sind, sondern gepuffert werden. Sie befinden sich in der sogenannten *Queue* und können somit nacheinander abgearbeitet werden. Zum Schluss wird auch der Verbindungssocket geschlossen.



Hier klicken, um das Bild zu vergrößern

Abbildung 20.2 Das Client/Server-Modell

Die Struktur des Clients ist vergleichsweise einfach. So gibt es beispielsweise nur einen Kommunikationssocket, über den mithilfe der Methode `connect` eine Verbindungsanfrage an einen bestimmten Server gesendet werden kann. Danach erfolgt, ähnlich wie beim Server, die tatsächliche Kommunikation über Methoden wie `send` oder `recv`. Nach dem Ende der Kommunikation wird der Verbindungssocket geschlossen.

Grundsätzlich kann für die Datenübertragung zwischen Server und Client aus zwei verfügbaren Netzwerkprotokollen gewählt werden: *UDP* und *TCP*. In den folgenden beiden Abschnitten sollen kleine

Beispielserver und -clients für beide dieser Protokolle implementiert werden.

Beachten Sie, dass sich das hier vorgestellte Flussdiagramm auf das verbindungsbehaftete und üblichere TCP-Protokoll bezieht. Die Handhabung des verbindungslosen UDP-Protokolls unterscheidet sich davon in einigen wesentlichen Punkten. Näheres dazu finden Sie im folgenden Abschnitt.

top

20.1.2 UDP ▼▲

Das Netzwerkprotokoll *UDP* (*User Datagram Protocol*) wurde 1977 als Alternative zu TCP für die Übertragung menschlicher Sprache entwickelt. Charakteristisch ist, dass UDP verbindungslos und nicht zuverlässig ist. Diese beiden Begriffe gehen miteinander einher und bedeuten zum einen, dass keine explizite Verbindung zwischen den Kommunikationspartnern aufgebaut wird, und zum anderen, dass UDP weder garantiert, dass gesendete Pakete in der Reihenfolge ankommen, in der sie gesendet wurden, noch dass sie überhaupt ankommen. Aufgrund dieser Einschränkungen können mit UDP jedoch vergleichsweise schnelle Übertragungen stattfinden, da beispielsweise keine Pakete neu angefordert oder gepuffert werden müssen.

Damit eignet sich UDP insbesondere für Multimedia-Anwendungen wie VoIP, Audio- oder Videostreaming, bei denen es auf eine schnelle Übertragung der Daten ankommt und kleinere Übertragungsfehler toleriert werden können.

Das im Folgenden entwickelte Beispielprojekt besteht aus einem Server- und einem Clientprogramm. Der Client schickt eine Textnachricht per UDP an eine bestimmte Adresse. Das dort laufende Serverprogramm nimmt die Nachricht entgegen und zeigt sie an. Betrachten wir zunächst den Quellcode des Clients:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

ip = raw_input("IP-Adresse: ")
nachricht = raw_input("Nachricht: ")

s.sendto(nachricht, (ip, 50000))
s.close()
```

Zunächst wird durch Aufruf der Funktion socket eine Socket-Instanz erzeugt. Dabei können zwei Parameter übergeben werden: zum einen der zu verwendende Adresstyp und zum anderen das zu verwendende Netzwerkprotokoll. Die Konstanten AF_INET und SOCK_DGRAM stehen dabei für Internet/IPv4 und UDP.

Danach werden zwei Angaben vom Benutzer eingelesen: die IP-Adresse, an die die Nachricht zu schicken ist, und die Nachricht selbst.

Zum Schluss wird die Nachricht unter Verwendung der Socket-Methode sendto zur angegebenen IP-Adresse geschickt, wozu der Port 50000 verwendet wird, und der Socket mittels close geschlossen.

Das Clientprogramm allein ist so gut wie wertlos, solange es kein dazu passendes Serverprogramm auf der anderen Seite gibt, das die Nachricht entgegennehmen und verwerten kann. Beachten Sie, dass UDP verbindungslos ist und sich die Implementation daher etwas vom Flussdiagramm eines Servers aus Abschnitt 20.1.1 unterscheidet.

Der Quelltext des Servers sieht folgendermaßen aus:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

try:
    s.bind(("", 50000))
    while True:
        daten, addr = s.recvfrom(1024)
```

```
        print "[%s] %s" % (addr[0], daten)
finally:
    s.close()
```

Auch hier wird zunächst eine Socket-Instanz erstellt. In der darauf folgenden `try/finally`-Anweisung wird dieser Socket durch Aufruf der Methode `bind` an eine Adresse gebunden. Beachten Sie, dass diese Methode ein Adressobjekt als Parameter übergeben bekommt. Immer wenn im Zusammenhang mit Sockets von einem Adressobjekt die Rede ist, ist damit schlicht ein Tupel mit zwei Elementen gemeint: einer IP-Adresse als String und einer Portnummer als ganze Zahl.

Das Binden eines Sockets an eine Adresse legt fest, über welche interne Schnittstelle der Socket Pakete empfangen kann. Wenn keine IP-Adresse angegeben wurde, bedeutet dies, dass Pakete über alle dem Server zugeordneten Adressen empfangen werden können, beispielsweise also auch über `127.0.0.1` oder `localhost`.

Nachdem der Socket an eine Adresse gebunden worden ist, können Daten empfangen werden. Dazu wird die Methode `recvfrom` (für *receive from*) in einer Endlosschleife aufgerufen. Die Methode wartet so lange, bis ein Paket eingegangen ist, und gibt die gelesenen Daten mitsamt den Absenderinformationen als Tupel zurück. Der Parameter von `recvfrom` kennzeichnet die maximale Paketgröße und sollte eine Zweierpotenz sein.

An dieser Stelle wird auch der Sinn der `try/finally`-Anweisung deutlich: Das Programm wartet in einer Endlosschleife auf eintreffende Pakete und kann daher nur mit einem Programmabbruch durch Tastenkombination, also durch eine `KeyboardInterrupt`–Exception, beendet werden. In einem solchen Fall muss der Socket trotzdem noch mittels `close` geschlossen werden.

top

20.1.3 TCP ▼▲

TCP (*Transmission Control Protocol*) ist kein Konkurrenzprodukt zu UDP, sondern füllt mit seinen Möglichkeiten die Lücken auf, die UDP offen lässt. So ist TCP vor allem verbindungsorientiert und zuverlässig. Verbindungsorientiert bedeutet, dass nicht, wie bei UDP, einfach Datenpakete an bestimmte IP-Adressen geschickt werden, sondern dass zuvor eine Verbindung aufgebaut wird und auf Basis dieser Verbindung weitere Operationen durchgeführt werden. Zuverlässig bedeutet, dass es mit TCP nicht, wie bei UDP, vorkommen kann, dass Pakete verloren gehen, fehlerhaft oder in falscher Reihenfolge ankommen. Solche Vorkommnisse korrigiert das TCP-Protokoll intern, indem es beispielsweise unvollständige oder fehlerhafte Pakete neu anfordert.

Aus diesem Grund ist TCP zumeist die erste Wahl, wenn es um eine Netzwerkschnittstelle geht. Bedenken Sie aber unbedingt, dass jedes Paket, das neu angefordert werden muss, Zeit kostet und die Latenz der Verbindung somit steigen kann. Außerdem sind fehlerhafte Übertragungen in einem LAN äußerst selten, weswegen man gerade dort die Performance von UDP und die Verbindungsqualität von TCP gegeneinander abwägen sollte.

Im Folgenden soll auch die Verwendung von TCP anhand eines kleinen Beispielprojekts erläutert werden: Es soll ein rudimentäres Chatprogramm entstehen, bei dem der Client eine Nachricht an den Server sendet, auf die der Server wieder antworten kann. Die Kommunikation soll also immer abwechselnd erfolgen. Der Quelltext des Servers sieht folgendermaßen aus:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("", 50000))
s.listen(1)

try:
```

```
    while True:
        komm, addr = s.accept()
        while True:
            data = komm.recv(1024)

            if not data:
                komm.close()
                break

            print "[%s] %s" % (addr[0], data)
            nachricht = raw_input("Antwort: ")
            komm.send(nachricht)
finally:
    s.close()
```

Bei der Erzeugung des Verbindungssockets unterscheidet sich TCP von UDP nur in den zu übergebenden Werten. In diesem Fall wird `AF_INET` für das IPv4-Protokoll und `SOCK_STREAM` für die Verwendung von TCP übergeben. Damit ist allerdings nur der Socket in seiner Rohform instanziiert. Auch bei TCP muss der Socket an eine IP-Adresse und einen Port gebunden werden. Beachten Sie, dass `bind` ein Adressobjekt als Parameter erwartet – die Angaben von IP-Adresse und Port also noch in ein Tupel gefasst sind. Auch hier werden wieder alle IP-Adressen des Servers genutzt.

Danach wird der Server durch Aufruf der Methode `listen` in den passiven Modus geschaltet und instruiert, nach Verbindungsanfragen zu horchen. Beachten Sie, dass diese Methode noch keine Verbindung herstellt. Der übergebene Parameter bestimmt die maximale Anzahl von zu puffernden Verbindungsversuchen und sollte mindestens `1` sein.

In der darauf folgenden Endlosschleife wartet die aufgerufene Methode `accept` des Verbindungssockets nun auf eine eingehende Verbindungsanfrage und akzeptiert diese. Zurückgegeben wird ein Tupel, dessen erstes Element der Kommunikationssocket ist, der zur Kommunikation mit dem verbundenen Client verwendet werden kann. Das zweite Element des Tupels ist das Adressobjekt des Verbindungspartners.

Nachdem eine Verbindung hergestellt wurde, wird eine zweite Endlosschleife eingeleitet, deren Schleifenkörper im Prinzip aus zwei Teilen besteht: Zunächst wird immer eine Nachricht per `komm.recv` vom Verbindungspartner erwartet und ausgegeben. Sollte von `komm.recv` ein leerer String zurückgegeben werden, so bedeutet dies, dass der Verbindungspartner die Verbindung beendet hat. In einem solchen Fall wird die innere Schleife abgebrochen. Wenn eine wirkliche Nachricht angekommen ist, erlaubt es der Server dem Benutzer, eine Antwort einzugeben, und verschickt diese per `komm.send`.

Jetzt soll der Quelltext des Clients besprochen werden:

```
import socket

ip = raw_input("IP-Adresse: ")
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, 50000))

try:
    while True:
        nachricht = raw_input("Nachricht: ")
        s.send(nachricht)
        antwort = s.recv(1024)
        print "[%s] %s" % (ip,antwort)
finally:
    s.close()
```

Auf der Clientseite wird der instanziierte Socket `s` durch Aufruf der Methode `connect` mit dem Verbindungspartner verbunden. Die Methode `connect` verschickt genau die Verbindungsanfrage, die beim Server durch `accept` akzeptiert werden kann. Wenn die Verbindung abgelehnt wurde, wird eine Exception geworfen.

Die nachfolgende Endlosschleife funktioniert ähnlich wie die des Servers, mit dem Unterschied, dass zuerst eine Nachricht eingegeben und abgeschickt wird und danach auf eine Antwort des Servers gewartet wird. Damit wären Client und Server in einen Rhythmus gebracht, bei dem der Server immer dann auf

eine Nachricht wartet, wenn beim Client eine eingegeben wird und umgekehrt.

Genau dieser Rhythmus ist aber auch der größte Knackpunkt des Beispielprojekts, denn es ist für einen der Kommunikationspartner schlicht unmöglich, zwei Nachrichten direkt hintereinander abzusetzen. Für den praktischen Einsatz hätte das Programm also allenfalls Unterhaltungswert. Das Ziel war es auch nicht, eine möglichst perfekte Chat-Applikation zu schreiben, sondern eine einfache und kurze Beispielimplementation einer Client/Server-Kommunikation über TCP zu erstellen.

Betrachten Sie es also als Herausforderung, Client und Server, beispielsweise durch Threads, zu einem brauchbaren Chat-Programm zu erweitern. Das könnte so aussehen, dass ein Thread jeweils `s.recv` abhört und eingehende Nachrichten anzeigt und ein zweiter Thread es ermöglicht, dass die Benutzer Nachrichten per `raw_input` eingeben, und diese dann verschickt.

top

20.1.4 Blockierende und nicht-blockierende Sockets

Wenn ein Socket erstellt wird, befindet er sich standardmäßig im sogenannten *blockierenden Modus* (engl. *blocking mode*). Das bedeutet, dass alle Methodenaufrufe warten, bis die von ihnen angestoßene Operation durchgeführt wurde. So würde ein Aufruf der Methode `recv` eines Sockets so lange das komplette Programm blockieren, bis tatsächlich Daten eingegangen sind und aus dem internen Puffer des Sockets gelesen werden können.

In vielen Fällen ist dieses Verhalten durchaus gewünscht, doch möchte man bei einem Programm, in dem viele verbundene Sockets verwaltet werden, beispielsweise nicht, dass einer dieser Sockets mit seiner `recv`-Methode das komplette Programm blockiert, nur weil noch keine Daten eingegangen sind, während an einem anderen Socket Daten zum Lesen bereitstehen. Um solche Probleme zu umgehen, lässt sich der Socket in den sogenannten *nicht-blockierenden Modus* (engl. *non-blocking mode*) versetzen. Dies wirkt sich folgendermaßen auf diverse Socket-Operationen aus:

- Die Methoden `recv` und `recvfrom` des Socket-Objekts geben nur noch ankommende Daten zurück, wenn sich diese bereits im internen Puffer des Sockets befinden. Sobald die Methode auf weitere Daten zu warten hätte, wirft sie eine `socket.error`–Exception und gibt damit den Kontrollfluss wieder an das Programm ab.

- Die Methoden `send` und `sendto` versenden die angegebenen Daten nur, wenn sie direkt in den Ausgangspuffer des Sockets geschrieben werden können. Gelegentlich kommt es vor, dass dieser Puffer voll ist und `send` bzw. `sendto` zu warten hätten, bis der Puffer weitere Daten aufnehmen kann. In einem solchen Fall wird im nicht-blockierenden Modus eine `socket.error`-Exception geworfen und der Kontrollfluss damit an das Programm zurückgegeben.

- Die Methode `connect` sendet eine Verbindungsanfrage an den Zielsocket und wartet nicht, bis diese Verbindung zustande kommt. Wenn `connect` aufgerufen wird und die Verbindungsanfrage noch läuft, wird eine `socket.error`-Exception mit der Fehlermeldung »*Operation now in progress*« geworfen. Durch mehrmaligen Aufruf von `connect` lässt sich feststellen, ob die Operation immer noch durchgeführt wird.

  Alternativ kann im nicht-blockierenden Modus die Methode `connect_ex` für Verbindungsanfragen verwendet werden. Diese wirft keine `socket.error`-Exception, sondern zeigt eine erfolgreiche Verbindung anhand eines Rückgabewertes von `0` an. Bei echten Fehlern, die bei der Verbindung auftreten, wird auch bei `connect_ex` eine Exception

geworfen.

Ein Socket lässt sich durch Aufruf seiner Methode `setblocking` in den nicht-blockierenden Zustand versetzen:

```
s.setblocking(0)
```

In diesem Fall würden sich Methodenaufrufe des Sockets `s` wie oben beschrieben verhalten. Ein Parameter von `1` versetzt den Socket wieder in den ursprünglichen blockierenden Modus.

Socket-Operationen werden im Falle des blockierenden Modus auch *synchrone Operationen* und im Falle des nicht-blockierenden Modus *asynchrone Operationen* genannt.

Es ist durchaus möglich, auch während des Betriebs zwischen dem blockierenden und nicht-blockierenden Modus eines Sockets umzuschalten. So könnte beispielsweise die Methode `connect` blockierend und anschließend die Methode `read` nicht-blockierend verwendet werden.

top

20.1.5 Verwendung des Moduls ▼▲

Da die Funktionen des Moduls oder die Methoden des Socket-Objekts in den beiden vorherigen Abschnitten vielleicht etwas zu kurz gekommen sind, möchten wir in diesem Abschnitt noch einmal eine Auflistung der wichtigsten dieser Funktionen und Methoden bringen. Wir beginnen mit den Funktionen und Konstanten des Moduls `socket`.

Funktionen

socket.getfqdn([name])

Gibt den vollständigen Domainnamen (FQDN, *Fully qualified domain name*) der Domain *name* zurück. Wenn *name* weggelassen wird, wird der vollständige Domainname des lokalen Hosts zurückgegeben.

```
>>> socket.getfqdn()
'HOSTNAME.localdomain'
```

socket.gethostbyname(hostname)

Gibt die IPv4-Adresse des Hosts *hostname* als String zurück.

```
>>> socket.gethostbyname("HOSTNAME")
'192.168.1.23'
```

socket.gethostname()

Gibt den Hostnamen des Systems als String zurück.

```
>>> socket.gethostname()
'HOSTNAME'
```

socket.getservbyname(servicename[, protocolname])

Gibt den Port für den Service *servicename* mit dem Netzwerkprotokoll *protocolname* zurück. Bekannte Services wären beispielsweise `"http"` oder `"ftp"` mit den Portnummern `80` bzw. `21`. Der Parameter *protocolname* sollte entweder `"tcp"` oder `"udp"` sein.

```
>>> socket.getservbyname("http", "tcp")
80
```

socket.getservbyport(port[, protocolname])

Diese Funktion ist das Gegenstück zu `getservbyname`.

```
>>> socket.getservbyport(21)
'ftp'
```

socket.socket([family[, type]])

Erzeugt einen neuen Socket. Der erste Parameter *family* kennzeichnet dabei die Adressfamilie und sollte entweder `socket.AF_INET` für den IPv4-Namensraum oder `socket.AF_INET6` für den IPv6-Namensraum sein.

Der zweite Parameter *type* kennzeichnet das zu verwendende Netzwerkprotokoll und sollte entweder `socket.SOCK_STREAM` für TCP oder `socket.SOCK_DGRAM` für UDP sein.

socket.getdefaulttimeout(), socket.setdefaulttimeout(timeout)

Gibt in Form einer Gleitkommazahl die maximale Anzahl an Sekunden zurück, die beispielsweise die Methode `recv` eines Socket-Objekts auf ein eingehendes Paket wartet. Durch die Funktion `setdefaulttimeout` kann dieser Wert für alle neu erzeugten `Socket`-Instanzen verändert werden.

Die Socket-Klasse

Nachdem durch die Funktion `socket` des Moduls `socket` eine neue Instanz der Klasse `Socket` erzeugt wurde, stellt diese natürlich weitere Funktionalität bereit, um sich mit einem zweiten Socket zu verbinden oder Daten an den Verbindungspartner zu übermitteln. Die Methoden der `Socket`-Klasse sollen im Folgenden beschrieben werden.

Beachten Sie, dass sich das Verhalten der Methoden im blockierenden und nicht-blockierenden Modus unterscheidet. Näheres dazu finden Sie in Abschnitt 20.1.4, »Blockierende und nicht-blockierende Sockets«. Im Folgenden sei *s* eine Instanz der Klasse `socket.Socket`.

s.accept()

Wartet auf eine eingehende Verbindungsanfrage und akzeptiert diese. Die `Socket`-Instanz muss zuvor durch Aufruf der Methode `bind` an eine bestimmte Adresse und einen Port gebunden worden sein und Verbindungsanfragen erwarten. Letzteres geschieht durch Aufruf der Methode `listen`.

Die Methode `accept` gibt ein Tupel zurück, dessen erstes Element eine neue `Socket`-Instanz, auch Connection-Objekt genannt, ist, über die die Kommunikation mit dem Verbindungspartner geschehen kann. Das zweite Element des Tupels ist ein weiteres Tupel, das IP-Adresse und Port des verbundenen Sockets enthält.

Diese Methode ist für TCP gedacht.

s.bind(address)

Bindet den Socket an die Adresse *address*. Der Parameter *address* muss ein Tupel der Form sein, wie es `accept` zurückgibt.

Nachdem ein Socket an eine bestimmte Adresse gebunden wurde, kann er, im Falle von TCP, in den passiven Modus geschaltet werden oder, im Falle von UDP, direkt Datenpakete empfangen.

s.close()

Schließt den Socket. Das bedeutet, dass keine Daten mehr über

ihn gesendet oder empfangen werden können.

s.connect(address)

Verbindet zu einem Server mit der Adresse *address*. Beachten Sie, dass dort ein Socket existieren muss, der auf dem gleichen Port auf Verbindungsanfragen wartet, damit die Verbindung zustande kommen kann. Der Parameter *address* muss im Falle des IPv4-Protokolls ein Tupel sein, das aus der IP-Adresse und der Portnummer besteht.

Diese Methode ist für TCP gedacht.

s.connect_ex(address)

Unterscheidet sich von `connect` nur darin, dass im nicht-blockierenden Modus keine Exception geworfen wird, wenn die Verbindung nicht sofort zustande kommt. Der Verbindungsstatus wird über einen ganzzahligen Rückgabewert angezeigt. Ein Rückgabewert von `0` bedeutet, dass der Verbindungsversuch erfolgreich durchgeführt wurde.

Beachten Sie, dass bei echten Fehlern, die beim Verbindungsversuch auftreten, weiterhin Exceptions geworfen werden, beispielsweise wenn der Zielsocket nicht erreicht werden konnte.

Diese Methode ist für TCP gedacht.

s.getpeername()

Gibt das Adressobjekt des mit diesem Socket verbundenen Sockets zurück. Das Adressobjekt ist ein Tupel, das IP-Adresse und Portnummer enthält.

Diese Methode ist für TCP gedacht.

s.getsockname()

Gibt das Adressobjekt zurück, über das dieser Socket mit dem verbundenen Socket kommuniziert.

s.listen(backlog)

Versetzt einen Serversocket in den sogenannten Listen-Modus, das heißt, der Socket achtet auf Sockets, die sich mit ihm verbinden wollen. Nachdem diese Methode aufgerufen worden ist, können eingehende Verbindungswünsche mit `accept` akzeptiert werden.

Der Parameter *backlog* legt die maximale Anzahl an gepufferten Verbindungsanfragen fest und sollte mindestens `1` sein. Den größtmöglichen Wert für *backlog* legt das Betriebssystem fest, meistens liegt er bei `5`.

Diese Methode ist für TCP gedacht.

s.recv(bufsize)

Liest beim Socket eingegangene Daten. Durch den Parameter *bufsize* wird die maximale Anzahl von zu lesenden Bytes festgelegt. Die gelesenen Daten werden in Form eines Strings zurückgegeben.

Diese Methode ist für TCP gedacht.

s.recvfrom(bufsize)

Unterscheidet sich von `recv` in Bezug auf den Rückgabewert. Dieser ist bei `recvfrom` ein Tupel, das als erstes Element die gelesenen Daten als String und als zweites Element das

Adressobjekt des Verbindungspartners enthält.

Diese Methode ist für UDP gedacht.

s.send(string)

Sendet den String *string* zum verbundenen Socket. Die Anzahl der gesendeten Bytes wird zurückgegeben. Beachten Sie, dass es unter Umständen vorkommt, dass die Daten nicht vollständig gesendet wurden. In einem solchen Fall ist die Anwendung dafür verantwortlich, die verbleibenden Daten erneut zu senden.

Diese Methode ist für TCP gedacht.

s.sendall(string)

Unterscheidet sich von send darin, dass `sendall` so lange versucht, die Daten zu senden, bis entweder der vollständige Datensatz *string* versendet wurde oder ein Fehler aufgetreten ist. Im Fehlerfall wird eine entsprechende Exception geworfen.

Diese Methode ist für TCP gedacht.

s.sendto(string, address)

Versendet den Datensatz *string* an einen Socket mit der Adresse *address*. Da der Verbindungspartner explizit angegeben wird, brauchen die beiden Sockets nicht untereinander verbunden zu sein. Der Parameter *address* muss ein Adressobjekt sein.

Diese Methode ist für UDP gedacht.

s.setblocking(flag)

Wenn für *flag* `0` übergeben wird, wird der Socket in den nicht-blockierenden Modus versetzt, sonst in den blockierenden Modus.

Im blockierenden Modus warten Methoden wie send oder `recv`, bis Daten versendet bzw. gelesen werden konnten. Im nicht-blockierenden Modus würde ein Aufruf von `recv` beispielsweise eine Exception verursachen, wenn keine Daten eingegangen sind, die gelesen werden könnten.

s.settimeout(value), gettimeout()

Setzt einen Timeout-Wert für diesen Socket. Dieser Wert bestimmt im blockierenden Modus, wie lange auf das Eintreffen bzw. Versenden von Daten gewartet werden soll. Dabei kann für *value* die Anzahl an Sekunden in Form einer Gleitkommazahl oder `None` übergeben werden.

Über die Methode `gettimeout` kann der Timeout-Wert ausgelesen werden.

Wenn ein Aufruf von beispielsweise send oder `recv` die maximale Wartezeit überschreitet, wird eine `socket.timeout`-Exception geworfen.

top

20.1.6 Netzwerk-Byte-Order ▼▲

Das Schöne an standardisierten Protokollen wir TCP oder UDP ist, dass Computer verschiedenster Bauart eine gemeinsame Schnittstelle haben, über die sie miteinander kommunizieren können. Allerdings hören diese Gemeinsamkeiten hinter der Schnittstelle unter Umständen wieder auf. So ist beispielsweise die sogenannte *Byte-Order* ein signifikanter Unterschied zwischen diversen Systemen. Diese Byte-Order legt die Speicherreihenfolge von Zahlen fest, die mehr als ein Byte Speicher benötigen.

Bei der Übertragung von Binärdaten führt es zu Problemen, wenn

diese ohne Konvertierung zwischen zwei Systemen mit verschiedener Byte-Order ausgetauscht werden. Das Protokoll TCP garantiert dabei nur, dass die gesendeten Bytes in der Reihenfolge ankommen, in der sie abgeschickt wurden.

Solange Sie sich bei der Netzwerkkommunikation auf reine ASCII-Strings beschränken, können keine Probleme auftreten, da ASCII-Zeichen nie mehr als ein Byte Speicher benötigen. Außerdem sind Verbindungen zwischen zwei Computern derselben Plattform problemlos. So können beispielsweise Binärdaten zwischen zwei x86er-PCs übertragen werden, ohne Probleme befürchten zu müssen.

Allerdings möchte man bei einer Netzwerkverbindung in der Regel Daten übertragen, ohne sich über die Plattform des verbundenen Rechners Gedanken zu machen. Dazu hat man die sogenannte *Netzwerk-Byte-Order* definiert. Das ist die Byte-Order, die für Binärdaten im Netzwerk zu verwenden ist. Um diese Netzwerk-Byte-Order sinnvoll umzusetzen, sind im Modul `socket` vier Funktionen enthalten, die entweder Daten von der Host-Byte-Order in die Netzwerk-Byte-Order (»hton«) oder umgekehrt (»ntoh«) konvertieren. Die folgende Tabelle listet diese Funktionen auf und erläutert ihre Bedeutung:

Alias	Bedeutung
`socket.ntohl(x)`	Konvertiert eine 32-Bit-Zahl von der Netzwerk- in die Host-Byte-Order.
`socket.ntohs(x)`	Konvertiert eine 16-Bit-Zahl von der Netzwerk- in die Host-Byte-Order.
`socket.htonl(x)`	Konvertiert eine 32-Bit-Zahl von der Host- in die Netzwerk-Byte-Order.
`socket.htons(x)`	Konvertiert eine 16-Bit-Zahl von der Host- in die Netzwerk-Byte-Order.

Tabelle 20.2 Konvertierung von Binärdaten

Der Aufruf dieser Funktionen ist möglicherweise überflüssig, wenn das entsprechende System bereits die Netzwerk-Byte-Order verwendet. Der gebräuchliche x86er-PC verwendet übrigens nicht die Netzwerk-Byte-Order.

An dieser Stelle möchten wir noch einmal darauf hinweisen, dass eine Konvertierung von Binärdaten in einem professionellen Programm selbstverständlich dazugehört. Solange Sie jedoch im privaten Umfeld kleinere Netzwerkanwendungen schreiben, die Binärdaten ausschließlich zwischen x86er-PCs austauschen, brauchen Sie sich über die Byte-Order keine Gedanken zu machen. Zudem können ASCII-Zeichen, wie gesagt, problemlos auch zwischen Systemen mit verschiedener Byte-Order ausgetauscht werden, sodass auch in diesem Fall keine explizite Konvertierung nötig ist.

top

20.1.7 Multiplexende Server – select ▼▲

Ein Server ist in den meisten Fällen nicht dazu gedacht, immer nur einen Client zu bedienen, wie es in den bisherigen Beispielen vereinfacht angenommen wurde. In der Regel muss ein Server eine ganze Reihe von verbundenen Clients verwalten, die sich in verschiedenen Phasen der Kommunikation befinden. Es stellt sich die Frage, wie so etwas sinnvoll in einem Prozess, also ohne den Einsatz von Threads, durchgeführt werden kann.

Selbstverständlich könnte man alle verwendeten Sockets in den nicht-blockierenden Modus schalten und die Verwaltung selbst in die Hand nehmen. Das ist aber nur auf den ersten Blick eine Lösung, denn der blockierende Modus besitzt einen unschätzbaren Vorteil: Ein blockierender Socket veranlasst, dass das Programm bei einer Netzwerkoperation so lange schlafen gelegt wird, bis die Operation durchgeführt werden kann. Auf diese Weise kann die

Prozessorauslastung reduziert werden.

Im Gegensatz dazu müssten wir beim Einsatz von nicht-blockierenden Sockets in einer Schleife ständig über alle verbundenen Sockets iterieren und prüfen, ob sich etwas getan hat, also ob beispielsweise Daten zum Auslesen bereitstehen. Dieser Ansatz, auch *Busy Waiting* genannt, ermöglicht uns zwar das quasi-parallele Auslesen mehrerer Sockets, das Programm lastet den Prozessor aber wesentlich mehr aus, da es über den gesamten Zeitraum aktiv ist.

Das Modul `select` ermöglicht es, im gleichen Prozess mehrere blockierende Sockets zu verwalten, sodass die Vorteile blockierender Sockets erhalten bleiben. Ein Server, der `select` verwendet, wird *multiplexender Server* genannt. Im Modul ist im Wesentlichen die Funktion `select` enthalten, die im Folgenden besprochen werden soll.

select.select(iwtd, owtd, ewtd[, timeout])

Im einfachsten Fall bekommt die Funktion `select` als ersten Parameter *iwtd* eine Liste von Sockets übergeben, mit denen eine Leseoperation durchgeführt werden soll. Nehmen wir einmal an, für die weiteren Parameter *owtd* und *ewtd* würde jeweils eine leere Liste übergeben. In diesem Fall würde die Funktion `select` das Programm so lange schlafen legen, bis an mindestens einem der übergebenen Sockets Daten vorliegen, die ausgelesen werden können.

Ähnlich verhält es sich mit dem zweiten Parameter, *owtd*. Hier wird eine Liste von Sockets übergeben, mit denen eine Schreiboperation durchgeführt werden soll. Die Funktion `select` weckt das Programm auf, sobald einer der hier übergebenen Sockets zum Schreiben bereit ist.

Für den dritten Parameter, *ewtd*, wird eine Liste von Sockets übergeben, bei denen möglicherweise sogenannte *out-of-band data* eingegangen sind. Das sind TCP-Pakete, die als besonders wichtig (engl. *urgent*) eingestuft sind und somit privilegiert übertragen werden. Mithilfe solcher Nachrichten kann ein Programm wichtige Ausnahmefälle signalisieren. Dennoch werden wir hier nicht näher darauf eingehen, da solche OOB-Pakete so gut wie nie verwendet werden.

Als vierter, optionaler und letzter Parameter kann ein Timeout-Wert in Sekunden angegeben werden. Dieser veranlasst die Funktion `select`, das Programm nach einer gewissen Zeit aufzuwecken, auch wenn sich bei keinem der übergebenen Sockets etwas getan hat. Wenn ein Timeout-Wert von `0.0` übergeben wird, gibt `select` nur die Sockets zurück, die beim Aufruf bereits bereit zum Lesen bzw. Schreiben sind.

Es ist möglich, für *iwtd*, *owtd* oder *ewtd* leere Listen zu übergeben, vor allem, weil in der Regel nur der erste dieser Parameter benötigt wird, denn es ist das klassische Anwendungsgebiet von `select`, auf eintreffende Daten zu warten. Der zweite Parameter ist deshalb weniger wichtig, weil ein Socket in der Regel zu jeder Zeit zum Versenden von Daten bereitsteht. Und in den seltenen Fällen, bei denen dies nicht der Fall ist, ist die »Verstopfung« des Ausgangspuffers nur von kurzer Dauer und ein blockierender Aufruf von send somit nicht weiter tragisch.

Die Funktion `select` gibt in jedem Fall ein Tupel zurück, das aus drei Listen besteht. Diese Listen enthalten jeweils die Sockets, bei denen entweder Daten gelesen oder geschrieben werden können oder, wie erwähnt, dringende Pakete vorliegen. Beachten Sie, dass dieselbe Socket-Instanz beim Aufruf von `select` durchaus in mehreren der übergebenen Listen vorkommen darf.

Im folgenden Beispiel soll ein Server geschrieben werden, der Verbindungen von beliebig vielen Clients akzeptiert und verwaltet. Diese Clients sollen dazu in der Lage sein, dem Server mehrere Nachrichten zu schicken, die von diesem dann am Bildschirm angezeigt werden. Aus Gründen der Einfachheit verzichten wir auf

eine Antwortmöglichkeit des Servers.

```
import socket
import select

server = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
server.bind(("", 50000))
server.listen(1)

clients = []

try:
    while True:
        lesen, schreiben, oob = select.select([server] +
clients,
                                              [], [])

        for sock in lesen:
            if sock is server:
                client, addr = server.accept()
                clients.append(client)
                print "+++ Client %s verbunden" % addr[0]
            else:
                nachricht = sock.recv(1024)
                ip = sock.getpeername()[0]
                if nachricht:
                    print "[%s] %s" % (ip, nachricht)
                else:
                    print "+++ Verbindung zu %s beendet" %
ip
                    sock.close()
                    clients.remove(sock)
finally:
    for c in clients:
        c.close()
    server.close()
```

Zunächst wird ein Socket `server` erzeugt, der dazu gedacht ist, eingehende Verbindungsanfragen zu akzeptieren. Zudem wird die leere Liste `clients` angelegt, die später alle verbundenen Client-Sockets enthalten soll. Die darauf folgende `try`/`except`-Anweisung hat die Aufgabe, alle verbundenen Sockets ordnungsgemäß durch Aufruf von `close` zu schließen, wenn das Programm beendet wird.

Viel interessanter ist aber die Endlosschleife innerhalb des `try`-Zweiges, in der zunächst die Funktion `select` aufgerufen wird. Dabei werden alle geöffneten Sockets, inklusive des Server-Sockets, als erster Parameter übergeben. Die von `select` zurückgegebenen Listen werden von `lesen`, `schreiben` und `oob` referenziert, wobei wir uns nur für die Liste `lesen` näher interessieren.

Nach dem Aufruf von `select` wird über die zurückgegebene Liste `lesen` iteriert und in jedem Iterationsschritt überprüft, ob es sich bei dem betrachteten Socket um den Server-Socket handelt. Wenn das der Fall ist, wenn also beim Server-Socket Daten zum Einlesen bereitstehen, bedeutet dies, dass eine Verbindungsanfrage vorliegt. Wir akzeptieren die Verbindung, fügen den neuen Socket in die Liste `clients` ein und geben eine entsprechende Meldung aus.

Wenn Daten bei einem Client-Socket eingegangen sind, bedeutet dies, dass entweder eine Nachricht von diesem eingetroffen ist oder dass die Verbindung beendet wurde. Um zu testen, welcher der beiden Fälle eingetreten ist, lesen wir die vorhandenen Daten mittels `recv` aus. Wenn die Verbindung seitens des Clients beendet wurde, gibt `recv` einen leeren String zurück. In diesem Fall löschen wir diesen Socket aus der Liste `clients` und geben eine entsprechende Meldung aus.

Der Vollständigkeit halber folgt hier noch der Quelltext des zu diesem Server passenden Clients:

```
import socket

ip = raw_input("IP-Adresse: ")
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, 50000))

try:
    while True:
        nachricht = raw_input("Nachricht: ")
        s.send(nachricht)
finally:
```

```
    s.close()
```

Dabei handelt es sich tatsächlich um reine Socket-Programmierung, wie wir sie bereits in den vorherigen Kapiteln behandelt haben. Beachten Sie, dass der Client, abgesehen von eventuell auftretenden Latenzen, nicht bemerkt, ob er von einem seriellen oder einem multiplexenden Server bedient wird.

20.1.8 SocketServer

Sie werden festgestellt haben, dass das Schreiben eines Servers unter Verwendung des Moduls socket mitunter eine sehr komplexe Aufgabe sein kann. Aus diesem Grund enthält Pythons Standardbibliothek das Modul SocketServer, das es erleichtern soll, einen Server zu schreiben, der in der Lage ist, mehrere Clients zu bedienen.

Im folgenden Beispiel soll der Chat-Server des vorherigen Abschnitts mit dem Modul SocketServer nachgebaut werden. Dazu muss zunächst ein sogenannter *Request Handler* erstellt werden. Das ist eine Klasse, die von der Basisklasse SocketServer.BaseRequestHandler abgeleitet wird. Im Wesentlichen muss in dieser Klasse die Methode handle überschrieben werden, in der die Kommunikation mit einem Client ablaufen soll:

```
import SocketServer

class ChatRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        addr = self.client_address[0]
        print "[%s] Verbindung hergestellt" % addr
        while True:
            s = self.request.recv(1024)
            if s:
                print "[%s] %s" % (addr, s)
            else:
                print "[%s] Verbindung geschlossen" % addr
                break
```

Hier wurde die Klasse ChatRequestHandler erzeugt, die von BaseRequestHand ler erbt. Später wird von der SocketServer-Instanz bei jeder hergestellten Verbindung eine neue Instanz dieser Klasse erzeugt und die Methode handle aufgerufen. In dieser Methode läuft dann die Kommunikation mit dem verbundenen Client ab. Zusätzlich zur Methode handle können noch die Methoden setup und finish überschrieben werden, die entweder vor (setup) oder nach (finish) dem Aufruf von handle aufgerufen werden.

In unserem Beispiel werden innerhalb der Methode handle in einer Endlosschleife eingehende Daten eingelesen. Wenn ein leerer String eingelesen wurde, wird die Verbindung vom Kommunikationspartner geschlossen. Andernfalls wird der gelesene String ausgegeben.

Damit ist die Arbeit am Request Handler beendet. Was jetzt noch fehlt, ist der Server, der eingehende Verbindungen akzeptiert und daraufhin den Request Handler instanziiert:

```
server = SocketServer.ThreadingTCPServer(("", 50000),
                                         ChatRequestHandler)
server.serve_forever()
```

Um den tatsächlichen Server zu erzeugen, wird eine Instanz der Klasse Threa dingTCPServer erzeugt. Dem Konstruktor wird dabei ein Adress-Tupel und die soeben erstellte Request Handler-Klasse ChatRequestHandler übergeben. Durch Aufruf der Methode serve_forever der ThreadingTCPServer-Instanz wird der Server dazu instruiert, eine unbestimmte Anzahl an Verbindungen einzugehen.

Beachten Sie, dass der Programmierer selbst Verantwortung für eventuell von mehreren Threads gemeinsam genutzte Ressourcen trägt. Diese müssen gegebenenfalls durch Critical Sections abgesichert werden.

Neben der Klasse `ThreadingTCPServer` können auch andere Klassen instanziiert werden, je nachdem, wie sich der Server verhalten soll. Die Schnittstelle der Konstruktoren ist immer dieselbe.

SocketServer.TCPServer, SocketServer.UDPServer

Ein einfacher TCP- bzw. UDP-Server. Beachten Sie, dass diese Server immer nur eine Verbindung gleichzeitig eingehen können. Aus diesem Grund ist die Klasse `TCPServer` für unser Beispielprogramm nicht einsetzbar.

SocketServer.ThreadingTCPServer, SocketServer.ThreadingUDPServer

Diese Klassen implementieren einen TCP- bzw. UDP-Server, der jede Anfrage eines Clients in einem eigenen Thread behandelt, sodass der Server mit mehreren Clients gleichzeitig in Kontakt sein kann. Damit ist die Klasse `ThreadingTCPSer ver` ideal für unser obiges Beispiel.

SocketServer.ForkingTCPServer, SocketServer.ForkingUDPServer

Diese Klassen implementieren einen TCP- bzw. UDP-Server, der jede Anfrage eines Clients in einem eigenen Prozess behandelt, sodass der Server mit mehreren Clients gleichzeitig in Kontakt sein kann. Beachten Sie dabei, dass die Methode `handle` des Request Handlers in einem eigenen Prozess ausgeführt wird, also nicht auf Instanzen des Hauptprozesses zugreifen kann.

Die Klasse SocketServer

An dieser Stelle sollen die wichtigsten Attribute und Methoden der Klasse `SoketServer` besprochen werden. Im Folgenden sei *s* eine Instanz der Klasse `SocketServer`.

s.address_family

Dieses Attribut referenziert ein Adress-Tupel, das die IP-Adresse und die Portnummer enthält, auf denen der Server *s* nach eingehenden Verbindungsanfragen horcht.

s.socket

Dieses Attribut referenziert die von dem Server verwendete `Socket`-Instanz.

s.fileno()

Gibt den Dateideskriptor des Server-Sockets zurück.

s.handle_request()

Instruiert den Server, genau eine Verbindungsanfrage zu akzeptieren und zu behandeln.

s.serve_forever()

Instruiert den Server, eine unbestimmte Anzahl von Verbindungsanfragen zu akzeptieren und zu behandeln.

Die Klasse BaseRequestHandler

Die Klasse `BaseRequestHandler` bietet einige Methoden und Attribute, die überschrieben oder verwendet werden können.

Beachten Sie, dass eine Instanz der Klasse `BaseRequestHandler` immer für einen verbundenen Client zuständig ist. Im Folgenden sei *rh* eine Instanz der Klasse `BaseRequestHandler`.

rh.request

Über das Attribut `request` können Informationen über die aktuelle Anfrage eines Clients herausgefunden werden. Bei einem TCP-Server referenziert `request` die Socket-Instanz, die zur Kommunikation mit dem Client verwendet wird. Mit dieser können Daten gesendet oder empfangen werden. Bei Verwendung des verbindungslosen UDP-Protokolls referenziert `request` einen String, der die gesendeten Daten enthält.

rh.client_address

Das Attribut `client_address` referenziert ein Adress-Tupel, das die IP-Adresse und die Portnummer des Clients enthält, dessen Anfrage mit dieser `BaseRequest Handler`-Instanz behandelt wird.

rh.server

Das Attribut `server` referenziert den verwendeten Server, also eine Instanz der Klassen `TCPServer`, `UDPServer`, `ThreadingTCPServer`, `ThreadingUDPServer`, `ForkingTCPServer` oder `ForkingUDPServer`.

rh.handle()

Diese Methode sollte überschrieben werden. Wenn der Server eine Verbindungsanfrage eines Clients akzeptiert hat, wird eine neue Instanz der Request-Handler-Klasse erzeugt und diese Methode aufgerufen.

rh.setup()

Diese Methode kann überschrieben werden und wird stets vor dem Aufruf von `handle` aufgerufen.

rh.finish()

Diese Methode kann überschrieben werden und wird stets nach dem Aufruf von `handle` aufgerufen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Intelligente Fehler zu machen, ist eine große Kunst« – Federico Fellini

21 Debugging

Das sogenannte *Debugging* bezeichnet das Aufspüren und Beseitigen von Fehlern, sogenannten *Bugs*, in einem Programm. Üblicherweise steht dem Programmierer dabei ein sogenannter *Debugger* zur Verfügung. Das ist ein wichtiges Entwicklerwerkzeug, das es ermöglicht, den Ablauf eines Programms zu überwachen und an bestimmten Stellen anzuhalten. Wenn der Programmablauf in einem Debugger angehalten wurde, kann der momentane Programmstatus genau analysiert werden. Auf diese Weise können Fehler sehr viel schneller gefunden werden als durch bloßes gedankliches Durchgehen des Quellcodes.

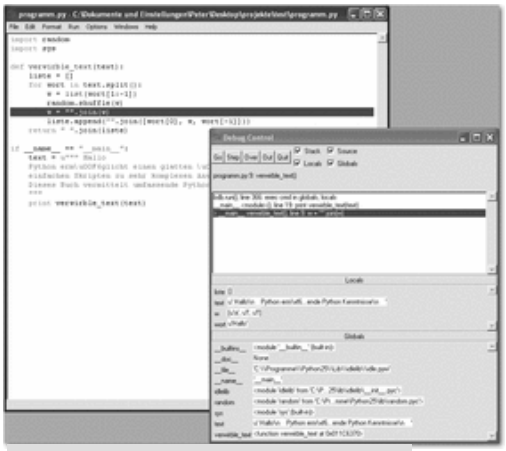
Im ersten Abschnitt widmen wir uns dem Debugging allgemein. Danach sollen Module erläutert werden, die nicht direkt etwas mit dem Debugger zu tun haben, sondern allgemein bei der Fehlersuche in einem Programm hilfreich sind. So wird beispielsweise erklärt, welche Möglichkeiten Python zum Erstellen einer Laufzeitanalyse oder automatisierter Tests bietet.

top

21.1 Der Debugger

Im Lieferumfang von Python ist ein Programm zum Debuggen von Python-Code enthalten, der sogenannte *PDB* (*Python Debugger*). Dieser Debugger läuft in einem Konsolenfenster und ist damit weder übersichtlich noch intuitiv. Aus diesem Grund haben wir uns dagegen entschieden, den PDB an dieser Stelle zu besprechen. Sollten Sie dennoch Interesse an diesem Debugger haben, beispielsweise gerade wegen seiner kommandozeilenbasierenden Benutzerschnittstelle, so finden Sie nähere Informationen dazu in der Python-Dokumentation.

Viele moderne Entwicklungsumgebungen [Eine Auflistung der gängigsten Entwicklungsumgebungen für Python finden Sie im Anhang.] für Python bieten einen umfangreichen, integrierten Debugger mit grafischer Benutzeroberfläche, mit dem sich die Fehlersuche in einem Python-Programm recht komfortabel gestaltet. Auch IDLE bietet einen rudimentären grafischen Debugger (siehe Abbildung 21.1).



Hier klicken, um das Bild zu vergrößern

Abbildung 21.1 Der grafische Debugger von IDLE

Um den Debugger in IDLE zu aktivieren, müssen Sie in der Python Shell auf den Menüpunkt Debug • Debugger klicken und dann das auf Fehler zu untersuchende Programm ganz normal per Run • Run Module ausführen. Es erscheint zusätzlich zum Editorfenster ein Fenster, in dem die aktuell ausgeführte Codezeile steht. Durch einen Doppelklick auf diese Zeile wird sie im Programmcode hervorgehoben, sodass man stets weiß, wo genau man sich im Programmablauf befindet.

Da es abgesehen von IDLE noch eine Menge weitere Python-IDEs gibt und IDLE bei Weitem nicht das Nonplusultra ist, wäre es müßig, an dieser Stelle eine detaillierte Einführung in den grafischen Debugger von IDLE zu geben. Allerdings ähneln sich die Funktionen der diversen grafischen Debugger sehr stark, sodass wir allgemein darauf eingehen möchten, welche Funktionen ein grafischer Debugger in der Regel anbietet.

Das grundsätzliche Prinzip eines Debuggers ist es, dem Programmierer das schrittweise Ausführen eines Programms zu ermöglichen, um sich somit von Zeile zu Zeile ein genaues Bild davon zu machen, welche Änderungen sich ergeben haben und wie sich diese im Laufe des Programms auswirken. Eine sogenannte *Debugging-Session* beginnt zumeist damit, dass der Programmierer sogenannte *Breakpoints* im Programm verteilt. Beim Starten des Debuggers wird das Programm normal ausgeführt, bis der Programmfluss auf den ersten Breakpoint stößt. An dieser Stelle hält der Debugger den Programmlauf an und erlaubt das Eingreifen des Programmierers. Viele Debugger halten auch direkt nach dem Starten an der ersten Programmzeile und warten auf weitere Instruktionen des Programmierers.

Wenn das Programm angehalten wurde und der Programmfluss somit an einer bestimmten Zeile im Quellcode steht, hat der

Programmierer mehrere Möglichkeiten, den weiteren Programmlauf zu steuern. Diese Möglichkeiten, im Folgenden Befehle genannt, finden Sie in einem grafischen Debugger üblicherweise an prominenter Stelle in einer Toolbar am oberen Rand des Fensters, da es sich dabei wirklich um die essenziellen Fähigkeiten eines Debuggers handelt.

- Mit dem Befehl *Step over* wird der Debugger dazu veranlasst, zur nächsten Quellcodezeile zu springen und dort erneut zu halten.
- Der Befehl *Step into* verhält sich ähnlich wie *Step over*, mit dem Unterschied, dass bei *Step into* auch in Funktions- oder Methodenaufrufe hineingesprungen wird, während diese bei *Step over* übergangen werden.
- Der Befehl *Step out* springt aus der momentanen Unterfunktion heraus wieder dorthin, wo die Funktion aufgerufen wurde. *Step out* kann damit gewissermaßen als Umkehrfunktion zu *Step into* gesehen werden.
- Der Befehl *Run* führt das Programm weiter aus, bis der Programmfluss auf den nächsten Breakpoint stößt oder das Programmende eintritt. Einige Debugger erlauben es mit einem ähnlichen Befehl, zu einer bestimmten Quellcodezeile zu springen oder den Programmcode bis zur Cursorposition auszuführen.

Neben diesen Befehlen, mit denen sich der Programmlauf steuern lässt, stellt ein Debugger einige Hilfsmittel bereit, mit deren Hilfe der Programmierer den Zustand des angehaltenen Programms vollständig erfassen kann. Welche dieser Hilfsmittel vorhanden sind und wie sie bezeichnet werden, ist von Debugger zu Debugger verschieden, dennoch möchten wir an dieser Stelle eine Übersicht über die gebräuchlichsten Hilfsmittel geben.

Das grundlegendste Hilfsmittel ist das Anzeigen einer Liste aller lokalen und globalen Referenzen mitsamt referenzierter Instanz, die im momentanen Programmkontext existieren. Auf diese Weise lassen sich Wertänderungen verfolgen und Fehler, die dabei entstehen, relativ leicht aufspüren.

Zusätzlich zu den lokalen und globalen Referenzen ist der sogenannte *Stack* von Interesse. In diesem wird die momentane Funktionshierarchie aufgelistet, sodass sich genau verfolgen lässt, welche Funktion welche Unterfunktion aufgerufen hat.

Gerade in Bezug auf die Programmiersprache Python bieten einige Debugger eine interaktive Shell, die sich im Kontext des angehaltenen Programms befindet und es dem Programmierer erlaubt, komfortabel Referenzen zu verändern, um somit in den Programmfluss einzugreifen.

Ein sogenannter *Post Mortem Debugger* kann in Anlehnung an den vorherigen Punkt betrachtet werden. In einem solchen Modus hält der Debugger das Programm erst an, wenn eine nicht abgefangene Exception aufgetreten ist. Im angehaltenen Zustand verfügt der Programmierer wieder über eine Shell sowie über die genannten Hilfsmittel, um dem Fehler auf die Spur zu kommen. Diese Form des Debuggens wird »post mortem« genannt, da sie erst nach dem Auftreten des tatsächlichen Fehlers, also nach dem »Tod« des Programms, aktiviert wird.

Mithilfe dieser Einführung in die Techniken des Debuggens und mit ein wenig Spieltrieb dürfte es für Sie kein Problem darstellen, den Debugger Ihrer favorisierten IDE in den Griff zu bekommen.

Abgesehen von dem eigentlichen Debugger, beinhaltet die Standardbibliothek Pythons noch einige Module, die speziell im Kontext des Debuggens von Bedeutung sind – sei es innerhalb der interaktiven Python Shell eines Debuggers oder völlig losgelöst vom Debugger. Diese Module sollen in den folgenden Abschnitten besprochen werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.” – Donald E. Knuth

22 Distribution von Python-Projekten

Es ist anzunehmen, dass Sie im Laufe dieses Buches bereits das eine oder andere eigenständige Python-Programm geschrieben haben. Vielleicht haben Sie sogar schon ein Programm oder Modul in Python geschrieben, das auch für andere Leute von Nutzen sein könnte. In diesem Moment stellt sich zwangsläufig die Frage, wie ein Python-Programm oder -Modul adäquat veröffentlicht werden kann. Idealerweise sollte es so geschehen, dass der Benutzer kein Experte sein muss, um es zu installieren.

Dazu ist in Pythons Standardbibliothek vor allem das Modul `distutils` enthalten, mit dem sich fertige Distributionen Ihres Programms oder Moduls erstellen lassen. Danach werden wir uns mit dem Drittanbietermodul py2exe beschäftigen, das Ihr Python-Programm zusammen mit dem gesamten Python-Interpreter zu einer einzigen ausführbaren Datei zusammenschweißt.

Zum Schluss gehen wir noch auf das Drittanbietermodul epydoc ein, mit dem sich sehr bequem eine Dokumentation zu Ihrem Programm oder Modul generieren lässt.

top

22.1 Erstellen von Distributionen – distutils

Das Paket `distutils` zielt auf die Distribution von Python-Modulen ab. Selbstverständlich könnte man einfach den Quellcode des Moduls ins Internet stellen und dazu eine Installationsanweisung liefern. Das scheint zwar aus Sicht des

Entwicklers erst einmal sehr bequem zu sein, ist aber aus Sicht des Benutzers nicht gerade komfortabel. Der Benutzer muss dann, um das Programm zu installieren, von Hand allerlei Dateien kopieren und sich dabei immer strikt an gewisse Regeln halten. Das kann man von einem Benutzer nicht verlangen. Der Installationsprozess soll durch das Paket `distutils` automatisiert werden, indem sogenannte *Distributionen* erstellt werden.

Bei einer Distribution unterscheidet man grundsätzlich zwei Typen:

- Eine sogenannte *Quellcodedistribution* (engl. *source distribution*) ist ein Archiv, das den Quellcode Ihres Moduls enthält. Zusätzlich zu dem Quellcode existiert ein Installationsscript namens *setup.py*, das die Installation des Moduls durchführt. Der Benutzer braucht diese Art einer Distribution also nur herunterzuladen, zu entpacken und das Installationsscript zu starten. Der Vorteil einer Quellcodedistribution ist ihre Plattformunabhängigkeit. Es muss also nur eine Distribution erstellt werden, die für jedes unterstützte Betriebssystem verwendet werden kann.
- Eine sogenannte *Binärdistribution* (engl. *binary distribution*) ist eine ausführbare Datei, die die Installation Ihres Moduls automatisch durchführt. Der Benutzer braucht diese Art einer Distribution also nur herunterzuladen und auszuführen. Eine Binärdistribution ist für den Benutzer besonders komfortabel, da er nur zwei Arbeitsschritte auszuführen hat. Allerdings bedeutet eine Binärdistribution mehr Aufwand für den Entwickler, denn das Installationsprogramm muss für verschiedene Plattformen erstellt werden.

Zum Erstellen einer Distribution sind mit dem `distutils`-Paket im Allgemeinen folgende Arbeitsschritte nötig:

- Schreiben Ihres Moduls oder Pakets [Eigentlich handelt es sich dabei nicht um einen Arbeitsschritt zum Erstellen einer Distribution. Dennoch ist es einleuchtenderweise eine unverzichtbare Voraussetzung. Beachten Sie, dass Sie auch mehrere Module und/oder Pakete in eine gemeinsame Distribution verpacken können. Näheres dazu erfahren Sie im Laufe des Kapitels.]
- Schreiben des Installationsscripts *setup.py*
- Erstellen einer Quellcodedistribution bzw. einer Binärdistribution

Diese Arbeitsschritte sollen in den folgenden Abschnitten detailliert besprochen werden.

Hinweis

Grundsätzlich lassen sich mit `distutils` nicht nur Distributionen von Modulen oder Paketen erstellen, sondern auch von *Extensions* (dt. *Erweiterungen*). Solche Extensions können später wie ein Modul oder Paket eingebunden werden, sind aber im Gegensatz zu normalen Modulen oder Paketen in einer anderen Programmiersprache, üblicherweise C oder C++, geschrieben. Wir werden in Abschnitt 26.2.3, »Erzeugen der Extension«, auch auf die Verwendung von `distutils` im Zusammenhang mit Extensions eingehen.

top

22.1.1 Schreiben des Moduls ▼▲

Dieser Punkt sollte so weit klar sein. Rufen Sie sich aber noch mal ins Gedächtnis, dass es einen Unterschied zwischen einem *Modul* und einem *Paket* gibt. Während ein Modul aus nur einer Programmdatei besteht, ist ein Paket ein Ordner, der mehrere

Untermodule oder -Pakete enthalten kann. Ein Paket erkennt man an der Programmdatei *__init__.py* im Paketverzeichnis. Die Unterscheidung der Begriffe *Modul* und *Paket* wird beim Erstellen des Installationsscripts noch eine Rolle spielen.

An dieser Stelle soll das Beispielmodul entwickelt werden, auf das wir uns im gesamten Kapitel beziehen werden. Dabei handelt es sich um ein sehr einfaches Modul, um die grundlegende Funktionalität von `distutils` zu demonstrieren. Bemerkungen zu komplexeren Distributionen, die beispielsweise Pakete oder Ähnliches enthalten, finden Sie an der jeweiligen Stelle im Text.

Sinn und Zweck des Beispielmoduls ist es, einen beliebigen Text so zu verändern, dass er sich ähnlich wie dieser liest:

Nach eienr Stidue der Cmabridge Uinverstiaet, ist es eagl in wlehcer Reiehnfogle die Bchustebaen in Woeretrn vokrmomen.

Es ist nur withcig, dsas der ertse und lettze Bchusatbe an der ricthgien Stlele snid. Der Rset knan total falcsh sein und man knan es onhe Porbelme leesn.

Das ist so, wiel das mneschilcge Geihrn nihct jeedn Bchustbaen liset sodnern das Wrot als gaznes.

Das Modul stellt dabei eine Funktion `verwirble_text` bereit, die einen String übergeben bekommt und diesen dann so »verwirbelt« zurückgibt, dass nur der erste und letzte Buchstabe sicher auf ihrem Platz bleiben.

```
import random

def verwirble_text(text):
    liste = []
    for wort in text.split():
        w = list(wort[1:-1])
        random.shuffle(w)
        liste.append("".join([wort[0], "".join(w),
wort[-1]]))
    return " ".join(liste)
```

Die Funktion iteriert in einer Schleife über alle im übergebenen String enthaltenen Wörter. Bei jedem Schleifendurchlauf wird der Teilstring aus dem jeweiligen Wort extrahiert, der verwirbelt werden soll. Dabei wird sichergestellt, dass der erste und der letzte Buchstabe nicht in diesen Teilstring mit aufgenommen werden. Zum Verwirbeln der Buchstaben wird die Funktion `shuffle` des Moduls `random` verwendet. Schlussendlich werden der verwirbelte String, der Anfangsbuchstabe und der Endbuchstabe zusammengefügt und an den resultierenden Text angehängt.

Beachten Sie, dass die Funktion der Einfachheit halber von einem absolut gutartigen String ausgeht. Das bedeutet insbesondere, dass der String keine Satzzeichen enthalten sollte.

Im Folgenden soll nun eine Distribution dieses Moduls `verwirbeln` erstellt werden, damit auch andere Python-Programmierer möglichst komfortabel in den Genuss dieses überaus mächtigen Werkzeugs gelangen können.

top

22.1.2 Das Installationsscript ▼▲

Der erste Schritt zur Distribution des eigenen Moduls ist das Erstellen eines *Installationsscripts*. Dies ist eine Python-Programmdatei namens *setup.py*, über die später das Erstellen der Distribution abläuft. Auch die Installation einer Quellcodedistribution aufseiten des Benutzers geschieht durch Aufruf dieser Programmdatei.

In unserem Beispiel muss im Installationsscript nur die Funktion `setup` des Moduls `distutils.core` aufgerufen werden.

```
from distutils.core import setup

setup(
     name = "verwirbeln",
     version = "1.0",
     author = "Micky Maus",
     author_email = "micky@maus.de",
     py_modules = ["verwirbeln"]
     )
```

Diese Funktion bekommt verschiedene *keyword arguments* übergeben, die Informationen über das Modul enthalten. Zusätzlich bekommt die Funktion über den Parameter py_modules alle Programmdateien übergeben, die der Distribution angehören sollen. Auf diese Weise ist es auch möglich, mehrere selbst geschriebene Module in einer Distribution anzubieten.

Das ist tatsächlich schon alles. Diese Programmdatei kann jetzt dazu verwendet werden, das Modul auf einem beliebigen Rechner mit Python-Installation zu installieren oder eine Distribution des Moduls zu erstellen. Wie das im Einzelnen funktioniert, klären wir später, zunächst betrachten wir einmal die Funktion setup.

distutils.core.setup(arguments)

Die Funktion setup des Moduls distutils.core muss in der Programmdatei *setup.py* aufgerufen werden und stößt den jeweils gewünschten Installationsprozess an. Dazu müssen der Funktion verschiedene *keyword arguments* übergeben werden, die Informationen über das Modul bzw. Paket bereitstellen. Die folgende Tabelle listet die wichtigsten möglichen Argumente auf und klärt kurz ihre Bedeutung.

Wenn nichts anderes angegeben ist, handelt es sich bei den jeweiligen Parametern um Strings.

Parametername	Beschreibung
name	Der Name der Distribution
version	Die Versionsnummer der Distribution
description	Eine kurze Beschreibung der Distribution
long_description	Eine ausführliche Beschreibung der Distribution
author	Der Name des Autors
author_email	Die E-Mail-Adresse des Autors
maintainer	Der Name des Paketverwalters (Maintainer), sofern dies nicht der Autor selbst ist
maintainer_email	Die E-Mail-Adresse des Paketverwalters
url	Die URL einer Homepage mit weiteren Informationen zur Distribution
download_url	Die URL, unter der die Distribution direkt heruntergeladen werden kann
packages	Eine Liste von Strings, die die Namen aller Pakete enthält, die in der Distribution enthalten sein sollen.
package_dir	Ein Dictionary, über das Pakete in Unterverzeichnissen in die Distribution aufgenommen werden können. Näheres zur Verwendung von package_dir folgt weiter unten.
package_data	Ein Dictionary, über das Dateien, die zu einem Paket gehören, mit in die Distribution aufgenommen werden können. Näheres zur Verwendung von package_data finden Sie weiter unten.
py_modules	Eine Liste von Strings, die die Namen aller Python-Module enthält, die in der Distribution enthalten sein sollen
	Eine Liste von Strings, die die Namen aller

scripts	Scriptdateien enthält, die in der Distribution enthalten sein sollen
data_files	Eine Liste von Tupeln, über die zusätzliche Dateien in die Distribution mit aufgenommen werden können. Näheres zur Verwendung von data_files finden Sie weiter unten.
ext_modules	Eine Liste von distutils.core.Extension-Instanzen, die die Namen aller Python-Erweiterungen enthält, die kompiliert werden und in der Distribution enthalten sein sollen. Näheres zu diesem Thema erfahren Sie in Abschnitt 26.2.
script_name	Der Name des Installationsscripts, das in der Distribution verwendet werden soll. Dieser Parameter ist mit sys.argv[0], also dem Namen des Scripts vorbelegt, das gerade ausgeführt wird.
license	Ein String, der die Lizenz angibt, unter der die Distribution veröffentlicht wird.
console	Der Pfad zu einer Programmdatei, die mithilfe von py2exe zu einer ausführbaren Windows-Konsolenanwendung gemacht werden soll[2]
window	Der Pfad zu einer Programmdatei, die mithilfe von py2exe zu einer ausführbaren Windows-EXE gemacht werden soll. Dieser Parameter eignet sich für Programme mit grafischer Benutzeroberfläche, da kein Konsolenfenster angezeigt wird.

Tabelle 22.1 Mögliche Schlüsselwortparameter für »setup«

Distribution von Paketen

Wenn Ihr Projekt statt aus einzelnen Modulen aus einem oder mehreren Paketen besteht, müssen Sie die Namen aller Pakete, die in die Distribution aufgenommen werden sollen, über den Schlüsselwortparameter packages angeben: [Näheres zu py2exe erfahren Sie in Abschnitt 22.2.]

```
from distutils.core import setup
setup(
     […]
     packages = ["paket1", "paket2", "paket1.unterpaket1"]
     )
```

In diesem Fall werden die Pakete paket1 und paket2, die sich im Hauptverzeichnis befinden müssen, in die Distribution aufgenommen. Zusätzlich wird noch das Paket unterpaket1 aufgenommen, das sich innerhalb des Pakets paket1 befindet. Beachten Sie, dass Sie durchaus sowohl Pakete über packages, als auch einzelne Module über py_modules in die Distribution aufnehmen können.

Natürlich möchte Sie niemand dazu zwingen, das Installationsscript im Hauptordner abzulegen, in dem möglicherweise bereits relativ viele Programmdateien liegen. Oftmals existiert neben dem Installationsscript ein Ordner *src* oder *source*, in dem sich dann die Module oder Pakete der Distribution befinden. Um solch einen Unterordner im Installationsscript bekannt zu machen, wird der Schlüsselwortparameter package_dir beim Aufruf von setup übergeben:

```
from distutils.core import setup
setup(
     […]
     package_dir = {"" : "src"},
     packages = ["paket1", "paket2", "paket1.unterpaket1"]
     )
```

Distribution zusätzlicher Dateien

Neben Modulen und Paketen können noch weitere Dateien zu Ihrem Projekt gehören und sollten damit auch Platz in der Distribution finden. Dazu zählen zunächst einfache Scriptdateien. Diese implementieren beispielsweise ein Tool, das im Zusammenhang mit Ihrem Paket steht. Der Unterschied zwischen einem Modul und einer Scriptdatei ist der, dass das Modul selbst keinen Python-Code ausführt, sondern nur Funktionen oder Klassen bereitstellt, während eine Scriptdatei ein lauffähiges Programm enthält. Solche Scriptdateien können beim Aufruf von `setup` durch den Schlüsselwortparameter `scripts` übergeben werden. Dabei muss für `scripts`, wie für andere Parameter auch, eine Liste von Strings übergeben werden, die jeweils einen Dateinamen enthalten.

Ein kleiner Service, den das Paket `distutils` in Bezug auf Scriptdateien durchführt, ist das automatische Anpassen der Shebang-Zeile an das Betriebssystem, auf dem die Distribution installiert wird.

Die nächste Kategorie zusätzlicher Dateien sind Ressourcen, die von bestimmten Paketen benötigt werden und in diesen enthalten sind. Beispielsweise könnte das Paket `paket1` die beiden Dateien *hallo.txt* und *welt.txt* erfordern. In einem solchen Fall können diese Dateien über den Schlüsselwortparameter package_data in Form eines Dictionarys übergeben werden:

```
setup(
      […]
      packages = ["paket1", "paket2",
"paket1.unterpaket1"],
      package_data = {"paket1" : ["hallo.txt", "welt.txt"]}
      )
```

Anstatt jede Datei einzeln anzugeben, können auch Wildcards verwendet werden. So würde der Wert `["*.txt"]` alle Textdateien einbinden, die sich im Verzeichnis des Paketes `paket1` befinden.

Zu guter Letzt ist es möglich, sonstige Dateien mit in die Distribution aufzunehmen. Dazu zählen alle Dateien, die in keine der vorherigen Kategorien passen, beispielsweise Konfigurationsdateien, Hilfeseiten oder Ähnliches. Diese Dateien können über den Schlüsselwortparameter `data_files` beim Funktionsaufruf von `setup` als Liste von Tupeln übergeben werden:

```
setup(
      […]
      data_files = [("grafiken", ["test1.bmp",
"test2.bmp"])
                    ("config", ["programm.cfg"])]
      )
```

In diesem Fall werden die Dateien *test1.bmp* und *test2.bmp* aus dem Verzeichnis *grafiken* sowie die Datei *programm.cfg* aus dem Verzeichnis *config* in die Distribution übernommen. Die Verzeichnisse verstehen sich relativ zum Pfad des Installationsscripts. Beachten Sie, dass hier durchaus auch absolute Pfade, beispielsweise für eine systemweite Konfigurationsdatei, angegeben werden können.

Hinweis

Beachten Sie allgemein, dass Sie Ordner innerhalb eines Pfades immer durch einen einfachen Slash (/) voneinander trennen sollten. Das Paket `distutils` kümmert sich dann um die korrekte »Übersetzung« des Pfades in das Format des jeweiligen Betriebssystems.

top

22.1.3 Erstellen einer Quellcodedistribution

Nachdem das Installationsscript geschrieben wurde, kann mit dessen Hilfe eine Quellcodedistribution Ihres Pakets oder Moduls erstellt werden. Dazu wechseln Sie in das Verzeichnis, in dem das Installationsscript liegt, und führen es mit dem Argument `sdist` aus:

```
setup.py sdist
```

Dieser Befehl erzeugt die Quellcodedistribution im Unterordner `dist` nach dem Namensschema *Projektname-Version.Format*. Dabei kann das Format des Archivs über die Option `--formats` angegeben werden. Es ist zudem möglich, eine Distribution in mehreren Archivformaten zu erstellen:

```
setup.py sdist --formats=zip,gztar
```

Mögliche Werte sind dabei `zip` für ein zip-Archiv (**.zip*), `gztar` für ein gz-komprimiertes tar-Archiv (**.tar.gz*), `bztar` für ein bz2-komprimiertes tar-Archiv (**.tar.bz2*), `ztar` für ein Z-komprimiertes tar-Archiv (**.tar.Z*) sowie `tar` für ein unkomprimiertes tar-Archiv. Wenn die Option `--formats` nicht angegeben wurde, wird unter Windows ein zip-Archiv und unter Unix-Systemen ein gz-komprimiertes tar-Archiv erstellt.

In das Archiv werden alle Dateien aufgenommen, die im Installationsscript eingetragen wurden. Zusätzlich wird eine Datei namens *README* oder *README.txt* automatisch in das Archiv mit aufgenommen, sofern eine solche im selben Ordner wie das Installationsscript existiert.

Das resultierende Archiv, die Quellcodedistribution, kann jetzt veröffentlicht und verbreitet werden. Der Benutzer, der diese Distribution herunterlädt, kann Ihr Modul bzw. Ihr Paket so installieren, wie in Abschnitt 22.1.5 beschrieben wird.

Hinweis

Beim Erstellen einer Distribution wird eine Datei namens *MANIFEST* erzeugt. Diese Textdatei enthält die Pfade zu allen Dateien, die in die Distribution mit aufgenommen werden. Beim erneuten Erstellen der Distribution werden diese Pfade aus der *MANIFEST*-Datei wieder ausgelesen, sofern die Datei existiert.

Wenn das Installationsscript aktueller ist als die *MANIFEST*-Datei, wird die *MANIFEST*-Datei beim nächsten Erstellvorgang aktualisiert. Trotzdem ist es gelegentlich notwendig, dieses Aktualisieren explizit zu erzwingen:

```
setup.py sdist --force-manifest setup.py sdist --manifest-only
```

Mit diesen Aufrufen von *setup.py* wird das Aktualisieren der *MANIFEST*-Datei vor dem Erstellen der Distribution erzwungen bzw. ausschließlich die *MANIFEST*-Datei aktualisiert.

top

22.1.4 Erstellen einer Binärdistribution

Neben einer Quellcodedistribution ist das Erstellen einer Binärdistribution von besonderem Interesse, da diese den wenigsten Installationsaufwand hat. Umgekehrt bedeutet es allerdings mehr Arbeit für Sie, da für verschiedene Betriebssysteme ganz unterschiedliche Formate für Binärdistributionen erstellt werden müssen. Das prominenteste dieser Formate ist ein Windows Installer, aber auch RPM-Pakete für RPM-basierende Linux-Distributionen können erstellt werden.

Beachten Sie, dass Sie neben einer Binärdistribution stets auch

eine Quellcodedistribution Ihres Projekts veröffentlichen sollten, da es Betriebssysteme gibt, die weder mit einem RPM-Paket noch mit einem Windows Installer etwas anfangen können.

Zum Erzeugen einer Binärdistribution wird das Installationsscript mit den folgenden Argumenten aufgerufen werden:

Argument	Bedeutung
`bdist_rpm`	Erzeugt ein RPM-Paket für RPM-basierende Linux-Distributionen wie beispielsweise Fedora Core oder SuSE.
`bdist_wininst`	Erzeugt einen Windows Installer, der dazu da ist, ein Modul bzw. Paket auf einem Windows-System zu installieren. Es wird die ausführbare Datei *Projektname-Version-win32.exe* im Unterordner *dist* erzeugt.

Tabelle 22.2 Mögliche Schlüsselwortparameter für »setup«

Da alle Informationen, die zum Erstellen der Binärdistribution benötigt werden, bereits im Installationsscript angegeben wurden, ist das Erzeugen einer Binärdistribution tatsächlich mit den folgenden Aufrufen von *setup.py* erledigt:

```
setup.py bdist_wininst
setup.py bdist_rpm
```

Beachten Sie allgemein, dass eine Binärdistribution unabhängig von einer Quellcodedistribution erzeugt werden kann. Es ist aber sinnvoll, ein Projekt sowohl in Form einer Binärdistribution für Windows als auch in Form einer Quellcodedistribution für andere Betriebssysteme anzubieten.

Hinweis

Solange Ihr Projekt aus reinen Python-Modulen besteht, also weder Pakte noch Extensions beinhaltet, kann die Installationsdatei für Windows auch unter anderen Betriebssystemen, beispielsweise unter Linux, erzeugt werden. Sobald aber Pakete oder Erweiterungen enthalten sind, muss dafür ein Windows-System verwendet werden.

top

22.1.5 Beispiel für die Verwendung einer Distribution

Nachdem Sie jetzt das grundlegende Handwerkszeug zum Erstellen einer Binär- und Quellcodedistribution erlernt haben, sollen hier noch ein paar Worte zur Verwendung der Distributionen selbst folgen.

Zu einer Binärdistribution brauchen wir dabei nicht besonders viel zu sagen, denn die Installationsprozedur entspricht dem auf dem jeweiligen Betriebssystem üblichen Vorgehen.

Wie eine Quellcodedistribution installiert wird, ist hingegen nicht ganz intuitiv und sollte bei Ihren eigenen Distributionen unbedingt in einer Readme-Datei erklärt werden. Die Installation einer Quellcodedistribution verläuft nach dem folgenden Schema:

- Herunterladen der Distribution
- Entpacken des Archivs
- Ausführen der Programmdatei *setup.py* mit dem Argument `install`

Sie sehen, dass auch für die Installation einer Distribution die

Programmdatei *setup.py* verantwortlich ist:

```
setup.py install
```

Wenn die Programmdatei *setup.py* mit dem Argument `install` ausgeführt wird, installiert sie die Distribution in die Python-Umgebung, die auf dem System installiert ist. Beachten Sie, dass dafür, je nach System, Administrator- oder Root-Rechte erforderlich sind.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Premature optimization is the root of all evil« – Donald E. Knuth

23 Optimierung

Als Programmierer sollten Sie mit der Zeit einen gewissen Sinn für die Ästhetik und Eleganz eines Programms entwickeln, der Ihnen sagt, wann Ihr Programm »schön« ist. Zwei wichtige Punkte eines eleganten Programms sind die Einfachheit des Ansatzes und die Laufzeiteffizienz. Diese beiden Grundsätze stehen sich in gewissem Maße gegenüber, denn häufig ist der effizienteste Ansatz nicht gerade von Klarheit geprägt.

Aus diesem Grund ist es bei einem nicht-zeitkritischen Programm absolut legitim, Effizienz und Einfachheit gegeneinander abzuwägen und zu einem gesunden Kompromiss zu gelangen. Bei einem zeitkritischen Programm ist die beste Lösung hingegen immer die effizienteste. Wir werden uns in diesem Kapitel ausschließlich mit der Optimierung der Laufzeit eines Python-Programms beschäftigen. Beachten Sie, dass ein Programm abgesehen von der Laufzeit noch in Hinblick auf andere Bereiche optimiert werden kann. So ist es beispielsweise durchaus üblich, eine Speicherplatzoptimierung durchzuführen, und letztendlich kann ein Programm auch in Hinblick auf die Einfachheit und Klarheit des Quelltextes hin optimiert werden. Der Begriff »Optimierung« wird im Folgenden ausschließlich im Sinne von »Laufzeitoptimierung« verstanden.

Egal, welche Art von Programm Sie schreiben oder welchen Begriff von Eleganz Sie haben, es lohnt sich auf jeden Fall, einen Blick auf einige Optimierungsstrategien von Python-Programmen zu werfen. Viele der hier gezeigten Tipps sind sehr simpel und lassen sich problemlos zur Gewohnheit machen.

Beachten Sie, dass wir uns dabei rein auf Python-spezifische Optimierungsstrategien konzentrieren. Den höchsten Laufzeitgewinn erzielen Sie jedoch mit der Optimierung der Algorithmik selbst. Doch das Optimieren von Algorithmen ist ein Thema für sich und soll hier keine Erwähnung finden. Beachten Sie zudem, dass wir häufig kleinere Python-Codes einander gegenüberstellen werden, die den gleichen Effekt haben, sich

jedoch teils gravierend in ihrer Laufzeit unterscheiden. Die in diesem Zusammenhang als »falsch« dargestellte Alternative ist natürlich nicht tatsächlich falsch, sondern im Vergleich zur anderen Variante nur ineffizient. Dennoch führen beide Alternativen zum gesteckten Ziel, sodass Sie frei entscheiden können, welcher der Varianten Sie den Vorzug gewähren möchten.

top

23.1 Die Optimize-Option

Grundsätzlich können Sie das Laufzeitverhalten eines Python-Programms beeinflussen, indem Sie es mit der Kommandozeilenoption `-O` ausführen. Diese Option veranlasst den Interpreter dazu, den resultierenden Byte-Code zu optimieren. Das bedeutet, dass `assert`-Anweisungen und Konstrukte wie

```
if __debug__:
    mache_etwas()
```

nicht ins Kompilat aufgenommen werden und somit keinen Einfluss auf das Laufzeitverhalten des optimierten Programms mehr haben können. Der optimierte Byte-Code wird in Dateien mit der Dateiendung *.pyo* gespeichert.

Durch die Kommandozeilenoption `-OO` ist es möglich, das Programm über das normale Maß hinaus zu optimieren. Wenn dies gewünscht ist, werden alle im Quelltext enthaltenen Docstrings ignoriert und nicht mit in das Kompilat aufgenommen. Auch damit wird ein wenig mehr Laufzeiteffizienz erreicht, wenngleich sich der Gewinn in Grenzen halten sollte. Beachten Sie dann aber, dass Sie keine Docstrings mehr auslesen können, weil keine vorhanden sind. Es ist dann beispielsweise für die Built-in Function `help` nicht mehr möglich, eine Hilfeseite zu Elementen Ihres Moduls zu generieren.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»What you see is all you get« – Brian Kernighan

24 Grafische Benutzeroberflächen

Nachdem wir uns bisher ausschließlich mit schlichten Konsolenanwendungen beschäftigt haben, kann das Kribbeln in den Fingern und damit der Schritt zur grafischen Benutzeroberfläche nicht mehr länger unterdrückt werden. Im Gegensatz zur textorientierten Oberfläche von Konsolenanwendungen sind Programme mit grafischer Oberfläche intuitiver zu bedienen, grafisch ansprechender und werden im Allgemeinen als moderner empfunden. Die grafische Benutzeroberfläche eines Programms, auch *GUI* (*Graphical User Interface*) genannt, besteht zunächst aus sogenannten *Fenstern* (engl. *Windows*). Innerhalb dieser Fenster lassen sich beliebige sogenannte *Steuerelemente*, häufig auch Widgets oder Controls genannt, platzieren. Unter Steuerelementen versteht man einzelne Bedieneinheiten, aus denen sich die grafische Benutzeroberfläche als Ganzes zusammensetzt. So ist beispielsweise ein Button (dt. *Schaltfläche*) oder ein Textfeld ein Steuerelement.

Sowohl die Terminologie als auch die Implementierung einer grafischen Oberfläche hängt sehr stark davon ab, welche Bibliothek, auch *Toolkit* genannt, verwendet wird. Aus diesem Grund werden wir zunächst allgemein verschiedene Toolkits auflisten, die mit Python verwendet werden können, und erst im zweiten Abschnitt zur eigentlichen Programmierung einer grafischen Benutzeroberfläche kommen.

top

24.1 Toolkits

Unter einem Toolkit versteht man, wie bereits gesagt, eine Bibliothek, mit deren Hilfe sich Programme mit grafischer Benutzeroberfläche erstellen lassen. Neben einigen plattformabhängigen Toolkits, beispielsweise den MFC (*Microsoft Foundation Classes*) für Windows, sind gerade im Zusammenhang mit Python plattformunabhängige Toolkits wie Qt, Gtk oder wxWidgets interessant. Diese Toolkits sind zumeist für C (Gtk) oder C++ (Qt, wxWidgets) geschrieben, lassen sich jedoch durch sogenannte *Bindings* auch mit Python ansprechen. Im Folgenden sollen die wichtigsten Python-Bindings für GUI-Toolkits aufgelistet und kurz erläutert werden.

Tkinter

Website: http://wiki.python.org/moin/TkInter

Toolkit: Tk

Das Toolkit Tk wurde ursprünglich für die Sprache Tcl (*Tool Command Language*) entwickelt und ist das einzige Toolkit, das in der Standardbibliothek Pythons enthalten ist. Das Modul `Tkinter` (*Tk interface*) erlaubt es, Tk-Anwendungen zu schreiben, und bietet damit eine interessante Möglichkeit, kleinere Anwendungen mit einer grafischen Benutzeroberfläche zu versehen, für die der Benutzer später keine zusätzlichen Bibliotheken installieren muss. Besonders hervorzuheben ist die gute Unicode-Unterstützung des Tk Toolkits. Das Modul `Tkinter` ist in der Standardbibliothek nicht unumstritten und wird allgemein eher zur Entwicklung kleinerer Anwendungen eingesetzt.

Aus diesem Grund haben wir uns dagegen entschieden, das Modul `Tkinter` in diesem Buch zu beschreiben, sondern dem Qt-Framework den Vorrang gewährt. Wenn Ihr Interesse an `Tkinter`

geweckt wurde, finden Sie auf der angegebenen Website und in der Python-Dokumentation genügend Informationen, um schnell gute Resultate erzielen zu können.

Ein Beispiel für ein Tk-Programm ist die Entwicklungsumgebung IDLE, die jeder Python-Version beiliegt.

PyGtk

Website: http://www.pygtk.org

Toolkit: Gtk

Das Toolkit Gtk (GIMP Toolkit) wurde ursprünglich für das Grafikprogramm GIMP entwickelt und zählt heute neben Qt zu den am meisten verbreiteten plattformübergreifenden Toolkits. Sowohl das Toolkit selbst als auch die Python-Bindings PyGtk stehen unter der *GNU Lesser General Public License* und können frei heruntergeladen und verwendet werden.

Das Gtk-Toolkit ist die Grundlage der freien Desktop-Umgebung GNOME und erfreut sich daher, gerade unter Linux-Anwendern, einer hohen Beliebtheit. Obwohl es eigentlich für C geschrieben wurde, ist Gtk von Grund auf objektorientiert und kann somit gut mit Python verwendet werden.

Beachten Sie, dass ebenfalls Python-Bindings für die Bibliotheken der Desktop-Umgebung Gnome existieren. Diese heißen *PyGnome*. Mit diesen Bindings können Sie spezielle Features von Gnome nutzen und Ihre Anwendung somit perfekt in die Gnome-Umgebung integrieren.

PyQt

Website: http://www.riverbankcomputing.co.uk/pyqt

Toolkit: Qt

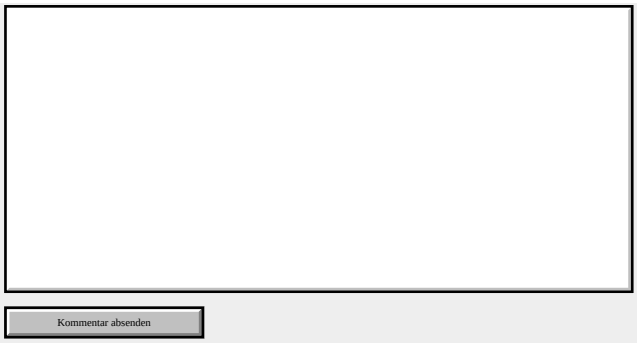
Bei Qt handelt es sich um ein umfassendes Framework, das von der norwegischen Firma *Trolltech* entwickelt wird und sowohl ein GUI-Toolkit als auch viel GUI-fremde Funktionalität enthält. Das durch und durch objektorientierte C++-Framework ist die Basis der freien Desktop-Umgebung KDE (*K Desktop Environment*) und aus diesem Grund ähnlich verbreitet und beliebt wie das Gtk-Toolkit. Sowohl Qt selbst als auch die Python-Bindings sind unter einem dualen Lizenzsystem erhältlich. Solange Sie Ihre Software unter einer freien Lizenz, beispielsweise der GPLv2, veröffentlichen, stehen Qt und PyQt unter der GPL. Für den kommerziellen Gebrauch muss eine Lizenzgebühr entrichtet werden. Qt wird für viele kommerzielle Projekte verwendet, darunter zum Beispiel Google Earth, Skype oder Opera.

Da Qt einerseits sehr mächtig ist und andererseits eine wirklich konsequente und gut strukturierte objektorientierte API bereitstellt, werden wir uns im folgenden praxisorientierten Einstieg in die Programmierung grafischer Benutzeroberflächen auf Qt bzw. PyQt beziehen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name	
E-Mail	
Ihr Kommentar	



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Ich sage voraus, dass sich das Internet bald zu einer Supernova aufbläht und 1996 katastrophal kollabieren wird.« – Robert Metcalfe

25 Python als serverseitige Programmiersprache im WWW mit Django

In der heutigen Zeit unterscheidet sich die Technik im World Wide Web drastisch von ihren Anfängen: Das rein informative Netzwerk aus statischen HTML-Seiten hat sich zu einer interaktiven Austauschplattform entwickelt, mit der praktisch alles möglich ist. Man kann über das Internet einkaufen, seinen nächsten Urlaub buchen, die Nachrichten verfolgen, seine sozialen Kontakte in Chats oder auf Community-Seiten pflegen oder sogar bei der Gestaltung von Wissensdatenbanken wie der Wikipedia mitwirken.

Alle diese neuen Möglichkeiten verdankt das Internet im Wesentlichen den Programmen, die bei einem Aufruf einer Webseite dynamisch HTML-Code mit den geforderten Daten generieren. Die Webprogrammierung war lange Zeit hauptsächlich eine Domäne für die Scriptsprache PHP [PHP (rekursive Abkürzung von *PHP: Hypertext Preprozessor*) ist eine Skriptsprache, die für die Einbettung in HTML-Seiten entwickelt wurde. Die Syntax von PHP ist an die Programmiersprache C angelehnt. Nahezu jeder Hosting-Service bietet heute Unterstützung für PHP-Skripte an.] , die sich in der Vergangenheit – mangels Alternativen – durchsetzen konnte. Scriptsprachen eignen sich deshalb besonders für die Programmierung von Webanwendungen, weil sich die Entwicklungszeiten gegenüber maschinennahen Programmiersprachen erheblich verkürzen lassen, sodass man schnell neue Funktionen einbauen kann. Außerdem ist gerade bei Programmen im WWW die Ausführungsgeschwindigkeit weniger wichtig, da die meiste Zeit in der Regel nach dem Ende des Programms durch die verhältnismäßig langsame Verbindung zwischen Server und Client verloren geht.

Heutzutage ist PHP nicht mehr die einzige serverseitige

Programmiersprache, die die oben genannten Kriterien erfüllt. An die Seite von PHP gesellen sich Sprachen wie Perl, JAVA, Ruby und im Besonderen auch Python.

Da Pythons Standardbibliothek nur wenige Module enthält, mit denen sich effiziente Webentwicklung betreiben lässt, wurde für diesen Zweck eine Reihe von sogenannten *Webframeworks* entwickelt. Das Ziel dieser Frameworks besteht darin, dem Programmierer die immer wiederkehrenden Aufgaben abzunehmen, die sich bei der Erstellung von dynamischen Internetseiten ergeben. Dazu zählen insbesondere:

- die Kommunikation mit dem Webserver
- die Generierung von HTML-Ausgaben durch Template-Systeme
- der Zugriff auf Datenbanken

Im Optimalfall braucht der Entwickler sich nicht mehr um die Besonderheiten der Datenbank und des Servers zu kümmern, sondern kann sich darauf konzentrieren, die Funktionen seiner Anwendung zu implementieren.

Es gibt sehr viele Webframeworks für Python, die alle ihre Vor- und Nachteile haben. Wir werden im Rahmen dieses Buchs ein Framework namens *Django* behandeln, das sich durch seinen Funktionsumfang, seine Eleganz und seine weite Verbreitung auszeichnet. Um Sie mit den Grundlagen von Django vertraut zu machen, werden wir eine kleine Webanwendung implementieren, die News-Beiträge verwalten kann. Außerdem wird es für die Besucher der Seite möglich sein, Kommentare zu den einzelnen Meldungen abzugeben.

Django ist ein relativ junges Framework, das 2005 zum ersten Mal veröffentlicht wurde. Es ist kostenlos unter der BSD-Lizenz verfügbar und ist besonders für die Entwicklung komplexer, datenbankgestützter Anwendungen konzipiert.

Aufgrund des großen Umfangs von Django werden wir Ihnen in diesem Kapitel nur grundlegendes Wissen vermitteln können. Es gibt aber im Internet sehr viel gute Dokumentation zum Thema Django. Besonders sei Ihnen dabei die Homepage des Projekts, http://www.djangoproject.com, ans Herz gelegt.

top

25.1 Installation

Wir werden in diesem Abschnitt die Installation des Moduls *Django* beschreiben, die notwendig ist, um Django-Anwendungen zu entwickeln. Wie Sie fertige Anwendungen auf einem Webserver der Öffentlichkeit zur Verfügung stellen, können Sie in der Django-Dokumentation auf http://www.djangoproject.org/ nachlesen. [Stichwörter dazu sind Apache-Webserver, `mod_python` und WSGI.]

Django befindet sich zum aktuellen Zeitpunkt (Oktober 2007) in einer sehr schnellen Entwicklung. Seit dem letzten offiziellen Release mit der Versionsnummer 0.96 vom 23.03.2007 hat sich Django stark verändert. So wurden viele neue Features hinzugefügt und Fehler beseitigt. Aus diesem Grund raten die Django-Entwickler von der Verwendung der Version 0.96 ab und empfehlen, die aktuelle Entwicklerversion direkt aus dem *SVN-Repository* [***Subversion*** (***SVN***) ist ein System, mit dem Entwickler ihren Quellcode komfortabel verwalten können. Außerdem ist es uns als Benutzern der Programme über SVN möglich, auf die aktuellen Quellen zuzugreifen.] zu laden.

Wegen dieser Empfehlung und der Tatsache, dass Django sich bis zur geplanten Version 1.0 in vielen Punkten von der Version 0.96 unterscheiden wird, haben wir uns entschlossen, die SVN-Version zu beschreiben.

Um auf das SVN-Repository von Django zugreifen zu können, müssen Sie die SVN-Software auf Ihrem Computer installieren. Wenn Sie unter Linux arbeiten, können Sie in der Paket-Datenbank Ihrer Distribution nachsehen. In der Regel sollte sich dort ein entsprechendes Paket finden. Windows-Benutzer finden eine Installationsdatei für SVN auf der Buch-CD.

Hinweis

Wenn Sie Linux einsetzen, ist es nicht unwahrscheinlich, dass Ihre Distribution bereits aktuelle SVN-Snapshots von Django als Paket anbietet. Ist dies der Fall, können Sie besser darauf zurückgreifen und sich so die manuelle Installation ersparen.

Die Installation von Django erfolgt in den folgenden Schritten:

Herunterladen des aktuellen Quellcodes

Erstellen Sie einen Ordner, in dem Sie den Quellcode von Django abspeichern möchten. Wir haben als Beispiel den Ordner *C:\django-svn* gewählt. Nun öffnen Sie eine Kommandozeile und navigieren in den neu erstellten Ordner.

Den aktuellen Quellcode können Sie mit der folgenden Anweisung aus dem SVN-Repository laden: [Die Zahl 6525 im Listing wird sich wahrscheinlich von der Ausgabe auf Ihrem Bildschirm unterscheiden, denn es handelt sich dabei um eine Art Versionsnummer.]

```
$ svn co http://code.djangoproject.com/svn/django/trunk/
...
Checked out revision 6525.
```

Das Herunterladen der Daten kann je nach Verbindungsgeschwindigkeit Ihres Rechners einige Zeit in Anspruch nehmen.

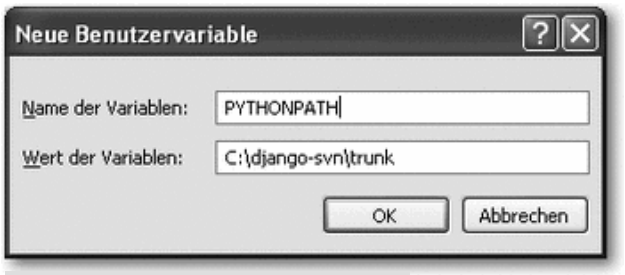
Das Django-Modul für Python importierbar machen

Damit Sie den Code nutzen können, muss Python das Django-Modul importieren können. Um das zu erreichen, gibt es mehrere Methoden, die je nach Ihrem Betriebssystem unterschiedlich gut für Sie geeignet sind.

Windows XP

Unter Windows erstellen Sie am besten eine Umgebungsvariable namens PYTHONPATH, die auf den Ordner *trunk* im Verzeichnis *django-svn* verweist. Dies können Sie in den Systemeigenschaften erledigen, die sich dadurch öffnen lassen, dass sie mit der rechten Maustaste auf **Arbeitsplatz** und im erscheinenden Kontextmenü auf **Eigenschaften** klicken.

Auf der Registerkarte **Erweitert** klicken Sie dann auf den Button **Umgebungsvariablen**. Anschließend erstellen Sie die neue Umgebungsvariable durch einen Klick auf den Button **Neu** oben im sich öffnenden Fenster. Nun können Sie die Werte für die neue Umgebungsvariable eingeben, wobei Sie den Pfad an den Ordner anpassen müssen, in dem Sie den Quellcode von Django gespeichert haben:



Hier klicken, um das Bild zu vergrößern

Abbildung 25.1 Umgebungsvariable PYTHONPATH anlegen

Nun können Sie Python starten und prüfen, ob Sie `django` importieren können:

```
>>> import django
>>>
```

Erscheint keine Fehlermeldung beim Importieren des Moduls `django`, ist der größte Teil der Konfiguration abgeschlossen.

Linux und Mac OS X

Unter Linux oder Mac OS X setzen Sie am besten einen symbolischen Link auf das Verzeichnis *django-svn/trunk/django* in den Python-Ordner *site-packages*. Wo sich *site-packages* befindet, können Sie folgendermaßen ermitteln:

```
$ python -c "from distutils.sysconfig import
get_python_lib; print get_python_lib()"
/usr/lib/python2.5/site-packages
```

Jetzt können Sie den symbolischen Link erzeugen, wenn Sie sich mit der Kommandozeile in dem Verzeichnis befinden, in das Sie den Django-Code heruntergeladen haben:

```
$ ln -s `pwd`/django-trunk/django
/usr/lib/python2.5/site-packages/django
```

Abschluss der Installation

Wir werden im weiteren Verlauf des Abschnitts das Programm *django-admin.py* im Verzeichnis *C:\django-svn\trunk\django\bin* verwenden. Damit Sie nicht immer den kompletten Pfad angeben müssen, sollten Sie das Programm *django-admin.py* in einen Ordner kopieren, der in der Umgebungsvariable `PATH` angegeben ist.

Dafür kommt unter Linux beispielsweise der Pfad */usr/local/bin* und unter Windows üblicherweise der Pfad *C:\Windows* infrage.

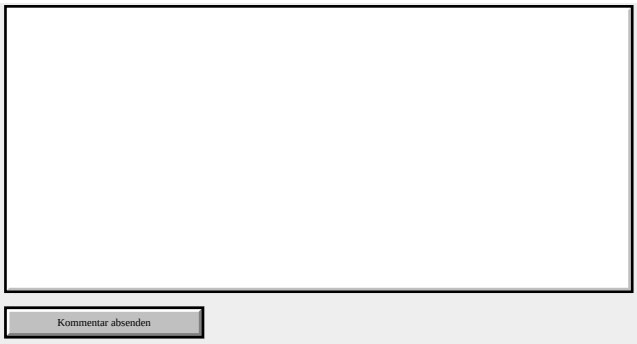
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Wenn die Sprache nicht stimmt, ist das, was gesagt wird, nicht das, was gemeint ist.« – Konfuzius

26 Anbindung an andere Programmiersprachen

Dieser Abschnitt beschäftigt sich mit der Interoperabilität zwischen Python und anderen Programmiersprachen, hier ausschließlich C. Grundsätzlich gibt es dabei zwei Strategien, ein Programm zu schreiben, das Python mit einer anderen Sprache kombiniert:

1. In einem Python-Programm soll C-Code ausgeführt werden.
2. In einem C-Programm soll ein Python-Script ausgeführt werden.

Zu 1.: In einem größeren Projekt kann Python als sehr komfortable und gut zu wartende Sprache beispielsweise für die Programmlogik eingesetzt werden, während man einige wenige zeitkritische Algorithmen des Projekts aus Effizienzgründen in einer nicht interpretierten Sprache wie C oder C++ schreibt. Zu diesem Ansatz besprechen wir im ersten Abschnitt, wie Sie mit dem Modul ctypes der Standardbibliothek auf dynamische Bibliotheken, beispielsweise Windows-DLLs, zugreifen können. Der zweite Abschnitt soll Möglichkeiten aufzeigen, C- oder C++-Code direkt in den Python-Quelltext einzubetten.

Zu 2.: Häufig möchte man auch den umgekehrten Weg beschreiten und in einem größeren C/C++-Projekt Python als

eingebettete Scriptsprache für dynamische Elemente des Programms verwenden. In einem Computerspiel könnte beispielsweise C/C++ für die hauptsächlich laufzeitkritische Hauptanwendung und die gesamte Algorithmik verwendet werden, während Python für die dynamischen Elemente des Spiels, beispielsweise Ereignisse in bestimmten Leveln oder das Verhalten verschiedener Spielfiguren, verwendet wird. Dieser Ansatz soll in Abschnitt 26.3 verfolgt werden.

Grundsätzlich benötigen Sie zum Verständnis der folgenden Kapitel zumindest rudimentäre Kenntnisse der Programmiersprache C.

top

26.1 Dynamisch ladbare Bibliotheken – ctypes

Mit dem Modul ctypes ist es möglich, Funktionen einer sogenannten *dynamisch ladbaren Bibliothek*, im Folgenden *dynamische Bibliothek* genannt, aufzurufen. Zu solchen dynamischen Bibliotheken zählen beispielsweise DLL-Dateien (*Dynamic Link Library*) unter Windows oder SO-Dateien (Shared Object) unter Linux bzw. Unix.

Das Aufrufen von Funktionen einer dynamischen Bibliothek ist besonders dann sinnvoll, wenn bestimmte laufzeitkritische Teile eines Python-Programms in einer hardwarenäheren und damit effizienteren Programmiersprache geschrieben werden sollen oder wenn man schlicht eine in C oder C++ geschriebene Bibliothek in Python nutzen möchte.

Beachten Sie grundsätzlich, dass das Erstellen einer dynamischen Bibliothek keine Eigenschaft der Programmiersprache C ist. Im Gegenteil: Eine dynamische Bibliothek kann als eine sprachunabhängige Schnittstelle zwischen verschiedenen Programmen betrachtet werden. Es ist absolut möglich, ein Python-Programm zu schreiben, das auf eine in C geschriebene dynamische Bibliothek zugreift, die ihrerseits auf eine dynamische Bibliothek zugreift, die in Pascal geschrieben wurde. Dies gilt allerdings nur für Sprachen, die sich zu einer dynamischen Bibliothek kompilieren lassen: PHP würde beispielsweise außen vor bleiben.

top

26.1.1 Ein einfaches Beispiel

Zum Einstieg in das Modul ctypes soll ein einfaches Beispiel dienen. Im Beispielprogramm soll die dynamische Bibliothek der *C Runtime Library* eingebunden und die darin enthaltene Funktion printf dazu genutzt werden, den Text »Hallo Welt« auszugeben. Die C Runtime Library ist unter Windows unter dem Namen *msvcrt.dll* und unter Unix-ähnlichen Systemen unter dem Namen *libc.so.6* zu finden. Dazu betrachten wir zunächst den Quellcode des Beispielprogramms:

```
import ctypes
bibliothek = ctypes.CDLL("MSVCRT")
bibliothek.printf("Hallo Welt\n")
```

Zunächst wird das Modul ctypes eingebunden und dann eine Instanz der Klasse CDLL erzeugt. Eine Instanz dieser Klasse repräsentiert eine geladene dynamische Bibliothek. Beachten Sie, dass die Dateiendung *.dll* unter Windows weggelassen und der Name der Bibliothek großgeschrieben werden muss, wenn eine System-Bibliothek geladen werden soll. Unter Linux sähe die Instanziierung der Klasse CDLL beispielsweise so aus:

```
bibliothek = ctypes.CDLL("libc.so.6")
```

Nachdem die dynamische Bibliothek eingebunden worden ist, kann

die Funktion `printf` wie eine Methode der `CDLL`-Instanz aufgerufen werden. [Beachten Sie, dass die Funktion `printf` nicht nach `sys.stdout`, sondern in den tatsächlichen `stdout`-Stream des Betriebssystems schreibt. Das bedeutet für Sie, dass Sie im obigen Beispiel nur dann eine Ausgabe sehen, wenn Sie das Programm in einer Python-Shell ausführen. Keine Ausgabe erscheint dagegen beispielsweise in IDLE.] In diesem Fall können wir ganz unbesorgt einen Python-String übergeben. Behalten Sie bei der Arbeit mit `ctypes` aber immer im Hinterkopf, dass es teilweise große Unterschiede zwischen den Datentypen von C/C++ und denen von Python gibt. Es können also nicht alle Funktionen so einfach verwendet werden. Im Laufe dieses Kapitels werden wir noch eingehend darauf zurückkommen.

top

26.1.2 Die eigene Bibliothek ▼▲

An dieser Stelle soll eine dynamische Bibliothek erstellt werden, auf die wir dann in den folgenden Abschnitten zugreifen werden. Die Bibliothek ist in C geschrieben und enthält drei Funktionen mit unterschiedlich komplexer Schnittstelle. Wir werden hier nur den Quelltext der drei Funktionen zeigen, auf die wir uns in den folgenden Beispielen beziehen. Es wird keine Anleitung geben, wie Sie den C-Code zu einer dynamischen Bibliothek kompilieren können oder Ähnliches. Das ist nicht Gegenstand dieses Buches und würde sich zudem bei den verschiedensten Betriebssystemen und Entwicklungsumgebungen zum Teil stark unterscheiden. Sie finden dazu nach einer kurzen Recherche im Internet genügend Anleitungen.

```
// Berechnet die Fakultaet einer ganzen Zahl
int fakultaet(int n)
    {
    int i;
    int ret = 1;
    for(i = 1; i <= n; i++)
        ret *= i;
    return ret;
    }

// Berechnet die Laenge eines Vektors im R3
double veclen(double x, double y, double z)
    {
    return sqrt(x*x + y*y + z*z);
    }

// Bubblesort
void sortiere(int *array, int len)
    {
    int i, j, tmp;
    for(i = 0; i < len; i++)
        {
        for(j = 0; j < i; j++)
            {
            if(array[j] > array[i])
                {
                tmp = array[j];
                array[j] = array[i];
                array[i] = tmp;
                }
            }
        }
    }
```

Die erste Funktion, `fakultaet`, berechnet die Fakultät einer ganzen Zahl und gibt das Ergebnis ebenfalls in Form einer ganzen Zahl zurück. Die zweite Funktion, `veclen`, berechnet die Länge eines dreidimensionalen Vektors. Sie bekommt dazu die drei Koordinaten des Vektors in Form von drei Gleitkommazahlen übergeben und gibt die Länge des Vektors ebenfalls in Form einer Gleitkommazahl zurück.

Die dritte, etwas komplexere Funktion, `sortiere`, implementiert den sogenannten *Bubblesort*-Algorithmus, um ein Array von beliebig vielen ganzen Zahlen aufsteigend zu sortieren. Dazu bekommt die Funktion einen Pointer auf das erste Element sowie die Anzahl der Elemente des Arrays übergeben.

Im Folgenden gehen wir davon aus, dass der oben stehende Quellcode zu einer dynamischen Bibliothek kompiliert wurde und unter dem Namen *bibliothek.dll* im jeweiligen Programmverzeichnis

der kommenden Beispielprogramme zu finden ist. Sollten Sie ein Unix-ähnliches Betriebssystem wie beispielsweise Linux einsetzen, müssen Sie zur Adaption der Beispielprogramme den Hinweis des vorherigen Abschnitts beachten.

Datentypen

An dieser Stelle haben wir eine fertige dynamische Bibliothek mit drei Funktionen, die wir jetzt mittels ctypes aus einem Python-Programm heraus aufrufen können. Der praktischen Umsetzung dieses Vorhabens stehen jedoch die teilweise inkompatiblen Datentypen von C und Python im Wege. Solange Instanzen der Datentypen long, int, str, unicode oder NoneType [Wenn die Instanz None an eine C-Funktion übergeben wird, kommt sie dort als NULL-Pointer an. Umgekehrt wird ein von einer C-Funktion zurückgegebener NULL-Pointer in Python zu None.] übergeben werden, funktioniert der Funktionsaufruf einwandfrei, denn diese Instanzen können eins zu eins nach C übertragen werden. So ist beispielsweise der Aufruf der Funktion fakultaet mit keinerlei Problemen behaftet:

```
from ctypes import CDLL
bib = CDLL("bibliothek.dll")
print bib.fakultaet(5)
```

Doch bereits das Übergeben einer float-Instanz scheitert. Für diesen und andere Datentypen von C implementiert das Modul ctypes entsprechende Gegenstücke in Python, deren Instanzen über die Schnittstelle geschickt werden können. Die folgende Tabelle listet alle in ctypes enthaltenen Datentypen sowie ihre Entsprechungen in C und Python auf.

Datentyp (ctypes)	Datentyp (C)	Datentyp (Python)
c_char	char	str (ein Zeichen)
c_wchar	wchar_t	unicode (ein Zeichen)
c_byte	char	int, long
c_ubyte	unsigned char	int, long
c_short	short	int, long
c_ushort	unsigned short	int, long
c_int	int	int, long
c_uint	unsigned int	int, long
c_long	long	int, long
c_ulong	unsigned long	int, long
c_longlong	__int64, long long	int, long
c_ulonglong	unsigned __int64, unsigned long long	int, long
c_float	float	int, long
c_double	double	int, long
c_char_p	char *	str, None
c_wchar_p	wchar_t *	unicode, None
c_void_p	void *	int, long, None

Tabelle 26.1 Datentypen des Moduls »ctypes«

All diese ctypes-Datentypen können durch Aufruf ihres Konstruktors instanziiert und mit einer Instanz des angegebenen Python-Datentyps initialisiert werden. Über das Attribut value kann der jeweilige Wert eines ctypes-Datentyps verändert werden.

```
>>> import ctypes
>>> f = ctypes.c_float(1.337)
>>> print f
c_float(1.3370000123977661)
>>> d = ctypes.c_double(1.337)
```

```
>>> print d
c_double(1.337)
>>> s = ctypes.c_char_p("Hallo Welt")
>>> print s
c_char_p('Hallo Welt')
>>> null = ctypes.c_void_p(None)
>>> print null
c_void_p(None)
```

Um ein Array eines bestimmten Datentyps anzulegen, wird der zugrunde liegende Datentyp mit der Anzahl der Elemente, die er aufnehmen soll, multipliziert. Das Ergebnis ist ein Datentyp, der das gewünschte Array speichert. Im konkreten Beispiel sieht das so aus:

```
>>> arraytyp = ctypes.c_int * 5
>>> a = arraytyp(1, 5, 2, 1, 9)
>>> a
<__main__.c_long_Array_5 object at 0xb7af82fc>
```

Einen solchen Array-Typ können wir beispielsweise der Funktion `sortiere` übergeben, die ein Array von ganzen Zahlen sortiert:

```
from ctypes import CDLL, c_int
bib = CDLL("bibliothek.dll")
arraytyp = c_int * 10
a = arraytyp(0,2,5,2,8,1,4,7,3,8)
print "Vorher: ", [i for i in a]
bib.sortiere(a, 10)
print "Nachher: ", [i for i in a]
```

Die Ausgabe dieses Beispielprogramms lautet:

```
Vorher:  [0, 2, 5, 2, 8, 1, 4, 7, 3, 8]
Nachher:  [0, 1, 2, 2, 3, 4, 5, 7, 8, 8]
```

Achtung

Häufig verlangen C-Funktionen einen Pointer auf einen String als Parameter, über den der String dann manipuliert wird. Beachten Sie unbedingt, dass Sie dann keine Instanz des Python-Datentyps `str` übergeben dürfen. Das liegt daran, dass `str` zu den unveränderlichen Datentypen gehört und auch von einer C-Funktion nicht neu beschrieben werden kann.

Um einer solchen Funktion dennoch einen beschreibbaren String zur Verfügung zu stellen, dient die Funktion `create_string_buffer`, die wir zusammen mit den anderen Funktionen des Moduls `ctypes` gegen Ende dieses Kapitels besprechen werden.

top

26.1.3 Schnittstellenbeschreibung ▼▲

Im folgenden Beispiel sollen die Parameter der Funktion `veclen`, wie von der Funktion verlangt, als Gleitkommazahlen übergeben werden.

```
from ctypes import CDLL, c_double
bib = CDLL("bibliothek.dll")
print bib.veclen(c_double(1.5), c_double (2.7), c_double
(3.9))
```

Wird dieser Code ausgeführt, so erhält man

```
1080623743
```

als Ergebnis, was nun wirklich nicht der gesuchten Vektorlänge entspricht. Wie es zu diesem Fehler kommen konnte und wie er sich vermeiden lässt, soll Thema dieses Abschnitts sein.

Der Rückgabewert und die Parameter einer Funktion, also ihre

Schnittstelle, sind in C anders als in Python an bestimmte Datentypen gebunden. Der oben beschriebene Problemfall resultiert daher, dass nach dem Laden einer dynamischen Bibliothek von ctypes angenommen wird, dass jede enthaltene C-Funktion eine ganze Zahl zurückgibt, was natürlich in vielen Fällen falsch ist. Die eigentliche Gleitkommazahl wurde also aus Unwissenheit als ganze Zahl interpretiert und entsprechend ausgegeben. Damit dies in Zukunft nicht mehr passiert, kann über das Attribut restype eines Funktionsobjekts der Datentyp des Rückgabewertes explizit angegeben werden:

```
from ctypes import CDLL, c_double
bib = CDLL("bibliothek.dll")
bib.veclen.restype = c_double
print bib.veclen(c_double(1.5), c_double(2.7),
c_double(3.9))
```

Bei diesem Beispielprogramm wird der Rückgabewert der C-Funktion korrekt interpretiert, wie die Ausgabe zeigt:

```
4.97493718553
```

Es wurde angesprochen, dass auch die Parameter einer Funktion in C an einen bestimmten Datentyp gebunden sind. Würden Sie beispielsweise im obigen Programm statt des Datentyps c_double Instanzen des Datentyps c_float übergeben, so würde bereits bei der Parameterübergabe ein Fehler in der Interpretation der Daten passieren, der letztlich in einem falschen Rückgabewert mündet.

Python bietet es Ihnen an, über das Attribut argtypes die Datentypen der Parameter festzulegen. Wenn das gemacht wird, werden übergebene Instanzen eines falschen Datentyps in den korrekten Datentyp konvertiert, oder es wird, wenn dies nicht möglich ist, eine ArgumentError-Exception geworfen. Im folgenden Programm wird die vollständige Schnittstelle der Funktion veclen vorgegeben:

```
from ctypes import CDLL, c_double
bib = CDLL("bibliothek.dll")
bib.veclen.restype = c_double
bib.veclen.argtypes = [c_double, c_double, c_double]
print bib.veclen(c_double(1.5), c_double(2.7),
c_double(3.9))
```

Es ist zwar so, dass unter Verwendung des Moduls ctypes in vielen Fehlerfällen Exceptions geworfen werden, beispielsweise wenn zu viele, zu wenige oder die falschen Parameter übergeben werden, doch Sie sollten sich immer vergegenwärtigen, dass Sie mit ctypes viele Schutzmechanismen von Python umgehen und möglicherweise direkt im Speicher arbeiten. Es ist also durchaus möglich, unter Verwendung von ctypes den Python-Interpreter zum Absturz zu bringen. Und mit »Absturz« ist keine Exception im bisherigen Sinne gemeint, sondern ein tatsächlicher Absturz durch einen Speicherzugriffsfehler beispielsweise.

top

26.1.4 Verwendung des Moduls

An dieser Stelle möchten wir noch einen kurzen Überblick über die wichtigsten im Modul ctypes enthaltenen Funktionen bieten, die einem die Arbeit mit C–Funktionen teils erheblich erleichtern.

ctypes.addressof(obj)

Gibt die Speicheradresse der Instanz *obj* zurück. Für den Parameter *obj* muss dabei eine Instanz eines ctypes-Datentyps übergeben werden.

ctypes.byref(obj)

Erzeugt einen Pointer auf die Instanz *obj* eines ctypes-Datentyps.

Der zurückgegebene Pointer kann einer C-Funktion übergeben werden.

ctypes.cast(obj, type)

Die Funktion `cast` bildet den Cast-Operator von C in Python ab. Die Funktion gibt eine neue Instanz des `ctypes`-Datentyps *type* zurück, die auf die gleiche Speicherstelle verweist wie *obj*.

ctypes.create_string_buffer(init_or_size[, size])

Diese Funktion erzeugt einen veränderlichen String-Buffer, in den aus einer C–Funktion heraus geschrieben werden kann. Zurückgegeben wird ein Array von `c_char`-Instanzen. Für den ersten Parameter *init_or_size* kann entweder eine ganze Zahl übergeben werden, die die Länge des zu erzeugenden Buffers enthält, oder ein String, mit dem der Buffer initialisiert werden soll. Beachten Sie im Falle eines Strings, dass der Buffer ein Zeichen größer gemacht wird, als der String lang ist. In dieses letzte Zeichen wird der Terminator `\0` geschrieben.

Wenn für *init_or_size* ein String übergeben wurde, kann über den Parameter *size* die Größe des Buffers festgelegt werden, sofern nicht die Länge des Strings genommen werden soll.

ctypes.create_unicode_buffer(init_or_size[, size])

Diese Funktion verhält sich wie `create_string_buffer`, mit dem Unterschied, dass ein Array von veränderlichen `c_wchar`-Instanzen, also ein Unicode-Buffer, erzeugt und zurückgegeben wird.

ctypes.sizeof(obj_or_type)

Die Funktion `sizeof` bildet den `sizeof`-Operator von C auf Python ab. Zurückgegeben wird die Größe der übergebenen Instanz bzw. des übergebenen `ctypes`-Datentyps in Byte.

ctypes.string_at(address[, size])

Gibt den String zurück, der an der Speicheradresse *address* steht. Sollte der String im Speicher nicht null-terminiert sein, so kann über den Parameter *size* die genaue Länge des Strings übergeben werden.

Für *address* muss eine ganze Zahl übergeben werden, die sinnvollerweise mit `addressof` geholt und verändert wurde.

ctypes.wstring_at(address[, size])

Die Funktion `wstring_at` funktioniert wie `string_at`, nur für Unicode-Strings.

In diesem Abschnitt konnte Ihnen nur ein Einblick in die Funktionalität von `ctypes` gegeben werden. So enthält das Modul `ctypes` noch weitere Konzepte zur Verwendung von Pointern, Strukturen und Unions beispielsweise. Sollte Ihr Interesse am Modul `ctypes` geweckt worden sein und sollten Sie mehr darüber herausfinden möchten, sei Ihnen die ausführliche Python-Dokumentation zu diesem Thema ans Herz gelegt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Warum glauben einem Leute sofort, wenn man ihnen sagt, dass es am Himmel 400 Billionen Sterne gibt, aber wenn man ihnen sagt, dass die Bank frisch gestrichen ist, müssen sie draufpatschen?« – Unbekannter Autor

27 Insiderwissen

In diesem Kapitel sollen kleinere Module vorgestellt werden, die im Programmieralltag von Nutzen sein können. Die hier vorgestellten Module implementieren keine neuen Konzepte oder irgendetwas Weltbewegendes, sie sind einfach nur praktisch. Es lohnt sich also allemal, einen Blick auf die folgenden Abschnitte zu werfen.

top

27.1 Dateien direkt mit einem bestimmten Encoding lesen

Das Modul codecs bietet Zugriff auf Funktionen und Typen, die den Umgang mit Encodings betreffen. Insbesondere stellt codecs dabei eine Funktion open zur Verfügung, mit der das Lesen und Schreiben von nicht mit ASCII kodierten Text-Dateien zum Kinderspiel wird:

```
>>> import codecs
>>> datei = codecs.open("ist das einfach.txt", "wb",
"utf8")
>>> datei.write(u"Häääälllööööö")
>>> datei.close()
>>> datei = codecs.open("ist das einfach.txt",
encoding="utf8")
>>> hallo = datei.read()
>>> hallo
u'H\xe4\xe4\xe4\xe4lll\xf6\xf6\xf6\xf6\xf6'
>>> print hallo
Häääälllööööö
```

Wie Sie sehen, unterscheidet sich die Schnittstelle von codecs.open von der der Built-in Function open dadurch, dass Sie zusätzlich einen Parameter encoding übergeben können, der das Encoding angibt, mit dem gelesen bzw. geschrieben werden soll.

Sie müssen nur dafür sorgen, dass alle Strings, die Sie an ein mit `codecs.open` erzeugtes Dateiobjekt übergeben, `unicode`-Instanzen sind, da sonst Fehler bei der Umwandlung auftreten können. In der Regel sind alle aus einer solchen Datei gelesenen Strings auch wieder `unicode`-Instanzen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Mehr als die Vergangenheit interessiert mich die Zukunft, denn in ihr gedenke ich zu leben.« – Albert Einstein

28 Zukunft von Python

Nun, da sich dieses Buch dem Ende entgegenneigt und viele wichtige und interessante Themen behandelt worden sind, möchten wir Ihnen einen kleinen Ausblick geben, wohin sich Python in Zukunft entwickeln wird. Generell lässt sich die zukünftige Entwicklung in zwei grobe Richtungen aufteilen: *Python 2.6* und *Python 3000*. Beide Versionen möchten wir in diesem Kapitel kurz erläutern, wobei Python 3000 das langfristigere und sicherlich interessantere Ziel ist und daher zuerst behandelt werden soll.

top

28.1 Python 3000

Bislang wurde bei jeder neuen Python-Version darauf geachtet, dass alter Code nach Möglichkeit kompatibel zur neuen Version bleibt. Dies ist für die Entwickler von Python mitunter ein großes Problem, denn die Sprache Python entwickelt sich weiter, und alte Designentscheidungen haben sich als inkonsequent oder schlecht herausgestellt. Trotzdem müssen überflüssig gewordene Module oder Built-in Functions in jedem neuen Python-Release weiterhin mitgeschleppt werden, solange man die Abwärtskompatibilität wahren möchte.

Mit *Python 3000*, was der Codename für Python 3.0 ist, möchten die Entwickler mit der Abwärtskompatibilität brechen und die Sprache insgesamt konsequenter und logischer machen. Dazu werden einige teils gravierende Änderungen an der Sprache selbst durchgeführt. Außerdem wird die Standardbibliothek überarbeitet und werden viele, bereits als *deprecated* eingestufte Module entfernt.

Wir möchten hier die Änderungen, die mit Python 3000 in die Sprache einfließen, nicht im Detail diskutieren, sondern nur einen groben Überblick darüber geben, wohin sich Python entwickeln

wird. Aus diesem Grund haben wir im Folgenden eine kurze Liste über die signifikantesten Änderungen von Python 3000 zusammengestellt:

- Das Schlüsselwort `print` wird durch eine Built-in Function `print` ersetzt.
- Der Divisionsoperator `/` wird bei numerischen Operanden immer eine Gleitkommadivision durchführen. Für eine Integer-Division muss der Operator `//` verwendet werden.
- In Anlehnung an List Comprehensions wird Python 3000 *Dictionary Comprehensions* und *Set Comprehensions* mit einem ähnlichen Zweck für die Datentypen `dict` und `set` unterstützen.
- Python 3000 wird mit *Function Annotations* eine flexible und optionale Syntax einführen, um beispielsweise den Datentyp von übergebenen Funktionsparametern zu überprüfen.

  Es werden standardmäßig Unicode-Strings verwendet. Zusätzlich wird es einen Datentyp `bytes` geben, der den »normalen« Byte-String speichert.

Um den Umstieg von Python 2.x auf Python 3000 bzw. Python 3.0 zu erleichtern, wird Python 3000 ein Programm namens `2to3` enthalten, das alten Python-Code automatisch an Python 3000 anpasst. Da dieses Programm lediglich eine syntaktische Konvertierung vornimmt und die Semantik Ihrer Programme nicht erfassen kann, bleibt Ihnen ein gewisses Maß an Arbeit beim Umstieg von Python 2.x auf Python 3000 wohl nicht erspart.

Im Folgenden werden einige Tipps aufgeführt, mit denen Sie den Aufwand beim Umstieg minimieren können:

- Verwenden Sie Python 2.6 und insbesondere den Python–3000-Kompatibilitätsmodus von Python 2.6. Näheres dazu erfahren Sie im nächsten Abschnitt.
- Leiten Sie alle eigenen Exception-Typen von der Basisklasse `Exception` ab.
- Leiten Sie alle eigenen Klassen von der Basisklasse `object` ab.
- Verwenden Sie den Operator `//` für Integer-Divisionen.
- Erstellen Sie Unit- oder Doctests mit größtmöglicher Abdeckung für Ihr Projekt. Auf diese Weise können Sie leichter Fehler entdecken, die aufgrund der Umstellung entstanden sind. Näheres zu Unit- und Doctests finden Sie in Abschnitt 21.5, »Automatisiertes Testen«.

  Nachdem wir uns mit der Zukunftsvision Python 3000 beschäftigt haben, wenden wir uns Python 2.6 zu.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

»Alles, was einen Anfang hat, hat auch ein Ende – und meistens hat das, was ein Ende hat, auch eine Fortsetzung« (Sprichwort)

A Anhang

top

A.1 Entwicklungsumgebungen ▼

Wie Sie wissen, ist bei Python bereits die rudimentäre Entwicklungsumgebung IDLE enthalten, die die grundlegenden Eigenschaften einer guten IDE hat. Dazu gehören unter anderem *Syntax Highlighting*, *Code Completion* und ein *integrierter Debugger*. Allerdings fügt sich IDLE aufgrund des verwendeten Tk-Toolkits nicht in das Look & Feel des Betriebssystems ein und hat eine für viele Anwender nicht intuitive Oberfläche. Aus diesem Grund möchten wir an dieser Stelle einen Überblick geben, welche IDEs für Python existieren, und eine kurze Zusammenfassung über deren Eigenschaften geben.

Beachten Sie, dass wir hier aus naheliegenden Gründen nur die bekanntesten IDEs besprechen werden. Es existiert eine große Anzahl weiterer IDEs bzw. Texteditoren für Python. Eine umfangreiche Liste finden Sie im offiziellen Python-Wiki auf *http://wiki.python.org/moin* unter dem Stichwort »Python Editors«.

Alle hier aufgeführten Entwicklungsumgebungen werden auf der angegebenen Website als Voll- oder Testversion zum Download angeboten.

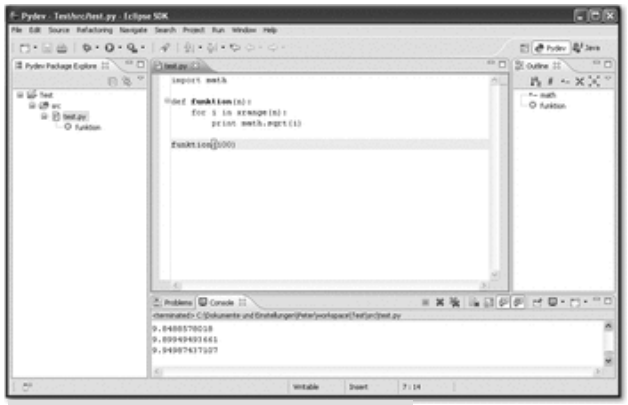
top

A.1.1 Eclipse ▼▲

Website: http://www.eclipse.org

Python-Plugin PyDev: http://pydev.sourceforge.net

Plattformen: Windows, Linux, Mac OS X



Hier klicken, um das Bild zu vergrößern

Abbildung A.1 Eclipse im Einsatz

Die von IBM entwickelte IDE *Eclipse* war ursprünglich für Java gedacht, kann aber inzwischen mithilfe von Plugins für viele Programmiersprachen verwendet werden. Eclipse wurde 2001 von IBM unter einer Open-Source-Lizenz freigegeben. Seit 2004 ist die *Eclipse Foundation* für die Weiterentwicklung der IDE verantwortlich. Eclipse erfreut sich großer Beliebtheit und kann mithilfe des *PyDev*-Plugins auch für die Entwicklung mit Python eingesetzt werden.

Das PyDev-Plugin verfügt über Quellcodevervollständigung, Syntaxüberprüfung und viele weitere Annehmlichkeiten. Zudem beinhaltet PyDev einen umfangreichen integrierten Python-Debugger, der auch mit Threads sinnvoll umgehen kann.

Beachten Sie, dass Eclipse ein installiertes Java SDK benötigt. Näheres dazu finden Sie auf der Eclipse-Website.

top

A.1.2 Eric4 ▼▲

Website: http://www.die-offenbachs.de/eric/

Plattformen: Windows, Linux, Mac OS X

Hier klicken, um das Bild zu vergrößern

Abbildung A.2 Eric4 im Einsatz

Die freie Entwicklungsumgebung Eric4 ist selbst mittels PyQt in Python geschrieben und spezialisiert sich auf PyQt4-Anwendungen. [Analog dazu existiert *Eric3* für PyQt3-Anwendungen.] Für die Entwicklung von Python-Programmen mit Qt-Oberfläche bietet die IDE viele interessante Hilfsmittel an.

top

A.1.3 Komodo IDE ▼▲

Website: http://www.activestate.com

Plattformen: Windows, Linux, Mac OS X

Die kommerzielle Entwicklungsumgebung *Komodo IDE* wird von der kanadischen Firma *ActiveState* entwickelt und unterstützt neben Python noch viele andere Sprachen. Eine abgespeckte Version der Komodo IDE namens *Komodo Edit* kann von der Homepage der Firma kostenlos heruntergeladen werden (siehe Abbildung A.3).

Hier klicken, um das Bild zu vergrößern

Abbildung A.3 Komodo IDE im Einsatz

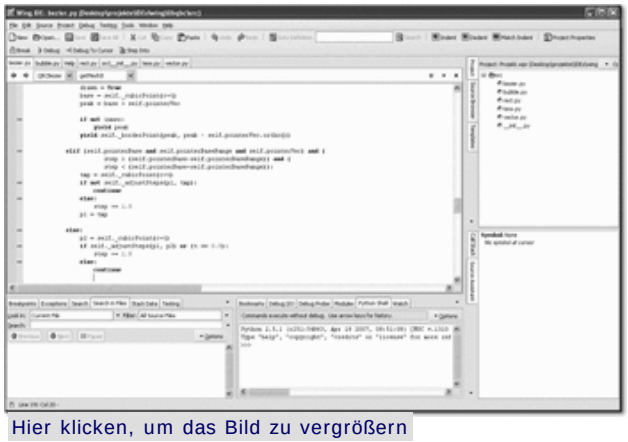
Die Entwicklungsumgebung unterstützt eine *Codevervollständigung*, *Calltips*, *Syntaxüberprüfung* sowie einen komfortablen und umfassenden *grafischen Debugger*, der auch eine *Post-Mortem-Debugging*-Funktionalität unterstützt.

top

A.1.4 Wing IDE

Website: http://www.wingware.com

Plattformen: Windows, Linux, Mac OS X



Hier klicken, um das Bild zu vergrößern

Abbildung A.4 Wing IDE im Einsatz

Die kommerzielle Entwicklungsumgebung *Wing IDE* wird von der amerikanischen Firma *Wingware* speziell für Python entwickelt und bietet komfortable Funktionen, die beim Programmieren helfen – darunter zum Beispiel einen Klassen- oder Modulbrowser und die bereits von anderen IDEs bekannte *Autovervollständigung*. Zudem bietet die IDE einen *grafischen Debugger*, mit dem auch Multithread-Anwendungen auf Fehler untersucht werden können.

Grundsätzlich ist die Wing IDE in drei Versionen verfügbar, die sich in ihren Features unterscheiden. Eine stark abgespeckte

Version können Sie kostenlos herunterladen und frei verwenden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _

Stichwortverzeichnis

top

C ▼▲

top

D ▼▲

top

E ▼▲

top

F ▼▲

Funktionsschnittstelle

top

G ▼▲

Gammaverteilung

Ganze Zahlen

Ganze Zahlen

Gaußverteilung

Gaußverteilung

Generatoren

Generator Expression

Geplantes Sprachelement

Geschwisterelement

GET

Getter-Methode

Gleichverteilung

Gleitkommazahlen

Gleitkommazahlen

global

Globale Referenz

Globaler Namensraum

Globale Variable

GNU gettext API

Grafische Benutzeroberfläche → GUI

Gruppe

Gtk

GUI

Guido van Rossum

top

H ▼▲

Hash-Funktion

Hash-Wert

Hash-Wert

Hauptdialog

Hexadezimalsystem

Hook

HTTP

top

I ▼▲

I18N

Identität

IDLE

IDLE

IDLE

if

if

Imaginärteil

top

J ▼▲

top

K ▼▲

top

L ▼▲

top

M ▼▲

top

N ▼▲

top

O ▼▲

top

P ▼▲

Q ▼▲

top

R ▼▲

top

S ▼▲

T ▼▲

U ▼▲

V ▼▲

W ▼▲

X ▼▲

Y ▼▲

Z ▼▲

_ ▲

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

1.2 Was leistet dieses Buch und was nicht?

Das Ziel dieses Buchs ist es, dem Leser fundierte Python-Kenntnisse zu vermitteln, damit er auch professionellen Aufgaben gewachsen ist. Dazu wird die Sprache Python umfassend eingeführt. Die Einführung erfolgt systematisch vom ersten einfachen Programm bis hin zu komplexen objektorientierten Programmen. Das Buch stellt den praxisbezogenen Umgang mit Python in den Vordergrund. Es ist nicht das Ziel dieses Buchs, Ihnen fundierte theoretische Kenntnisse über Disziplinen der Informatik zu vermitteln.

Abgesehen von der Einführung in die Sprache selbst, werden große Teile von Pythons Standardbibliothek besprochen. Bei der Standardbibliothek handelt es sich um eine Sammlung von Hilfsmitteln, die das Arbeiten mit Python erleichtern und eine der größten Stärken von Python darstellen. Abhängig von der Bedeutung und Komplexität des jeweiligen Themas werden konkrete Beispielprogramme zur Demonstration erstellt, was zum einen im Umgang mit der Sprache Python schult und zum anderen als Grundlage für eigene Projekte dienen kann. Der Quelltext der Beispielprogramme ist sofort ausführbar und befindet sich auf der CD, die diesem Buch beiliegt. Bei wichtigen Themen wird zusätzlich eine Referenz geboten, die das Buch auch als Nachschlagewerk nutzbar macht.

Dieses Buch ist keinesfalls als Einführung in die Programmierung allgemein oder gar in die Informatik anzusehen. Wir behandeln weder Datenstrukturen noch Algorithmen noch die dahinter stehende Theorie. Der Hauptfokus liegt auf der praktischen Arbeit mit Python, weshalb wir uns auf die Lösung von Problemen mithilfe der Sprache konzentrieren.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer

über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

1.3 Wie ist dieses Buch aufgebaut?

Dieses Buch ist in vier Teile gegliedert, deren Inhalt im Folgenden kurz zusammengefasst wird. Sollten Sie mit den Begriffen im Moment noch nichts anfangen können, seien Sie unbesorgt. An dieser Stelle dienen alle genannten Begriffe zur Orientierung und werden im jeweiligen Kapitel des Buchs ausführlich erklärt.

1. Der erste Teil bietet einen Einstieg in die Arbeit mit Python. Dabei legen wir sehr viel Wert darauf, dass der Leser schon früh seine ersten eigenen Programme entwickeln und testen kann, denn wie bei der Programmierung allgemein gilt auch in Python, dass learning by doing die erfolgversprechendste Lernmethode ist. Die Einführung in die Grundelemente von Python haben wir so aufgebaut, dass größtenteils auf das Begriffsgebäude der Objektorientierung verzichtet wurde, um Umsteigern von nicht objektorientierten Sprachen den Einstieg zu erleichtern. Neben der Sprache selbst werden die eingebauten Datentypen und ihre Verwendung behandelt.
2. Im zweiten Teil stehen dann die Konzepte im Vordergrund, die die Arbeit mit Python erst so richtig angenehm machen, allerdings für den unerfahrenen Leser auch völliges Neuland darstellen können. Als große Oberthemen sind dabei Modularisierung und Objektorientierung zu nennen, die in Python eine zentrale Rolle spielen. Außerdem werden moderne Programmiertechniken wie Exception-Handling, Iteratoren und Generatoren behandelt.
3. Der dritte Teil konzentriert sich auf Pythons Batteries-included-Philosophie, wonach Python nach Möglichkeit alles in der Standardbibliothek mitbringen sollte, was für die Entwicklung eigener Anwendungen erforderlich ist. Wir werden in diesem Teil auf viele der mitgelieferten Module eingehen und auch das ein oder andere Drittanbietermodul erklären. Insbesondere ist auch die Suche nach Fehlern in Python-Programmen und deren Behebung Thema dieses Teils.

   Der dritte Teil ist eher als Nachschlagewerk zu konkreten

Problemen gedacht und sollte nicht einfach in einem Rutsch von vorn bis hinten gelesen werden.

4. Im letzten Teil werden wir weiterführende Themen wie die Weitergabe von fertigen Python-Programmen und -Modulen an Endanwender bzw. andere Entwickler behandeln. Neben der Programmoptimierung und der Auslagerung laufzeitkritischer Programmteile in effizientere Sprachen wie C werden auch die Entwicklung von grafischen Benutzeroberflächen mit PyQt und die Erstellung von Webanwendungen mit dem populären Framework Django besprochen. Außerdem werden kleine Kniffe gezeigt, die das Arbeiten mit Python noch effektiver machen können. Am Ende des Buchs geben wir einen kurzen Ausblick auf die kommenden Python-Versionen 2.6 und 3000 und weisen Sie auf mögliche Inkompatibilitäten hin – und darauf, wie man diese umgeht.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

1.4 Wer sollte dieses Buch wie lesen?

Dieses Buch richtet sich im Wesentlichen an zwei Typen von Lesern: diejenigen, die in die Programmierung mit Python einsteigen möchten und idealerweise bereits grundlegende Kenntnisse der Programmierung besitzen und diejenigen, die mit der Sprache Python bereits mehr oder weniger vertraut sind und ihr Wissen vertiefen möchten. Für beide Typen ist dieses Buch bestens geeignet, da sowohl eine vollständige Einführung in die Programmiersprache als auch eine umfassende Referenz zur Anwendung von Python in vielen Bereichen geboten wird.

Im Folgenden möchten wir eine Empfehlung an Sie richten, wie Sie dieses Buch, abhängig von Ihrem Kenntnisstand, lesen sollten.

Sollten Sie bereits einige grundlegende Erfahrungen in einer Programmiersprache, beispielsweise PHP oder Java, gesammelt haben, so bringen Sie im Prinzip bereits alle Voraussetzungen zum Lesen dieses Buchs mit, da der erste Teil des Buchs einen umfassenden Einstieg in die Sprache Python beinhaltet und wichtige Begriffe an Ort und Stelle erklärt werden. Dennoch sollten Sie sich darauf gefasst machen, dass der Anspruch in den folgenden drei Teilen des Buchs rasch zunimmt, denn unser Buch soll Sie in die Lage versetzen, Python professionell einsetzen zu können. Ein Leser, der das Buch zum Einstieg in die Programmiersprache Python verwenden möchte, sollte sich auf die beiden ersten Teile konzentrieren und diese vollständig durcharbeiten.

Beachten Sie, dass dieses Buch nicht für den generellen Einstieg in die Programmierung gedacht ist.

Wenn Sie selbst als »alter Hase« von C oder einer anderen Programmiersprache wechseln und eine moderne Sprache kennenlernen möchten, haben Sie mit diesem Buch die richtige Wahl getroffen. Sie können die ersten beiden Teile einfach sequenziell lesen, um Ihre Kenntnisse im Anschluss daran in Teil 4 zu vertiefen. Die Besprechung zentraler Module ist in Teil 3 angesiedelt, der als Nachschlagewerk dient.

Als letzte Zielgruppe kommen erfahrene Python-Programmierer in Betracht. Sollte der Umgang mit Python für Sie zum alltäglichen Geschäft gehören, können Sie im ersten und zweiten Teil Ihr Wissen vertiefen und festigen oder beide Teile einfach querlesen. Für Sie werden die letzten beiden Teile interessanter sein, die Ihnen als hilfreiches Nachschlagewerk dienen und weiterführende Informationen zu speziellen Themen wie zur Entwicklung grafischer Benutzeroberflächen anbieten können. Außerdem bietet dieses Buch einige interessante Praxistipps, mit denen Sie Ihre Ziele schneller als bisher erreichen können.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

1.5 Danksagung

Nachdem wir Ihnen das Buch vorgestellt und hoffentlich schmackhaft gemacht haben, möchten wir uns noch bei denjenigen bedanken, die uns bei der Ausarbeitung des Manuskripts begleitet, unterstützt und uns immer wieder zum Schreiben angetrieben haben.

Besonderer Dank gilt Prof. Dr. Ulrich Kaiser, der mit seiner konstruktiven Kritik und unzähligen Stunden des Korrekturlesens die Qualität des Buchs deutlich verbessert hat. Außerdem ist es seiner Initiative zu verdanken, dass wir überhaupt dazu gekommen sind, ein Buch zu schreiben. Wir sind sehr glücklich, dass wir von seiner Sachkenntnis und Erfahrung profitieren konnten.

Neben der fachlichen Korrektheit trägt auch die verwendete Sprache maßgeblich zur Qualität des Buchs bei. Dass sich dieses Buch so gut liest, wie es sich liest, haben wir Angelika Kaiser zu verdanken, die auch noch so kompliziert verschachtelte Satzgefüge in klare, gut verständliche Formulierungen verwandeln konnte.

Außerdem möchten wir Herbert Ernesti dafür danken, dass er das fertige Werk noch einmal als Ganzes unter die Lupe genommen hat und viele nützliche Verbesserungsvorschläge machen konnte.

Die Anfängerfreundlichkeit der Erklärungen wurde von Anne Kaiser experimentell erprobt und für gut befunden – vielen Dank dafür.

Zum Schluss danken wir noch allen Mitarbeitern von Galileo Press, die an der Erstellung dieses Buchs beteiligt waren. Namentlich hervorheben möchten wir dabei unsere Lektorin Judith Stevens-Lemoine, die uns geholfen hat, sich durch den Autorendschungel zu schlagen und uns dabei alle Freiheiten für eigene Ideen gelassen hat.

Johannes Ernesti, je@revelation-soft.de **Peter Kaiser**,

p@penguin-p.de

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

2.2 Grundlegende Konzepte

Grundsätzlich handelt es sich bei Python um eine imperative Programmiersprache, die jedoch noch weitere *Programmierparadigmen* in sich vereint. So ist es beispielsweise möglich, mit Python objektorientiert und funktional zu programmieren. Sollten Sie mit diesen Begriffen im Moment noch nichts anfangen können, seien Sie unbesorgt, schließlich soll Ihnen die Programmierung mit Python und damit die Anwendung der verschiedenen Paradigmen in diesem Buch beigebracht werden.

Obwohl Python viele Sprachelemente gängiger Scriptsprachen implementiert, handelt es sich um eine interpretierte Programmiersprache. Der Unterschied zwischen einer Programmier- und einer Scriptsprache liegt im sogenannten *Compiler*. Ähnlich wie Java oder C# verfügt Python über einen Compiler, der aus dem Quelltext ein Kompilat, den sogenannten *Byte-Code* erzeugt. Dieser Byte-Code wird dann in einer virtuellen Maschine, dem *Python-Interpreter*, ausgeführt.

Ein weiteres Konzept, das Python zum Beispiel mit Java gemeinsam hat, ist die *Plattformunabhängigkeit*. Der Python-Interpreter läuft unter verschiedensten Betriebssystemen und ermöglicht, dass ein und dasselbe Python-Programm unmodifiziert unter verschiedenen Betriebssystemen lauffähig ist. Insbesondere werden die drei großen Desktop-Betriebssysteme Windows, Linux und Mac OS X unterstützt.

Im Lieferumfang von Python ist neben dem Interpreter und dem Compiler eine umfangreiche *Standardbibliothek* enthalten. Diese Standardbibliothek ermöglicht es dem Programmierer, in kurzer Zeit relativ einfach strukturierte Programme zu schreiben, die allerdings sehr komplexe Aufgaben verrichten können. So bietet die Standardbibliothek beispielsweise umfassende Möglichkeiten zur Netzwerkkommunikation oder der Datenspeicherung. Da die Standardbibliothek die Programmiermöglichkeiten in Python wesentlich bereichert, widmen wir ihr im dritten und teilweise auch vierten Teil dieses Buchs besondere Aufmerksamkeit.

Ein Nachteil der Programmiersprache ABC, den Guido van Rossum bei der Entwicklung von Python beheben wollte, war ihre fehlende Flexibilität. Ein grundlegendes Konzept von Python ist es daher, es dem Programmierer so einfach wie möglich zu machen, die Standardbibliothek beliebig zu erweitern. Da Python selbst, als abstrakte Programmiersprache, nur eingeschränkte Möglichkeiten zur maschinennahen Programmierung bietet, können maschinennahe oder zeitkritische Erweiterungen problemlos in C geschrieben werden. Das ermöglicht die sogenannte *Python API*.

Als letztes grundlegendes Konzept von Python soll erwähnt werden, dass Python unter der *PSF-Lizenz* steht. Das ist eine von der Python Software Foundation entworfene Lizenz für Open-Source-Software, die wesentlich weniger restriktiv ist als beispielsweise die GNU General Public License. So erlaubt es die PSF-Lizenz, den Python-Interpreter lizenzkostenfrei in größere, kommerzielle Anwendungen einzubetten und mit diesen auszuliefern. Diese Politik macht Python auch für kommerzielle Anwendungen attraktiv.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

2.3 Einsatzmöglichkeiten und Stärken

Die größte Stärke von Python ist *Flexibilität*. So kann Python beispielsweise als Programmiersprache für kleine und große Applikationen, als serverseitige Programmiersprache im Internet oder als Scriptsprache für eine größere C- oder C++-Anwendung verwendet werden. Auch abseits des klassischen Markts breitet sich Python beispielsweise im Embedded-Bereich aus. So existieren bereits Python-Interpreter für diverse Mobiltelefone oder PDAs.

Python ist aufgrund seiner *einfachen Syntax* sehr leicht zu erlernen und gut zu lesen. Außerdem erlauben es die automatische Speicherverwaltung und die umfangreiche Standardbibliothek, mit relativ kleinen Programmen bereits sehr komplexe Probleme anzugehen. Aus diesem Grund eignet sich Python zum sogenannten *Rapid Prototyping*. Bei dieser Art der Entwicklung geht es darum, in möglichst kurzer Zeit einen lauffähigen Prototyp als eine Art Machbarkeitsstudie einer größeren Software zu erstellen, die dann später in einer anderen Programmiersprache implementiert werden soll. Mithilfe eines solchen Prototyps lassen sich Probleme und Designfehler bereits entdecken, bevor die tatsächliche Entwicklung der Software begonnen wird.

Eine weitere Stärke Pythons ist die bereits im vorherigen Abschnitt angesprochene *Erweiterbarkeit*. Aufgrund dieser Erweiterbarkeit können Python-Entwickler aus einem reichen Fundus von Drittanbieterbibliotheken und Anbindungen an viele bekannte Bibliotheken schöpfen. So existieren beispielsweise Anbindungen an die gängigsten GUI-Toolkits, die somit das Erstellen eines Python-Programms mit grafischer Benutzeroberfläche ermöglichen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

2.4 Aktuelle Einsatzgebiete

Python erfreut sich immer größerer Bekanntheit und Verbreitung. Viele große Firmen setzen bereits erfolgreich die freie Programmiersprache ein. Die wohl bekannteste dieser Firmen ist *Google*, bei der der Python-Erfinder Guido van Rossum arbeitet. Neben Google setzen beispielsweise auch die amerikanische Spezialeffekte-Schmiede *Industrial Light & Magic* und sogar die *NASA* Python ein. Auch die bekannte Website *YouTube* ist fast vollständig in Python geschrieben.

Ein weiteres interessantes Einsatzgebiet ist der von der gemeinnützigen Organisation *One Laptop per Child* entwickelte 100-Dollar-Laptop. Dabei wurde die Benutzeroberfläche des Laptops in Python geschrieben.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

3.2 Tippen, kompilieren, testen

In diesem Abschnitt sollen die Arbeitsabläufe besprochen werden, die nötig sind, um ein Python-Programm zu erstellen und auszuführen. Ganz allgemein sollten Sie sich darauf einstellen, dass wir in einem Großteil des Buchs ausschließlich sogenannte *Konsolenanwendungen* in Python schreiben werden. Eine Konsolenanwendung hat eine rein textbasierte Schnittstelle zum Benutzer und läuft in der Konsole des jeweiligen Betriebssystems ab.

Grundsätzlich besteht ein Python-Programm aus einer oder mehreren Programmdateien. Diese Programmdateien haben die Dateiendung *.py* und enthalten den Python-Quelltext. Dabei handelt es sich im Prinzip um nichts anderes als um Textdateien. Programmdateien können also mit einem normalen Texteditor bearbeitet werden.

Nachdem eine Programmdateien geschrieben worden ist, besteht der nächste logische Schritt darin, sie auszuführen. Wenn Sie IDLE verwenden, kann die Programmdatei bequem über den Menüpunkt Run • Run Module ausgeführt werden. Sollten Sie einen Editor verwenden, der keine vergleichbare Funktion unterstützt, müssen Sie in einer Kommandozeile in das Verzeichnis der Programmdatei wechseln und, abhängig von Ihrem Betriebssystem, verschiedene Kommandos ausführen.

Unter Windows reicht es, den Namen der Programmdatei einzugeben und mit ↵ zu bestätigen. Im folgenden Beispiel soll die Programmdatei *programm.py* im Ordner *C:\Ordner* ausgeführt werden. Dazu müssen Sie ein Konsolenfenster unter Start • Programme • Zubehör • Eingabeaufforderung starten.

Hier klicken, um das Bild zu vergrößern

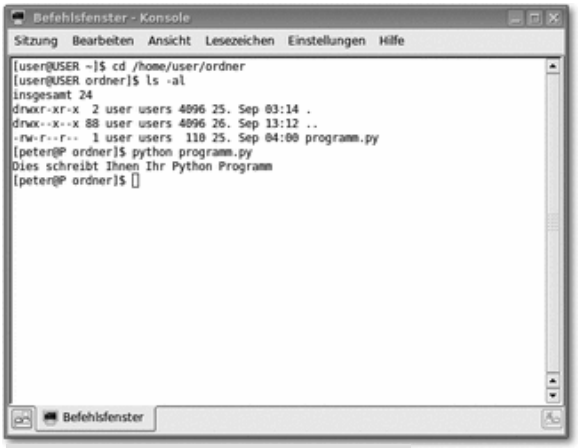
Abbildung 3.3 Ausführen eines Python-Programms unter Windows

Bei »Dies schreibt Ihnen Ihr Python Programm« handelt es sich um eine Ausgabe des Python-Programms in der Datei *programm.py*, die beweist, dass das Python-Programm tatsächlich ausgeführt wurde.

Hinweis

Unter Windows ist es auch möglich, ein Python-Programm durch einen Doppelklick auf die jeweilige Programmdatei auszuführen. Das hat aber gegenüber der soeben besprochenen Methode den Nachteil, dass sich das Konsolenfenster sofort nach Beenden des Programms schließt und die Ausgaben des Programms somit nicht erkennbar sind.

Unter Unix-ähnlichen Betriebssystemen wie Linux oder Mac OS X müssen Sie ebenfalls in das Verzeichnis wechseln, in dem die Programmdatei liegt, und dann den Python-Interpreter mit dem Kommando python, gefolgt von dem Namen der auszuführenden Programmdatei, starten. Im folgenden Beispiel soll die Programmdatei *programm.py* unter Linux ausgeführt werden, die sich im Verzeichnis */home/user/ordner* befindet.



Hier klicken, um das Bild zu vergrößern

Abbildung 3.4 Ausführen eines Python-Programms unter Linux

top

3.2.1 Shebang ▼▲

Unter einem Unix-ähnlichen Betriebssystem wie beispielsweise Linux können Python-Programmdateien mithilfe eines sogenannten *Shebangs*, auch *Magic Line* genannt, direkt ausführbar gemacht werden. Dazu muss die erste Zeile der Programmdatei in der Regel folgendermaßen lauten:

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

In diesem Fall wird das Betriebssystem dazu angehalten, diese Programmdatei immer mit dem Python-Interpreter auszuführen. Unter anderen Betriebssystemen, beispielsweise Windows, wird die Shebang-Zeile ignoriert.

Beachten Sie, dass der Python-Interpreter auf Ihrem System in einem anderen Verzeichnis als dem hier angegebenen installiert sein könnte. Allgemein ist daher folgende Shebang-Zeile besser, da sie vom tatsächlichen Installationsort Pythons unabhängig ist:

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

Beachten Sie, dass das Executable-Flag der Programmdatei gesetzt werden muss, bevor die Datei tatsächlich ausführbar ist. Das geschieht mit dem Befehl

```
chmod +x dateiname
```

Die in diesem Buch gezeigten Beispiele enthalten aus Gründen der Übersicht keine Shebang-Zeile. Das bedeutet aber ausdrücklich nicht, dass vom Einsatz einer Shebang-Zeile grundsätzlich abzuraten wäre.

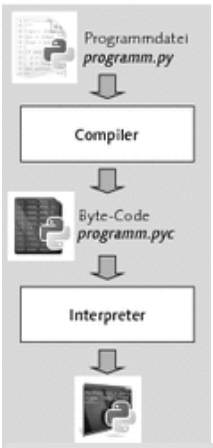
top

3.2.2 Interne Abläufe

Bislang haben Sie einen ungefähren Begriff davon, was Python ausmacht und wo die Stärken der Programmiersprache liegen. Außerdem wurde das theoretische Grundwissen zum Erstellen und Ausführen einer Python-Programmdatei vermittelt. Doch in den vorherigen Abschnitten sind Begriffe wie Compiler oder Interpreter gefallen, ohne tatsächlich erklärt worden zu sein. In diesem Abschnitt möchten wir uns daher den internen Vorgängen widmen, die beim Ausführen einer Python-Programmdatei ablaufen. Die folgende Grafik soll veranschaulichen, was beim Ausführen einer Programmdatei namens *programm.py* geschieht.

Wenn die Programmdatei *programm.py* wie zu Beginn des Kapitels beschrieben ausgeführt wird, passiert sie zunächst den sogenannten *Compiler*. Ein Compiler ist ein allgemeiner Begriff der Informatik und bezeichnet ein Programm, das von einer formalen Sprache in eine andere übersetzt. In Falle von Python übersetzt der Compiler von der Sprache Python in den sogenannten *Byte-Code*. Dabei steht es dem Compiler frei, den generierten Byte-Code im Arbeitsspeicher zu behalten oder als *programm.pyc* auf der Festplatte zu speichern.

Beachten Sie, dass das vom Compiler generierte Kompilat, im Gegensatz zu beispielsweise C- oder C++-Kompilaten, nicht direkt auf dem Prozessor ausgeführt werden kann. Zur Ausführung des Byte-Codes wird eine weitere Abstraktionsschicht, der sogenannte *Interpreter,* benötigt. Der Interpreter, häufig auch *virtuelle Maschine* (engl. *virtual machine*) genannt, liest den vom Compiler erzeugten Byte-Code ein und führt ihn aus.



Hier klicken, um das Bild zu vergrößern

Abbildung 3.5 Kompilieren und Interpretieren einer Programmdatei

Dieses Prinzip einer interpretierten Programmiersprache hat verschiedene Vorteile. So kann derselbe Python-Code beispielsweise unmodifiziert auf allen Plattformen ausgeführt werden, für die ein Python-Interpreter existiert. Allerdings laufen Programme interpretierter Programmiersprachen aufgrund des zwischengeschalteten Interpreters auch immer langsamer als ein vergleichbares C-Programm, das direkt auf dem Prozessor ausgeführt wird.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

4.2 Gleitkommazahlen

Das Literal für eine Gleitkommazahl besteht aus einem Vorkommaanteil, einem Dezimalpunkt und einem Nachkommaanteil. Wie schon bei den ganzen Zahlen ist es möglich, ein Vorzeichen anzugeben:

```
>>> 0.5
0.5
>>> -123.456
-123.456
>>> +1.337
1.337
```

Beachten Sie, dass es sich bei dem Dezimaltrennzeichen um einen Punkt handeln muss. Die in Deutschland übliche Schreibweise mit einem Komma ist nicht zulässig. Gleitkommazahlen lassen sich ebenso intuitiv in Termen verwenden wie die ganzen Zahlen:

```
>>> 1.0 / 4.0
0.25
```

Die Integer-Division kann umgangen werden, indem mindestens ein Quotient als Gleitkommazahl geschrieben wird. Das könnte folgendermaßen aussehen:

```
>>> 6 / 4.0
1.5
```

Statt durch die ganze Zahl 4 wird hier durch die Gleitkommazahl 4.0 dividiert. Das Ergebnis einer solchen Operation ist dann eine Gleitkommazahl.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Einstieg in SQL

IT-Handbuch für Fachinformatiker

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

4.3 Zeichenketten

Neben den Zahlen sind Zeichenketten, auch *Strings* genannt, von entscheidender Bedeutung. Strings ermöglichen es, Text vom Benutzer einzulesen, zu speichern, zu bearbeiten oder auszugeben.

Um einen konstanten String zu erzeugen, wird der zugehörige Text in doppelte Hochkommata geschrieben:

```
>>> "Hallo Welt"
'Hallo Welt'
>>> "abc123"
'abc123'
```

Dass der Interpreter den Wert des Strings in einfachen Hochkommata ausgibt, sollte Sie im Moment nicht weiter stören, wir werden zu gegebener Zeit darauf zurückkommen.

Ähnlich wie bei Ganz- und Gleitkommazahlen gibt es auch Operatoren für Strings. So fügt der Operator + beispielsweise zwei Strings zusammen:

```
>>> "Hallo" + " " + "Welt"
'Hallo Welt'
```

Abgesehen davon kann ein String unter Verwendung des Operators * mit einer ganzen Zahl multipliziert werden:

```
>>> "Hallo" * 3
'HalloHalloHallo'
>>> 3 * "Hallo"
'HalloHalloHallo'
```

Die Operatoren - und /, die wir für die Ganz- und Gleitkommazahlen eingeführt haben, sind für Strings nicht definiert.

Achtung!

Beachten Sie, dass die Verwendung von Umlauten und anderen Sonderzeichen unter Umständen zu Problemen führen kann. Im interaktiven Modus funktioniert dies zwar, in einem Python-Programm sind dafür jedoch zusätzliche Vorkehrungen zu treffen. Wir empfehlen Ihnen daher, vorerst auf Sonderzeichen zu verzichten, bis wir Sie in Abschnitt 8.5.3 in die Thematik eingeführt haben.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

top

4.4 Variablen

Es ist in Python möglich, einer Zahl oder Zeichenkette einen Namen zu geben. Dazu werden der Name auf der linken und das entsprechende Literal auf der rechten Seite eines Gleichheitszeichens geschrieben. Eine solche Operation wird *Zuweisung* genannt.

```
>>> name = 0.5
>>> var123 = 12
>>> string = "Hallo Welt!"
```

Die mit den Namen verknüpften Werte können später ausgegeben oder in Berechnungen verwendet werden, indem der Name anstelle des jeweiligen Werts eingegeben wird:

```
>>> name
0.5
>>> 2 * name
1.0
>>> (var123 + var123) / 3
8
>>> var123 + name
12.5
```

Es ist genauso möglich, dem Ergebnis einer Berechnung einen Namen zu geben:

```
>>> a = 1 + 2
>>> b = var123 / 4
```

Dabei wird immer zuerst die Seite rechts vom Gleichheitszeichen ausgewertet. So wird beispielsweise bei der Anweisung `a = 1 + 2` stets zuerst das Ergebnis von `1 + 2` bestimmt, bevor dem entstandenen Wert ein Name zugewiesen wird.

Ein Variablenname, auch *Bezeichner* genannt, darf aus allen Buchstaben des englischen Alphabets und dem Unterstrich (»_«) zusammengesetzt werden. Nach mindestens einem führenden

Buchstaben oder dem Unterstrich dürfen auch Ziffern verwendet werden. Bestimmte sogenannte *Schlüsselwörter* sind in Python für die Sprache selbst reserviert und dürfen nicht als Bezeichner verwendet werden. Eine Übersicht über alle reservierten Wörter finden Sie im Anhang.

Zum Schluss möchten wir noch einen weiteren Begriff einführen. Alles, was mit numerischen Literalen, also Ganz- oder Gleitkommazahlen, Variablen und den bisher vorgestellten Operatoren formuliert werden kann, wird als *arithmetischer Ausdruck* bezeichnet. Ein solcher Ausdruck könnte also so aussehen:

```
(a * a + b) / 12
```

Alle bisher eingeführten Operatoren +, -, * und / werden folgerichtig als *arithmetische Operatoren* bezeichnet.

Beachten Sie bei der Verwendung von Variablen, dass Python *case sensitive* ist. Dies bedeutet, dass bei Bezeichnern zwischen Groß- und Kleinschreibung unterschieden wird. In der Praxis heißt das, dass die Bezeichner `otto` und `Otto` nicht identisch sind, sondern durchaus zwei verschiedene Werte haben können.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

4.6 Bildschirmausgaben

Auch wenn wir hin und wieder auf den interaktiven Modus zurückgreifen werden, ist es natürlich unser Ziel, möglichst schnell echte Python-Programme zu schreiben. Es ist eine Besonderheit des interaktiven Modus, dass der Wert eines eingegebenen Ausdrucks automatisch ausgegeben wird. In einem normalen Programm müssen Bildschirmausgaben dagegen vom Programmierer erzeugt werden. Um den Wert einer Variablen auszugeben, wird in Python der Befehl `print` verwendet:

```
>>> print 1.2
1.2
```

Beachten Sie, dass mittels `print`, im Gegensatz zur automatischen Ausgabe des interaktiven Modus, nur der Wert an sich ausgegeben wird. So wird bei der automatischen Ausgabe der Wert eines Strings in Hochkommata geschrieben, während dies bei `print` nicht der Fall ist:

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

Auch hier ist es problemlos möglich, anstatt eines konstanten Wertes einen Variablennamen zu verwenden:

```
>>> var = 9
>>> print var
9
```

oder das Ergebnis eines Ausdrucks direkt auszugeben:

```
>>> print -3 * 4
-12
```

Außerdem ermöglicht `print` es, mehrere Variablen oder Konstanten in einer Zeile auszugeben. Dazu werden die Werte durch Kommata getrennt angegeben. Jedes Komma wird bei der Ausgabe durch ein Leerzeichen ersetzt:

```
>>> print -3, 12, "Python rockt"
-3 12 Python rockt
```

Das ist insbesondere dann hilfreich, wenn nicht nur einzelne Werte, sondern auch ein kurzer erklärender Text dazu ausgegeben werden soll. So etwas könnte folgendermaßen erreicht werden:

```
>>> var = 9
>>> print "Die magische Zahl ist:", var
Die magische Zahl ist: 9
```

Abschließend ist noch zu sagen, dass `print` nach jeder Ausgabe einen Zeilenvorschub ausgibt. Es wird also stets in eine neue Zeile geschrieben.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

5.2 Das erste Programm

Als Einstieg in die Programmierung mit Python bieten wir hier ein kleines Beispielprogramm, das Spiel Zahlenraten. Die Spielidee ist folgende:

Der Spieler soll eine im Programm festgelegte Zahl erraten. Dazu stehen ihm beliebig viele Versuche zur Verfügung. Nach jedem Versuch informiert ihn das Programm darüber, ob die geratene Zahl zu groß, zu klein oder genau richtig gewesen ist. Sobald der Spieler die Zahl erraten hat, gibt das Programm die Anzahl der Versuche aus und wird beendet. Aus Sicht des Spielers soll das Ganze folgendermaßen aussehen:

```
Raten Sie: 42
Zu klein
Raten Sie: 10000
Zu gross
Raten Sie: 999
Zu klein
Raten Sie: 1337
Super, Sie haben es in  4 Versuchen geschafft!
```

Kommen wir vom Ablaufprotokoll zur konkreten Implementierung in Python (siehe Abbildung 5.2).

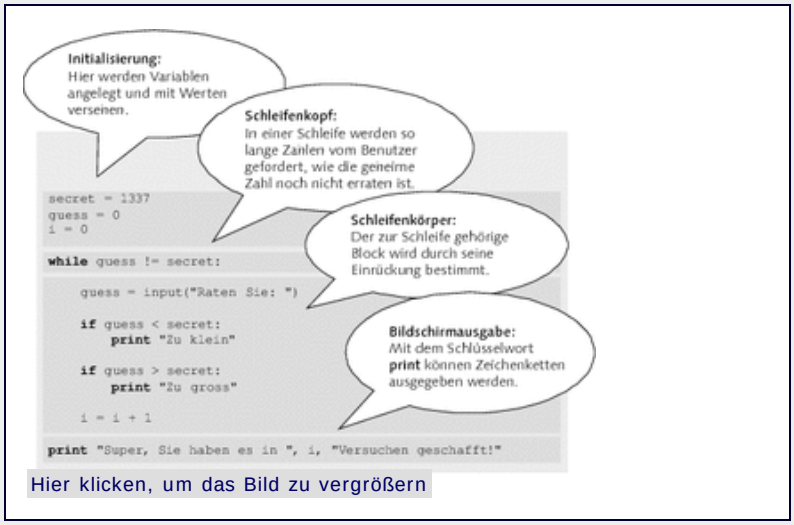
Jetzt möchten wir die einzelnen Bereiche des Programms noch einmal ausführlich diskutieren.

Initialisierung

Hier werden die für das Spiel benötigten Variablen angelegt. Python unterscheidet zwischen verschiedenen *Variablentypen*, wie Zeichenketten, Ganz- oder Fließkommazahlen. Der Typ einer Variablen wird zur Laufzeit des Programms anhand des ihr zugewiesenen Wertes bestimmt. Es ist also nicht nötig, einen Variablentyp explizit anzugeben. Eine Variable kann im Laufe des Programms ihren Typ ändern.

In unserem Spiel werden Variablen für die gesuchte Zahl

(secret), die Benutzereingabe (guess) und den Versuchszähler (i) angelegt und mit Anfangswerten versehen. Dadurch, dass guess und secret zu Beginn des Programms verschiedene Werte haben, ist sichergestellt, dass die Schleife anläuft.



Hier klicken, um das Bild zu vergrößern

Abbildung 5.2 Zahlenraten, ein einfaches Beispiel

Schleifenkopf

Eine while-Schleife wird eingeleitet. Eine while-Schleife läuft so lange, wie die im Schleifenkopf genannte Bedingung (guess != secret) erfüllt ist, also in diesem Fall, bis die Variablen guess und secret den gleichen Wert haben. Aus Benutzersicht bedeutet dies: Die Schleife läuft so lange, bis die Benutzereingabe mit der gespeicherten Zahl übereinstimmt.

Den zum Schleifenkopf gehörigen Schleifenkörper erkennt man daran, dass die nachfolgenden Zeilen um eine Stufe weiter eingerückt wurden. Sobald die Einrückung wieder um einen Schritt nach links geht, endet der Schleifenkörper.

Schleifenkörper

In der ersten Zeile des Schleifenkörpers wird eine Zahl vom Spieler eingelesen und in der Variablen guess gespeichert. Dazu wird die Zeile input("Raten Sie: ") benötigt. Der String "Raten Sie: " wird dabei vor der Eingabe ausgegeben und dient dazu, den Benutzer zur Eingabe der Zahl aufzufordern.

Nach dem Einlesen wird einzeln geprüft, ob die eingegebene Zahl guess größer oder kleiner als die gesuchte Zahl secret ist, und mittels print eine entsprechende Meldung ausgegeben. Schlussendlich wird der Versuchszähler i um eins erhöht.

Nach dem Hochzählen des Versuchszählers endet der Schleifenkörper, da die nächste Zeile nicht mehr unter dem Schleifenkopf eingerückt ist.

Bildschirmausgabe

Die letzte Programmzeile gehört nicht mehr zum Schleifenkörper. Das bedeutet, dass sie erst ausgeführt wird, wenn die Schleife vollständig durchlaufen, das Spiel also gewonnen ist. In diesem Fall werden eine Erfolgsmeldung sowie die Anzahl der benötigten Versuche ausgegeben. Das Spiel ist beendet.

Erstellen Sie jetzt Ihr erstes Python-Programm, indem Sie den Programmcode in eine Datei namens *spiel.py* schreiben und zur Ausführung bringen. Ändern Sie den Startwert von guess, und spielen Sie das Spiel.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

5.3 Kommentare

Sie können sich sicherlich vorstellen, dass es nicht das Ziel ist, Programme zu schreiben, die auf eine Postkarte passen würden. Mit der Zeit wird der Quelltext Ihrer Programme umfangreicher und komplexer werden. Irgendwann ist der Zeitpunkt erreicht, da bloßes Gedächtnistraining nicht mehr ausreicht, um die Übersicht zu bewahren. Spätestens dann kommen Kommentare ins Spiel.

Ein *Kommentar* ist ein kleiner Text, der eine bestimmte Stelle des Quellcodes kurz erläutert und auf Probleme, offene Aufgaben oder Ähnliches hinweisen kann. Ein Kommentar wird vom Interpreter einfach ignoriert, ändert also am Ablauf des Programms selbst nichts.

Die einfachste Möglichkeit, einen Kommentar zu verfassen, ist der sogenannte *Zeilenkommentar*. Diese Art des Kommentars wird mit dem #-Zeichen begonnen und endet mit dem Ende der Zeile:

```
# Ein Beispiel mit Kommentaren
print "Hallo Welt!" # Simple Hallo-Welt-Ausgabe
```

Für längere Kommentare bietet sich ein *Blockkommentar* an. Ein Blockkommentar beginnt und endet mit drei aufeinanderfolgenden Anführungszeichen (""").

```
""" Dies ist ein Blockkommentar,
er kann sich über mehrere Zeilen erstrecken. """
```

Kommentare sollten nur gesetzt werden, wenn sie zum Verständnis des Quelltextes beitragen oder sonstige wertvolle Informationen enthalten. Jede noch so unwichtige Zeile zu kommentieren führt dazu, dass man den Wald vor lauter Bäumen nicht mehr sieht.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

5.4 Der Fehlerfall

Vielleicht haben Sie bereits ein wenig mit dem Beispielprogramm aus Abschnitt 5.2 gespielt und sind dabei auf eine solche Ausgabe des Interpreters gestoßen:

```
File "spiel.py", line 8
    if guess < secret
                ^
SyntaxError: invalid syntax
```

Es handelt sich dabei um eine Fehlermeldung, die in diesem Fall auf einen Syntaxfehler im Programm hinweist. Können Sie erkennen, welcher Fehler hier vorliegt? Richtig, es fehlt der Doppelpunkt am Ende der Zeile.

Python stellt bei der Ausgabe einer Fehlermeldung wichtige Informationen bereit, die bei der Fehlersuche hilfreich sind:

- Die erste Zeile der Fehlermeldung gibt Aufschluss darüber, in welcher Zeile innerhalb welcher Datei der Fehler aufgetreten ist. In diesem Fall handelt es sich um die Zeile 8 in der Datei *spiel.py*.
- Der mittlere Teil zeigt den betroffenen Ausschnitt des Quellcodes, wobei die genaue Stelle, auf die sich die Meldung bezieht, mit einem kleinen Pfeil markiert ist. Wichtig ist, dass dies die Stelle ist, an der der Interpreter den Fehler erstmalig feststellen konnte. Das ist nicht unbedingt gleichbedeutend mit der Stelle, an der der Fehler gemacht wurde.
- Die letzte Zeile spezifiziert den Typ der Fehlermeldung, in diesem Fall einen Syntax Error. Dies sind die am häufigsten auftretenden Fehlermeldungen. Sie zeigen an, dass der Compiler das Programm aufgrund eines formalen Fehlers nicht weiter übersetzen konnte.

Neben dem Syntaxfehler gibt es eine ganze Reihe weiterer

Fehlertypen, die hier nicht alle im Detail besprochen werden sollen. Wir möchten jedoch noch auf den `IndentationError` (dt. *Einrückungsfehler*) hinweisen, da er gerade bei Python-Anfängern häufig auftritt. Versuchen Sie dazu einmal folgendes Programm auszuführen:

```
i = 10
if i == 10:
print "Falsch eingerueckt"
```

Sie sehen, dass die letzte Zeile eigentlich einen Schritt weiter eingerückt sein müsste. So, wie das Programm jetzt geschrieben wurde, hat die `if`-Anweisung keinen Anweisungskörper. Das ist nicht zulässig, und es tritt ein `IndentationError` auf:

```
File "indent.py", line 3
    print "Falsch eingerueckt"
        ^
IndentationError: expected an indented block
```

Nachdem wir uns mit diesen Grundlagen vertraut gemacht haben, kommen wir zu einem wichtigen Sprachelement aller modernen Programmiersprachen, den Kontrollstrukturen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

6.2 Schleifen ▼

Eine sogenannte *Schleife* ermöglicht es ganz allgemein, einen Codeblock, den sogenannten *Schleifenkörper*, mehrmals hintereinander auszuführen. Python unterscheidet zwei Typen von Schleifen: eine `while`-Schleife als sehr simples Konstrukt und eine `for`-Schleife zum Durchlaufen komplexerer Datentypen.

top

6.2.1 While-Schleife ▼▲

Die `while`-Schleife haben wir bereits in unserem Spiel »Zahlenraten« verwendet. Sie dient dazu, einen Codeblock so lange auszuführen, wie eine bestimmte Bedingung erfüllt ist. In unserem ersten Programm aus Abschnitt 5.2 wurde mithilfe einer `while`-Schleife so lange eine neue Zahl vom Spieler eingelesen, bis die eingegebene Zahl mit der gesuchten Zahl übereinstimmte.

Grundsätzlich besteht eine `while`-Schleife aus einem Schleifenkopf, in dem die Bedingung steht, sowie einem Schleifenkörper, der dem auszuführenden Codeblock entspricht. Beachten Sie, dass die Schleife läuft, *solange* die Bedingung erfüllt ist, und nicht, *bis* sie erfüllt ist.

```
while Bedingung:
    Anweisung
        :
    Anweisung
```

Hier klicken, um das Bild zu vergrößern

Abbildung 6.5 Struktur einer while-Schleife

Das folgende Beispiel ist ein etwas verknappter Ausschnitt des »Zahlenraten«-Spiels und soll die Verwendung der `while`-Schleife veranschaulichen:

```
secret = 1337
guess = 0
while guess != secret:
    guess = input("Raten Sie: ")
```

Das Schlüsselwort `while` leitet den Schleifenkopf ein und wird von der gewünschten Bedingung und einem Doppelpunkt gefolgt. In den nächsten Zeilen folgt, um eine Stufe weiter eingerückt, der Schleifenkörper. Dort wird eine Zahl vom Benutzer eingelesen und mit dem Namen `guess` versehen. Dieser Prozess läuft so lange, bis die im Schleifenkopf genannte Bedingung erfüllt ist, bis also die Eingabe des Benutzers (`guess`) mit der geheimen Zahl (`secret`) übereinstimmt.

Ähnlich wie eine `if`-Anweisung kann eine `while`-Schleife um einen `else`-Zweig erweitert werden. Der Codeblock, der zu diesem Zweig gehört, wird genau einmal ausgeführt, nämlich dann, wenn die Schleife vollständig abgearbeitet wurde, also die Bedingung zum ersten Mal `False` ergibt:

```
while Bedingung:
    Anweisung
        :
    Anweisung
else:
    Anweisung
        :
    Anweisung
```

Hier klicken, um das Bild zu vergrößern

Abbildung 6.6 Struktur einer while-Schleife mit else-Zweig

Betrachten wir dies an einem konkreten Beispiel:

```
secret = 1337
guess = 0
while guess != secret:
    guess = input("Raten Sie: ")
else:
    print "Sie haben es geschafft!"
```

Aus Benutzersicht bedeutet dies, dass die Erfolgsmeldung ausgegeben wird, wenn die richtige Zahl geraten wurde:

```
Raten Sie: 100
Raten Sie: 200
Raten Sie: 1337
Sie haben es geschafft!
```

Momentan scheint dieser `else`-Zweig überflüssig, da der gleiche Effekt durch folgenden Code erreicht werden kann:

```
secret = 1337
guess = 0
while guess != secret:
    guess = input("Raten Sie: ")
print "Sie haben es geschafft!"
```

Die beiden Beispiele sind in diesem Anwendungsfall völlig äquivalent zu verwenden. Dies ist nicht immer der Fall. Dass der `else`-Zweig einer `while`-Schleife durchaus seine Berechtigung hat, werden Sie im nächsten Abschnitt sehen.

top

6.2.2 Vorzeitiger Abbruch einer Schleife ▼▲

Stellen Sie sich einmal vor, wir wollten das Beispiel, das wir im vorherigen Abschnitt eingeführt haben, dahingehend erweitern, dass das Spiel durch Eingabe einer `0` beendet werden kann. Dies

ist mit Ihrem bisherigen Kenntnisstand zwar möglich, jedoch nur über Umwege zu erreichen. Was wirklich fehlt, ist eine Möglichkeit, eine Schleife in besonderen Fällen vorzeitig zu beenden. Genau dies wird mit der sogenannten break-Anweisung erreicht:

```
secret = 1337
guess = 0
while guess != secret:
    guess = input("Raten Sie: ")
    if guess == 0:
        print "Das Spiel wird beendet"
        break
else:
    print "Sie haben es geschafft!"
```

Das Beispiel wurde durch eine if-Anweisung erweitert. Direkt nachdem eine Zahl vom Spieler eingegeben und mit dem Namen guess versehen wurde, wird geprüft, ob es sich bei der Eingabe um eine 0 handelt (guess == 0). Sollte dies der Fall sein, wird eine entsprechende Meldung ausgegeben und die while-Schleife mittels break beendet.

In Kombination mit break zeigt sich auch die eigentliche Bedeutung des else-Zweigs einer Schleife. Der else-Zweig wird nur ausgeführt, wenn die Schleife vollständig durchlaufen wurde, und nicht, wenn sie durch break vorzeitig beendet wurde. Das Ablaufprotokoll gibt uns hier recht:

```
Raten Sie: 100
Raten Sie: 200
Raten Sie: 0
Das Spiel wird beendet
```

top

6.2.3 Vorzeitiger Abbruch eines Schleifendurchlaufs

Wir haben mit break bereits eine Möglichkeit vorgestellt, den Ablauf einer Schleife zu beeinflussen. Die sogenannte continue-Anweisung bricht im Gegensatz zu break jedoch nicht die gesamte Schleife ab, sondern nur den aktuellen Schleifendurchlauf. Um dies zu veranschaulichen, betrachten wir das folgende Beispiel, das bisher noch ohne continue-Anweisung auskommt:

```
while True:
    zahl = input("Geben Sie eine Zahl ein: ")
    ergebnis = 1
    while zahl > 0:
        ergebnis = ergebnis * zahl
        zahl = zahl - 1
    print "Ergebnis: ", ergebnis
```

Zur Erklärung des Beispiels: In einer Endlosschleife, also einer while-Schleife, deren Bedingung unter allen Umständen erfüllt ist (while True), wird eine Zahl eingelesen und die Variable ergebnis mit 1 initialisiert. In einer darauf folgenden weiteren while-Schleife wird ergebnis so lange mit zahl multipliziert, wie die Bedingung zahl > 0 erfüllt ist. Zudem wird in jedem Durchlauf der inneren Schleife der Wert von zahl um 1 verringert.

Nachdem die innere Schleife durchlaufen ist, wird die Variable ergebnis ausgegeben. Wie Sie vermutlich bereits erkannt haben, berechnet das Beispielprogramm die Fakultät einer jeden eingegebenen Zahl:

```
Geben Sie eine Zahl ein: 4
Ergebnis:  24
Geben Sie eine Zahl ein: 5
Ergebnis:  120
Geben Sie eine Zahl ein: 6
Ergebnis:  720
```

Allerdings erlaubt der obige Code auch eine solche Eingabe:

```
Geben Sie eine Zahl ein: -10
Ergebnis:  1
```

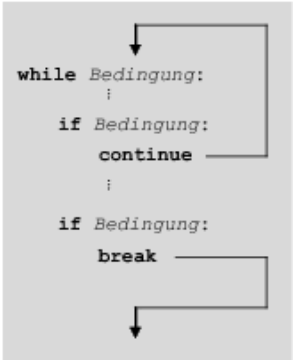
Durch die Eingabe einer negativen Zahl ist die Bedingung der inneren Schleife (`zahl > 0`) von vornherein `False`, die Schleife wird also gar nicht erst ausgeführt. Aus diesem Grund wird sofort der Wert von `ergebnis` ausgegeben, der in diesem Fall `1` ist.

Das ist allerdings nicht ganz das, was in diesem Fall erwartet werden würde. Bei einer negativen Zahl handelt es sich um eine ungültige Eingabe. Idealerweise sollte das Programm also bei Eingabe einer ungültigen Zahl die Berechnung abbrechen und kein Ergebnis anzeigen. Eben dies wird durch Verwendung einer `continue`-Anweisung erreicht:

```
while True:
    zahl = input("Geben Sie eine Zahl ein: ")
    if zahl < 0:
        print "Negative Zahlen sind nicht erlaubt"
        continue
    ergebnis = 1
    while zahl > 0:
        ergebnis = ergebnis * zahl
        zahl = zahl - 1
    print "Ergebnis: ", ergebnis
```

Direkt nachdem die Eingabe des Benutzers eingelesen wurde, wird in einer `if`-Abfrage überprüft, ob es sich um eine negative Zahl handelt (`zahl < 0`). Sollte das der Fall sein, so wird mittels `print` eine entsprechende Fehlermeldung ausgegeben und der aktuelle Schleifendurchlauf mit `continue` abgebrochen. Das heißt, dass alle Codezeilen, die zur Schleife gehören und hinter `continue` stehen, erst im nächsten Schleifendurchlauf wieder interpretiert werden. Aus Benutzersicht bedeutet das, dass nach Eingabe einer negativen Zahl kein Ergebnis, sondern eine Fehlermeldung ausgegeben wird. Danach wird zur Eingabe der nächsten Zahl aufgefordert:

```
Geben Sie eine Zahl ein: 4
Ergebnis:  24
Geben Sie eine Zahl ein: 5
Ergebnis:  120
Geben Sie eine Zahl ein: -10
Negative Zahlen sind nicht erlaubt
Geben Sie eine Zahl ein: -100
Negative Zahlen sind nicht erlaubt
```



Hier klicken, um das Bild zu vergrößern

Abbildung 6.7 Eine Schleife mit break und continue

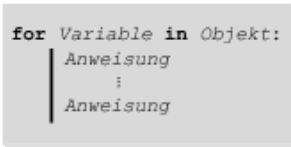
Rückblickend möchten wir an dieser Stelle noch einmal den Unterschied zwischen `break` und `continue` herausarbeiten (siehe Abbildung 6.7).

Während `break` die Schleife vollständig abbricht, beendet `continue` nur den aktuellen Schleifendurchlauf, die Schleife an sich läuft aber weiter.

top

6.2.4 For-Schleife

Neben der bisher behandelten `while`-Schleife existiert in Python ein weiteres Schleifenkonstrukt, die sogenannte `for`-Schleife. Eine `for`-Schleife kann im einfachsten Fall als *Zählschleife* verwendet werden. Das ist eine Schleife, die es dem Programmierer ermöglicht, festzulegen wie oft ein Codeblock erneut ausgeführt werden soll. Die Anzahl der bisherigen Schleifendurchläufe steht im Codeblock als Variable, dem sogenannten *Schleifenzähler*, zur Verfügung (siehe Abbildung 6.8).



Hier klicken, um das Bild zu vergrößern

Abbildung 6.8 Struktur einer for-Schleife

Was genau für `Objekt` eingesetzt werden kann, werden wir im Folgenden erklären. Konkret im Quelltext sieht eine `for`-Schleife beispielsweise so aus:

```
for i in range(5):
    print i
```

In diesem Beispiel werden alle ganzen Zahlen von `0` bis einschließlich `4` ausgegeben. `range` kann dabei nicht nur ein Limit setzen, sondern allgemein in drei Varianten verwendet werden:

`range(`*stop*`)` `range(`*start*`, `*stop*`)` `range(`*start*`, `*stop*`, `*step*`)`

Der Platzhalter *start* steht dabei für die Zahl, mit der begonnen wird. Die Schleife wird beendet, sobald *stop* erreicht wurde. Wichtig ist zu wissen, dass der Schleifenzähler selbst niemals den Wert *stop* erreicht, er bleibt stets kleiner. In jedem Schleifendurchlauf wird der Schleifenzähler um *step* erhöht. Sowohl *start* als auch *stop* und *step* müssen ganze Zahlen sein. Wenn alle Werte angegeben sind, sieht die `for`-Schleife folgendermaßen aus:

```
for i in range(1, 10, 2):
    print i
```

Die Zählvariable `i` beginnt jetzt mit dem Wert `1`, die Schleife wird ausgeführt, solange `i` kleiner ist als `10`, und in jedem Schleifendurchlauf wird `i` um `2` erhöht. Damit gibt die Schleife die Werte `1`, `3`, `5`, `7` und `9` auf dem Bildschirm aus.

Eine `for`-Schleife kann nicht nur in positiver Richtung verwendet werden, es ist auch möglich, herunterzuzählen:

```
for i in range(10, 1, -2):
    print i
```

In diesem Fall wird `i` zu Beginn der Schleife auf den Wert `10` gesetzt und in jedem Durchlauf um `2` verringert. Die Schleife läuft, solange `i` größer ist als `1`, und würde die Werte `10`, `8`, `6`, `4` und `2` auf dem Bildschirm ausgeben.

Damit bietet sich die `for`-Schleife geradezu an, um das Beispiel des letzten Abschnitts zur Berechnung der Fakultät einer Zahl zu überarbeiten. Es ist gleichzeitig ein Beispiel dafür, dass `while`- und `for`-Schleifen wie selbstverständlich ineinander verwendet werden können:

```
while True:
    zahl = input("Geben Sie eine Zahl ein: ")
    if zahl < 0:
        print "Negative Zahlen sind nicht erlaubt"
        continue
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis = ergebnis * i
    print "Ergebnis: ", ergebnis
```

Nachdem eine Eingabe durch den Benutzer erfolgt ist und diese auf ihr Vorzeichen hin überprüft wurde, wird eine `for`-Schleife eingeleitet. Der Schleifenzähler `i` der Schleife beginnt mit dem Wert 2. Die Schleife läuft, solange `i` kleiner als `zahl+1` ist: Der höchstmögliche Wert von `i` ist also `zahl`. In jedem Schleifendurchlauf wird dann die Variable `ergebnis` mit `i` multipliziert.

Es wurde bereits angedeutet, dass eine Zählschleife nur eine mögliche Verwendung der `for`-Schleife darstellt. Ganz allgemein durchläuft eine `for`-Schleife ein sogenanntes *iterierbares Objekt*. Ohne näher ins Detail gehen zu wollen, ist zu sagen, dass ein solches Objekt in der Regel eine Art Container für eine Reihe verschiedener Werte darstellt. Sie werden im Laufe dieses Buchs viele solcher Objekte kennenlernen, und wir werden dabei jedes Mal auf die Verwendung der `for`-Schleife zurückkommen.

Eines dieser iterierbaren Objekte haben Sie jedoch bereits kennengelernt: Ein String kann ganz allgemein als ein Container für eine Reihe von Buchstaben betrachtet werden, und als solcher auch mithilfe der `for`-Schleife durchlaufen werden. Der Vorgang wird auch *Iterieren* genannt. Es wird »über einen String iteriert«:

```
for c in "Hallo Welt":
    print c
```

Die Ausgabe dieses Beispiels lautet:

```
H
a
l
l
o

W
e
l
t
```

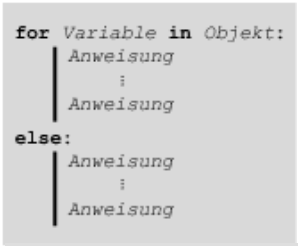
Der Namenswechsel der Schleifenvariable von `i` nach `c` hat übrigens keine syntaktische Bedeutung, sondern eher eine assoziative: `i` kann als Abkürzung für »integer« (dt. *ganze Zahl*) und `c` für »character« (dt. *Buchstabe*) angesehen werden.

Dass jeder Buchstabe in eine neue Zeile geschrieben wurde, hat nichts mit der Schleife zu tun, sondern ist ein normales Verhalten der `print`-Anweisung.

Abschließend ist noch zu sagen, dass eine `for`-Schleife genauso über einen `else`-Zweig verfügen kann wie eine `while`-Schleife. Auch bei einer `for`-Schleife ist ein `else`-Zweig nur in Kombination mit `break` sinnvoll (siehe Abbildung 6.9).

Konkret:

```
for c in "abc":
    print c
else:
    print "abc"
```



Hier klicken, um das Bild zu vergrößern

Abbildung 6.9 Struktur einer for-Schleife mit else-Zweig

Dies führt zu folgender Ausgabe:

```
a
b
c
abc
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

6.3 Die pass-Anweisung

Während der Entwicklung eines Programms kommt es vor, dass eine Kontrollstruktur vorerst nur teilweise implementiert wird. Der Programmierer erstellt einen Anweisungskopf, fügt aber keinen Anweisungskörper an, da er sich vielleicht zuerst um andere, wichtigere Dinge kümmern möchte. Ein in der Luft hängender Anweisungskopf ohne entsprechenden Körper ist aber ein Syntaxfehler.

Zu diesem Zweck existiert die pass-Anweisung. Es ist eine Anweisung, die gar nichts macht. Sie könnte folgendermaßen angewendet werden:

```
if x == 1:
    pass
elif x == 2:
    print "x hat den Wert 2"
```

In diesem Fall ist im Körper der if-Anweisung nur pass zu finden. Sollte x also den Wert 1 haben, passiert schlicht und einfach nichts.

Die pass-Anweisung hat den Zweck, Syntaxfehler in vorläufigen Programmversionen zu vermeiden. Fertige Programme enthalten in der Regel keine pass-Anweisungen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

7.2 Referenzen und Instanzen freigeben

Während eines Programmlaufs werden in der Regel sehr viele Instanzen angelegt, die aber nicht alle die ganze Zeit gebraucht werden. Betrachten wir einmal den folgenden fiktiven Programmanfang:

```
willkommen = "Herzlich willkommen im Beispielprogramm"
print willkommen
# Hier würde es jetzt mit dem restlichen Programm
weitergehen
```

Es ist leicht ersichtlich, dass die von `willkommen` referenzierte Instanz nach der Begrüßung nicht mehr gebraucht wird und somit während der restlichen Programmlaufzeit sinnlos Speicher verschwendet. Wünschenswert wäre also eine Möglichkeit, nicht mehr benötigte Instanzen auf Anfrage entfernen zu können.

Python lässt den Programmierer den Speicher nicht direkt verwalten, sondern übernimmt dies für ihn. Als Folge davon können wir bestehende Instanzen nicht manuell löschen, sondern müssen uns auf einen Automatismus verlassen, die sogenannte *Garbage Collection* [Die ***Garbage Collection*** (dt. *Müllabfuhr*) ist ein System, das nicht mehr benötigte Datenobjekte entfernt und den dazugehörigen Speicher wieder freigibt. Sie arbeitet für den Programmierer unsichtbar im Hintergrund. Für technisch Interessierte: Pythons Garbage Collection ist durch ein Reference-Counting-System implementiert, das durch einen Algorithmus zur Erkennung zyklischer Referenzen ergänzt wird.] .

Trotzdem gibt es eine Form der Einflussnahme:

Instanzen, auf die keine Referenzen mehr verweisen, werden von Python als nicht mehr benötigt eingestuft und dementsprechend wieder freigegeben. Wollen wir also eine Instanz entfernen, müssen wir nur die dazugehörigen Referenzen freigeben. Für diesen Zweck gibt es in Python die `del`-Anweisung. Nach ihrer Freigabe existiert die Referenz nicht mehr, und ein versuchter `NameError`

Zugriff führt zu einem :

```
>>> v1 = 1337
>>> v1
1337
>>> del v1
>>> v1

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'v1' is not defined
```

Möchte man mehrere Instanzen auf einmal freigeben, trennt man sie einfach durch Kommata voneinander ab:

```
>>> v1 = 1337
>>> v2 = 2674
>>> v3 = 4011
>>> del v1, v2, v3
>>> v1

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'v1' is not defined
```

Um zu erkennen, wann für eine Instanz keine Referenzen mehr existieren, speichert Python intern für jede Instanz einen Zähler, den sogenannten *Referenzzähler* (engl. *Reference Count*). Für frisch erzeugte Instanzen hat er den Wert null. Immer wenn eine neue Referenz auf eine Instanz erzeugt wird, erhöht sich der Referenzzähler der Instanz um eins, und immer, wenn eine Referenz freigegeben wird, wird er um eins verringert. Damit gibt der Referenzzähler einer Instanz immer die aktuelle Anzahl von Referenzen an, die auf sie verweisen. Erreicht der Zähler den Wert null, gibt es für die Instanz keine Referenz mehr. Da Instanzen für den Programmierer nur über Referenzen zugänglich sind, ist der Zugriff auf eine solche Instanz nicht mehr möglich – sie kann gelöscht werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

7.3 Mutable vs. immutable Datentypen

Vielleicht sind Sie beim Ausprobieren des gerade Beschriebenen schon auf den folgenden Scheinwiderspruch gestoßen:

```
>>> a = 1
>>> b = 1
>>> id(a)
9656320
>>> id(b)
9656320
>>> a is b
True
```

Warum referenzieren a und b dieselbe Ganzzahl-Instanz, wie es der Identitätenvergleich zeigt, obwohl wir in den ersten beiden Zeilen ausdrücklich zwei Instanzen mit dem Wert 1 erzeugt haben?

Um diese Frage zu beantworten, müssen wir wissen, das Python grundlegend zwischen zwei Arten von Datentypen unterscheidet: zwischen *mutable* (dt. *veränderlichen*) Datentypen und *immutable* (dt. *unveränderlichen*) Datentypen. Wie die Namen schon sagen, besteht der Unterschied zwischen den beiden Arten darin, ob sich der Wert einer Instanz zur Laufzeit ändern kann, ob sie also veränderbar ist. Instanzen eines mutable Typs sind dazu in der Lage, nach ihrer Erzeugung andere Werte anzunehmen, während dies bei immutable Datentypen nicht der Fall ist.

Wenn sich der Wert einer Instanz aber nicht ändern kann, macht es auch keinen Sinn, mehrere immutable Instanzen des gleichen Werts im Speicher zu verwalten, weil im Optimalfall genau eine Instanz ausreicht, auf die dann alle entsprechenden Referenzen verweisen. Wie Sie sich nun sicherlich denken, handelt es sich bei Ganzzahlen eben um so einen immutable Datentyp, und Python hat aus Optimierungsgründen bei beiden Einsen auf dieselbe Instanz verweisen lassen. Auch Strings sind immutable. [Das bedeutet natürlich nicht, dass Strings und Ganzzahlen aus Sicht des Programmierers unveränderlich sind. Es wird nur bei jeder Manipulation eines immutable Datentyps eine neue Instanz des

Datentyps erzeugt, anstatt die alte zu verändern.]

Es ist allerdings nicht so, dass es immer nur genau eine Instanz zu jedem benötigten Wert eines unveränderlichen Datentyps gibt, obwohl dies theoretisch möglich wäre. Der Grund dafür liegt in der Optimierung:

Wird eine neue Instanz eines immutable Typs vom Programm angefordert, gibt es für Python zwei Möglichkeiten. Entweder wird eine neue Instanz im Speicher erstellt oder eine vorhandene ein weiteres Mal referenziert. Eine neue Instanz im Speicher zu erzeugen »kostet« Python Rechenzeit und Speicherplatz. Python muss Speicher anfordern und diesen mit den entsprechenden Informationen füllen. Eine bestehende Instanz ein weiteres Mal zu referenzieren ist um ein Vielfaches »billiger«, da sowohl das Bereitstellen als auch das Befüllen des Speichers entfallen und stattdessen nur ein Referenzzähler erhöht sowie eine Speicheradresse kopiert werden muss. Das stimmt aber nur dann, wenn der Interpreter schon weiß, an welcher Stelle im Speicher eine Instanz mit dem gleichen Wert wie der neu angeforderten Instanz liegt. Je »länger« der Wert der neuen Instanz ist und je mehr Instanzen es bereits gibt, desto aufwendiger gestaltet sich die Suche nach einer bereits bestehenden passenden Instanz. Ab einem gewissen Punkt ist es dann nicht mehr effizient, eine bereits existierende Instanz erneut zu referenzieren, weil die Suche mehr Rechenzeit kostet als das Erstellen einer neuen Instanz. Python entscheidet unabhängig vom Programmierer, welchen der beiden Wege es beschreitet. Beispielsweise haben wir im letzten Abschnitt die Arbeitsweise von `id` mit dem String `"Hallo Welt"` verdeutlicht und festgestellt, dass sich die Identitäten der beiden Instanzen unterscheiden: Python hat in diesem Fall aus den oben genannten Optimierungsgründen zwei Instanzen des Strings erstellt, obwohl dies nicht nötig gewesen wäre.

Bei den mutable, also den veränderlichen Datentypen sieht es anders aus: Weil Python damit rechnen muss, dass sich der Wert einer solchen Instanz nachträglich ändern wird, ist das obige System, nach Möglichkeit bereits vorhandene Instanzen erneut zu referenzieren, nicht sinnvoll. Hier kann man sich also darauf verlassen, dass immer eine neue Instanz erzeugt wird.

Weil wir bisher noch keinen veränderbaren Datentyp eingeführt haben, muss an dieser Stelle auf ein Beispiel verzichtet werden.

Wir werden im Folgenden bei der Einführung neuer Datentypen angeben, zu welcher der beiden Kategorien sie gehören.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

8.2 Das Nichts – NoneType

Beginnen wir mit dem einfachsten Datentyp überhaupt: dem Nichts. Der dazu gehörige Basisdatentyp wird *NoneType* genannt. Es drängt sich natürlich die Frage auf, wieso es eines Datentyps bedarf, der einzig und allein dazu da ist, »nichts« zu repräsentieren. Nun, es ist eigentlich nur konsequent. Stellen Sie sich einmal folgende Situation vor: Sie implementieren ein Verfahren, bei dem jede reelle Zahl ein mögliches Ergebnis ist. Allerdings kann es in einigen Fällen vorkommen, dass die Berechnung nicht durchführbar ist. Welcher Wert soll als Ergebnis zurückgegeben werden? Richtig: »Nichts«. Auch dass das »Nichts« in Python ein eigener Datentyp ist, hat durchaus seine Berechtigung, denn dadurch kann man Variablen explizit auf den Wert »Nichts« testen.

Kommen wir zur konkreten Verwendung des Datentyps: Es gibt nur eine einzige Instanz des »Nichts« namens `None`. Dies ist eine Konstante, die Sie jederzeit im Quelltext verwenden können:

```
>>> ref = None
>>> ref
>>> print ref
None
```

Im Beispiel wurde eine Referenz namens `ref` auf `None` angelegt. Dass `None` tatsächlich dem »Nichts« entspricht, merken wir in der

zweiten Zeile. Wir versuchen, ref vom Interpreter ausgeben zu lassen, und erhalten tatsächlich kein Ergebnis. Um den Wert dennoch auf dem Bildschirm ausgeben zu können, müssen wir uns des Schlüsselwortes print bedienen.

Es wurde bereits gesagt, dass None die einzige Instanz des »Nichts« ist. Diese Besonderheit können wir uns zunutze machen, um sehr effizient zu überprüfen, ob eine Referenz auf None verweist oder nicht:

```
if ref is None:
    print "ref ist None"
```

Mit dem Schlüsselwort is wird überprüft, ob die von ref referenzierte Instanz mit None identisch ist. Diese Art, einen Wert auf None zu testen, kann vom Interpreter schneller ausgeführt werden, als der wertbezogene Vergleich mit dem Operator ==, der selbstverständlich auch möglich ist. Beachten Sie, dass diese beiden Operationen nur in diesem Fall und auch hier nur vordergründig äquivalent sind: Mit == werden zwei Werte und mit is zwei Identitäten auf Gleichheit geprüft.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

8.3 Numerische Datentypen

Die numerischen Datentypen sind eine Kategorie, zu der fünf Basisdatentypen gehören: `int` und `long` zum Speichern von ganzen Zahlen, `float` für Gleitkommazahlen, `complex` für komplexe Zahlen und `bool` für boolesche Werte. Alle numerischen Datentypen sind immutable, also unveränderlich. Beachten Sie, dass dies nicht bedeutet, dass es keine Operatoren gibt, um Zahlen zu verändern, sondern vielmehr, dass nach jeder Veränderung eine neue Instanz des jeweiligen Datentyps erzeugt werden muss. Aus Sicht des Programmierers besteht also zunächst kaum ein Unterschied.

Für alle numerischen Datentypen sind folgende Operatoren definiert:

Operator	Ergebnis
`x + y`	Summe von x und y
`x – y`	Differenz von x und y
`x * y`	Produkt von x und y
`x / y`	Quotient von x und y
`x % y`	Rest beim Teilen von x durch y (außer bei `complex`)
`+x`	Positives Vorzeichen, lässt x unverändert

-x	Negatives Vorzeichen – Vorzeichenwechsel bei x
x ** y	x hoch y
x // y	Abgerundeter Quotient von x und y (außer bei complex)

Tabelle 8.1 Gemeinsame Operatoren numerischer Datentypen

Hinweis

Sollten Sie bereits eine C-ähnliche Programmiersprache beherrschen, wundern Sie sich zu Recht, denn in Python existiert kein Operator für Inkrementierungen (x++) oder Dekrementierungen (x--).

Neben diesen grundlegenden Operatoren existiert in Python eine Reihe zusätzlicher Operatoren. Oftmals möchte man beispielsweise die Summe von x und y berechnen und das Ergebnis in x speichern, x also um y erhöhen. Dazu ist mit den obigen Operatoren folgende Anweisung nötig:

```
x = x + y
```

Für solche Fälle gibt es in Python sogenannte *erweiterte Zuweisungen* (engl. *augmented assignments*), die als eine Art Abkürzung für die obige Anweisung angesehen werden können.

Operator	Entsprechung
x += y	x = x + y
x -= y	x = x – y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x **= y	x = x ** y
x //= y	x = x // y

Tabelle 8.2 Gemeinsame Operatoren numerischer Datentypen

Wichtig ist, dass Sie hier für y einen beliebigen arithmetischen Ausdruck einsetzen können, während x ein Ausdruck sein muss, der auch als Ziel einer normalen Zuweisung eingesetzt werden könnte.

Für die Datentypen `int`, `long`, `float` und `bool` sind außerdem *vergleichende Operatoren* definiert. Da komplexe Zahlen prinzipiell nicht sinnvoll anzuordnen sind, lässt der Datentyp `complex` nur die Verwendung der ersten drei Operatoren zu:

Operator	Ergebnis
==	Wahr, wenn x und y gleich sind
!=	Wahr, wenn x und y verschieden sind
<>	Analog zu !=, bitte nicht verwenden
<	Wahr, wenn x kleiner ist als y (außer bei complex)
<=	Wahr, wenn x kleiner oder gleich y ist (außer bei complex)
>	Wahr, wenn x größer ist als y (außer bei complex)
>=	Wahr, wenn x größer oder gleich y ist (außer bei complex)

Tabelle 8.3 Gemeinsame Operatoren numerischer Datentypen

Jeder dieser vergleichenden Operatoren liefert als Ergebnis einen

Wahrheitswert. Ein solcher Wert wird zum Beispiel als Bedingung einer `if`-Anweisung erwartet. Die Operatoren könnten also folgendermaßen verwendet werden:

```
if x < 4:
    print "x ist kleiner als 4"
```

Es können beliebig viele der vergleichenden Operatoren zu einer Reihe verkettet werden. Das obere Beispiel ist genau genommen nur ein Spezialfall dieser Regel, mit lediglich zwei Operanden. Die Bedeutung einer solchen Verkettung entspricht der mathematischen Sichtweise und soll am folgenden Beispiel zu erkennen sein:

```
if 2 < x < 4:
    print "x liegt zwischen 2 und 4"
```

Mehr zu booleschen Werten folgt im Unterabschnitt des entsprechenden Basisdatentyps `bool`.

Numerische Datentypen können ineinander umgeformt werden. Dabei können je nach Umformung Informationen verloren gehen. Als Beispiel betrachten wir einige Konvertierungen im interaktiven Modus:

```
>>> float(33)
33.0
>>> int(33.5)
33
>>> bool(12)
True
>>> complex(True)
 (1+0j)
```

Allgemein wird zunächst der Name des Datentyps geschrieben, in den konvertiert werden soll, gefolgt von dem zu konvertierenden Wert in Klammern. Statt eines konkreten Literals kann auch eine Referenz eingesetzt werden bzw. eine Referenz mit dem entstehenden Wert verknüpft werden:

```
>>> var1 = 12.5
>>> int(var1)
12
>>> var2 = int(40.25)
>>> var2
40
```

So viel zur allgemeinen Einführung in die numerischen Datentypen. Die folgenden Abschnitt werden jeden dieser Datentypen im Detail behandeln.

8.3.1 Ganzzahlen – int, long ▼▲

Im Raum der ganzen Zahlen gibt es in Python zwei Datentypen: den Typ `int` für den begrenzten Zahlenraum von -2^{31} bis $2^{31} - 1$ (auf 32-Bit-Systemen) und den Typ `long` für ganze Zahlen, deren Länge theoretisch unbegrenzt ist.

Der Datentyp `int` hat dabei durchaus seine Berechtigung, da er intern als ein Datenwort der zugrunde liegenden Rechnerarchitektur gespeichert wird und somit sehr schnell verarbeitet werden kann. Um mit dem Datentyp `long` zu arbeiten, muss intern sehr viel mehr Aufwand betrieben werden, weshalb die Verarbeitung von `int` im Verhältnis zu `long` grundsätzlich schneller ist. Falls das Ergebnis einer Operation nicht mehr durch den Datentyp `int` abgebildet werden kann, erzeugt Python automatisch eine Instanz vom Typ `long`. Vor dem Programmierer bleibt das in den meisten Fällen verborgen.

Der bevorzugte Datentyp für ganze Zahlen ist `int`. Möchte man eine Zahl explizit als `long` definieren, so kennzeichnet man dies

durch ein L oder l am Ende des Literals. Auch wenn beides zulässig ist, empfehlen wir hier, stets ein großes L zu verwenden, da das kleine allzu häufig, und gerade am Ende einer Zahl, mit der 1 (Eins) verwechselt wird:

```
v_int = 12345
v_long = 123489751285373945938732451
```

Zahlensysteme

Ganze Zahlen, egal ob int oder long, können in Python in mehreren Zahlensystemen geschrieben werden.

- Zahlen, die, wie im obigen Beispiel, ohne ein spezielles Präfix geschrieben sind, werden im *Dezimalsystem* interpretiert. Zu beachten ist, dass einer solchen Zahl keine führenden Nullen vorangestellt werden dürfen:

```
v_dez1 = 1337
v_dez2 = 1337L
```

- Eine führende Null kennzeichnet eine Zahl, die im *Oktalsystem* geschrieben wurde. Die Verwendung des Oktalsystems ist ein Relikt aus älteren Zeiten und wird heute kaum noch benötigt. Beachten Sie, dass hier nur Ziffern von 0 bis 7 erlaubt sind:

```
v_okt1 = 02471
v_okt2 = 02471L
```

- Die nächste und weitaus gebräuchlichere Variante ist das *Hexadezimalsystem*, das durch das Präfix 0x bzw. 0X gekennzeichnet wird. Die Zahl selbst darf aus den Ziffern 0 - 9 und den Buchstaben A–F bzw. a–f gebildet werden:

```
v_hex1 = 0x5A3F
v_hex2 = 0X5a3fL
```

Für alle diese Literale ist die Verwendung eines negativen Vorzeichens möglich:

```
>>> -1234
-1234
>>> -0777
-511
>>> -0xFF
-255
```

Vielleicht möchten Sie sich nicht auf diese drei Zahlensysteme beschränken, die von Python explizit unterstützt werden, sondern ein exotischeres verwenden. Natürlich gibt es in Python nicht für jedes mögliche Zahlensystem ein eigenes Literal. Stattdessen können Sie sich folgender Schreibweise bedienen:

```
v_6 = int("54425", 6)
```

Es handelt sich um eine alternative Methode, eine Instanz des Datentyps int zu erzeugen und mit einem Anfangswert zu versehen. Dazu werden in den Klammern ein String, der den gewünschten Initialwert in dem gewählten Zahlensystem enthält, sowie die Basis dieses Zahlensystems als ganze Zahl geschrieben. Beide Werte müssen durch ein Komma getrennt werden. Im Beispiel wurde das Sechsersystem verwendet.

Python unterstützt Zahlensysteme mit einer Basis zwischen 2 und 36. Wenn ein Zahlensystem mehr als zehn verschiedene Ziffern zur Darstellung einer Zahl benötigt, werden zusätzlich zu den Ziffern 0 bis 9 die Buchstaben des englischen Alphabets A bis Z verwendet.

v_6 hat jetzt den Wert 7505 (im Dezimalsystem).

Beachten Sie, dass es sich bei den Zahlensystemen nur um eine alternative Schreibweise des gleichen Wertes handelt. Der Datentyp `int` springt beispielsweise nicht in eine Art Hexadezimalmodus, sobald er einen solchen Wert enthält. Ein Zahlensystem ist nur bei Wertzuweisungen oder -ausgaben von Bedeutung. Standardmäßig werden alle Zahlen im Dezimalsystem ausgegeben:

```
>>> v1 = 0xFF
>>> v2 = 0777
>>> v1
255
>>> v2
511
```

Wir werden später, im Zusammenhang mit Strings, darauf zurückkommen, wie sich Zahlen in anderen Zahlensystemen ausgeben lassen.

Bit-Operationen

Neben dem Hexadezimal- und dem Oktalsystem ist in der Informatik das Dualsystem von großer Bedeutung. Das Dualsystem, oder auch Binärsystem, ist ein Zahlensystem mit der Basis 2. Eine ganze Zahl wird also als Folge von Einsen und Nullen dargestellt. In Python existiert kein Literal, mit dem Zahlen in Dualschreibweise direkt verwendet werden könnten, jedoch sind für die Datentypen `int` und `long` einige Operatoren definiert, die sich explizit auf die binäre Darstellung der Zahl beziehen:

Operator	Ergebnis
`x & y`	Bitweises UND von x und y (AND)
`x | y`	Bitweises nicht ausschließendes ODER von x und y (OR)
`x ^ y`	Bitweises ausschließendes ODER von x und y (XOR)
`~x`	Bitweises Komplement von x
`x << n`	Bitverschiebung um n Stellen nach links
`x >> n`	Bitverschiebung um n Stellen nach rechts

Tabelle 8.4 Bit-Operatoren der Datentypen int und long

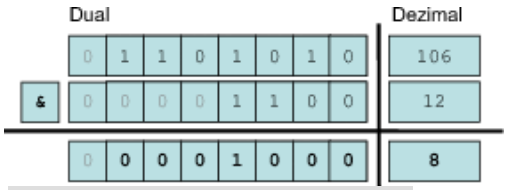
Auch hier sind erweiterte Zuweisungen mithilfe der folgenden Operatoren möglich:

Operator	Entsprechung
`x &= y`	`x = x & y`
`x |= y`	`x = x | y`
`x ^= y`	`x = x ^ y`
`x <<= n`	`x = x << n`
`x >>= n`	`x = x >> n`

Tabelle 8.5 Bit-Operatoren der Datentypen int und long

Da vielleicht nicht jedem unmittelbar klar ist, was die einzelnen Operationen bewirken, möchten wir sie im Folgenden im Detail besprechen.

Das *bitweise UND* zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verknüpft werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine `1`, wo die jeweiligen Bits der Operanden übereinstimmen, und es hat da eine `0`, wo sich diese unterscheiden. Dies soll durch die folgende Grafik veranschaulicht werden:



Hier klicken, um das Bild zu vergrößern

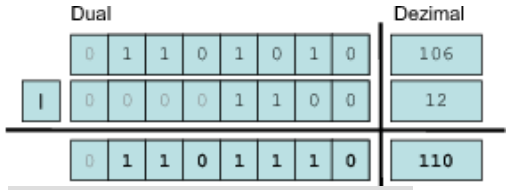
Abbildung 8.1 Bitweises UND

Im interaktiven Modus von Python probieren wir aus, ob das bitweise UND mit den in der Grafik gewählten Operanden tatsächlich das erwartete Ergebnis zurückgibt:

```
>>> 106 & 12
8
```

Diese Prüfung des Ergebnisses werden wir nicht für jede Operation einzeln durchführen. Um allerdings mit den bitweisen Operatoren vertrauter zu werden, lohnt es sich, hier ein wenig zu experimentieren.

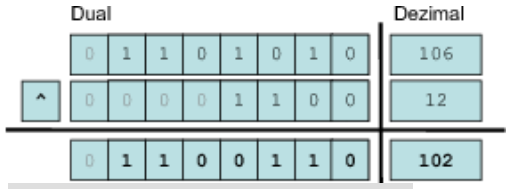
Das *bitweise ODER* zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verglichen werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine 1, wo mindestens eines der jeweiligen Bits der Operanden 1 ist. Abbildung 8.2 veranschaulicht dies.



Hier klicken, um das Bild zu vergrößern

Abbildung 8.2 Bitweises nicht ausschließendes ODER

Das *bitweise ausschließende ODER* zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verglichen werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine 1, wo sich die jeweiligen Bits der Operanden voneinander unterscheiden, und da eine 0, wo sie gleich sind. Dies wird von Abbildung 8.3 veranschaulicht.



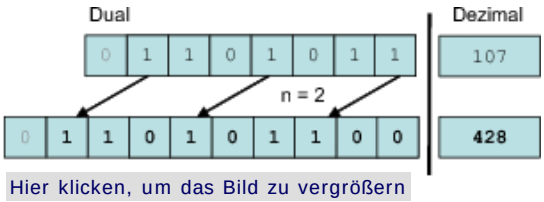
Hier klicken, um das Bild zu vergrößern

Abbildung 8.3 Bitweises exklusives ODER

Das *bitweise Komplement* bildet das sogenannte Einerkomplement einer Dualzahl, das der Negation aller vorkommenden Bits entspricht. In Python ist dies auf Bitebene nicht möglich, da eine ganze Zahl in ihrer Länge unbegrenzt ist und das Komplement immer in einem abgeschlossenen Zahlenraum gebildet werden muss. Deswegen wird die eigentliche Bit-Operation zur arithmetischen Operation und folgendermaßen definiert: [Das ist sinnvoll, da man zur Darstellung negativer Zahlen in abgeschlossenen Zahlenräumen das sogenannte Zweierkomplement verwendet. Dieses erhält man, indem man zum Einerkomplement 1 addiert. Also: $-x$ = Zweierkomplement von x = $\sim x + 1$ Daraus folgt: $\sim x = -x - 1$]

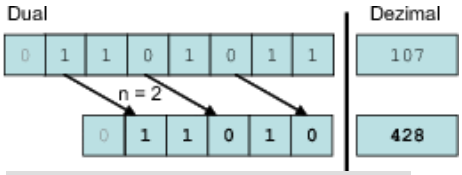
$\sim x = -x - 1$

Bei der *Bitverschiebung* wird die Bitfolge in der binären Darstellung des ersten Operanden um die durch den zweiten Operanden gegebene Anzahl Stellen nach links bzw. rechts verschoben. Die entstandene Lücke wird mit Nullen gefüllt. Abbildung 8.4 und Abbildung 8.5 veranschaulichen eine Verschiebung um zwei Stellen nach links bzw. nach rechts.



Hier klicken, um das Bild zu vergrößern

Abbildung 8.4 Bitverschiebung um zwei Stellen nach links



Hier klicken, um das Bild zu vergrößern

Abbildung 8.5 Bitverschiebung um zwei Stellen nach rechts

Die in der Bitdarstellung entstehenden Lücken auf der rechten bzw. linken Seite werden mit Nullen aufgefüllt.

top

8.3.2 Gleitkommazahlen – float

Zu Beginn dieses Teils des Buches sind wir bereits oberflächlich auf Gleitkommazahlen eingegangen, was wir hier ein wenig vertiefen möchten. Zum Speichern einer Gleitkommazahl mit begrenzter Genauigkeit wird der Datentyp `float` verwendet.

Wie bereits besprochen wurde, sieht eine Gleitkommazahl im einfachsten Fall folgendermaßen aus:

```
v = 3.141
```

Python unterstützt außerdem eine Notation, die es ermöglicht, die Exponentialschreibweise zu verwenden:

```
v = 3.141e-12
```

Durch ein kleines oder großes e wird die *Mantisse* (`3.141`) vom *Exponenten* (`-12`) getrennt. Übertragen in die mathematische Schreibweise, entspricht `3.141e-12` $3{,}141 \cdot 10^{-12}$. Beachten Sie, dass sowohl die Mantisse als auch der Exponent im Dezimalsystem anzugeben sind. Andere Zahlensysteme sind nicht vorgesehen, was die gefahrlose Verwendung von führenden Nullen ermöglicht:

```
v = 03.141e-0012
```

Es gibt noch weitere Varianten, eine gültige Gleitkommazahl zu definieren. Es handelt sich dabei um Spezialfälle der obigen Notation, weswegen sie etwas exotisch wirken. Sie sollten der Vollständigkeit halber trotzdem erwähnt werden. Pythons interaktiver Modus gibt nach jeder Eingabe ihren Wert aus. Das machen wir uns zunutze und lassen zu jedem Spezialfall den normal formatierten Wert automatisch ausgeben:

```
>>> -3.
-3.0
>>> .001
0.001
>>> 3e2
```

```
300.0
```

Eventuell haben Sie gerade schon etwas mit den Gleitkommazahlen experimentiert und sind dabei auf einen vermeintlichen Fehler des Interpreters gestoßen:

```
>>> 0.9
0.90000000000000002
```

Aufgrund der Begrenztheit von `float` können rationale Zahlen nicht unendlich präzise gespeichert werden. Stattdessen werden sie mit einer bestimmten Genauigkeit angenähert. In diesem Fall konnte keine präzisere Annäherung an die `0.9` gefunden werden. Es ist unter Verwendung der Basisdatentypen nicht möglich mit beliebig genauen Dezimalzahlen zu rechnen. Dazu muss die Standardbibliothek bemüht werden, was wir zu gegebener Zeit behandeln werden.

Gleitkommazahlen können nicht beliebig genau gespeichert werden. Das impliziert auch, dass es sowohl eine Ober- als auch eine Untergrenze für diesen Datentyp geben muss. Und tatsächlich können Gleitkommazahlen, die in ihrer Größe ein bestimmtes Limit überschreiten, in Python nicht mehr dargestellt werden. Wenn das Limit überschritten wird, wird die Zahl als `inf` gespeichert, bzw. als `-inf`, wenn das untere Limit unterschritten wurde. Es kommt also zu keinem Fehler, und es ist immer noch möglich, eine übergroße Zahl mit anderen zu vergleichen:

```
>>> 3.0e999
inf
>>> -3.0e999
-inf
>>> 3.0e999 < 12.0
False
>>> 3.0e999 > 12.0
True
>>> 3.0e999 == 3.0e999999999999
True
```

Es ist zwar möglich, zwei unendlich große Gleitkommazahlen miteinander zu vergleichen, jedoch lässt sich nur bedingt mit ihnen rechnen. Dazu folgendes Beispiel:

```
>>> 3.0e999 + 1.5e999999
inf
>>> 3.0e999 - 1.5e999999
nan
>>> 3.0e999 * 1.5e999999
inf
>>> 3.0e999 / 1.5e999999
nan
```

Zwei unendlich große Gleitkommazahlen lassen sich problemlos addieren oder multiplizieren. Das Ergebnis ist in beiden Fällen wieder `inf`. Ein Problem gibt es aber, wenn versucht wird, zwei solche Zahlen zu subtrahieren bzw. zu dividieren. Da diese Rechenoperationen nicht sinnvoll sind, ergeben sie `nan`. Der Status `nan` ist vom Typ her ähnlich wie `inf`, bedeutet jedoch »not a number«, also so viel wie »nicht berechenbar«.

Beachten Sie, dass weder `inf` noch `nan` eine Konstante ist, die Sie selbst in einem Python-Programm verwenden könnten.

top

8.3.3 Boolesche Werte – bool

Eine Instanz des Datentyps `bool` kann nur zwei verschiedene Werte annehmen: »Wahr« oder »Falsch« oder, um innerhalb der Python-Syntax zu bleiben, `True` bzw. `False`. Deshalb ist es auf den ersten Blick absurd, `bool` den numerischen Datentypen unterzuordnen. Python sieht hier jedoch `True` analog zur `1` und `False` analog zur `0`, sodass sich mit booleschen Werten genauso rechnen lässt wie beispielsweise schon mit den ganzen Zahlen. Bei den Namen `True` und `False` handelt es sich um Konstanten, die im Quelltext verwendet werden können. Zu beachten ist

besonders, dass die Konstanten mit einem Großbuchstaben beginnen:

```
v1 = True
v2 = False
```

Logische Operatoren

Ein oder mehrere boolesche Werte lassen sich mithilfe von bestimmten Operatoren zu einem booleschen Ausdruck kombinieren. Ein solcher Ausdruck resultiert, wenn er ausgewertet wurde, wieder in einem booleschen Wert, also in `True` oder `False`. Bevor es zu theoretisch wird, folgt hier zunächst die Tabelle der sogenannten *logischen Operatoren*, und darunter sehen Sie weitere Erklärungen mit konkreten Beispielen.

Operator	Ergebnis
not x	Logische Negierung von x
x and y	Logisches UND zwischen x und y
x or y	Logisches (nicht ausschließendes) ODER zwischen x und y

Tabelle 8.6 Logische Operatoren des Datentyps bool

Die *logische Negierung* eines booleschen Wertes ist schnell erklärt: Der entsprechende Operator `not` macht `True` zu `False` und `False` zu `True`. In einem konkreten Beispiel würde das folgendermaßen aussehen:

```
if not x:
    print "x ist False"
else:
    print "x ist True"
```

Das *logische UND* zwischen zwei Wahrheitswerten ergibt nur dann `True`, wenn beide Operanden bereits `True` sind. In folgender Tabelle sind alle möglichen Fälle aufgelistet:

x	y	Ausdruck: a and b
True	True	True
False	True	False
True	False	False
False	False	False

Tabelle 8.7 Mögliche Fälle des logischen UNDs

In einem konkreten Beispiel würde die Anwendung des logischen UNDs so aussehen:

```
if x and y:
    print "x und y sind True"
```

Das *logische ODER* zwischen zwei Wahrheitswerten ergibt genau dann eine wahre Aussage, wenn mindestens einer der beiden Operanden wahr ist. Es handelt sich demnach um ein nicht ausschließendes ODER. Ein Operator für ein logisches ausschließendes (exklusives) ODER existiert in Python nicht. Folgende Tabelle listet alle möglichen Fälle auf:

x	y	Ausdruck: a or b
True	True	True
False	True	True
True	False	True

False	False	False

Tabelle 8.8 Mögliche Fälle des logischen ODERs

Ein logisches ODER könnte folgendermaßen implementiert werden:

```
if x or y:
    print "x oder y ist True"
```

Selbstverständlich können all diese Operatoren miteinander kombiniert und in einem komplexen Ausdruck verwendet werden. Das könnte etwa folgendermaßen aussehen:

```
if x and y or y and z and not x:
    print "Holla die Waldfee"
```

Wir möchten diesen Ausdruck hier nicht im Einzelnen besprechen. Es sei nur gesagt, dass der Einsatz von Klammern den erwarteten Effekt hat, nämlich dass umklammerte Ausdrücke zuerst ausgewertet werden. Die folgende Tabelle zeigt den Wahrheitswert des Ausdruckes auf, und zwar in Abhängigkeit von den drei Parametern x, y und z:

x	y	z	Ausdruck: x and y or y and z and not x
True	True	True	True
False	True	True	True
True	False	True	False
True	True	False	True
False	False	True	False
False	True	False	False
True	False	False	False
False	False	False	False

Tabelle 8.9 Mögliche Ergebnisse des Ausdrucks

Zu Beginn des Abschnitts über numerische Datentypen haben wir einige vergleichende Operatoren eingeführt, die eine Wahrheitsaussage in Form eines booleschen Wertes ergeben. Das folgende Beispiel zeigt, dass diese ganz selbstverständlich zusammen mit den logischen Operatoren verwendet werden können:

```
if x > y or (y > z and x != 0):
    print "Mein lieber Schwan"
```

In diesem Fall muss es sich bei x, y und z um Variablen der Typen int, float oder auch bool handeln.

Wahrheitswerte anderer Datentypen

In Python lassen sich Instanzen eines jeden Basisdatentyps in einen booleschen Wert überführen. Dies ist eine sinnvolle Eigenschaft, da sich eine Instanz der Basisdatentypen häufig in zwei Stadien befinden kann: »leer« und »nicht leer«. Oftmals möchte man beispielsweise testen, ob ein String Buchstaben enthält oder nicht. Dadurch, dass ein String in einen booleschen Wert konvertiert werden kann, wird ein solcher Test sehr einfach durch logische Operatoren möglich:

```
>>> not ""
True
>>> not "abc"
False
```

Durch Verwendung eines logischen Operators wird der Operand

automatisch als Wahrheitswert interpretiert.

Für jeden Basisdatentyp wurde ein bestimmter Wert als `False` definiert. Alle davon abweichenden Werte sind `True`. Die folgende Tabelle listet für jeden Datentyp den entsprechenden `False`-Wert auf. Einige der Datentypen wurden noch nicht eingeführt, woran Sie sich an dieser Stelle jedoch nicht weiter stören sollten.

Basisdatentyp	False-Wert	Beschreibung
NoneType	None	Der Wert None
Numerische Datentypen		
int, long	0	Der Wert Null
float	0.0	Der Wert Null
complex	0 + 0j	Der Wert Null
Sequenzielle Datentypen		
str	""	Eine leerer String
list	[]	Eine leere Liste
tuple	()	Ein leeres Tupel
Assoziative Datentypen		
dict	{}	Ein leeres Dictionary
Mengen		
set, frozenset	set(), frozenset()	Eine leere Menge

Tabelle 8.10 Wahrheitswerte anderer Datentypen

Alle anderen Werte ergeben `True`.

Betrachten wir die Konvertierung eines Wertes in einen Wahrheitswert anhand einer Gleitkommazahl:

```
>>> bool(0.0)
False
>>> bool(0.0e12)
False
>>> bool(1.0)
True
>>> bool(123.456)
True
```

Auswertung logischer Operatoren

Python wertet logische Ausdrücke grundsätzlich von links nach rechts, also Im folgenden Beispiel zuerst a und dann b, aus:

```
if a or b:
    print "a oder b sind True"
```

Es wird aber nicht garantiert, dass jeder Teil des Ausdruckes tatsächlich ausgewertet wird. Aus Optimierungsgründen bricht Python die Auswertung des Ausdruckes sofort ab, wenn das Ergebnis feststeht. Wenn im obigen Beispiel also a bereits den Wert `True` hat, ist der Wert von b nicht weiter von Belang. b würde dann nicht mehr ausgewertet werden. Dieses Detail scheint unwichtig, kann aber zu schwer auffindbaren Fehlern führen.

Zu Beginn dieses Kapitels wurde gesagt, dass ein boolescher Ausdruck stets einen booleschen Wert ergibt, wenn er ausgewertet wurde. Das ist nicht ganz korrekt, denn auch hier wurde die Arbeitsweise des Interpreters in einer Weise optimiert, über die man Bescheid wissen sollte. Deutlich wird dies an folgendem Beispiel aus dem interaktiven Modus:

```
>>> 0 or 1
1
```

Nach dem, was wir bisher besprochen haben, sollte das Ergebnis

des Ausdrucks `True` sein, was mitnichten der Fall ist. Stattdessen gibt Python hier den ersten Operanden mit dem Wahrheitswert `True` zurück. Das ist um einiges effizienter, da keine neue Instanz erzeugt werden muss, und hat in vielen Fällen trotzdem den erwünschten Effekt, denn der zurückgegebene Wert wird problemlos automatisch in den Wahrheitswert `True` überführt. Die Auswertung der beiden Operatoren `or` und `and` läuft dabei folgendermaßen ab:

- Das logische ODER (`or`) nimmt den Wert des ersten Operanden an, der den Wahrheitswert `True` besitzt, oder – wenn es einen solchen nicht gibt – den Wert des letzten Operanden.
- Das logische UND (`and`) nimmt den Wert des ersten Operanden an, der den Wahrheitswert `False` besitzt, oder – wenn es einen solchen nicht gibt – den Wert des letzten Operanden.

Diese Details haben dabei auch durchaus ihren unterhaltsamen Wert:

```
>>> "Python" or "Java"
'Python'
```

top

8.3.4 Komplexe Zahlen – complex

Überraschenderweise findet sich ein Datentyp zur Speicherung komplexer Zahlen unter Pythons Basisdatentypen. In vielen Programmiersprachen würden komplexe Zahlen eher eine Randnotiz in der Standardbibliothek darstellen oder ganz außen vor bleiben. Sollten Sie nicht mit komplexen Zahlen vertraut sein, können Sie dieses Kapitel gefahrlos überspringen. Es wird nichts behandelt, was für das weitere Erlernen von Python vorausgesetzt würde.

Komplexe Zahlen bestehen aus einem reellen Realteil und einem Imaginärteil, der aus einer reellen Zahl besteht, die mit der imaginären Einheit *j* multipliziert wird. Das in der Mathematik eigentlich übliche Symbol der imaginären Einheit ist *i*. Python hält sich hier an die Notationen der Elektrotechnik. Die imaginäre Einheit *j* kann als Lösung der Gleichung

$$j^2 = -1$$

verstanden werden. Im folgenden Beispiel weisen wir einer komplexen Zahl den Namen `v` zu:

```
v = 4j
```

Wenn man, wie im Beispiel, nur einen Imaginärteil angibt, wird der Realteil automatisch als `0` angenommen. Um den Realteil festzulegen, wird dieser auf den Imaginärteil addiert. Die beiden folgenden Schreibweisen sind äquivalent:

```
v1 = 3 + 4j
v2 = 4j + 3
```

Statt des kleinen `j` ist auch ein großes `J` als Literal für den Imaginärteil einer komplexen Zahl zulässig. Entscheiden Sie hier ganz nach Ihren Vorlieben, welche der beiden Möglichkeiten Sie verwenden möchten.

Sowohl der Real- als auch der Imaginärteil kann eine beliebige reelle Zahl sein, also Instanzen der Typen `int` oder `float`. Folgende Schreibweise ist demnach auch korrekt:

```
v3 = 3.4 + 4e2j
```

Zu Beginn des Abschnitts über numerische Datentypen wurde bereits angedeutet, dass sich komplexe Zahlen von den anderen numerischen Datentypen unterscheiden. Da für komplexe Zahlen keine mathematische Reihenfolge definiert ist, können Instanzen des Datentyps `complex` nur auf Gleichheit oder Ungleichheit verglichen werden. Die Menge der vergleichenden Operatoren ist also auf `==`, `!=` und `<>` beschränkt.

Des Weiteren sind sowohl der Modulo-Operator `%` als auch der Operator `//` für eine ganzzahlige Division im Komplexen zwar formal möglich, haben jedoch keinen mathematischen Sinn. Deswegen sind sie in Python inzwischen als *deprecated* (dt. *abgelehnt*), also als nicht mehr zu verwenden, eingestuft. Sollten Sie die Operatoren dennoch verwenden, wird eine entsprechende Warnung ausgegeben:

```
>>> 4j % 2+3j
sys:1: DeprecationWarning: complex divmod(), // and % are
deprecated
7j
```

Der Datentyp `complex` besitzt zwei sogenannte *Attribute*, die das Arbeiten mit ihm erheblich erleichtern. Es kommt zum Beispiel vor, dass man Berechnungen nur mit dem Realteil oder nur mit dem Imaginärteil der gespeicherten Zahl anstellen möchte. Um einen der beiden Teile zu isolieren, erlaubt Python folgende Notationen, die hier exemplarisch an einer Referenz auf eine komplexe Zahl namens `x` gezeigt werden:

Attribut	Beschreibung
`x.real`	Realteil von `x` als reelle Zahl (`float`)
`x.imag`	Imaginärteil von `x` als reelle Zahl (`float`)

Tabelle 8.11 Attribute des Datentyps complex

Diese können im Code ganz selbstverständlich verwendet werden:

```
>>> c = 23 + 4j
>>> c.real
23.0
>>> c.imag
4.0
```

Wir werden im Zusammenhang mit objektorientierter Programmierung darauf zurückkommen und näher darauf eingehen, was ein Attribut genau ist.

Außer über seine zwei Attribute verfügt der Datentyp `complex` über eine sogenannte *Methode*, die in der Tabelle exemplarisch für eine Referenz auf eine komplexe Zahl namens `x` erklärt wird.

Methode	Beschreibung
`x.conjugate()`	Liefert die zu `x` konjugiert komplexe Zahl

Tabelle 8.12 Methoden des Datentyps complex

Im Quelltext kann eine Methode ähnlich einfach verwendet werden wie ein Attribut:

```
>>> c = 23 + 4j
>>> c.conjugate()
(23-4j)
```

Das Ergebnis von `conjugate` ist wieder eine komplexe Zahl, der selbstverständlich ein Name zugewiesen werden kann. Außerdem verfügt natürlich auch das Ergebnis über eine Methode `conjugate`:

```
>>> c = 23 + 4j
>>> c2 = c.conjugate()
>>> c2
(23-4j)

>>> c3 = c2.conjugate()
>>> c3
(23+4j)
```

Näheres zur Verwendung von Methoden erfahren Sie im nächsten Abschnitt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

8.4 Methoden und Parameter

Die bisher behandelten numerischen Datentypen waren sehr einfach aufgebaut: Ihre Werte ließen sich mit einer Zahl oder – bei `complex` – mit zwei Zahlen beschreiben, und der Umgang mit ihnen beschränkte sich auf Rechen-, Bit- und Vergleichsoperationen.

Im Folgenden werden wir uns mit umfassenderen Datentypen beschäftigen, für die es Operationen gibt, die nicht durch solche Operatoren abgebildet werden können. Um diese Funktionalität trotzdem zu ermöglichen, bedient man sich sogenannter *Methoden*. Methoden beziehen sich immer auf Instanzen bestimmter Datentypen und werden durch einen sogenannten *Methodenaufruf* verwendet. Der Aufruf einer Methode sieht folgendermaßen aus:

```
referenz.methode()
```

Das bedeutet dann: »Führe die von `methode` definierten Operationen mit der Instanz aus, auf die `referenz` verweist.« Welche Methoden für eine Instanz verfügbar sind, hängt von ihrem Datentyp ab.

Viele Methoden benötigen neben der Instanz noch weitere

Informationen, um zu funktionieren. Hierfür gibt es sogenannte *Parameter*, die durch Kommata getrennt in die Klammern am Ende des Methodenaufrufs geschrieben werden:

```
referenz.methode(parameter1, parameter2)
```

Als Parameter können formal sowohl Referenzen als auch Literale verwendet werden:

```
var = 12
referenz.methode(var, "Hallo Welt!")
```

Es gibt auch *optionale Parameter*, die nur bei Bedarf übergeben werden müssen. Wenn wir Methoden mit solchen Parametern einführen, werden diese in der Parameterliste durch eckige Klammern gekennzeichnet:

```
referenz.methode(param1, param2[, param3])
```

In diesem Beispiel wären param1 und param2 reguläre, d. h. erforderliche Parameter, und param3 wäre ein optionaler Parameter. Die Methode könnte also mit zwei verschiedenen Konfigurationen aufgerufen werden:

```
referenz.methode(1, 2, 3)
referenz.methode(1, 2)
```

Bei dem ersten Aufruf wäre der Wert 3 für den optionalen Parameter param3 übergeben worden, während er beim zweiten ausgelassen wurde.

Bei den bisher besprochenen Parameterübergaben war immer die Position eines Übergabewertes entscheidend dafür, für welchen formalen Parameter er eingesetzt wurde. Im letzten Beispiel stand die 1 als Übergabewert an erster Stelle und wurde dadurch mit dem ersten Parameter der Liste, param1, verknüpft. Gleiches gilt für die 2 und param2.

Man kann einer Methode Parameter auch als sogenannte *Schlüsselwortparameter* (engl. *keyword arguments*) übergeben. Schlüsselwortparameter werden direkt mit dem formalen Parameternamen verknüpft, und ihre Reihenfolge in der Liste spielt keine Rolle mehr. Um einen Wert als Schlüsselwortparameter zu übergeben, weist man dem Parameternamen innerhalb des Aufrufs den zu übergebenden Wert mithilfe des Gleichheitszeichens zu. Die beiden folgenden Methodenaufrufe sind demnach vollkommen gleichwertig:

```
>>> referenz.methode(1, 2, 3)
>>> referenz.methode(param2=2, param1=1, param3=3)
```

Wie Sie sehen, spielt es dabei keine Rolle, ob es sich bei solchen Übergaben um optionale oder um erforderliche Parameter handelt. Man kann auch positions- und schlüsselwortbezogene Parameter mischen, wobei allerdings alle Schlüsselwortparameter am Ende der Parameterliste stehen müssen. Damit ist der nachstehende Aufruf äquivalent zu den beiden vorhergehenden:

```
>>> referenz.methode(1, param3=3, param2=2)
```

param1 wurde als positionsbezogener Parameter übergeben, während param2 und param3 als Schlüsselwortparameter übergeben wurden.

Welche der beiden Übergabemethoden man in der Praxis bevorzugt, ist größtenteils Geschmackssache. Schlüsselwortparameter haben den Vorteil, dass man nicht an die

Reihenfolge der Parameter in der Funktionsdefinition gebunden ist. Deshalb bleiben solche Aufrufe auch dann noch korrekt, wenn sich die Parameterliste in ihrer Reihenfolge ändert. Außerdem sieht man schon an der Stelle des Aufrufs anhand des Parameternamens, wofür der übergebene Wert innerhalb der Funktion benutzt wird. Dadurch kann man die Lesbarkeit eines Programms verbessern. Demgegenüber ist die Übergabe positionsbezogener Parameter mit weniger Schreibaufwand verbunden, weil nicht immer der Parametername mit angegeben werden muss. Wenn sich die Namen der formalen Parameter in der Funktionsdefinition ändern, funktionieren positionsbezogene Übergaben auch ohne Änderung, wohingegen Übergaben mit Schlüsselwortparametern angepasst werden müssen. Es ist in der Regel so, dass positionsbezogene Parameter häufiger verwendet werden, was wahrscheinlich an dem geringeren Schreibaufwand liegt.

Die meisten Methoden erzeugen ein Ergebnis, das uns als Aufrufendem zur Verfügung steht. Beispielsweise verfügen Zeichenketten über eine Methode `lower`, mit deren Hilfe man einen neuen String erzeugen kann, in dem alle Großbuchstaben des Ursprungstrings in Kleinbuchstaben konvertiert wurden:

```
>>> s = "DaS sIeHt AbEr KoMiScH aUs"
>>> s.lower()
'das sieht aber komisch aus'
```

Bei diesem sogenannten *Rückgabewert* der Methode handelt es sich um eine neue Instanz, in diesem Fall um eine String-Instanz, die wir wie gewohnt mit Referenzen versehen und anschließend weiterverwenden können:

```
>>> s = "DaS sIeHt AbEr KoMiScH aUs"
>>> low = s.lower()
>>> low.upper()
'DAS SIEHT ABER KOMISCH AUS'
```

Die Referenz `low` zeigt auf die von `s.lower()` zurückgegebene String-Instanz mit dem Wert `'das sieht aber komisch aus'`. Wie alle Strings besitzt diese ihrerseits eine Methode `upper`, die alle Kleinbuchstaben in Großbuchstaben umwandelt und von uns mit `low.upper()` aufgerufen wird.

Neben diesen Methoden, die immer an einen bestimmten Datentyp gebunden sind, existieren Operationen, die global und damit unabhängig von bestimmten Typen zur Verfügung stehen. Sie werden *Built-in Functions* (dt. *eingebaute Funktionen*) genannt und sind fast genauso zu verwenden wie Methoden, außer dass ihnen keine Referenz auf eine Instanz vorangestellt werden muss und auch der Punkt entfällt. Die Instanz, auf die sich die Operation bezieht, wird in der Regel als Parameter übergeben.

```
builtin_name(referenz)
```

Sie haben schon solche Funktionen kennengelernt, wie zum Beispiel `type` und `id` in Abschnitt 7.1:

```
>>> t = type(1337)
>>> t
<type 'int'>
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

8.6 Mappings ▼

Die Kategorie *Mappings* (dt. *Zuordnungen*) enthält Datentypen, die eine Zuordnung zwischen verschiedenen Objekten herstellen.

top

8.6.1 Dictionary – dict ▲

Der einzige Datentyp der Kategorie Mappings ist das *Dictionary*, wofür in Python der Name `dict` verwendet wird. Der Name des Datentyps gibt dabei schon einen guten Hinweis darauf, was sich dahinter verbirgt. Ein Dictionary enthält beliebig viele *Schlüssel/Wert-Paare* (engl. *key/value pairs*), wobei der Schlüssel nicht unbedingt, wie bei einer Liste, eine ganze Zahl sein muss. Vielleicht ist Ihnen dieser Datentyp schon von einer anderen Programmiersprache her bekannt, wo er als *assoziatives Array* (u. a. in PHP), *Map* (u. a. in C++) oder *Hash* (u. a. in Perl) bezeichnet wird. Der Datentyp `dict` ist mutable, also veränderlich.

Im folgenden Beispiel wird erklärt, wie ein `dict` mit mehreren Schlüssel/Wert-Paaren innerhalb von geschweiften Klammern erzeugt wird. Außerdem wird die Assoziation mit einem Wörterbuch ersichtlich:

```
woerterbuch = {"Germany" : "Deutschland", "Spain" :
"Spanien"}
```

In diesem Fall wurde ein dict mit zwei Einträgen angelegt, die durch ein Komma getrennt werden. Beim ersten wurde dem Schlüssel "Germany" der Wert "Deutschland" zugewiesen. Schlüssel und Wert werden durch einen Doppelpunkt voneinander getrennt. Beachten Sie, dass Sie nicht gezwungen sind, alle Paare in eine Zeile zu schreiben. Innerhalb der geschweiften Klammern können Sie Ihren Quellcode beliebig formatieren:

```
woerterbuch = {
              "Germany" : "Deutschland",
              "Spain" : "Spanien",
              "France" : "Frankreich"
              }
```

Hinter dem letzten Schlüssel/Wert-Paar kann ein weiteres Komma stehen, es wird aber nicht benötigt. Jeder Schlüssel muss im Dictionary eindeutig sein, es darf also kein zweiter Schlüssel mit demselben Namen existieren. Formal ist folgendes zwar möglich, es bewirkt aber nur, dass das erste Schlüssel/Wert-Paar überschrieben wird.

```
d = {
    "Germany" : "Deutschland",
    "Germany" : "Pusemuckel"
    }
```

Im Gegensatz dazu brauchen die Werte eines Dictionarys nicht eindeutig sein, dürfen also ruhig mehrfach vorkommen:

```
d = {
    "Germany" : "Deutschland",
    "Allemagne" : "Deutschland"
    }
```

In den bisherigen Beispielen waren bei allen Paaren sowohl der Schlüssel als auch der Wert ein String. Das muss nicht unbedingt sein:

```
mapping = {
          0 : 1,
          "abc" : 0.5,
          1.2e22 : [1,2,3,4],
          (1,3,3,7) : "def"
          }
```

In einem Dictionary können beliebige Instanzen, seien sie mutable oder immutable, als Werte verwendet werden. Bei dem Schlüssel ist zu beachten, dass nur Instanzen unveränderlicher (immutable) Datentypen verwendet werden dürfen. Dabei handelt es sich um alle bisher besprochenen Datentypen mit Ausnahme der Listen und der Dictionarys selbst. Versuchen wir beispielsweise, ein Dictionary zu erstellen, in dem eine Liste als Schlüssel verwendet wird, so meldet sich der Interpreter mit einem entsprechenden Fehler:

```
>>> d = {[1,2,3] : "abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Diese Beschränkung rührt daher, dass die Schlüssel eines Dictionarys anhand eines aus ihrem Wert errechneten *Hash-Werts* verwaltet werden. Prinzipiell lässt sich aus jedem Objekt ein Hash-Wert berechnen, bei veränderlichen Objekten macht dies jedoch wenig Sinn, da sich der Hash-Wert bei Veränderung des Objekts ebenfalls ändern würde. Eine solche Veränderung würde

beispielsweise die Schlüsselverwaltung eines Dictionarys zerstören. Aus diesem Grund sind veränderliche Objekte »unhashable«, wie in der obigen Fehlermeldung gesagt wird.

Bei einem Dictionary handelt es sich um ein iterierbares Objekt. Es ist daher möglich, ein Dictionary in einer `for`-Schleife zu durchlaufen. Dabei wird nicht über das komplette Dictionary iteriert, sondern nur über alle Schlüssel. Im folgenden Beispiel durchlaufen wir alle Schlüssel unseres Wörterbuchs und geben sie mittels `print` aus:

```
for key in woerterbuch:
    print key
```

Die Ausgabe des Codes sieht erwartungsgemäß folgendermaßen aus:

```
Germany
Spain
France
```

Selbstverständlich kann in einer solchen Schleife auch auf die Werte des Dictionarys zugegriffen werden. Dazu bedient man sich des Zugriffsoperators, den wir im Folgenden unter anderem behandeln werden. Beachten Sie, dass Sie die Größe des Dictionarys nicht verändern dürfen, während es in einer Schleife durchlaufen wird. Die Größe des Dictionarys würde zum Beispiel durch das Hinzufügen oder Löschen eines Schlüssel/Wert-Paares beeinflusst. Sollten Sie es dennoch versuchen, bekommen Sie folgende Fehlermeldung angezeigt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

Diese Beschränkung gilt ausschließlich für Operationen, die die Größe des Dictionarys beeinflussen, also beispielsweise das Hinzufügen und Entfernen von Einträgen. Sollten Sie in einer Schleife lediglich den korrelierenden Wert eines Schlüssels ändern, tritt keinerlei Fehler auf.

Operatoren

Bisher haben Sie gelernt, was ein Dictionary ist und wie es erzeugt wird. Außerdem sind wir auf einige Besonderheiten eingegangen. Nachfolgend besprechen wir die für Dictionarys verfügbaren Operatoren.

Operator	Beschreibung
`len(s)`	Liefert die Anzahl aller im Dictionary `s` enthaltenen Elemente.
`d[k]`	Zugriff auf den Wert mit dem Schlüssel `k`
`del d[k]`	Löschen des Schlüssels `k` und seines Wertes
`k in d`	True, wenn sich der Schlüssel `k` in `d` befindet
`k not in d`	True, wenn sich der Schlüssel `k` nicht in `d` befindet

Tabelle 8.26 Operatoren eines Dictionarys

Nachfolgend besprechen wir die Operatoren eines Dictionarys im Detail. Die meisten der Operatoren werden anhand des Dictionarys `woerterbuch` erklärt, das wir zu Beginn dieses Kapitels eingeführt haben.

Länge eines Dictionarys

Um die Länge eines Dictionarys zu bestimmen, wird die eingebaute Funktion `len` verwendet. Die Länge entspricht dabei

der Anzahl von Schlüssel/Wert-Paaren:

```
>>> len(woerterbuch)
3
```

Zugriff auf einen Wert

Um in einem Dictionary auf einen Wert zugreifen zu können, wird der entsprechende Schlüssel in eckigen Klammern hinter den Namen des Dictionarys geschrieben. Bei dem im zweiten Beispiel angelegten Wörterbuch könnte ein solcher Zugriff folgendermaßen aussehen:

```
>>> woerterbuch["Germany"]
'Deutschland'
```

Dabei erfolgt der Zugriff, indem Werte miteinander verglichen werden und nicht Identitäten. Das liegt daran, dass die Schlüssel eines Dictionarys intern durch ihren Hash-Wert repräsentiert werden, der ausschließlich anhand des Wertes einer Instanz gebildet wird. In der Praxis bedeutet dies, dass beispielsweise die Zugriffe `d[1]` und `d[1.0]` äquivalent sind.

Zu guter Letzt werfen wir noch einen Blick darauf, was passiert, wenn auf einen Wert zugegriffen werden soll, der nicht existiert. Der Interpreter antwortet mit einer Fehlermeldung:

```
>>> d = {}
>>> d[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100
```

Löschen eines Schlüssel/Wert-Paares

Um in einem Dictionary einen Eintrag zu löschen, kann das Schlüsselwort `del` in Kombination mit dem Zugriffsoperator verwendet werden. Im folgenden Beispiel wird der Eintrag `"Germany" : "Deutschland"` aus dem Dictionary entfernt werden.

```
del woerterbuch["Germany"]
```

Das Dictionary selbst existiert auch dann noch, wenn es durch Entfernen des letzten Eintrags leer geworden ist.

Auf bestimmte Schlüssel testen

Um ein Dictionary auf bestimmte Schlüssel zu testen, werden die Operatoren `in` und `not in` verwendet. Sie prüfen, ob sich ein Schlüssel im Dictionary befindet oder nicht, und geben das entsprechende Ergebnis als Wahrheitswert zurück:

```
>>> "France" in woerterbuch
True
>>> "Spain" not in woerterbuch
False
```

Methoden

Neben den Operatoren sind für Dictionarys eine ganze Reihe von Methoden definiert, die die Arbeit mit Dictionarys erleichtern. Parameter in eckigen Klammern können, je nach Verwendung der Methode, weggelassen werden.

Methode	Beschreibung
	Löscht den Inhalt des Dictionarys d. Das

`d.clear()`	Dictionary selbst bleibt bestehen.
`d.copy()`	Erzeugt eine Kopie von `d`. Beachten Sie, dass nur das Dictionary selbst kopiert wird. Alle Werte bleiben Referenzen auf dieselben Instanzen.
`d.has_key(k)`	Äquivalent zu `k in d`. Bitte nicht benutzen.
`d.items()`	Erzeugt eine Liste, die alle Schlüssel/Wert-Paare von `d` als Tupel enthält.
`d.keys()`	Erzeugt eine Liste aller Schlüssel von `d`.
`d.values()`	Erzeugt eine Liste aller Werte von `d`.
`d.update(d2)`	Fügt ein Dictionary `d2` zu `d` hinzu und überschreibt gegebenenfalls die Werte von bereits vorhandenen Schlüsseln.
`d.fromkeys(seq[, value])`	Erstellt ein neues Dictionary mit den Werten der Liste `seq` als Schlüssel und setzt jeden Wert initial auf `value`. Beachten Sie, dass diese Methode nichts am Dictionary `d` ändert.
`d.get(k[, x])`	Liefert `d[k]`, wenn der Schlüssel `k` vorhanden ist, ansonsten `x`.
`d.setdefault(k[, x])`	Das Gegenteil von `get`. Setzt `d[k] = x`, wenn der Schlüssel `k` nicht vorhanden ist.
`d.popitem()`	Gibt ein willkürliches Schlüssel/Wert-Paar von `d` zurück und entfernt es aus dem Dictionary.
`d.iteritems()`	Erlaubt es, in einer `for`-Schleife alle Schlüssel/Wert-Paare von `d` zu durchlaufen.
`d.iterkeys()`	Erlaubt es, in einer `for`-Schleife alle Schlüssel von `d` zu durchlaufen.
`d.itervalues()`	Erlaubt es, in einer `for`-Schleife alle Werte von `d` zu durchlaufen.

Tabelle 8.27 Methoden eines Dictionarys

Jetzt möchten wir alle Methoden detailliert und jeweils mit einem kurzen Beispiel im interaktiven Modus erläutern. Alle Beispiele werden dabei in folgendem Kontext erklärt:

```
>>> d = {"k1" : "v1", "k2": "v2", "k3": "v3"}
```

Es ist also in jedem Beispiel ein Dictionary `d` mit drei Schlüssel/Wert-Paaren vorhanden. In den Beispielen werden wir das Dictionary verändern und uns vom Interpreter seinen Wert ausgeben lassen. Die Ausgabe des unveränderten Dictionarys sieht folgendermaßen aus:

```
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}
```

Sie können dabei jedes Beispiel für sich betrachten und von diesen Grundvoraussetzungen ausgehen. Änderungen, die in einem Beispiel an dem Dictionary `d` durchgeführt werden, wirken sich nicht auf die Folgebeispiele aus.

d.clear()

Die Methode `clear` löscht alle Schlüssel/Wert-Paare von `d`. Sie hat dabei nicht den gleichen Effekt wie `del d`, da das Dictionary selbst nicht gelöscht, sondern nur geleert wird:

```
>>> d.clear()
>>> d
{}
```

d.copy()

Die Methode copy erzeugt eine Kopie des Dictionarys d. Beachten Sie, dass zwar das Dictionary selbst kopiert wird, es sich bei den Werten aber nach wie vor um Referenzen auf dieselben Objekte handelt.

```
>>> e = d.copy()
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}
```

d.has_key(k)

Die Methode has_key prüft, ob ein bestimmter Schlüssel im Dictionary d enthalten ist, und gibt das Ergebnis als booleschen Wert zurück. Der Schlüssel, nach dem gesucht wird, wird der Methode als Parameter übergeben:

```
>>> d.has_key("k3")
True
>>> d.has_key(123)
False
```

Die Methode has_key kann damit als äquivalent zum Operator in betrachtet werden. Sie sollten die Verwendung des Operators bevorzugen, da has_key in zukünftigen Versionen von Python entfernt werden könnte.

d.items()

Die Methode items erzeugt eine Liste, die alle Schlüssel/Wert-Paare des Dictionarys d enthält. Jedes Paar wird dabei in einem Tupel gespeichert. Diese Liste ist, wie das Dictionary selbst, nicht sortiert:

```
>>> d.items()
[('k3', 'v3'), ('k2', 'v2'), ('k1', 'v1')]
```

d.keys()

Die Methode keys funktioniert ähnlich wie items, mit dem Unterschied, dass die erzeugte Liste nicht die Schlüssel/Wert-Paare enthält, sondern nur die Schlüssel:

```
>>> d.keys()
['k3', 'k2', 'k1']
```

d.values()

Die Methode values ist das Gegenstück zu keys. Statt der Schlüssel enthält die erzeugte Liste alle Werte des Dictionarys:

```
>>> d.values()
['v3', 'v2', 'v1']
```

d.update(d2)

Die Methode update erweitert das Dictionary d um die Schlüssel und Werte des Dictionarys *d2*, das der Methode als Parameter übergeben wird:

```
>>> d.update({"k4" : "v4"})
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1', 'k4': 'v4'}
```

Sollten beide Dictionarys über einen gleichen Schlüssel verfügen, so wird der mit diesem Schlüssel verbundene Wert in d mit dem

aus *d2* überschrieben:

```
>>> d.update({"k1" : "python rulez"})
{'k3': 'v3', 'k2': 'v2', 'k1': 'python rulez'}
```

d.fromkeys(seq[, value])

Die Methode `fromkeys` erzeugt ein neues Dictionary und verwendet dabei die Einträge der Liste *seq* als Schlüssel. Der Parameter *value* ist optional. Sollte er jedoch angegeben werden, so wird er als Wert eines jeden Schlüssel/Wert-Paars verwendet:

```
>>> d.fromkeys([1,2,3], "python")
{1: 'python', 2: 'python', 3: 'python'}
```

Wird der Parameter *value* ausgelassen, so wird stets `None` als Wert eingetragen:

```
>>> d.fromkeys([1,2,3])
{1: None, 2: None, 3: None}
```

d.get(k[, x])

Die Methode `get` ermöglicht den Zugriff auf einen Wert des Dictionarys. Im Gegensatz zum Zugriffsoperator wird aber keine Exception erzeugt, wenn der Schlüssel nicht vorhanden ist. Stattdessen wird in diesem Fall der optionale Parameter *x* zurückgegeben. Sollte *x* nicht angegeben worden sein, so wird er als `None` angenommen. Die Methode `get` kann also als Ersatz für folgenden Code gesehen werden:

```
if k in d:
    wert = k[d]
else:
    wert = x
```

Die Methode `get` kann folgendermaßen verwendet werden:

```
>>> d.get("k2", 1337)
'v2'
>>> d.get("k5", 1337)
1337
```

d.setdefault(k[, x])

Die Methode `setdefault` fügt das Schlüssel/Wert-Paar {*k* : *x*} zum Dictionary d hinzu, sollte der Schlüssel *k* nicht vorhanden sein:

```
>>> d.setdefault("k2", 1337)
'v2'
>>> d.setdefault("k5", 1337)
1337
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1', 'k5': 1337}
```

d.popitem()

Die Methode `popitem` gibt ein willkürliches Schlüssel/Wert-Paar als Tupel zurück und entfernt es aus dem Dictionary. Beachten Sie, dass das zurückgegebene Paar zwar willkürlich, aber nicht zufällig ist:

```
>>> d.popitem()
('k3', 'v3')
>>> d
{'k2': 'v2', 'k1': 'v1'}
```

Sollte d leer sein, so wird eine entsprechende Exception erzeugt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

d.iteritems()

Die Methode iteritems erlaubt es, in einer for-Schleife über alle Schlüssel/Wert-Paare zu iterieren. Das könnte zum Beispiel folgendermaßen aussehen:

```
for paar in d.iteritems():
    print paar
```

In jedem Schleifendurchlauf enthält die Variable paar das jeweilige Schlüssel/Wert-Paar als Tupel. Dementsprechend sieht die Ausgabe des Beispiels aus:

```
('k3', 'v3')
('k2', 'v2')
('k1', 'v1')
```

d.iterkeys()

Die Methode iterkeys erlaubt es, in einer for-Schleife alle Schlüssel zu durchlaufen. Im folgenden Beispiel werden alle Schlüssel mittels print ausgegeben:

```
for key in d.iterkeys():
    print key
```

Wir haben eingangs gesagt, dass es keiner speziellen Methode bedarf, um alle Schlüssel eines Dictionarys zu durchlaufen. Die Methode iterkeys kann problemlos durch folgenden Code umgangen werden:

```
for key in d:
    print key
```

Beide Beispiele sind äquivalent und erzeugen folgende Ausgabe:

```
k3
k2
k1
```

d.itervalues()

Die Methode itervalues verhält sich ähnlich wie iterkeys, mit dem Unterschied, dass sie es ermöglicht, alle Werte zu durchlaufen:

```
for value in d.itervalues():
    print value
```

Das Beispiel erzeugt folgende Ausgabe:

```
v3
v2
v1
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

8.7 Mengen

Eine *Menge* (engl. *set*) ist eine ungeordnete Ansammlung von Elementen. Jedes Element kann sich dabei nur einmal in der Menge befinden. In Python gibt es zur Darstellung von Mengen zwei Basisdatentypen: `set` für eine veränderliche Menge sowie `frozenset` für eine unveränderliche Menge: `set` ist demnach mutable, `frozenset` immutable.

Eine leere Instanz der Datentypen `set` und `frozenset` wird folgendermaßen erzeugt:

```
s = set()
fs = frozenset()
```

Wenn die Menge bereits zum Zeitpunkt der Instanziierung Elemente enthalten soll, so muss ein iterierbares Objekt als Parameter übergeben werden, das alle gewünschten Elemente enthält, in diesem Fall zum Beispiel eine Liste:

```
>>> set([1,2,3,99,-7])
set([99, 1, 2, 3, -7])
```

Wir werden hier stets Listen verwenden, da ihre Verwendung am einfachsten zu durchschauen ist. Tupel, Strings, Dictionarys oder

gar Mengen selbst funktionieren zur Initialisierung eines Sets oder Frozensets problemlos. Im Falle eines Strings wird das set aus allen Buchstaben und im Falle eines Dictionarys aus allen Schlüsseln zusammengesetzt. Da in einer Menge kein Element mehrfach vorkommen darf, werden Dubletten herausgefiltert. Es kommt deswegen jedoch nicht zu einem Fehler.

```
>>> set("Python")
set(['h', 'o', 'n', 'P', 't', 'y'])
>>> frozenset({0:"a", 1:"b", 2:"c"})
frozenset([0, 1, 2])
```

An diesem Beispiel ist sehr schön zu sehen, dass die Reihenfolge der in einem set enthaltenen Werte willkürlich ist. Beachten Sie, dass sie dennoch nicht zufällig ist, sondern von bestimmten interpreterspezifischen Faktoren abhängt.

Mithilfe der Methode values eines Dictionarys lassen sich auch die Werte ins Set übertragen:

```
>>> d = ({0 : "a", 1 : "b", 2 : "c"})
>>> set(d.values())
set(['a', 'c', 'b'])
```

Bei einer Menge handelt es sich um ein iterierbares Objekt, das problemlos in einer for-Schleife durchlaufen werden kann. Dazu folgendes Beispiel:

```
menge = set([1, 100, "a", 0.5])
for element in menge:
    print element
```

Dieser Code würde folgende Ausgabe erzeugen:

```
a
1
100
0.5
```

Operatoren

Die Datentypen set und frozenset verfügen über eine gemeinsame Schnittstelle, die im Folgenden näher erläutert werden soll. Wir möchten damit beginnen, alle gemeinsamen Operatoren zu behandeln. Der Einfachheit halber werden wir uns bei der Beschreibung der Operatoren ausschließlich auf den Datentyp set beziehen. Dennoch können sie und auch die Methoden, die später beschrieben werden, für frozenset genauso verwendet werden. [Eine Menge T wird »echte Teilmenge« einer zweiten Menge M genannt, wenn T Teilmenge von M ist und weniger Elemente als M enthält.]

Operator	Beschreibung
len(s)	Liefert die Anzahl aller im Set s enthaltenen Elemente.
x in s	True, wenn x im Set s enthalten ist, andernfalls False.
x not in s	True, wenn x nicht im Set s enthalten ist. Andernfalls False.
s <= t	True, wenn es sich bei der Menge s um eine Teilmenge der Menge t handelt, andernfalls False.
s < t	True, wenn es sich bei der Menge s um eine echte Teilmenge11 der Menge t handelt, andernfalls False.
s >= t	True, wenn es sich bei der Menge t um eine Teilmenge der Menge s handelt, andernfalls False.
s > t	True, wenn es sich bei der Menge t um eine echte Teilmenge der Menge s handelt, andernfalls False.

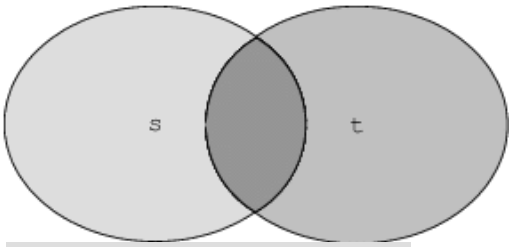
`s | t`	Erzeugt ein neues Set, das alle Elemente von `s` und `t` enthält. Diese Operation bildet also die Vereinigungsmenge zweier Mengen.
`s & t`	Erzeugt ein neues Set, das die Objekte enthält, die sowohl Element der Menge `s` als auch Element der Menge `t` sind. Diese Operation bildet also die Schnittmenge zweier Sets.
`s – t`	Erzeugt ein neues Set mit allen Elementen von `s`, außer denen, die auch in `t` enthalten sind. Diese Operation erzeugt also die Differenz zweier Mengen.
`s ^ t`	Erzeugt ein neues Set, das alle Objekte enthält, die entweder in `s` oder in `t` vorkommen, nicht aber in beiden. Diese Operation bildet also die symmetrische Differenz zweier Mengen.

Tabelle 8.28 Operatoren der Datentypen set und frozenset

Im Folgenden werden alle Operatoren anhand von Beispielen anschaulich beschrieben. Die Beispiele sind dabei in diesem Kontext zu sehen:

```
>>> s = set([0,1,2,3,4,5,6,7,8,9])
>>> t = set([6,7,8,9,10,11,12,13,14,15])
```

Es existieren also zwei Mengen namens `s` und `t`, die aus Gründen der Übersichtlichkeit jeweils ausschließlich über numerische Elemente verfügen. Die Mengen überschneiden sich in einem gewissen Bereich. Grafisch kann die Ausgangssituation folgendermaßen veranschaulicht werden. Der dunkelgraue Bereich entspricht der Schnittmenge von `s` und `t`.



Hier klicken, um das Bild zu vergrößern

Abbildung 8.7 Die Ausgangssituation

Anzahl der Elemente

Um die Anzahl der Elemente zu bestimmen, die in einer Menge enthalten sind, wird – wie schon bei den sequenziellen Datentypen sowie dem Dictionary – die eingebaute Funktion `len` verwendet:

```
>>> len(s)
10
```

Ist ein Element im Set enthalten?

Um zu testen, ob ein Element in einem Set enthalten ist, dient der Operator `in`. Zudem kann sein Gegenstück `not in` verwendet werden, um das Gegenteil zu prüfen:

```
>>> 10 in s
False
>>> 10 not in t
False
```

Handelt es sich um eine Teilmenge?

Um zu testen, ob es sich bei einem Set um eine Teilmenge eines anderen Sets handelt, werden die Operatoren <= und >=, sowie < und > für echte Teilmengen, verwendet:

```
>>> u = set([4,5,6])
>>> u <= s
True
>>> u < s
True
>>> u >= s
False
>>> u <= t
False
```

Beachten Sie den Unterschied zwischen *Teilmenge* (<=, >=) und *echter Teilmenge* (<, >) an folgendem Beispiel:

```
>>> m = set([1,2,3])
>>> n = set([1,2,3])
>>> m <= n
True
>>> m < n
False
```

Vereinigung zweier Mengen

Um zwei Mengen zusammenzufügen, kann der Operator | verwendet werden. Es wird ein neues Set erzeugt, das alle Elemente enthält, die in s oder in t enthalten sind:

```
>>> s | t
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15])
```

Bezogen auf unsere Grafik bedeutet das Folgendes:

Hier klicken, um das Bild zu vergrößern

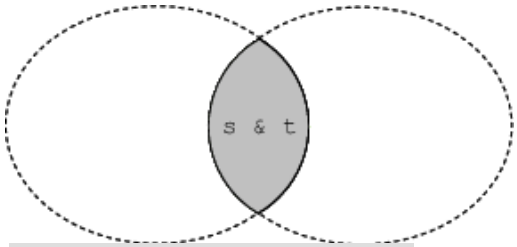
Abbildung 8.8 Vereinigungsmenge von s und t

Schnittmenge

Um die Schnittmenge zweier Mengen zu bestimmen, wird der Operator & verwendet. Es wird ein neues Set erzeugt, das alle Elemente enthält, die sowohl im ersten als auch im zweiten Operanden enthalten sind.

```
>>> s & t
set([8, 9, 6, 7])
```

Auch die Auswirkungen dieses Operators können wir veranschaulichen:



Hier klicken, um das Bild zu vergrößern

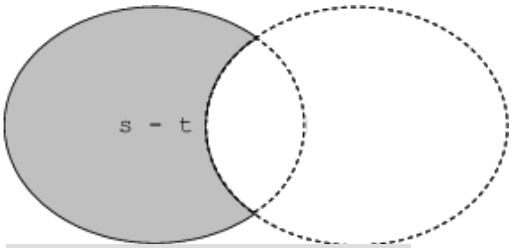
Abbildung 8.9 Schnittmenge von s und t

Differenz zweier Mengen

Um die Differenz zweier Mengen zu bestimmen, wird der Operator - verwendet. Es wird ein neues Set erzeugt, das alle Elemente des ersten Operanden enthält, die nicht zugleich im zweiten Operanden enthalten sind:

```
>>> s - t
set([0, 1, 2, 3, 4, 5])
>>> t - s
set([10, 11, 12, 13, 14, 15])
```

Grafisch bedeutet dies:



Hier klicken, um das Bild zu vergrößern

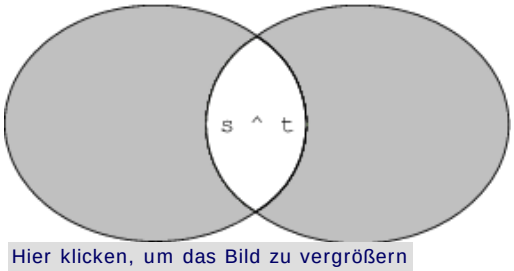
Abbildung 8.10 Differenz von s und t

Symmetrische Differenz zweier Mengen

Um die symmetrische Differenz zweier Mengen zu bestimmen, wird der Operator ^ verwendet. Es wird ein neues Set erzeugt, das alle Elemente enthält, die entweder im ersten oder im zweiten Operanden vorkommen, nicht aber in beiden gleichzeitig:

```
>>> s ^ t
set([0, 1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15])
```

Gönnen wir uns einen letzten Blick auf unsere Grafik:



Hier klicken, um das Bild zu vergrößern

Abbildung 8.11 Symmetrische Differenz von s und t

Methoden

Die Datentypen set und frozenset verfügen über eine recht überschaubare Liste von Methoden, die in ihrem Zweck sogar größtenteils gleichbedeutend mit einem der bereits diskutierten

Operatoren sind. Sie haben dennoch ihre Daseinsberechtigung, da sie aufgrund ihres Namens im Quelltext selbsterklärend sind – ganz im Gegensatz zu einem Operator, dessen Sinn sich erst nach intensiver Beschäftigung mit set und frozenset erschließt:

Methode	Beschreibung
s.issubset(t)	Äquivalent zu s <= t
s.issuperset(t)	Äquivalent zu s >= t
s.union(t)	Äquivalent zu s | t
s.intersection(t)	Äquivalent zu s & t
s.difference(t)	Äquivalent zu s – t
s.symmetric_difference(t)	Äquivalent zu s ^ t
s.copy()	Erzeugt eine Kopie des Sets s

Tabelle 8.29 Methoden der Datentypen set und frozenset

s.copy()

Um eine Kopie eines Sets zu erzeugen, wird die Methode copy verwendet:

```
>>> m = s.copy()
>>> m
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> m is s
False
>>> m == s
True
```

Wichtig ist, dass nur das Set selbst kopiert wird. Bei den enthaltenen Elementen handelt es sich sowohl in der ursprünglichen Menge als auch in der Kopie um Referenzen auf dieselben Objekte. Dies ist Ihnen bereits aus Abschnitt 8.5.1, »Listen – list«, geläufig.

top

8.7.1 Mengen – set

Das set bietet, als Datentyp für veränderliche Mengen, einige Operatoren und Methoden, die über den eben besprochenen Grundbestand hinausgehen. Beachten Sie, dass alle hier eingeführten Operatoren und Methoden nicht für frozenset verfügbar sind. Zuallererst möchten wir darauf eingehen, dass auch hier wie selbstverständlich erweiterte Zuweisungen für die oben eingeführten Operatoren verwendet werden dürfen:

Operator	Entsprechung
s |= t	s = s | t
s &= t	s = s & t
s -= t	s = s – t
s ^= t	s = s ^ t

Tabelle 8.30 Operatoren des Datentyps set

Neben diesen neuen Operatoren stellt ein set fünf neue Methoden bereit, die es vom frozenset unterscheiden. Des Weiteren existieren auch für erweiterte Zuweisungsoperatoren Alternativen in Form einer Methode:

Methode	Beschreibung
s.update(t)	Äquivalent zu s |= t
s.intersection_update(t)	Äquivalent zu s &= t
s.difference_update(t)	Äquivalent zu s -= t

s.symmetric_difference_update(t)	Äquivalent zu s ^= t
s.add(e)	Fügt das Objekt e als Element in das Set s ein.
s.remove(e)	Löscht das Element e aus dem Set s. Sollte e nicht vorhanden sein, wird eine Exception erzeugt.
s.discard(e)	Löscht das Element e aus dem Set s. Sollte e nicht vorhanden sein, wird dies ignoriert.
s.clear()	Löscht alle Elemente des Sets s, jedoch nicht das Set selbst.

Tabelle 8.31 Methoden des Datentyps set

Diese Methoden möchten wir nachfolgend anhand eines Beispiels erläutern. Die Beispiele sind dabei in diesem Kontext zu sehen:

```
>>> s = set([1,2,3,4,5])
>>> s
set([1, 2, 3, 4, 5])
```

s.add(e)

Die Methode add fügt ein Element *e* in das Set s ein:

```
>>> s.add(6)
>>> s
set([1, 2, 3, 4, 5, 6])
```

Sollte *e* bereits im Set vorhanden sein, so wird dies ignoriert.

s.remove(e)

Die Methode remove löscht das Element *e* aus dem Set s:

```
>>> s.remove(5)
>>> s
set([1, 2, 3, 4, 6])
```

Sollte das zu löschende Element nicht im Set vorhanden sein, so wird eine Fehlermeldung erzeugt:

```
>>> s.remove(17)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 17
```

s.discard(e)

Die Methode discard löscht ein Element *e* aus dem Set s. Der einzige Unterschied zur Methode remove besteht darin, dass keine Fehlermeldung erzeugt wird, wenn *e* nicht in s vorhanden ist:

```
>>> s.discard(5)
>>> s
set([1, 2, 3, 4])
>>> s.discard(17)
>>> s
set([1, 2, 3, 4])
```

s.clear()

Die Methode clear entfernt alle Elemente aus dem Set s. Das Set selbst bleibt nach dem Aufruf von clear jedoch weiterhin vorhanden:

```
>>> s.clear()
>>> s
set([])
```

8.7.2 Unveränderliche Mengen – frozenset

Da es sich bei einem `frozenset` lediglich um eine Version des `set` handelt, die nach dem Erstellen nicht mehr verändert werden darf, wurden alle Operatoren und Methoden bereits im Rahmen der Grundfunktionalität zu Beginn des Kapitels erklärt. Beachten Sie, dass ein `frozenset` nicht nur selbst unveränderlich ist, sondern auch nur unveränderliche Elemente enthalten darf:

```
>>> frozenset([1, 2, 3, 4])
frozenset([1, 2, 3, 4])
>>> frozenset([[1, 2], [3, 4]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Welche Vorteile bietet nun das explizite Behandeln einer Menge als unveränderlich? Nun, neben gewissen Vorteilen in puncto Geschwindigkeit und Speichereffizienz kommt, wir erinnern uns, als Schlüssel eines Dictionarys nur ein unveränderliches Objekt in Frage. Innerhalb eines Dictionarys kann also ein `frozenset` sowohl als Schlüssel als auch als Wert verwendet werden. Das möchten wir im folgenden Beispiel veranschaulichen:

```
>>> d = {frozenset([1,2,3,4]) : "Hello World"}
>>> d
{frozenset([1, 2, 3, 4]): 'Hello World'}
```

Im Gegensatz dazu passiert Folgendes, wenn versucht wird, ein `set` als Schlüssel zu verwenden:

```
>>> d = {set([1,2,3,4]) : "Hello World"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: set objects are unhashable
```

Mit dem `set` haben wir den letzten Basisdatentyp behandelt. Freuen Sie sich nun darauf, das Gelernte anzuwenden. Im nächsten Kapitel werden wir über die verschiedenen Wege sprechen, wie ein Programm mit dem Benutzer interagieren kann.

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

9.2 Tastatureingaben

Sie erinnern sich sicherlich noch an unser kleines Beispielprogramm zum Einstieg in die Sprache Python, das Spiel »Zahlenraten«. Dort haben wir bereits eine Tastatureingabe des Benutzers verwendet. Dazu wurde folgender Code verwendet:

```
guess = input("Raten Sie: ")
```

Hier wird zunächst der String "Raten Sie:" auf dem Bildschirm ausgegeben und ein Eingabeprompt angezeigt. Jetzt wird der Programmablauf so lange angehalten, bis der Benutzer eine Eingabe getätigt und diese mit ↵ bestätigt hat. Anschließend wird der eingelesene Wert mit der Referenz guess verknüpft.

Bei der Verwendung von input müssen Sie vor allem eine Besonderheit beachten: Alle Eingaben des Benutzers werden zunächst als Python-Code interpretiert und können erst danach an eine Referenz gebunden werden. So würde guess hier nach der Eingabe von »2+2« keineswegs einen String mit dem Wert "2+2" referenzieren, sondern eine ganze Zahl mit dem Wert 4.

Diese Eigenschaft von input lässt sich in unserem Beispiel zum »Cheaten« verwenden, indem der Name der Referenz auf den zu ratenden Begriff eingegeben wird:

```
Raten Sie: secret
Super, Sie haben es in 1 Versuchen geschafft!
```

Was ist da genau passiert? Nun, dazu sollten wir uns zunächst die ersten paar Zeilen des Programms vergegenwärtigen:

```
secret = 1337
guess = 0
i = 0
while guess != secret:
    guess = input("Raten Sie: ")
    [...]
```

Wir erinnern uns, dass die Eingabe des Spielers bei Verwendung von input als Python-Code interpretiert wird. Sie könnten sich also vorstellen, dass input("Raten Sie: ") durch die Eingabe ersetzt wird. Nach der Eingabe des Spielers referenzieren sowohl guess als auch secret denselben Wert, nämlich die geheime Zahl 1337. Damit ist die Abbruchbedingung der Schleife sofort erfüllt und der Spieler hat das Spiel gewonnen, ohne die geheime Zahl erraten zu haben.

Das ist natürlich nicht im Sinne des Programmierers und kann, je nachdem in welchem Kontext ein Programm eingesetzt wird, als massive Sicherheitslücke betrachtet werden. In der Regel ist es daher nicht erwünscht, dass eine Eingabe des Benutzers interpretiert wird, sondern man möchte sie als String weiterverarbeiten können. Für diesen Zweck existiert in Python eine Alternative zu input namens raw_input:

```
guess = raw_input("Raten Sie: ")
```

Nach der Verwendung von raw_input referenziert guess die Eingabe als String. Sollte der Benutzer also, wie im eingangs erwähnten Beispiel, »2+2« eingeben, so referenziert guess einen String mit dem Wert "2+2". Allerdings ist es mit dem Ersetzen von input durch raw_input noch nicht ganz getan, denn es wird im weiteren Verlauf des Spiels erwartet, dass guess eine ganze Zahl und keinen String referenziert. Dazu müssen wir das Ergebnis von raw_input explizit in einen Wert des Datentyps int konvertieren:

```
guess = int(raw_input("Raten Sie: "))
```

Spätestens jetzt werden alle nicht numerischen Eingaben mit einer Fehlermeldung quittiert:

```
Raten Sie: abc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
```

Von den Verwendungsmöglichkeiten her sind input und raw_input gleich. Der Text, der vor der Eingabe auf dem Bildschirm ausgegeben wird, ist optional und kann ersatzlos gestrichen werden:

```
guess = raw_input()
```

Dies hat eine Eingabe ohne vorherigen Prompt zur Folge.

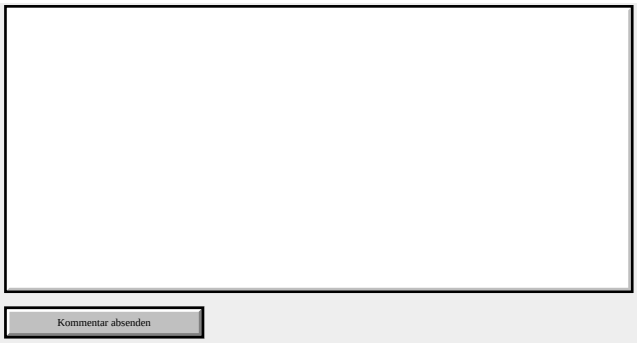
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

9.3 Dateien

Sie wissen jetzt, wie man Daten auf dem Bildschirm ausgeben und vom Benutzer einlesen kann. Fehlt noch die dritte Disziplin, die in diesem Kapitel behandelt werden soll: das Lesen und Schreiben von Dateien.

Dies sollte zum Standardrepertoire eines jeden Programmierers gehören – sei es, um Daten abzuspeichern, die später wieder verwendet werden sollen, oder um eine Logdatei zu führen, die den Programmablauf protokolliert.

Bevor das Lesen und Schreiben von Dateien in Python behandelt wird, müssen wir uns ganz allgemein mit Datenströmen befassen.

top

9.3.1 Datenströme

Unter einem *Datenstrom* (engl. *data stream*) versteht man eine kontinuierliche Folge von Daten. Dabei werden zwei Typen unterschieden: Von eingehenden Datenströmen (engl. *downstreams*) können Daten gelesen und in ausgehende Datenströme (engl. *upstreams*) geschrieben werden. Bildschirmausgaben, Tastatureingaben sowie Dateien und sogar Netzwerkverbindungen werden als Datenstrom betrachtet.

Es gibt zwei Standarddatenströme, die Sie, ohne es zu wissen, bereits verwendet haben. Sowohl die Ausgabe eines Strings auf dem Bildschirm als auch eine Benutzereingabe sind nichts anderes als Operationen auf den Standardein- bzw. -ausgabeströmen `stdin` und `stdout`.

Einige Betriebssysteme, darunter vor allem Windows, erlauben es, Datenströme im Text- und Binärmodus zu öffnen. Der Unterschied besteht darin, dass im Textmodus bestimmte binäre Steuerzeichen berücksichtigt werden. So wird ein im Textmodus geöffneter Strom beispielsweise nur bis zum ersten Auftreten des sogenannten EOF-Zeichens gelesen. Im Binärmodus hingegen wird

der vollständige Inhalt des Datenstroms eingelesen.

Als letzte Unterscheidung gibt es Datenströme, in denen man sich beliebig positionieren kann, und solche, in denen das nicht geht. Eine Datei stellt zum Beispiel einen Datenstrom dar, in dem die Schreib-/Leseposition beliebig festgelegt werden kann. Ein Beispiel für einen Datenstrom, in dem das nicht funktioniert, wäre der Standardeingabestrom (`stdin`) oder eine Netzwerkverbindung.

top

9.3.2 Daten aus einer Datei auslesen ▼▲

Wir beginnen damit, Daten aus einer Datei auszulesen. Dazu müssen wir lesend auf diese Datei zugreifen. Bei der Testdatei, die wir in diesem Beispiel verwenden werden, handelt es sich um ein Wörterbuch, das in jeder Zeile ein englisches Wort und, durch ein Leerzeichen davon getrennt, seine deutsche Übersetzung enthält. Die Datei soll *woerterbuch.txt* heißen:

```
Spain Spanien
Germany Deutschland
Sweden Schweden
France Frankreich
Italy Italien
```

Im Programm würden wir die Daten, die in dieser Datei stehen, gerne so aufbereiten, dass wir später in einem Dictionary bequem auf sie zugreifen können. Als kleine Zugabe werden wir das Programm noch dahingehend erweitern, dass der Benutzer das Programm nach der Übersetzung eines englischen Begriffes fragen kann.

Zunächst einmal muss die Datei zum Lesen geöffnet werden. Dazu wird die Built-in Function open verwendet. Diese gibt das in Abschnitt 9.1 bereits angesprochene *Dateiobjekt* zurück:

```
fobj = open("woerterbuch.txt", "r")
```

Nachdem open aufgerufen wurde, können mit dem Dateiobjekt Daten aus der Datei gelesen werden. Nachdem das Lesen der Datei beendet worden ist, muss sie explizit durch Aufrufen der Methode `close` geschlossen werden:

```
fobj.close()
```

Als erster Parameter von open wird ein String übergeben, der den Dateinamen enthält. Beachten Sie, dass hier sowohl relative als auch absolute Dateinamen erlaubt sind. In diesem Fall handelt es sich um einen relativen Dateinamen, die Datei muss sich also im selben Verzeichnis wie das Programm befinden. Der zweite Parameter ist ebenfalls ein String und spezifiziert den Modus, in dem die Datei geöffnet werden soll, wobei `"r"` für »read« steht und bedeutet, dass die Datei zum Lesen geöffnet wird. Das von der Funktion zurückgegebene Dateiobjekt wird mit der Referenz `fobj` verknüpft. Sollte die Datei nicht vorhanden sein, wird ein `IOError` erzeugt:

```
Traceback (most recent call last):
  File "woerterbuch.py", line 1, in <module>
    fobj = open("woerterbuch.txt", "r")
IOError: [Errno 2] No such file or directory:
'woerterbuch.txt'
```

Wenn ein Dateiobjekt nicht mehr benötigt wird, muss es durch Aufruf der Methode `close` geschlossen werden. Nach Aufruf dieser Methode können keine weiteren Daten mehr gelesen werden.

Im nächsten Schritt möchten wir die Datei zeilenweise auslesen. Dies ist relativ einfach, da das Dateiobjekt zeilenweise iterierbar ist. Wir können also die altbekannte `for`-Schleife verwenden:

```
fobj = open("woerterbuch.txt", "r")
for line in fobj:
    print line
fobj.close()
```

In der for-Schleife iterieren wir zeilenweise über das Dateiobjekt, wobei line jeweils den Inhalt der aktuellen Zeile referenziert. Momentan wird jede Zeile im Schleifenkörper lediglich ausgegeben. Wir möchten jedoch im Programm ein Dictionary aufbauen, das nach dem Einlesen der Datei die englischen Begriffe als Schlüssel und den jeweiligen deutschen Begriff als Wert enthält.

Dazu legen wir zunächst ein leeres Dictionary an:

```
woerter = {}
```

Dann wird die Datei *woerterbuch.txt* zum Lesen geöffnet und in einer Schleife über alle Zeilen der Datei iteriert:

```
fobj = open("woerterbuch.txt", "r")
for line in fobj:
    zuordnung = line.split(" ")
    woerter[zuordnung[0]] = zuordnung[1]
fobj.close()
```

Im Schleifenkörper verwenden wir nun die Methode split eines Strings, um die aktuell eingelesene Zeile in zwei Teile einer Liste aufzubrechen: in den Teil links vom Leerzeichen, also das englische Wort, und in den Teil rechts vom Leerzeichen, also das deutsche Wort. In der nächsten Zeile des Schleifenkörpers wird dann ein neuer Eintrag im Dictionary angelegt, mit dem Schlüssel zuordnung[0] (dem englischen Wort) und dem Wert zuordnung[1] (dem deutschen Wort).

Verändern Sie einmal den obigen Code dahingehend, dass nach dem Schließen des Dateiobjekts das erzeugte Dictionary mittels print ausgegeben wird. Diese Ausgabe wird etwa so aussehen:

```
{'Italy': 'Italien', 'Sweden': 'Schweden\n', 'Germany':
'Deutschland\n', 'Spain': 'Spanien\n', 'France':
'Frankreich\n'}
```

Sie sehen, dass hinter jedem Wert ein \n, also die Escape-Sequenz für einen Zeilenumbruch, steht. Das liegt daran, dass ein Zeilenumbruch in Python als Buchstabe und damit als Teil des Dateiinhaltes angesehen wird. Deswegen wird jede Zeile einer Datei vollständig, also inklusive eines möglichen Zeilenumbruchs am Ende, eingelesen. Der Zeilenumbruch wird natürlich nur eingelesen, wenn er wirklich vorhanden ist. Das bedeutet, dass die letzte Zeile (in diesem Fall Italy Italien) ohne Zeilenumbruch am Ende eingelesen wird.

Den Zeilenumbruch möchten wir im endgültigen Dictionary nicht wiederfinden. Aus diesem Grund rufen wir in jedem Schleifendurchlauf die strip-Methode des Strings line auf. Diese entfernt alle Whitespace-Zeichen, unter anderem also einen Zeilenumbruch, am Anfang und Ende des Strings.

```
woerter = {}
fobj = open("woerterbuch.txt", "r")
for line in fobj:
    line = line.strip()
    zuordnung = line.split(" ")
    woerter[zuordnung[0]] = zuordnung[1]
fobj.close()
```

Damit ist der Inhalt der Datei vollständig in ein Dictionary überführt worden. Als kleine Zugabe haben wir uns vorgenommen, es dem Benutzer zu ermöglichen, Anfragen an das Programm zu senden. Im Ablaufprotokoll soll das folgendermaßen aussehen:

```
Geben Sie ein Wort ein: Germany
Das deutsche Wort lautet: Deutschland
Geben Sie ein Wort ein: Italy
Das deutsche Wort lautet: Italien
Geben Sie ein Wort ein: Greece
Das Wort ist unbekannt
```

Im Programm lesen wir in einer Endlosschleife Anfragen vom Benutzer ein. Mit dem `in`-Operator prüfen wir, ob das eingelesene Wort als Schlüssel im Dictionary vorhanden ist. Ist das der Fall, so wird die entsprechende deutsche Übersetzung ausgegeben. Sollte das eingegebene Wort nicht vorhanden sein, so wird eine Fehlermeldung ausgegeben.

```
woerter = {}

fobj = open("woerterbuch.txt", "r")
for line in fobj:
    line = line.strip()
    zuordnung = line.split(" ")
    woerter[zuordnung[0]] = zuordnung[1]
fobj.close()

while True:
    wort = raw_input("Geben Sie ein Wort ein: ")
    if wort in woerter:
        print "Das deutsche Wort lautet:", woerter[wort]
    else:
        print "Das Wort ist unbekannt"
```

Das hier vorgestellte Beispielprogramm ist weit davon entfernt, perfekt zu sein, jedoch zeigt es sehr schön, wie Dateiobjekte und auch Dictionarys sinnvoll eingesetzt werden können. Fühlen Sie sich dazu ermutigt, das Programm zu erweitern. Sie könnten es dem Benutzer beispielsweise ermöglichen, das Programm zu beenden, Übersetzungen in beide Richtungen anbieten oder das Verwenden mehrerer Quelldateien erlauben.

top

9.3.3 Daten in eine Datei schreiben ▼▲

Im letzten Kapitel haben wir uns dem Lesen von Dateien gewidmet. Dass es auch andersherum geht, soll in diesem Kapitel das Thema sein. Um eine Datei zum Schreiben zu öffnen, wird ebenfalls die Built-in Function open verwendet. Sie erinnern sich, dass diese Funktion einen Modus als zweiten Parameter erwartet, der im letzten Abschnitt `"r"` für »read« sein musste. Analog dazu muss `"w"` (für »write«) angegeben werden, wenn die Datei zum Schreiben geöffnet werden soll. Sollte die gewünschte Datei bereits vorhanden sein, so wird sie geleert. Nicht vorhandene Dateien werden erstellt.

```
fobj = open("ausgabe.txt", "w")
```

Nachdem alle Daten in die Datei geschrieben wurden, muss das Dateiobjekt durch Aufruf der Methode `close` geschlossen werden.

```
fobj.close()
```

Das Schreiben eines Strings in die geöffnete Datei erfolgt durch Aufruf der Methode `write` des Dateiobjekts. Das nachfolgende Beispielprogramm versteht sich als Gegenstück zu dem im vorherigen Kapitel. Wir gehen davon aus, dass `woerter` ein Dictionary referenziert, das englische Begriffe als Schlüssel und die deutschen Übersetzungen als Werte enthält. Es handelt sich also genau um das Dictionary, das von dem Beispielprogramm des letzten Kapitels erzeugt wurde.

```
fobj = open("ausgabe.txt", "w")
for engl in woerter:
    fobj.write(engl + " " + woerter[engl] + "\n")
fobj.close()
```

Zunächst wird eine Datei namens *ausgabe.txt* zum Schreiben geöffnet. Danach werden alle Schlüssel des Dictionarys `woerter` durchlaufen. In jedem Schleifendurchlauf wird mittels `fobj.write` ein entsprechend formatierter String in die Datei geschrieben. Beachten Sie zum einen, dass beim Schreiben einer Datei explizit durch Ausgabe eines `\n` in eine neue Zeile gesprungen werden muss, und zum anderen, dass die Initialisierung des Dictionarys `woerter` im Codebeispiel fehlt.

Die von diesem Beispiel geschriebene Datei kann problemlos durch das Beispielprogramm aus dem letzten Abschnitt wieder eingelesen werden.

In Abschnitt 9.1, »Bildschirmausgaben«, wurde gesagt, dass sich Strings auch mittels `print` in eine Datei schreiben lassen. Das ist korrekt. Das obige Beispiel lässt sich durch diese Notation folgendermaßen formulieren:

```
fobj = open("ausgabe.txt", "w")
for engl in woerter:
    print >> fobj, engl, woerter[engl]
fobj.close()
```

Beachten Sie, dass nach einer Ausgabe mit `print` automatisch in eine neue Zeile gesprungen sowie für jedes Komma ein Leerzeichen in der Ausgabe ergänzt wird.

Hinweis

Um Sonderzeichen innerhalb einer Textdatei verwenden zu können, wird die Datei, wie Sie es bereits von Sonderzeichen in Strings her kennen, in einer bestimmten Kodierung gespeichert.

Um solche kodiert gespeicherten Dateien komfortabel lesen oder schreiben zu können, verwendet man das Modul codecs der Standardbibliothek, das in Abschnitt 27.1 beschrieben wird.

top

9.3.4 Verwendung des Dateiobjekts ▲

Das Dateiobjekt besitzt, wie beispielsweise die komplexeren Datentypen auch, Methoden und Attribute. Einige von ihnen haben wir in den beiden vorherigen Unterabschnitten bereits besprochen. Wir möchten auf das Dateiobjekt bezogene Attribute, Methoden und Built-in Functions noch einmal tabellarisch erklären. Die Bedeutung ist in der Tabelle jeweils zusammenfassend dargestellt. Einige der Methoden werden im Anschluss an die Tabelle ausführlich erklärt.

Built-in Function	Beschreibung
`open(filename[, mode[, bufsize]])`	Öffnet eine Datei im gewünschten Modus und gibt das Dateiobjekt zurück. Beachten Sie, dass es sich um eine Built-in Function handelt und nicht um eine Methode des Dateiobjekts `f`. Eine ausführliche Beschreibung von open finden Sie am Ende des Kapitels.

Tabelle 9.1 Built-in Functions für Dateiobjekte

In der nun folgenden Tabelle sei `f` stets ein mit open erfolgreich erzeugtes Dateiobjekt. [In diesem Zusammenhang bedeutet »ungefähr«, dass die Anzahl der zu lesenden Bytes möglicherweise zu einer internen Puffergröße aufgerundet wird.]

Methode	Beschreibung
`f.close()`	Schließt ein bestehendes Dateiobjekt. Beachten Sie, dass danach keine Lese- oder Schreiboperationen mehr durchgeführt werden dürfen.
`f.flush()`	Verfügt, dass anstehende Schreiboperationen sofort ausgeführt werden.
`f.fileno()`	Gibt den Deskriptor der geöffneten Datei als ganze Zahl zurück.
`f.isatty()`	`True`, wenn das Dateiobjekt auf einem Datenstrom geöffnet wurde, der nicht an beliebiger Stelle geschrieben oder gelesen werden kann.
`f.next()`	Liest die nächste Zeile der Datei ein und gibt diese als String zurück.
`f.read([size])`	Liest `size` Bytes der Datei ein, oder weniger, wenn vorher das Ende der Datei erreicht wurde. Sollte `size` nicht angegeben sein, so wird die Datei vollständig eingelesen. Die Daten werden als String zurückgegeben.
`f.readline([size])`	Liest eine Zeile der Datei ein. Durch Angabe von `size` lässt sich die Anzahl der zu lesenden Bytes begrenzen.[1]
`f.readlines([sizehint])`	Liest alle Zeilen und gibt sie in Form einer Liste von Strings zurück. Sollte `sizehint` angegeben sein, so wird nur gelesen, bis ungefähr `sizehint` Bytes gelesen wurden.
`f.seek(offset[, whence])`	Setzt die aktuelle Schreib-/Leseposition in der Datei auf `offset`. Eine ausführliche Beschreibung von `f.seek` finden Sie am Ende des Kapitels.
`f.tell()`	Liefert die aktuelle Schreib-/Leseposition in der Datei.
`f.truncate([size])`	Löscht in der Datei alle Daten, die hinter der aktuellen Schreib-/Leseposition bzw. – sofern angegeben – hinter `size` stehen.
`f.write(str)`	Schreibt den String `str` in die Datei.
`f.writelines(sequence)`	Schreibt mehrere Zeilen in die Datei. `sequence` muss eine Liste von Strings sein.

Tabelle 9.2 Methoden eines Dateiobjekts

Attribut	Beschreibung
`f.closed`	`True`, wenn die Datei geschlossen ist, andernfalls `False`.
`f.encoding`	Enthält das Encoding, das genutzt wird, um eine Datei im Textmodus zu schreiben. Ein Wert von `None` bedeutet, dass der Systemdefault verwendet wird.
`f.mode`	Enthält den Modus, der beim Öffnen der Datei angegeben wurde.
`f.name`	Enthält den Namen der geöffneten Datei.

`f.newlines`	Dieses Attribut enthält alle Typen von Newline-Zeichen, die bisher vorgekommen sind, da diese von System zu System sehr verschieden sind.

Tabelle 9.3 Attribute eines Dateiobjekts

Viele der oben beschriebenen Methoden sind durch vorangegangene Beispiele oder den erklärenden Text ausreichend beschrieben. Wir möchten uns trotzdem noch einmal eingehend mit der Built-in Function open sowie mit der Methode seek befassen.

open(filename[, mode[, bufsize]])

Die Built-in Function open öffnet eine Datei und gibt das erzeugte Dateiobjekt zurück. Mithilfe dieses Dateiobjekts können nachher die gewünschten Operationen an der Datei durchgeführt werden.

Die Funktion open erwartet drei Parameter. Zunächst einmal muss der Dateiname bzw. der Pfad zur zu öffnenden Datei angegeben werden (*filename*). Dann wird der Modus (*mode*) spezifiziert, in dem die Datei zu öffnen ist. Dieser Parameter muss ein String sein, wobei alle gültigen Werte und ihre Bedeutung in der folgenden Tabelle aufgelistet sind:

Modus	Beschreibung
`"r"`	Die Datei wird ausschließlich zum Lesen geöffnet (»r« für »read«).
`"w"`	Die Datei wird ausschließlich zum Schreiben geöffnet. Eine eventuell bestehende Datei gleichen Namens wird überschrieben (»w« steht für »write«).
`"a"`	Die Datei wird ausschließlich zum Schreiben geöffnet. Eine eventuell bestehende Datei gleichen Namens wird nicht überschrieben, sondern erweitert (»a« steht für »append«).
`"r+"`, `"w+"`, `"a+"`	Die Datei wird zum Lesen und Schreiben geöffnet. Beachten Sie, dass `"w+"` eine eventuell bestehende Datei gleichen Namens leert.
`"rb"`, `"wb"`, `"ab"`, `"r+b"`, `"w+b"`, `"a+b"`	Die Datei wird im Binärmodus geöffnet. Beachten Sie, dass Strings in Python auch binäre Daten erfassen können (»b« steht für »binary«).

Tabelle 9.4 Dateimodi

Der Parameter *mode* ist optional und wird als `"r"` angenommen, wenn er weggelassen wird.

Als letzter, ebenfalls optionaler Parameter kann der Puffermodus bzw. eine Puffergröße übergeben werden. Dabei bedeutet ein Wert von `0`, dass keine Pufferung verwendet werden soll, und `1` heißt, dass zeilenweise gepuffert werden soll. Jede ganze Zahl größer als `1` wird als Richtwert für die gewünschte Puffergröße betrachtet. Eine negative Zahl veranlasst dazu, den Systemdefault zu verwenden.

f.seek(offset[, whence])

Setzt die Schreib-/Leseposition innerhalb der Datei. Beachten Sie, dass diese Methode je nach Modus, in dem die Datei geöffnet wurde, keine Auswirkung hat (Modus `"a"`) oder dass die Schreibposition vor der nächsten Ausgabe zurückgesetzt werden kann (Modus `"a+"`). Sollte die Datei im Binärmodus geöffnet worden sein, wird der Parameter *offset* in Bytes vom Dateianfang aus gezählt. Diese Interpretation von *offset* lässt sich durch den optionalen Parameter *whence* beeinflussen:

Wert von whence	Interpretation von offset
0	Anzahl Bytes relativ zum Dateianfang
1	Anzahl Bytes relativ zur aktuellen Schreib-/Leseposition
2	Anzahl Bytes relativ zum Dateiende

Tabelle 9.5 Der Parameter whence

Beachten Sie, dass Sie seek nicht so unbeschwert verwenden können, wenn die Datei im Textmodus geöffnet wurde. Hier sollten als `offset` nur Rückgabewerte der Methode `tell` verwendet werden. Abweichende Werte können zu undefiniertem Verhalten führen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

10.2 Funktionsparameter ▼

Wir haben bereits oberflächlich besprochen, was Funktionsparameter sind und wie sie verwendet werden können, doch das ist bei Weitem noch nicht die ganze Wahrheit. In diesem Abschnitt sollen drei Techniken eingeführt werden, die die Verwendung von Funktionsparametern bequemer oder eleganter machen. Alle drei Techniken sind mehr oder weniger speziell und somit nicht für alle Einsatzgebiete von Funktionen geeignet.

top

10.2.1 Optionale Parameter ▼▲

Zu Beginn dieses Kapitels wurde die Verwendung einer Funktion anhand der Built-in Function `range` erklärt. Erinnern Sie sich noch daran, als `range` im Zusammenhang mit der `for`-Schleife eingeführt wurde? Wenn ja, dann wissen Sie sicherlich noch, dass unter anderem der letzte der drei Parameter optional war. Das bedeutet zunächst einmal, dass dieser Parameter beim Funktionsaufruf weggelassen werden kann. Ein optionaler Parameter muss funktionsintern mit einem Wert vorbelegt sein, üblicherweise einen Standardwert, der in einem Großteil der Funktionsaufrufe ausreichend ist. Bei der Funktion `range` regelt der dritte Parameter die Schrittweite und ist mit `1` vorbelegt. Folgende Aufrufe von `range` sind also äquivalent:

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

Dies ist ein interessantes Sprachmerkmal von Python, denn oftmals hat eine Funktion ein Standardverhalten, das sich durch zusätzliche Parameter an spezielle Gegebenheiten anpassen lassen

soll. In den überwiegenden Fällen, in denen das Standardverhalten jedoch genügt, wäre es umständlich, trotzdem die für diesen Aufruf völlig überflüssigen Parameter anzugeben. Deswegen sind vordefinierte Parameterwerte oft eine sinnvolle Ergänzung der eigenen Funktionsschnittstelle.

Um einen Funktionsparameter mit einem Defaultwert vorzubelegen, wird dieser Wert bei der Funktionsdefinition zusammen mit einem Gleichheitszeichen hinter den Parameternamen geschrieben. Die folgende Funktion soll, je nach Anwendung, die Summe von zwei, drei oder vier ganzen Zahlen berechnen und das Ergebnis zurückgeben. Dabei soll der Programmierer beim Aufruf der Funktion nur so viele Zahlen angeben müssen, wie er benötigt:

```
def summe(a, b, c=0, d=0):
    return a + b + c + d
```

Um eine Addition durchführen zu können, müssen mindestens zwei Parameter übergeben worden sein. Die anderen beiden werden mit `0` vorbelegt. Sollten sie beim Funktionsaufruf nicht explizit angegeben werden, so fließen sie in die Addition nicht ein. Die Funktion könnte folgendermaßen aufgerufen werden:

```
summe(1, 2)
summe(1, 2, 3)
summe(1, 2, 3, 4)
```

Beachten Sie, dass optionale Parameter nur am Ende einer Funktionsschnittstelle stehen dürfen. Das heißt, dass auf einen optionalen kein nicht-optionaler Parameter mehr folgen darf. Diese Einschränkung ist wichtig, damit alle angegebenen Parameter eindeutig zuzuordnen sind.

top

10.2.2 Schlüsselwortparameter ▼▲

Neben den bislang verwendeten sogenannten *positional arguments* gibt es in Python eine weitere Möglichkeit, Parameter zu übergeben. Solche Parameter werden *keyword arguments* genannt. Es handelt sich dabei lediglich um eine weitere Technik, Parameter beim Funktionsaufruf zu übergeben. An der Funktionsdefinition ändert sich nichts. Betrachten wir dazu unsere Summenfunktion, die wir im vorangegangenen Abschnitt geschrieben haben:

```
def summe(a, b, c=0, d=0):
    return a + b + c + d
```

Diese Funktion kann auch folgendermaßen aufgerufen werden:

```
summe(d=1, b=3, c=2, a=1)
```

Dazu werden im Funktionsaufruf die Parameter, wie bei einer Zuweisung, auf den gewünschten Wert gesetzt. Da bei der Übergabe der jeweilige Parametername angegeben werden muss, ist die Zuordnung unter allen Umständen eindeutig. Das erlaubt es dem Programmierer, Schlüsselwortparameter in beliebiger Reihenfolge anzugeben.

Es ist möglich, beide Formen der Parameterübergabe zu kombinieren. Dabei ist zu beachten, dass keine *positional arguments* auf *keyword arguments* folgen dürfen, Letztere also immer am Ende des Funktionsaufrufs stehen müssen.

```
summe(1, 2, c=10, d=11)
```

Beachten Sie außerdem, dass nur solche Parameter als *keyword arguments* übergeben werden dürfen, die im selben Funktionsaufruf nicht bereits als *positional argument* übergeben wurden.

Zum Schluss möchten wir noch anmerken, dass optionale Parameter auch unter Verwendung von *keyword arguments* wie erwartet funktionieren.

top

10.2.3 Beliebige Anzahl von Parametern ▼▲

Für beide Formen der Parameterübergabe (*positional* und *keyword*) gibt es eine Notation, die es einer Funktion ermöglicht, beliebig viele Parameter entgegenzunehmen. Bleiben wir zunächst einmal bei den *positional arguments*. Betrachten Sie dazu folgende Funktionsdefinition:

```
def funktion(a, b, *weitere):
    print "Feste Parameter:", a, b
    print "Weitere Parameter:", weitere
```

Zunächst einmal werden ganz klassisch zwei Parameter a und b festgelegt und zusätzlich ein dritter namens `weitere`. Wichtig ist der Stern vor seinem Namen. Bei einem Aufruf dieser Funktion würden a und b, wie Sie das bereits kennen, die ersten beiden übergebenen Instanzen referenzieren. Interessant ist, dass `weitere` fortan ein Tupel referenziert, das alle zusätzlich übergebenen Instanzen enthält. Anschaulich wird dies, wenn wir folgende Funktionsaufrufe betrachten:

```
funktion(1, 2)
funktion(1, 2, "Hallo Welt", 42, [1,2,3,4])
```

Die Ausgabe der Funktion im Falle des ersten Aufrufs wäre:

```
Feste Parameter: 1 2
Weitere Parameter: ()
```

Der Parameter `weitere` referenziert also ein leeres Tupel. Im Falle des zweiten Aufrufs sähe die Ausgabe folgendermaßen aus:

```
Feste Parameter: 1 2
Weitere Parameter: ('Hallo Welt', 42, [1, 2, 3, 4])
```

Der Parameter `weitere` referenziert nun ein Tupel, in dem alle über a und b hinausgehenden Instanzen in der Reihenfolge enthalten sind, wie sie übergeben wurden.

Diese Art, einer Funktion das Entgegennehmen beliebig vieler Parameter zu ermöglichen, funktioniert ebenso für *keyword arguments*. Der Unterschied besteht darin, dass der Parameter, der alle weiteren Instanzen enthalten soll, in der Funktionsdefinition mit zwei Sternen geschrieben werden muss, sowie darin, dass er später kein Tupel, sondern ein Dictionary referenziert. Dieses Dictionary enthält den jeweiligen Parameternamen als Schlüssel und die übergebene Instanz als Wert. Betrachten Sie dazu folgende Funktionsdefinition:

```
def funktion(a, b, **weitere):
    print "Feste Parameter:", a, b
    print "Weitere Parameter:", weitere
```

und diese beiden dazu passenden Funktionsaufrufe:

```
funktion(1, 2)
funktion(1, 2, johannes="ernesti", peter="kaiser")
```

Die Ausgabe nach dem ersten Funktionsaufruf sähe folgendermaßen aus:

```
Feste Parameter: 1 2
Weitere Parameter: {}
```

Der Parameter `weitere` referenziert also ein leeres Dictionary. Nach dem zweiten Aufruf sähe die Ausgabe so aus:

```
Feste Parameter: 1 2
Weitere Parameter: {'johannes': 'ernesti', 'peter':
'kaiser'}
```

Beide Techniken können zusammen verwendet werden, wie folgende Funktionsdefinition zeigt:

```
def funktion(*positional, **keyword):
    print positional
    print keyword
```

Der Funktionsaufruf

```
funktion(1, 2, 3, 4, hallo="welt", key="word")
```

gibt diese Werte aus:

```
(1, 2, 3, 4)
{'hallo': 'welt', 'key': 'word'}
```

Sie sehen, dass `positional` ein Tupel mit allen Positions- und `keyword` ein Dictionary mit allen Schlüsselwort-Parametern referenziert.

Entpacken einer Parameterliste

Wenn eine Funktion beliebige Parameter erwartet, kommen diese funktionsintern gesammelt entweder in Form eines Tupels (*positional arguments*) oder eines Dictionarys (*keyword arguments*) an. Gelegentlich möchte man die in diesem Tupel bzw. Dictionary enthaltenen Parameter an eine andere Funktion weiterreichen. Dabei soll aber jedes Element des Tupels bzw. jedes Schlüssel/Wert-Paar des Dictionarys beim Aufruf der zweiten Funktion als eigenständiger Parameter übergeben werden. Dieser Vorgang wird *Entpacken* eines Tupels oder eines Dictionarys genannt.

Das Entpacken eines Tupels soll an einem Beispiel verdeutlicht werden. Dazu definieren wir zwei Funktionen, `f1` und `f2`, wobei `f1` über eine feste Schnittstelle verfügt, während `f2` beliebig viele *positional arguments* akzeptiert. Die Funktion `f2` soll die ihr übergebenen Parameter, auf die sie in Form eines Tupels zugreifen kann, entpacken und an die Funktion `f1` weiterreichen.

```
def f1(a, b, c, d):
    print "Parameter:", a, b, c, d

def f2(*prm):
    f1(*prm)
```

Zur Funktion `f1` muss nicht viel gesagt werden: Sie erwartet vier Parameter und gibt diese auf dem Bildschirm aus. Viel interessanter ist die Funktion `f2`, die eine beliebige Anzahl Positionsparameter erwartet und diese im Funktionskörper an die Funktion `f1` weiterreichen soll. Das Tupel `prm`, das die der Funktion `f2` übergebenen Parameter enthält, kann im Funktionsaufruf von `f1` durch ein vorangestelltes Sternchen (`*`) entpackt werden. Wenn das Tupel vier Elemente enthält, kommen diese in Form der Parameter `a`, `b`, `c` und `d` bei `f1` an. Sollte das

Tupel weniger oder mehr Elemente enthalten, verursacht dies einen Fehler. So gibt `f1` bei einem Funktionsaufruf von

```
f2(1, 2, 3, 4)
```

den Text

```
Parameter: 1 2 3 4
```

auf dem Bildschirm aus.

Analog dazu kann ein Dictionary mit zwei vorangestellten Sternchen entpackt werden:

```
def f1(a, b, c, d):
    print "Parameter:", a, b, c, d

def f2(**prm):
    f1(**prm)
```

In diesem Fall verursacht der Funktionsaufruf

```
f2(a=5, b=6, c=7, d=8)
```

die erwartete Bildschirmausgabe:

```
Parameter: 5 6 7 8
```

Beachten Sie allgemein, dass die hier vorgestellte Syntax nur innerhalb eines Funktionsaufrufs verwendet werden darf und außerhalb dessen zu einem Fehler führt.

10.2.4 Seiteneffekte ▲

Bisher haben wir die Thematik der Seiteneffekte geschickt umschifft, doch Sie sollten immer im Hinterkopf behalten, dass sogenannte *Seiteneffekte* (engl. *side effects*) immer dann auftreten können, wenn eine Instanz eines mutable Datentyps, also zum Beispiel einer Liste oder eines Dictionarys, als Funktionsparameter übergeben wird.

Um dies verstehen zu können, müssen wir zunächst allgemein darüber sprechen, auf welchen Wegen Funktionsparameter übergeben werden. In der Programmierung unterscheidet man dabei grob zwei Arten:

Bei einem *call-by-value* wird funktionsintern mit Kopien der als Parameter übergebenen Instanzen gearbeitet. Das hat den Vorteil, dass eine Funktion keine ungewollten Änderungen im Hauptprogramm bewirken kann, erzeugt jedoch unter Umständen einen erheblichen Overhead, da auch größere Instanzen wie Listen oder Dictionarys bei jedem Funktionsaufruf kopiert werden müssten.

Das gegensätzliche Prinzip wird *call-by-reference* genannt und bedeutet, dass funktionsintern mit Referenzen auf die im Hauptprogramm befindlichen Instanzen gearbeitet wird. Der Vorteil dieser Methode liegt auf der Hand: Es müssen keine Instanzen kopiert werden, und ein Funktionsaufruf wird dementsprechend performant. Der größte Nachteil der Referenzparameter ist, dass innerhalb einer Funktion eine übergebene Instanz so verändert werden kann, dass sich dies auch im Hauptprogramm auswirkt. Solche Änderungen sind vom Programmierer meist nicht erwünscht und werden als Seiteneffekte bezeichnet.

In Python werden Funktionsparameter grundsätzlich »by

reference« übergeben. Betrachten Sie dazu folgendes Beispiel, das sich zunächst auf unveränderliche Datentypen wie `int` oder `float` beschränkt:

```
>>> def f(a, b):
...     print id(a)
...     print id(b)
...
>>> p = 1
>>> q = 2
>>> id(p)
134537016
>>> id(q)
134537004
>>> f(p, q)
134537016
134537004
```

Im interaktiven Modus definieren wir zuerst eine Funktion, die zwei Parameter `a` und `b` erwartet und ihre jeweilige Identität ausgibt. Nachfolgend werden zwei Referenzen `p` und `q` angelegt, die je eine Instanz des Datentyps `int` referenzieren. Dann lassen wir uns die Identitäten der beiden Referenzen ausgeben und rufen die angelegte Funktion `f` auf. Sie sehen, dass die ausgegebenen Identitäten gleich sind. Es handelt sich also sowohl bei `p` und `q` als auch bei `a` und `b` im Funktionskörper um Referenzen auf dieselben Instanzen.

Trotzdem ist die Verwendung eines immutable Datentyps grundsätzlich frei von Seiteneffekten, da dieser bei Veränderung automatisch kopiert wird und alte Referenzen davon nicht berührt werden. Sollten wir also beispielsweise `a` im Funktionskörper um eins erhöhen, so werden nachher `a` und `p` verschiedene Instanzen referenzieren. Dies ermöglicht es uns, mit unveränderlichen Parametern umzugehen, als wären sie »by value« übergeben worden.

Diese Sicherheit können uns mutable Datentypen nicht geben. Dazu folgendes Beispiel:

```
def f(liste):
    liste[0] = 42
    liste += [5,6,7,8,9]

zahlen = [1,2,3,4]

print zahlen
f(zahlen)
print zahlen
```

Zunächst wird eine Funktion definiert, die eine Liste als Parameter erwartet und diese im Funktionskörper verändert. Im Hauptprogramm wird eine Liste angelegt und ausgegeben. Danach wird die Funktion aufgerufen und die Liste erneut ausgegeben. Die Ausgabe des Beispiels sieht folgendermaßen aus:

```
[1, 2, 3, 4]
[42, 2, 3, 4, 5, 6, 7, 8, 9]
```

Es ist zu erkennen, dass sich die Änderungen nicht allein auf den Kontext der Funktion beschränken, sondern sich auch im Hauptprogramm auswirken. Wenn eine Funktion nicht nur lesend auf eine Instanz eines veränderlichen Datentyps zugreifen muss und Seiteneffekte nicht ausdrücklich erwünscht sind, sollten Sie innerhalb der Funktion oder bei der Parameterübergabe eine Kopie der Instanz erzeugen. Das könnte in Bezug auf das obige Beispiel so aussehen: [Sie erinnern sich, dass beim Slicen einer Liste stets eine Kopie derselben erzeugt wird. Im Beispiel wurde das Slicing genutzt, um eine vollständige Kopie der Liste zu erzeugen, indem weder ein Start- noch ein Endindex angegeben wurde.]

```
f(zahlen[:])
```

Neben den bisher besprochenen Referenzparametern existiert eine

weitere, seltenere Form von Seiteneffekten, die auftritt, wenn ein veränderlicher Datentyp als Defaultwert eines Parameters verwendet wird:

```
>>> def f(a=[1,2,3]):
...     a += [4,5]
...     print a
...
>>> f()
[1, 2, 3, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5, 4, 5]
```

Wir definieren im interaktiven Modus eine Funktion, die einen einzigen Parameter erwartet, der mit einer Liste vorbelegt ist. Im Funktionskörper wird diese Liste um zwei Elemente vergrößert und ausgegeben. Nach mehrmaligem Aufrufen der Funktion ist zu erkennen, dass es sich bei dem Defaultwert augenscheinlich immer um dieselbe Instanz gehandelt hat.

Das liegt daran, dass eine Instanz, die als Defaultwert genutzt wird, nur einmalig und nicht bei jedem Funktionsaufruf neu erzeugt wird. Grundsätzlich sollten Sie also darauf verzichten, Instanzen unveränderlicher Datentypen als Defaultwert zu verwenden. Schreiben Sie Ihre Funktionen stattdessen folgendermaßen:

```
def f(a=None):
    if a is None:
        a = [1,2,3]
```

Selbstverständlich können Sie statt None eine Instanz eines beliebigen anderen immutable Datentypen verwenden, ohne dass Seiteneffekte auftreten.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Verarbeitung in elektronischen Systemen.

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

10.3 Zugriff auf globale Variablen

Bisher wurde ein Funktionskörper als abgekapselter Bereich betrachtet, der ausschließlich über Parameter bzw. den Rückgabewert Informationen mit dem Hauptprogramm austauschen kann. Das ist zunächst auch gar keine schlechte Sichtweise, denn so hält man seine Schnittstelle »sauber«.

Zunächst einmal müssen zwei Begriffe unterschieden werden. Wenn wir uns im Kontext einer Funktion, also im Funktionskörper befinden, dann können dort selbstverständlich Referenzen und Instanzen erzeugt und verwendet werden. Diese haben jedoch nur unmittelbar in der Funktion selbst Gültigkeit. Sie existieren im sogenannten *lokalen Namensraum*. Im Gegensatz dazu existieren Referenzen des Hauptprogramms im *globalen Namensraum*. Begrifflich wird auch zwischen *globalen Referenzen* und *lokalen Referenzen* unterschieden. Dazu folgendes Beispiel:

```
def f():
    a = "lokaler String"
b = "globaler String"
```

Wie stark zwischen globalem und lokalem Namensraum unterschieden wird, zeigt das folgende Beispiel:

```
def f(a):
    print a

a = 10
f(100)
```

In diesem Beispiel existiert sowohl im globalen als auch im lokalen Namensraum eine Referenz namens a. Im globalen Namensraum referenziert sie die ganze Zahl 10 und im lokalen Namensraum der Funktion den übergebenen Parameter, in diesem Fall die ganze Zahl 100. Es ist wichtig zu verstehen, dass diese

beiden Referenzen nichts miteinander zu tun haben, da sie in verschiedenen Namensräumen existieren.

Im lokalen Namensraum des Funktionskörpers kann jederzeit lesend auf eine globale Referenz zugegriffen werden, solange keine lokale Referenz gleichen Namens existiert:

```
def f():
    print s

s = "globaler String"
f()
```

Sobald versucht wird, schreibend auf eine globale Referenz zuzugreifen, wird stattdessen eine entsprechende lokale Referenz erzeugt:

```
def f():
    s = "lokaler String"
    print s

s = "globaler String"
f()
print s
```

Die Ausgabe dieses Beispiels lautet:

```
lokaler String
globaler String
```

Eine Funktion kann dennoch, mithilfe der `global`-Anweisung, schreibend auf eine globale Referenz zugreifen. Dazu muss im Funktionskörper das Schlüsselwort `global`, gefolgt von einer oder mehreren globalen Referenzen, geschrieben werden:

```
def f():
    global s
    s = "lokaler String"
    print s

s = "globaler String"
f()
print s
```

Die Ausgabe des Beispiels lautet:

```
lokaler String
lokaler String
```

Im Funktionskörper von `f` wird `s` explizit als globale Referenz gekennzeichnet und kann fortan als solche verwendet werden.

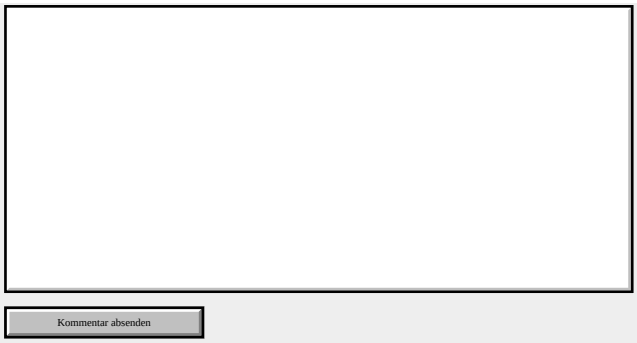
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

10.4 Lokale Funktionen

Es ist möglich, sogenannte *lokale Funktionen* zu definieren. Das sind Funktionen, die im lokalen Namensraum einer anderen Funktion angelegt werden und nur dort gültig sind. Das folgende Beispiel zeigt eine solche Funktion:

```
def globale_funktion(n):

    def lokale_funktion(n):
        return n**2

    return lokale_funktion(n)
```

Innerhalb der globalen Funktion `globale_funktion` wurde eine lokale Funktion namens `lokale_funktion` definiert. Beachten Sie, dass der jeweilige Parameter `n` trotz des gleichen Namens nicht zwangsläufig denselben Wert referenziert. Die lokale Funktion kann im Namensraum der globalen Funktion völlig selbstverständlich wie jede andere Funktion auch aufgerufen werden.

Da sie einen eigenen Namensraum besitzt, hat die lokale Funktion keinen Zugriff auf lokale Referenzen der globalen Funktion. Um dennoch einige ausgewählte Referenzen an die lokale Funktion durchzuschleusen, bedient man sich eines Tricks mit vorbelegten Funktionsparametern:

```
def globale_funktion(n):

    def lokale_funktion(n=n):
        return n**2

    return lokale_funktion()
```

Wie Sie sehen, muss der lokalen Funktion der Parameter `n` beim Aufruf nicht mehr explizit übergeben werden. Er wird vielmehr

implizit in Form eines vorbelegten Parameters übergeben.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

10.6 Rekursion

Python erlaubt es dem Programmierer, sogenannte *rekursive Funktionen* zu schreiben. Das sind Funktionen, die sich selbst aufrufen. Die aufgerufene Funktion ruft sich erneut selbst auf. Das geht so weiter, bis eine Abbruchbedingung diese – sonst endlose – Rekursion beendet. Die Anzahl der aufgerufenen Funktionen wird *Rekursionstiefe* genannt und ist auf einen bestimmten Wert begrenzt.

Jede rekursive Funktion kann, unter Umständen mit viel Aufwand, in eine iterative umgeformt werden. Eine iterative Funktion ruft sich selbst nicht auf, sondern löst das Problem allein durch Einsatz von Kontrollstrukturen, speziell Schleifen. Eine rekursive Funktion ist oft eleganter und kürzer als ihr iteratives Ebenbild, in der Regel aber auch langsamer.

Im folgenden Beispiel wurde eine rekursive Funktion zur Berechnung der Fakultät einer ganzen Zahl geschrieben:

```
def fak(n):
    if n > 0:
        return fak(n - 1) * n
    else:
        return 1
```

Es soll nicht Sinn und Zweck dieses Abschnitts sein, vollständig in die Thematik der Rekursion einzuführen. Stattdessen möchten wir hier nur einen kurzen Überblick geben. Sollten Sie das Beispiel nicht auf Anhieb verstehen, seien Sie nicht entmutigt, denn es lässt sich auch ohne Rekursion passabel in Python programmieren. Trotzdem sollten Sie nicht leichtfertig über die Rekursion hinwegsehen, denn es handelt sich dabei um einen höchst interessanten Weg, sehr elegante Programme zu schreiben.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

10.7 Vordefinierte Funktionen

Es war im Laufe des Buches schon oft von sogenannten *Built-in Functions* die Rede. Das sind vordefinierte Funktionen, die dem Programmierer jederzeit zur Verfügung stehen. Üblicherweise handelt es sich dabei um Hilfsfunktionen, die das Programmieren in Python erheblich erleichtern. Sie kennen bereits die Built-in Functions `len` und `range`. Im Folgenden werden alle bisher relevanten Built-in Functions ausführlich beschrieben. Im Anhang finden Sie eine vollständige tabellarische Übersicht.

abs(x)

Die Funktion `abs` berechnet den Betrag von `x`. Der Parameter `x` muss dabei ein numerischer Wert sein, also eine Instanz der Datentypen `int`, `long`, `float`, `bool` oder `complex`.

```
>>> abs(1)
1
>>> abs(-12.34)
12.34
>>> abs(3 + 4j)
5.0
```

all(iterable)

Die Funktion `all` gibt immer dann `True` zurück, wenn alle Elemente des als Parameter übergebenen iterierbaren Objekts, also beispielsweise einer Liste oder eines Tupels, den Wahrheitswert `True` ergeben. Sie wird folgendermaßen verwendet:

```
>>> all([True, True, False])
False
>>> all([True, True, True])
True
```

any(iterable)

Die Funktion any arbeitet ähnlich wie `all`. Sie gibt immer dann `True` zurück, wenn mindestens ein Element des als Parameter übergebenen iterierbaren Objekts, also zum Beispiel einer Liste oder eines Tupels, den Wahrheitswert `True` ergibt. Sie wird folgendermaßen verwendet:

```
>>> any([True, False, False])
True
>>> any([False, False, False])
False
```

bool([x])

Gibt den Wahrheitswert der Instanz *x* zurück. Wenn kein Parameter übergeben wurde, gibt die Funktion `bool` den booleschen Wert `False` zurück.

chr(i)

Die Funktion chr gibt einen String der Länge 1 zurück, der das Zeichen mit dem ASCII-Code *i* enthält. Der Parameter `i` repräsentiert ein Byte, muss also eine ganze Zahl zwischen `0` und 255 sein.

```
>>> chr(65)
'A'
>>> chr(33)
'!'
```

cmp(x, y)

Die Funktion `cmp` vergleicht zwei beliebige, aber vergleichbare Instanzen miteinander. Das Ergebnis ist negativ, wenn *x* < *y*, null, wenn *x* == *y*, und positiv, wenn *x* > *y*. Beachten Sie, dass nicht nur Zahlen untereinander vergleichbar sind, sondern beispielsweise auch Strings oder Listen.

```
>>> cmp("A", "B")
-1
>>> cmp(999, 99)
1
>>> cmp([1,2], [1,2])
0
```

complex([real[, imag]])

Dies erzeugt eine Instanz des Datentyps `complex` zur Speicherung einer komplexen Zahl. Die erzeugte Instanz hat den komplexen Wert *real* + *imag*`*j`. Fehlende Parameter werden als `0` angenommen.

Außerdem ist es möglich, der Funktion `complex` einen String zu übergeben, der das Literal einer komplexen Zahl enthält. In diesem Fall darf jedoch kein weiterer Parameter angegeben werden.

```
>>> complex(1, 3)
(1+3j)
>>> complex(1.2, 3.5)
(1.2+3.5j)
>>> complex("3+4j")
(3+4j)
>>> complex("3")
(3+0j)
```

Beachten Sie, dass ein eventuell übergebener String keine Leerzeichen um den +-Operator enthalten darf:

```
>>> complex("3 + 4j")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: complex() arg is a malformed string
```

Leerzeichen am Anfang oder Ende des Strings sind aber kein Problem.

dict([source])

Erzeugt eine Instanz des Datentyps `dict`. Wenn kein Parameter übergeben wird, wird ein leeres Dictionary erstellt. Durch einen der folgenden Aufrufe ist es möglich, das Dictionary beim Erzeugen mit Werten zu füllen.

- Wenn *source* ein Dictionary ist, werden die Schlüssel und Werte dieses Dictionarys in das neue übernommen. Beachten Sie, dass dabei keine Kopien der Werte entstehen, sondern diese weiterhin dieselben Instanzen referenzieren.

```
>>> dict({"a" : 1, "b" : 2})
{'a': 1, 'b': 2}
```

- Alternativ kann *source* eine Liste von Tupeln sein, wobei jedes Tupel zwei Elemente enthalten kann: Den Schlüssel und den damit assoziierten Wert. Die Liste muss also folgende Struktur haben: `[("a", 1), ("b", 2)]`:

```
>>> dict([("a", 1), ("b", 2)])
{'a': 1, 'b': 2}
```

- Zudem erlaubt es `dict`, Schlüssel und Werte als *keyword arguments* zu übergeben. Der Parametername wird dabei in einen String geschrieben und als Schlüssel verwendet. Beachten Sie, dass Sie damit bei der Namensgebung den Beschränkungen eines Bezeichners unterworfen sind:

```
>>> dict(a=1, b=2)
{'a': 1, 'b': 2}
```

divmod(a, b)

Die Funktion `divmod` gibt folgendes Tupel zurück: (*a*`//`*b*, *a*`%`*b*). Mit Ausnahme von `complex` können für *a* und *b* Instanzen beliebiger numerischer Datentypen übergeben werden:

```
>>> divmod(2.5, 1.3)
(1.0, 1.2)
>>> divmod(11, 4)
(2, 3)
```

enumerate(iterable)

Die Funktion `enumerate` erzeugt ein iterierbares Objekt, das nicht allein über die Elemente von *iterable* iteriert, sondern über Tupel der folgenden Form: (`i`, *iterable*`[i]`). Dabei ist `i` ein Schleifenzähler, der bei `0` beginnt. Die Schleife wird beendet, wenn `i` den Wert `len(`*iterable*`)-1` hat.

Damit eignet sich `enumerate` besonders für `for`-Schleifen, in denen ein numerischer Schleifenzähler mitgeführt werden soll. Innerhalb einer `for`-Schleife kann `enumerate` folgendermaßen verwendet werden:

```
for i, wert in enumerate(iterable):
    print "Der Wert von iterable an", i, "ter Stelle
ist:", wert
```

Angenommen, der obige Code würde für eine Liste `iterable = [1,2,3,4,5]` ausgeführt, so käme folgende Ausgabe zustande:

```
Der Wert von iterable an 0 ter Stelle ist: 1
Der Wert von iterable an 1 ter Stelle ist: 2
Der Wert von iterable an 2 ter Stelle ist: 3
Der Wert von iterable an 3 ter Stelle ist: 4
Der Wert von iterable an 4 ter Stelle ist: 5
```

file(filename[, mode[, bufsize]])

Konstruiert ein Dateiobjekt. Die Funktion `file` verfügt über die gleiche Schnittstelle wie die in Abschnitt 9.3 besprochene Built-in Function open und bewirkt das Gleiche. Dennoch sollten Sie zum Öffnen einer Datei stets open verwenden, anstatt den Konstruktor des Dateiobjekts direkt aufzurufen.

filter(function, list)

Die Funktion `filter` erwartet ein Funktionsobjekt als ersten und eine Liste als zweiten Parameter. Der Parameter *function* muss eine Funktion oder Lambda-Form sein, die einen Parameter erwartet und einen booleschen Wert zurückgibt.

Die Funktion `filter` ruft für jedes Element der Liste `list` die Funktion *function* auf und erzeugt eine neue Liste, die alle Elemente von `list` enthält, für die *function* `True` zurückgegeben hat. Dies soll an folgendem Beispiel erklärt werden, in dem `filter` dazu verwendet wird, um aus einer Liste von ganzen Zahlen die ungeraden Zahlen herauszufiltern:

```
def fun(prm):
    return (prm%2 == 0)
print filter(fun, [1,2,3,4,5,6,7,8,9,10])
```

Die Ausgabe des Beispiels lautet:

```
[2, 4, 6, 8, 10]
```

float([x])

Erzeugt eine Instanz des Datentyps `float`. Wenn der Parameter *x* nicht angegeben wurde, wird der Wert der Instanz mit `0.0`, andernfalls mit dem übergebenen Wert initialisiert. Mit Ausnahme von `complex` können Instanzen alle numerischen Datentypen für *x* übergeben werden.

```
>>> float()
0.0
>>> float(5)
5.0
```

Außerdem ist es möglich, für *x* einen String zu übergeben, der eine Gleitkommazahl enthält:

```
>>> float("1e30")
1e+30
>>> float("0.5")
0.5
```

frozenset([iterable])

Erzeugt eine Instanz des Datentyps `frozenset` zum Speichern einer unveränderlichen Menge. Wenn der Parameter *iterable* angegeben wurde, so werden die Elemente der erzeugten Menge diesem iterierbaren Objekt entnommen. Wenn der Parameter *iterable* nicht angegeben wurde, ist der Funktionsaufruf äquivalent zu `frozenset([])`. Beachten Sie zum einen, dass ein `frozenset` keine veränderlichen Elemente enthalten darf, und zum anderen, dass jedes Element nur einmal in einer Menge vorkommen kann.

```
>>> frozenset()
frozenset([])
>>> frozenset([1,2,3,4,5])
frozenset([1, 2, 3, 4, 5])
>>> frozenset("Pyyyyyyython")
frozenset(['h', 'o', 'n', 'P', 't', 'y'])
```

globals()

Die Built-in Function globals gibt ein Dictionary mit allen globalen Referenzen des aktuellen Namensraums zurück. Die Schlüssel entsprechen den Referenznamen als Strings und die Werte den jeweiligen Instanzen.

```
>>> a = 1
>>> b = {}
>>> c = [1,2,3]
>>> globals()
{'a': 1, 'c': [1, 2, 3], 'b': {}, '__builtins__': <module
'__builtin__'
 (built-in)>, '__name__': '__main__', '__doc__': None}
```

Das zurückgegebene Dictionary enthält neben den vorher angelegten noch weitere Instanzen, die im globalen Namensraum existieren. Diese vordefinierten Referenzen haben wir bisher noch nicht besprochen, lassen Sie sich davon also nicht stören.

hash(object)

Berechnet den Hash-Wert der Instanz *object* und gibt ihn zurück. Bei einem Hash-Wert handelt es sich um eine ganze Zahl, die aus Typ und Wert der Instanz erzeugt wird. Ein solcher Wert wird verwendet, um effektiv zwei komplexere Instanzen auf Gleichheit prüfen zu können. So werden beispielsweise die Schlüssel eines Dictionarys intern durch ihre Hash-Werte verwaltet.

```
>>> hash(12345)
12345
>>> hash("Hallo Welt")
-962533610
>>> hash((1,2,3,4))
89902565
```

Beachten Sie den Unterschied zwischen veränderlichen (mutable) und unveränderlichen (immutable) Instanzen. Aus letzteren kann zwar formal auch ein Hash-Wert errechnet werden, dieser wäre aber nur so lange gültig, wie die Instanz nicht verändert wurde. Aus diesem Grund ist es nicht sinnvoll, Hash-Werte von veränderlichen Instanzen zu berechnen; veränderliche Instanzen sind »unhashable«:

```
>>> hash([1,2,3,4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

help([object])

Die Funktion help startet die interaktive Hilfe von Python. Wenn der Parameter *object* ein String ist, wird dieser im Hilfesystem nachgeschlagen. Sollte es sich um eine andere Instanz handeln, wird eine dynamische Hilfeseite zu dieser generiert.

hex(x)

Erzeugt einen String, der die als Parameter *x* übergebene ganze Zahl in Hexadezimalschreibweise enthält. Die Zahl entspricht, wie sie im String erscheint, dem Python-Literal für Hexadezimalzahlen.

```
>>> hex(12)
'0xc'
>>> hex(0xFF)
'0xff'
>>> hex(-33)
'-0x21'
```

id(object)

Die Funktion id gibt die Identität einer beliebigen Instanz zurück. Bei der Identität einer Instanz handelt es sich um eine ganze Zahl, die die Instanz eindeutig identifiziert.

```
>>> id(1)
134537016
>>> id(2)
134537004
```

input([prompt])

Liest einen Python-Ausdruck vom Benutzer ein und evaluiert ihn. Das Ergebnis der Evaluation wird zurückgegeben. Der Parameter *prompt* ist optional. Hier kann ein String angegeben werden, der vor der Eingabeaufforderung ausgegeben werden soll.

```
>>> input()
2+2
4
>>> input("Geben Sie einen Ausdruck ein: ")
Geben Sie einen Ausdruck ein: (12 + 14)*9
234
```

int([x[, radix]])

Erzeugt eine Instanz des Datentyps int. Die Instanz kann, durch Angabe von *x*, mit einem Wert initialisiert werden. Wenn kein Parameter angegeben wird, erhält die erzeugte Instanz den Wert 0.

Wenn der Parameter *x* als String übergeben wird, so erwartet die Funktion int, dass dieser String den gewünschten Wert der Instanz enthält. Durch den optionalen Parameter *radix* kann die Basis des Zahlensystems angegeben werden, in dem die Zahl geschrieben wurde.

```
>>> int(5)
5
>>> int("FF", 16)
255
>>> int(hex(12), 16)
12
```

len(s)

Gibt die Länge bzw. die Anzahl der Elemente von *s* zurück.

```
>>> len("Hallo Welt")
10
>>> len([1,2,3,4,5])
5
```

list([sequence])

Erzeugt eine Instanz des Datentyps list aus den Elementen von *sequence*. Der Parameter *sequence* muss ein iterierbares Objekt sein. Wenn er weggelassen wird, wird eine leere Liste erzeugt.

```
>>> list()
[]
>>> list((1,2,3,4))
[1, 2, 3, 4]
>>> list({"a": 1, "b": 2})
['a', 'b']
```

locals()

Die Built-in Function locals gibt ein Dictionary mit allen lokalen

Referenzen des aktuellen Namensraums zurück. Die Schlüssel entsprechen den Referenznamen als Strings und die Werte den jeweiligen Instanzen. Dies soll an folgendem Beispiel deutlich werden:

```
def f(a, b, c):
    d = a + b + c
    print lokals()
f(1, 2, 3)
```

Dieses Beispiel erzeugt folgende Ausgabe:

```
{'a': 1, 'c': 3, 'b': 2, 'd': 6}
```

Beachten Sie, dass der Aufruf von `locals` im Namensraum des Hauptprogramms äquivalent ist zum Aufruf von `globals`.

long([x[, radix]])

Erzeugt eine Instanz des Datentyps `long`. Die Funktionsschnittstelle kann verwendet werden wie die von `int`.

```
>>> long(123)
123L
```

map(function, list ...)

Diese Funktion erwartet ein Funktionsobjekt als ersten und eine Liste als zweiten Parameter. Optional können weitere Listen übergeben werden, die aber die gleiche Länge wie die erste haben müssen. Die Funktion *function* muss genauso viele Parameter erwarten, wie Listen übergeben wurden, und aus den Parametern einen Rückgabewert erzeugen.

Die Funktion `map` ruft *function* für jedes Element der übergebenen Liste auf und gibt eine neue Liste zurück, die die jeweiligen Rückgabewerte von *function* enthält. Sollten mehrere Listen übergeben werden, so werden *function* die jeweils n-ten Elemente aller Listen übergeben. Beachten Sie, dass *function* aus diesem Grund unbedingt genau so viele Parameter erwarten muss, wie Listen übergeben werden, und dass alle übergebenen Listen gleich viele Elemente enthalten müssen.

Im folgenden Beispiel wird das Funktionsobjekt durch eine Lambda-Form erstellt. Es ist auch möglich, eine echte Funktion zu definieren und ihren Namen zu übergeben.

```
>>> f = lambda x: x**2
>>> map(f, [1,2,3,4])
[1, 4, 9, 16]
```

Hier wird `map` dazu verwendet, eine Liste mit den Quadraten der Elemente einer zweiten Liste zu erzeugen.

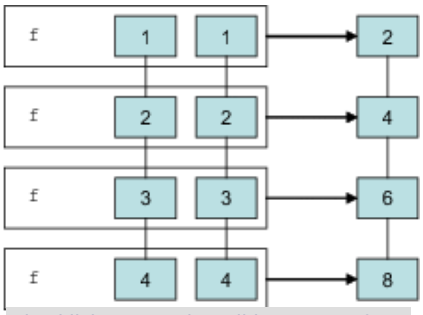
```
>>> f = lambda x, y: x+y
>>> map(f, [1,2,3,4], [1,2,3,4])
[2, 4, 6, 8]
```

Hier wird `map` dazu verwendet, aus zwei Listen eine zu erzeugen, die die Summen der jeweiligen Elemente beider Quelllisten enthält.

In beiden Beispielen wurden Listen verwendet, die ausschließlich numerische Elemente enthielten. Das muss nicht unbedingt sein. Welche Elemente eine Liste enthalten darf, hängt davon ab, welche Instanzen für *function* als Parameter verwendet werden dürfen.

Das obige Beispiel soll durch Abbildung 10.2 anschaulich erklärt

werden.



Hier klicken, um das Bild zu vergrößern

Abbildung 10.2 Arbeitsweise der Built-in Function map

Die eingehenden und ausgehenden Listen sind jeweils senkrecht dargestellt.

max(s[, args...][key])

Wenn keine zusätzlichen Parameter übergeben werden, erwartet `max` eine Sequenz und gibt ihr größtes Element zurück. Die übergebene Instanz eines sequenziellen Datentyps muss Elemente enthalten:

```
>>> max([2,4,1,9,5])
9
>>> max("Hallo Welt")
't'
```

Wenn mehrere Parameter übergeben werden, so verhält sich `max` so, dass der größte übergebene Parameter zurückgegeben wird:

```
>>> max(3, 5, 1, 99, 123, 45)
123
>>> max("Hallo", "Welt", "!")
'Welt'
```

Für beide Verwendungsarten von `max` kann eine optionale Funktion als Schlüsselwortparameter übergeben werden, die für jedes Element der übergebenen Sequenz bzw. jeden Parameter aufgerufen wird, bevor das größte Element festgestellt wird. So ist es mit *key* möglich, aus den übergebenen Datensätzen eine für die Ordnungsrelation relevante Information zu extrahieren.

In folgendem Beispiel soll *key* dazu verwendet werden, die Funktion `max` für Strings *case insensitive* zu machen. Dazu zeigen wir zunächst den normalen Aufruf ohne *key*:

```
>>> max("a", "P", "q", "X")
'q'
```

Ohne eigene *key*-Funktion wird der größte Parameter unter Berücksichtigung von Groß- und Kleinbuchstaben ermittelt. Folgende *key*-Funktion konvertiert zuvor alle Buchstaben in Kleinbuchstaben:

```
>>> f = lambda x: x.lower()
>>> max("a", "P", "q", "X", key=f)
'X'
```

Durch die *key*-Funktion wird der größte Parameter anhand der durch `f` modifizierten Werte ermittelt, jedoch unmodifiziert zurückgegeben.

min(s[, args...][key])

Die Funktion `min` verhält sich wie `max`, ermittelt jedoch das

kleinste Element einer Sequenz bzw. den kleinsten übergebenen Parameter.

oct(x)

Die Funktion `oct` erzeugt einen String, der die übergebene ganze Zahl *x* in Oktalschreibweise enthält.

```
>>> oct(123)
'0173'
>>> oct(0777)
'0777'
```

open(filename[, mode[, bufsize]])

Öffnet eine Datei im gewünschten Modus und gibt das erzeugte Dateiobjekt zurück. Eine vollständige Beschreibung der Funktion finden Sie im Abschnitt 9.3.4, »Verwendung des Dateiobjekts«.

ord(c)

Die Funktion `ord` erwartet einen String der Länge `1` und gibt den ASCII-Code des enthaltenen Zeichens zurück. Wenn es sich um einen Unicode-String handelt, wird der Unicode-Code des Zeichens zurückgegeben.

```
>>> ord("P")
80
```

pow(x, y[, z])

Berechnet *x* ** *y* oder, wenn *z* angegeben wurde, *x* ** *y* % *z*. Beachten Sie, dass diese Berechnung unter Verwendung des Parameters *z* performanter ist als die Ausdrücke `pow(`*x*`, `*y*`) % `*z* bzw. *x* ** *y* % *z*.

```
>>> 7 ** 5  % 4
3
>>> pow(7, 5, 4)
3
```

range([start,]stop[, step])

Die Funktion `range` erzeugt eine Liste mit fortlaufenden, numerischen Elementen. Dabei beginnen die Elemente der Liste mit *start*, enden vor *stop*, und jedes Element ist um *step* größer als das vorherige. Sowohl *start* als auch *step* sind optional und mit `0` bzw. `1` vorbelegt.

Beachten Sie insbesondere, dass *stop* eine Grenze angibt, die nicht erreicht wird. Die Nummerierung beginnt also bei `0` und endet einen Schritt, bevor *stop* erreicht würde.

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

Es ist möglich, eine negative Schrittweite anzugeben:

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

In allen bisherigen Beispielen des Buches, in denen eine Zählschleife benötigt wurde, wurde dafür `range` in Kombination mit einer `for`-Schleife verwendet. Dazu ist zu sagen, dass es in

nahezu allen Fällen performanter ist, stattdessen die Built-in Function xrange zu verwenden, auf die wir in diesem Kapitel noch zu sprechen kommen werden.

raw_input([prompt])

Liest eine Eingabe vom Benutzer ein und gibt diese als String zurück. Durch den optionalen Parameter *prompt* kann ein String angegeben werden, der vor der Eingabeaufforderung ausgegeben wird.

```
>>> raw_input()
2+2
'2+2'
>>> s = raw_input("Ihre Eingabe bitte: ")
Ihre Eingabe bitte: Oh, kein Thema, bitte sehr
>>> s
'Oh, kein Thema, bitte sehr'
```

Beachten Sie, dass raw_input im Gegensatz zu input die Eingabe nicht als Python-Ausdruck interpretiert, sondern als String zurückgibt. Damit ist raw_input in den meisten Fällen eine bessere Wahl als input, da der Input so im Programm ankommt, wie er von der Tastatur eingegeben wurde.

reduce(function, sequence[, initializer])

Die Funktion reduce erwartet ein Funktionsobjekt als ersten und eine Sequenz als zweiten Parameter. Die Funktion *function* muss zwei Parameter akzeptieren und aus ihnen einen Rückgabewert bestimmen.

Die Funktion reduce ruft *function* für die ersten beiden Elemente der Sequenz sequence auf und ersetzt beide Elemente durch den Rückgabewert von *function*. Diese Prozedur wiederholt sich so lange, bis *sequence* nur noch ein Element enthält. Dieses wird zurückgegeben.

Beachten Sie, dass die Funktion frei von Nebeneffekten ist, an *sequence* selbst also nichts geändert wird. Insofern ist reduce ein unglücklich gewählter Name, denn an der übergebenen Sequenz wird nichts geändert.

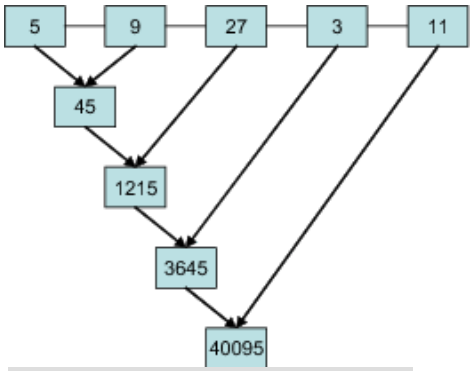
Bei der Liste [a, b, c, d] wäre der Aufruf von reduce gleichbedeutend mit:

```
function(function(function(a, b), c), d)
```

Im folgenden Beispiel wird reduce dazu verwendet, das Produkt aller Elemente von *sequence* zu errechnen.

```
>>> f = lambda x, y: x * y
>>> reduce(f, [5,9,27,3,11])
40095
```

Abbildung 10.3 soll die dem Ergebnis zugrunde liegende Berechnung verdeutlichen.



Hier klicken, um das Bild zu vergrößern

Abbildung 10.3 Funktionsweise von reduce

Wenn der optionale Parameter *initializer* angegeben wurde, so wird die hier übergebene Instanz als Startwert der Berechnung angenommen. Man kann sich das so vorstellen, dass die übergebene Instanz gedanklich als erstes Element in *sequence* eingefügt wird. Das bedeutet unter anderem, dass das Übergeben einer leeren Liste nur dann erlaubt ist, wenn der Parameter *initializer* angegeben wurde.

```
>>> f = lambda x, y: x * y
>>> reduce(f, [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial
value
>>> reduce(f, [], 100)
100
```

repr(object)

Gibt einen String zurück, der eine druckbare Repräsentation der Instanz *object* enthält. Für viele Instanzen versucht `repr`, den Python-Code in den String zu schreiben, der die entsprechende Instanz erzeugen würde. Für manche Instanzen ist dies jedoch nicht möglich bzw. nicht praktikabel. In einem solchen Fall gibt `repr` zumindest den Typ der Instanz aus.

```
>>> repr([1,2,3,4])
'[1, 2, 3, 4]'
>>> repr(0x34)
'52'
>>> repr(set([1,2,3,4]))
'set([1, 2, 3, 4])'
>>> repr(open("datei.txt", "w"))
"<open file 'datei.txt', mode 'w' at 0xb7bea0f8>"
```

reversed(seq)

Mit `reversed` kann eine Sequenz *seq* sehr effizient rückwärts durchlaufen werden: [Die Built-in-Function `reversed` ist nicht auf Sequenzen beschränkt, sondern funktioniert für jedes beliebige iterierbare Objekt. Was es mit iterierbaren Objekten auf sich hat, erfahren Sie in Abschnitt 13.5.]

```
>>> for i in reversed([1, 2, 3, 4, 5, 6]):
...     print i,
6 5 4 3 2 1
```

round(x[, n])

Rundet die Gleitkommazahl *x* auf *n* Nachkommastellen. Der Parameter *n* ist optional und mit `0` vorbelegt.

```
>>> round(0.5, 4)
0.5
>>> round(-0.5)
-1.0
>>> round(0.5234234234234, 5)
0.52342
```

set([iterable])

Erzeugt eine Instanz des Datentyps `set`. Wenn angegeben, werden alle Elemente des iterierbaren Objekts *iterable* in das Set übernommen. Beachten Sie, dass ein Set keine Dubletten enthalten darf, jedes in *iterable* mehrfach vorkommende Element also nur einmal eingetragen wird.

```
>>> set()
set([])
>>> set("Hallo Welt")
set(['a', ' ', 'e', 'H', 'l', 'o', 't', 'W'])
```

```
>>> set([1,2,3,4])
set([1, 2, 3, 4])
```

sorted(iterable[, cmp[, key[, reverse]]])

Die Funktion sorted erzeugt aus den Elementen von *iterable* eine sortierte Liste:

```
>>> sorted([3,1,6,2,9,1,8])
[1, 1, 2, 3, 6, 8, 9]
>>> sorted("Hallo Welt")
[' ', 'H', 'W', 'a', 'e', 'l', 'l', 'l', 'o', 't']
```

Die Funktion akzeptiert drei weitere Schlüsselwortparameter, um das Sortieren der Elemente zu beeinflussen:

- Durch den Schlüsselwortparameter *cmp* kann eine eigene Vergleichsfunktion angegeben werden. Diese muss über die gleiche Schnittstelle und das gleiche Verhalten verfügen wie die Built-in Function cmp. Im Beispiel wurde eine Vergleichsfunktion geschrieben, die das negative Ergebnis der Built-in Function cmp zurückgibt. Dadurch wird die Liste umgekehrt sortiert:

```
>>> f = lambda x, y: -cmp(x, y)
>>> sorted([3,1,6,2,9,1,8], cmp=f)
[9, 8, 6, 3, 2, 1, 1]
```

- Durch den Schlüsselwortparameter *key* kann eine Funktion übergeben werden, die die für den Vergleich wichtige Information aus den Elementen extrahiert. Die Funktion muss einen Parameter akzeptieren und einen Rückgabewert zurückgeben. Wie die Built-in Function max verfügt sorted ebenfalls über einen Parameter *key*, über den eine Funktion gleicher Schnittstelle und gleicher Bedeutung übergeben werden muss.

```
>>> f = lambda x: x.lower()
>>> sorted("Hallo Welt", key=f)
[' ', 'a', 'e', 'H', 'l', 'l', 'l', 'o', 't',
'W']
```

- Der Schlüsselwortparameter *reverse* muss ein boolescher Wert sein und ist mit False vorbelegt. Wird er auf True gesetzt, so veranlasst dies sorted, die Sortierreihenfolge umzukehren.

```
>>> sorted([3,1,6,2,9,1,8], reverse=True)
[9, 8, 6, 3, 2, 1, 1]
```

Die obigen Schlüsselwortparameter können selbstverständlich nicht nur isoliert, sondern auch gemeinsam übergeben werden.

Es ist in den meisten Fällen schneller, eine Liste durch Angabe von *key* und *reverse* in einer bestimmten Weise zu sortieren, als den gleichen Effekt durch *cmp* zu erreichen, da *cmp* für jedes Element mehrfach aufgerufen werden muss.

str([object])

Erzeugt eine Instanz des Datentyps str. Wenn der optionale Parameter *object* übergeben wird, so wird eine lesbare Repräsentation der Instanz in den String geschrieben. Der Unterschied zu repr besteht darin, dass str nicht immer versucht, die Ausgabe als gültigen Python-Code zu formulieren.

```
>>> str(None)
'None'
>>> str()
''
>>> str(None)
'None'
```

```
>>> str(str)
"<type 'str'>"
```

sum(sequence[, start])

Die Funktion sum berechnet die Summe aller Elemente von *sequence* und gibt das Ergebnis zurück. Wenn der optionale Parameter *start* angegeben wurde, so fließt dieser als Startwert der Berechnung ebenfalls in die Summe mit ein.

```
>>> sum([1,2,3,4])
10
>>> sum([1,2,3,4], 2)
12
>>> sum([4,3,2,1], 2)
12
```

tuple([sequence])

Erzeugt eine Instanz des Datentyps tuple und überträgt dabei, wenn angegeben, alle Elemente von *sequence* in diese neue Instanz.

```
>>> tuple()
()
>>> tuple([1,2,3,4])
(1, 2, 3, 4)
```

type(object)

Die Funktion type gibt den Datentyp der übergebenen Instanz *object* zurück.

```
>>> type(1)
<type 'int'>
>>> type("Hallo Welt") == str
True
>>> type(sum)
<type 'builtin_function_or_method'>
```

unichr(i)

Die Funktion unichr erzeugt einen Unicode-String der Länge 1, der das Zeichen mit dem Unicode-Code *i* enthält:

```
>>> unichr(64)
u'@'
>>> unichr(99)
u'c'
```

unicode([object[, encoding[, errors]]])

Erzeugt eine Instanz des Datentyps unicode, die, sofern angegeben, eine lesbare Repräsentation der Instanz *object* enthält.

Wenn *object* eine Instanz des Datentyps str ist, können zusätzlich die Parameter *encoding* und *errors* angegeben werden, die das Konvertieren des 8-Bit-Strings in einen Unicode-String beeinflussen. Der Parameter *encoding* muss ein String sein, der den Namen des Encodings enthält, das zum Decodieren des 8-Bit-Strings verwendet werden soll. Der Parameter *errors* muss ein String sein und regelt, wie verfahren werden soll, wenn ein Zeichen auftritt, das nicht decodiert werden kann. Wenn *errors* den Wert "strict", mit dem der Parameter vorbelegt ist, hat, wird ein ValueError erzeugt. Wenn *errors* den Wert "ignore" hat, werden Fehler beim Dekodieren ignoriert.

```
>>> unicode()
u''
>>> unicode("Hallo Welt")
u'Hallo Welt'
>>> unicode("Hallo Welt", "utf16")
```

```
u'\u6148\u6c6c\u206f\u6557\u746c'
```

xrange([start,]stop[, step])

Die Funktion xrange verfügt über die gleiche Schnittstelle wie die Funktion range und ist für ähnliche Zwecke gedacht. Jedoch gibt xrange keine vollständige Liste zurück, sondern ein sogenanntes XRange-Objekt. Das ist ein iterierbares Objekt, das sich wie eine Liste verhält, aber das nächste Element erst bei Bedarf erzeugt. Die Funktion ist dazu gedacht, range bei der Verwendung in einer for-Schleife abzulösen. Dort ist xrange, gerade wenn sonst sehr große Listen erzeugt werden müssten, schneller und effizienter.

Beachten Sie aber, dass es zwar möglich ist, auf beliebige Elemente des XRange-Objekts zuzugreifen, dass dies aber sehr langsam ist. Sollten solche Operationen gefordert sein, muss zu range gegriffen werden.

zip([iterable, ...])

Die Funktion zip nimmt beliebig viele, gleich lange iterierbare Objekte als Parameter. Sollten nicht alle die gleiche Länge haben, werden die längeren auf die Länge des kürzesten dieser Objekte beschnitten.

Als Rückgabewert wird eine neue Liste erzeugt, die als Element i ein Tupel enthält, das seinerseits die i-ten Elemente der übergebenen Listen enthält.

```
>>> zip([1,2,3,4], [5,6,7,8], [9,10,11,12])
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
>>> zip("Hallo Welt", "HaWe")
[('H', 'H'), ('a', 'a'), ('l', 'W'), ('l', 'e')]
```

Dies waren noch nicht alle Built-in Functions, da einige für Themen gedacht sind, die bisher noch nicht behandelt wurden. Im Anhang finden Sie eine tabellarische Übersicht über alle Built-in Functions, inklusive eines Verweises, wo die jeweilige Funktion detailliert besprochen wird.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

11.2 Eigene Module

Nachdem Sie in die unendlichen Weiten der `import`-Anweisung eingeführt wurden, möchten wir uns damit beschäftigen, wie Module selbst erstellt und eingebunden werden können. Beachten Sie, dass es sich hier nicht um eine Bibliothek handelt, die in jedem Python-Programm zur Verfügung steht, sondern um ein Modul, das nur lokal in Ihrem Python-Programm genutzt werden kann. Von der Verwendung her unterscheiden sich Module und Bibliotheken kaum. In diesem Abschnitt soll ein Programm erstellt werden, das eine ganze Zahl einliest, deren Fakultät und Kehrwert berechnet und die Ergebnisse ausgibt. Die mathematischen Berechnungen sollen dabei nicht nur in Funktionen, sondern auch in einem eigenen Modul gekapselt werden. Dazu schreiben wir diese zunächst in eine Datei namens *mathematik.py*:

```
def fak(n):
    ergebnis = 1
    for i in xrange(2, n+1):
        ergebnis *= i
    return ergebnis

def kehr(n):
    return 1.0 / n
```

Die Funktionen sollten selbsterklärend sein. Beachten Sie, dass die Datei *mathematik.py* selbst keinerlei Code ausführt, sondern nur Funktionen bereitstellt, die aus anderen Modulen heraus aufgerufen werden können.

Jetzt erstellen wir eine Programmdatei namens *programm.py*, in der das Hauptprogramm stehen soll. Beide Dateien müssen sich im selben Verzeichnis befinden. Im Hauptprogramm importieren wir zunächst das lokale Modul `mathematik`. Der Modulname eines lokalen Moduls entspricht dem Dateinamen der zugehörigen Programmdatei ohne Dateiendung. Beachten Sie, dass der Modulname den Regeln der Namensgebung eines Bezeichners folgen muss. Das bedeutet insbesondere, dass, abgesehen von dem Punkt vor der Dateiendung, kein Punkt im Dateinamen

erlaubt ist.

```
import mathematik

while True:
    zahl = int(raw_input("Geben Sie eine ganze Zahl ein:
"))
    print "Fakultaet: ", mathematik.fak(zahl)
    print "Kehrwert: ", mathematik.kehr(zahl)
```

Sie sehen, dass das lokale Modul im Hauptprogramm wie eine Bibliothek importiert und verwendet werden kann.

Durch das Erstellen eigener Module kann es leicht zu Namenskonflikten mit der Standardbibliothek kommen. Beispielsweise hätten wir unsere obige Programmdatei auch *math.py* und das Modul demzufolge `math` nennen können. Dieses Modul stünde im Konflikt mit der Bibliothek `math`. Für solche Fälle ist dem Interpreter eine Reihenfolge vorgegeben, nach der er zu verfahren hat, wenn ein Modul oder eine Bibliothek importiert werden soll:

- Zunächst wird der lokale Programmordner nach einer Datei mit dem entsprechenden Namen durchsucht. In dem oben geschilderten Konfliktfall würde bereits im ersten Schritt feststehen, dass ein lokales Modul namens `math` existiert. Wenn ein solches lokales Modul existiert, wird dieses eingebunden und keine weitere Suche durchgeführt.
- Wenn kein lokales Modul des angegebenen Namens gefunden wurde, wird die Suche auf Bibliotheken ausgeweitet.
- Wenn auch keine Bibliothek mit dem angegebenen Namen gefunden werden konnte, wird ein `ImportError` erzeugt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named abc
```

top

11.2.1 Modulinterne Referenzen

In jedem Modul existieren globale Variablen, die Informationen über das Modul selbst enthalten. An dieser Stelle soll ein Überblick über diese recht überschaubare Anzahl von Referenzen gegeben werden. Beachten Sie, dass es sich jeweils um zwei Unterstriche vor und hinter dem Namen der Referenz handelt.

Referenz	Beschreibung
`__builtins__`	Referenziert ein Dictionary, das die Namen aller eingebauten Typen und Funktionen als Schlüssel und die mit den Namen verknüpften Instanzen als Werte enthält.
`__file__`	Referenziert einen String, der den Namen der Programmdatei des Moduls inklusive Pfad enthält. Nicht bei Modulen der Standardbibliothek verfügbar.
`__name__`	Referenziert einen String, der den Namen des Moduls enthält.

Tabelle 11.1 Globale Variablen in einem Modul

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer

über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

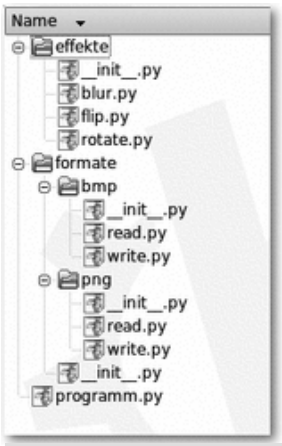
Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

11.3 Pakete

Python ermöglicht es Ihnen, mehrere Module in einem sogenannten *Paket* zu kapseln. Das ist vorteilhaft, wenn diese Module thematisch zusammengehören. Ein Paket kann, im Gegensatz zu einem einzelnen Modul, beliebig viele weitere Pakete enthalten, die ihrerseits wieder Module bzw. Pakete enthalten können.

Um ein Paket zu erstellen, muss im Wesentlichen ein Unterordner im Programmverzeichnis erzeugt werden. Der Name des Ordners entspricht dem Namen des Pakets. Zusätzlich muss in diesem Ordner eine Programmdatei namens *__init__.py* existieren. (Beachten Sie, dass es sich um jeweils zwei Unterstriche vor und hinter »init« handelt.) Diese Datei darf leer, muss aber vorhanden sein und enthält Initialisierungscode, der beim Einbinden des Paketes einmalig ausgeführt wird. Ein Programm mit mehreren Paketen und Unterpaketen hat also eine solche oder ähnliche Verzeichnisstruktur, wie Sie in Abbildung 11.1 sehen.



Hier klicken, um das Bild zu vergrößern

Zum Katalog

Python
bestellen

Ihre Meinung?

Wie hat Ihnen das <openbook> gefallen?
Ihre Meinung

Buchtipps

Linux

Ubuntu GNU/Linux

Praxisbuch Web 2.0

UML 2.0

Praxisbuch Objektorientierung

Abbildung 11.1 Paketstruktur eines Beispielprogramms

Im weiteren Verlauf des Kapitels werden wir immer wieder auf dieses Beispiel zurückkommen, deswegen soll es hier kurz besprochen werden. Es handelt sich um die Verzeichnisstruktur eines fiktiven Bildbearbeitungsprogramms. Das Hauptprogramm befindet sich in der Datei *programm.py*. Neben dem Hauptprogramm existieren im Programmverzeichnis zwei Pakete:

- Das Paket `effekte` soll bestimmte Effekte auf ein bereits geladenes Bild anwenden können. Dazu enthält das Paket neben der Datei *__init__.py* drei Module, die jeweils einen grundlegenden Effekt durchführen sollen. Es handelt sich um die Module `blur` (zum Verwischen des Bildes), `flip` (zum Spiegeln des Bildes) und `rotate` (zum Drehen des Bildes).
- Das Paket `formate` soll dazu in der Lage sein, bestimmte Grafikformate zu lesen und schreiben. Dazu definiert es in seiner *__init__.py* zwei Funktionen namens `leseBild` und `schreibeBild`. Wir möchten nicht näher auf Funktionsschnittstellen oder Ähnliches eingehen, sondern relativ abstrakt bleiben. Damit das Lesen und Schreiben von Grafiken diverser Formate möglich ist, enthält das Paket `formate` zwei Unterpakete namens `bmp` und `png`, die je zwei Module zum Lesen bzw. Schreiben des entsprechenden Formats enthalten.

Im Hauptprogramm sollen zunächst die Pakete `effekte` und `formate` eingebunden und verwendet werden. Dies ermöglicht die `import`-Anweisung:

```
import effekte, formate
```

bzw:

```
import effekte
import formate
```

Beachten Sie, dass es zu einem Namenskonflikt kommt, wenn beispielsweise neben dem Paket `effekte` ein Modul gleichen Namens, also eine Programmdatei namens *effekte.py,* existiert. Es ist grundsätzlich so, dass bei Namensgleichheit ein Paket Vorrang vor einem Modul hat, es also keine Möglichkeit mehr gibt, das Modul zu importieren.

Die `import`-Anweisung veranlasst, dass die Programmdatei *__init__.py* des einzubindenden Paketes ausgeführt und der Inhalt dieser Datei als Modul in einem eigenen Namensraum verfügbar gemacht wird. So könnte nach den obigen `import`-Anweisungen folgendermaßen auf die Funktionen `leseBild` und `schreibeBild` zugegriffen werden:

```
formate.leseBild()
formate.schreibeBild()
```

Um das nun geladene Bild zu modifizieren, soll diesmal ein Modul des Paketes `effekte` geladen werden. Auch dies ist mit der `import`-Anweisung möglich. Der Paketname wird durch einen Punkt vom Modulnamen getrennt. Auf diese Weise kann ein Modul aus einer beliebigen Paketstruktur importiert werden:

```
import effekte.blur
```

In diesem Fall wurde das Paket `effekte` vorher eingebunden. Wenn dies nicht der Fall gewesen wäre, so würde das Importieren von `effekte.blur` dafür sorgen, dass zunächst das Paket `effekte` eingebunden und die dazugehörige *__init__.py* ausgeführt werden würde. Danach wird das Untermodul `blur`

eingebunden. Das Modul kann fortan wie jedes andere verwendet werden:

```
effekte.blur.verschwemmeBild()
```

Auf diese Weise kann auch direkt auf ein Modul zugegriffen werden, das tiefer in der Paketstruktur steht. Hier greifen wir beispielsweise auf das Modul `formate.bmp.write` zu und rufen die Funktion `schreibeBMP` auf, die in diesem Modul definiert werden soll.

```
import formate.bmp.write
formate.bmp.write.schreibeBMP()
```

Auf ein eingebundenes Modul oder Paket kann nur über seinen vollen Namen, also inklusive aller übergeordneten Pakete, zugegriffen werden. Allerdings ist es auch hier möglich, durch as ein Alias zu vergeben:

```
import formate.bmp.write as w
w.schreibeBMP()
```

top

11.3.1 Importieren aller Module eines Pakets ▼▲

Bisher konnte mit

```
from abc import *
```

der gesamte Inhalt eines Moduls in den aktuellen Namensraum importiert werden. Dies funktioniert für Pakete nicht. Der Grund dafür ist, dass einige Betriebssysteme, darunter vor allem Windows, bei Datei- und Ordnernamen nicht zwischen Groß- und Kleinschreibung unterscheiden – Python aber sehr wohl. Angenommen, die obige Anweisung würde wie gehabt funktionieren und abc wäre ein Paket, so wäre es beispielsweise unter Windows völlig unklar, ob ein Untermodul namens `modul` als `Modul`, `MODUL` oder `modul` eingebunden werden soll.

Aus diesem Grund importiert die obige Anweisung nicht alle im Paket enthaltenen Module in den aktuellen Namensraum, sondern importiert nur das Paket an sich und führt den Initialisierungscode in *__init__.py* aus. Sowohl alle in dieser Datei angelegten Elemente als auch alle Elemente von eventuell vorher importierten Modulen dieses Pakets werden in den aktuellen Namensraum eingeführt.

Es gibt zwei Möglichkeiten, das gewünschte Verhalten der obigen Anweisung zu erreichen. Beide müssen vom Autor des Pakets implementiert werden.

Zum einen können alle Module des Pakets innerhalb der *__init__.py* per `import`-Anweisung importiert werden. Dies hätte zur Folge, dass sie beim Einbinden des Paketes und damit nach dem Ausführen des Codes der *__init__.py*-Datei eingebunden wären.

Zum anderen kann dies durch Anlegen einer Referenz namens `__all__` geschehen. Diese muss eine Liste von Strings mit den zu importierenden Modulnamen referenzieren:

```
__all__ = ["blur", "flip", "rotate"]
```

Es liegt im Ermessen des Programmierers, welches Verhalten **from** abc **import** * bei seinen Paketen zeigen soll. Beachten Sie aber, dass das Importieren des kompletten Modul- bzw. Paketinhalts in den aktuellen Namensraum zu unerwünschten

Namenskonflikten führen kann. Aus diesem Grund sollten Sie importierte Module stets in einem eigenen Namensraum führen.

11.3.2 Relative Importanweisungen

Bei der Verwendung der klassischen `import`-Anweisung, die bislang besprochen wurde, kann es, wie bereits angemerkt wurde, zu Namenskonflikten zwischen der Standardbibliothek und einem lokalen Modul oder Paket kommen. Wenn also beispielsweise ein lokales Modul namens `math` existiert, gibt es keine saubere Möglichkeit mehr, das gleichnamige Modul der Standardbibliothek einzubinden.

Um solchen Konfliktfällen entgegenzuwirken, soll sich das Verhalten der `import`-Anweisung in künftigen Python-Versionen dahingehend ändern, dass im Konfliktfall das globale Modul bzw. Paket eingebunden wird. Das explizite Einbinden eines lokalen Moduls oder Pakets geschieht über eine sogenannte *relative Importanweisung*. Auf diese Weise können Namenskonflikte vermieden werden.

Seit Version 2.5 von Python sind relative Importanweisungen in die Sprache aufgenommen worden. Das Verhalten der normalen `import`-Anweisung wurde jedoch nicht geändert. Allerdings ist es möglich, Python 2.5 in einen Modus zu schalten, in dem sich die `import`-Anweisung nach den zukünftigen Regeln verhält. Dazu muss folgende Codezeile zu Beginn der jeweiligen Programmdatei geschrieben werden:

```
from __future__ import absolute_imports
```

Näheres zum Modul `__future__` und seiner Bedeutung erfahren Sie in Abschnitt 13.7.

Eine relative Importanweisung wird folgendermaßen geschrieben:

```
from . import math
```

Als Basis für die relative Importanweisung dient die `from`/`import`-Anweisung, bei der ein Punkt für das aktuelle Paket steht. Im Beispiel würde also das lokale Modul `math` eingebunden, das sich im selben Paket befindet wie das Modul, in dem die relative Importanweisung steht.

Um ein Modul oder Paket einzubinden, das sich in einem Unterpaket befindet, wird folgende relative Importanweisung verwendet:

```
from .math import trig
```

Dieses Mal wurde vorausgesetzt, dass ein Unterpaket `math` existiert, in dem es ein Modul oder Paket namens `trig`, beispielsweise für trigonometrische Funktionen, gibt.

Mithilfe einer relativen Importanweisung ist es auch möglich, ein Modul oder Paket einzubinden, das sich in einem Paket befindet, das in der Pakethierarchie höher angeordnet ist. Im folgenden Beispiel wird das Modul `trig` des direkt übergeordneten Pakets `math` eingebunden, indem dem Paketnamen ein weiterer Punkt vorangestellt wurde:

```
from ..math import trig
```

Beachten Sie bei der Verwendung von relativen Importanweisungen, dass diese nur innerhalb einer Paketstruktur verwendet werden dürfen. Eine Programmdatei, die direkt

ausgeführt wird, befindet sich nicht in einer Paketstruktur und darf deshalb keine relativen Importanweisungen enthalten. Dies bezieht sich insbesondere auch auf den interaktiven Modus.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

11.4 Built-in Functions

Es existieren zwei Built-in Functions, die sich auf Modularisierung, also auf das Einbinden von Modulen und Paketen beziehen. Diese Funktionen wurden in Abschnitt 10.7, »Vordefinierte Funktionen«, nicht erläutert, da das Konzept der Modularisierung Ihnen zu diesem Zeitpunkt noch nicht bekannt war. Aus diesem Grund soll die Beschreibung der Built-in Functions `__import__` und `reload` an dieser Stelle nachgeholt werden. Beachten Sie, dass diese beiden Funktionen nur in wenigen Fällen benötigt werden und daher an dieser Stelle nur oberflächlich erläutert werden. Ausführliche Informationen über die Funktionen finden Sie in der Python-Dokumentation.

`__import__(name[, globals[, locals[, fromlist[, level]]]])`

Die Built-in Function `__import__` wird von der `import`-Anweisung verwendet, um ein Modul oder Paket einzubinden. Die Funktion existiert hauptsächlich, damit sie vom Programmierer überschrieben werden kann, um das Verhalten der `import`-Anweisung zu verändern. Zum Überschreiben der Funktion muss eine neue Funktion mit gleicher Schnittstelle erstellt und dem Namen `__import__` zugewiesen werden.

Die Funktion `__import__` bindet das Modul oder Paket *name* ein und gibt den erzeugten Namensraum zurück. Dabei kann für *globals* und *locals* jeweils ein Dictionary übergeben werden, das alle Referenzen des globalen bzw. lokalen Namensraums enthält. Ein solches Dictionary wird von den Built-in Functions `globals` und `locals` erstellt. Für den vierten Parameter, *fromlist,* kann eine Liste mit Namen übergeben werden, die aus dem Modul *name* eingebunden werden sollen. Der fünfte Parameter, *level*, gibt an, ob absolutes oder relatives Importverhalten verwendet werden soll (vgl. Abschnitt 11.3.2). Der voreingestellte Wert von `-1` weist die Funktion `__import__` dazu an, sowohl absolutes als auch relatives Importverhalten zu zeigen. Ein Wert von `0` würde absolutes Importverhalten vorschreiben, während ein positiver Wert größer null die Anzahl der übergeordneten Verzeichnisse

festlegt, die beim relativen Importverhalten mit einbezogen werden sollen.

Die beiden import-Anweisungen

```
import bla
from blubb import hallo, welt
```

resultieren intern in den folgenden Aufrufen von __import__:

```
__import__("bla")
__import__("blubb", globals(), locals(), ["hallo",
"welt"], -1)
```

reload(module)

Mithilfe der Funktion reload kann ein bereits eingebundenes Modul oder Paket erneut geladen und initialisiert werden. Das ist besonders dann sinnvoll, wenn ein selbst geschriebenes Modul in einer Sitzung im interaktiven Modus eingebunden ist und während der Sitzung Änderungen am Quelltext des Moduls durchgeführt wurden. In einem solchen Fall kann die Funktion reload dazu verwendet werden, das Modul neu einzubinden, ohne die Sitzung im interaktiven Modus beenden zu müssen. Beachten Sie, dass ein Modul nicht mit einer erneuten import-Anweisung neu eingebunden werden kann, da diese zuerst überprüft, ob das Modul bereits eingebunden und initialisiert wurde.

Allgemein ist von der Verwendung von reload vor allem im Quelltext eines Programms, also außerhalb des interaktiven Modus, abzuraten, da das Neuinitialisieren eines Moduls Probleme hervorrufen kann. Diese Probleme sind zum Teil sehr speziell und sollen hier nicht weiter erläutert werden. Eine genaue Beschreibung der möglichen Probleme finden Sie in der Python-Dokumentation.

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

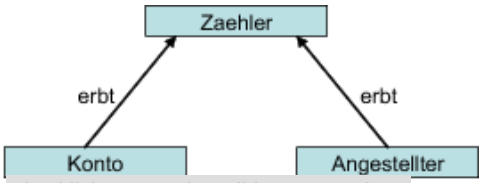
top

12.2 Vererbung

Bisher haben wir nur objektorientierte Techniken behandelt, die durch Kapselung von Daten und Definition von Schnittstellen die Konsistenz der Objekte sichern. Eines der zu Anfang des Kapitels angesprochenen Ziele der Objektorientierung war es aber auch, dass unsere Programme auch leicht veränderlich sind, sodass sie auf Probleme angewandt werden können, die dem ursprünglichen Problem ähnlich sind. Dieses Ziel wird aber mit den bis jetzt eingeführten Techniken noch nicht erreicht.

Wir haben im letzten Abschnitt unsere Klasse `Konto` so erweitert, dass sie mittels eines statischen Attributs die Anzahl ihrer Instanzen nachhalten konnte. Wenn wir nun eine neue Klasse definieren wollten – nehmen wir beispielhaft eine Klasse, die Angestellte der Bank beschreibt – und diese ebenfalls die Anzahl ihrer eigenen Instanzen – in dem Fall also die Zahl der Angestellten – ermitteln soll, so müssten wir den Quellcode für das Instanzzählen ein weiteres Mal in die Klasse `Angestellter` schreiben. Es wäre wünschenswert, einmal festzulegen, wie eine Klasse ihre eigenen Instanzen zählt, und diese Fähigkeit ohne erneutes Aufschreiben des Codes auf neue Klassen übertragen zu können.

Dieses Konzept, Fähigkeiten einer Klasse auf eine andere zu übertragen, nennt man *Vererbung*, wobei alle Member, also sowohl Attribute als auch Methoden, von der Mutter- auf die Tochterklasse übertragen werden. In unserem Beispiel hätten wir also eine Mutterklasse `Zaehler`, die die Instanzzählung implementiert und von der die Klassen `Konto` und `Angestellter` diese Fähigkeit erben:



Hier klicken, um das Bild zu vergrößern

Abbildung 12.2 Konto und Angestellter erben von Zaehler.

Man spricht auch davon, dass die *Basisklasse* `Zaehler` ihre Member an die beiden *Subklassen*, `Konto` und `Angestellter`, vererbt.

Wir wollen nun das angegebene Beispiel in Python implementieren, wobei wir uns zuerst der `Zaehler`-Klasse zuwenden:

```
class Zaehler(object):
    Anzahl = 0

    def __init__(self):
        type(self).Anzahl += 1

    def __del__(self):
        type(self).Anzahl -= 1
```

Die Definition enthält bis auf den Zugriff auf das Attribut `Anzahl` mittels `type(self)` nichts Neues. Wir können deshalb nicht mehr direkt über den Klassennamen per `Zaehler.Anzahl` auf das Attribut zugreifen, weil wir von der Klasse erben wollen und die Subklassen jeweils ihr eigenes statisches Attribut `Anzahl` haben sollen. Würden wir mit `Zaehler.Anzahl` arbeiten, könnten wir damit die Gesamtanzahl der `Konto`- und `Angestellter`-Instanzen berechnen. Mithilfe von `type` lässt sich der Datentyp einer Instanz ermitteln, und das nutzen wir, um den Zähler abhängig davon, welchen Typ `self` hat, für die richtige Klasse zu ändern.

Um nun unsere Klasse `Konto` von `Zaehler` erben zu lassen, müssen wir anstatt des `object` innerhalb der Klammern hinter dem Klassennamen `Zaehler` verwenden. Tatsächlich ist es so, dass wir bis hierher alle unsere Klassen von der Basisklasse `object` haben erben lassen, wodurch sie grundlegende Eigenschaften erhalten haben, damit sie überhaupt als Klasse nutzbar wurden.

Der Grund dafür, dass diese Basisklasse explizit angegeben werden muss, ist historisch bedingt. Früher wurde in Python streng zwischen eingebauten Datentypen und Klassen unterschieden, sodass es insbesondere nicht möglich war, eigene Klassen von diesen Datentypen erben zu lassen. Als man erkannte, dass dies ein großer Nachteil war, vereinigte man eingebaute Datentypen und selbstdefinierte Klassen. Allerdings führte dieser Schritt unter bestimmten Bedingungen zu Problemen mit Programmen, die noch die »alten« Klassen verwendeten. Deshalb kann man heute durch das Erben von `object` explizit angeben, dass man eine »neue« Klasse definieren möchte. Da die überholten *old-style classes* gegenüber den von `object` abgeleiteten *new-style classes* nur Nachteile haben, sollten Sie ausschließlich mit Letzteren arbeiten.

Nun wollen wir unsere `Konto`-Klasse von `Zaehler` erben lassen:

```
class Konto(Zaehler):
    def __init__(self, inhaber, kontonummer, kontostand,
                 max_tagesumsatz=1500):

        Zaehler.__init__(self) # Wichtige Zeile - siehe
unten

        self.__Inhaber = inhaber
        self.__Kontonummer = kontonummer
        self.__Kontostand = kontostand
        self.__MaxTagesumsatz = max_tagesumsatz
        self.__UmsatzHeute = 0

    # hier wären die restlichen Methoden
```

Im Wesentlichen haben sich bei der neuen Definition von Konto nur das schon angesprochene Ersetzen von object durch Zaehler und die erste Zeile des Konstruktors geändert. Mit Zaehler.__init__(self) rufen wir den Konstruktor der Basisklasse auf, um unser Konto auch als Zähler benutzen zu können. Dies ist deshalb notwendig, weil eine Klasse nur eine Methode __init__ haben kann. Bei der Vererbung tritt nun oft der Fall ein, dass die erbende Klasse Methoden definiert, die auch schon in der Basisklasse vorhanden waren – in unserem Beispiel eben der Konstruktor __init__. In einem solchen Fall werden die Methoden der Basisklasse mit denen, die die Subklasse selbst definiert, überschrieben, sodass im Beispiel self.__init__ eine Referenz auf den Konstruktor von Konto und nicht auf den von Zaehler enthält. Um trotzdem auf solche überschriebenen Methoden zugreifen zu können, ersetzt man beim Aufruf das self vor dem Punkt durch den Namen der entsprechenden Basisklasse und übergibt self explizit als Parameter. Würde Zaehler.__init__ noch weitere Parameter erwarten, so würden diese wie üblich durch Kommata getrennt dahinter geschrieben.

Sie sollten sich außerdem als wichtige Regel merken, dass Sie im Konstruktor einer abgeleiteten Klasse immer den Konstruktor der Basisklasse aufrufen müssen, weil Ihre Instanzen sonst aufgrund der fehlenden Initialisierung in einen nicht definierten Zustand übergehen oder sich auf andere Weise falsch verhalten können.

In unserem Fall würde die Instanzzählung ohne den Aufruf des Konstruktors der Basisklasse nicht funktionieren, da der Zähler nicht mit 0 initialisiert würde.

Natürlich können Sie von einer erbenden Klasse weitere Klassen erben lassen, sodass ganze »Stammbäume« entstehen. Wenn Sie beispielsweise bei der Speicherung der Bankangestellten eigene Klassen für jeden Tätigkeitsbereich definieren möchten, so könnten diese von der Klasse Angestellter erben, die wiederum Zaehler als Basisklasse hat:

```
class Angestellter(Zaehler):
    def __init__(self, name, stundenlohn,
stunden_pro_woche):
        Zaehler.__init__(self)
        self.Name = name
        self.Stundenlohn = stundenlohn
        self.StundenProWoche = stunden_pro_woche

    def befoerdere(self, neue_position):
        # hier würde der Code für eine Beförderung stehen
        pass
```

Unsere Angestellten haben der Einfachheit halber nur ihren Namen, ihren Stundenlohn und ihre durchschnittliche Arbeitszeit pro Woche in Stunden als Attribute. Nun könnten wir die beiden speziellen Angestellten, Sekretaerin und Bankdirektor, definieren, die jeweils von der Klasse Angestellter erben:

```
class Sekretaerin(Angestellter):
    def __init__(self, name):
        Angestellter.__init__(self, name, 15, 30)

class Bankdirektor(Angestellter):
    def __init__(self, name, dienstwagen):
        Angestellter.__init__(self, name, 150, 50)
        self.Dienstwagen = dienstwagen
```

Da es in unserer Bank Standardarbeitszeiten und einheitliche Gehälter für jede Position gibt, brauchen diese Informationen nicht mehr an den Konstruktor der abgeleiteten Klassen übergeben zu werden, sondern werden bei dem Aufruf des Konstruktors der Basisklasse intern weitergegeben. Die Sekretaerin hat in unserem einfachen Beispiel neben den von Angestellter geerbten Members keine weiteren Attribute oder Methoden, und der Bankdirektor bekommt neben dem »Erbgut« nur noch ein neues Attribut für seinen Dienstwagen dazu.

Mithilfe des Konzepts der Vererbung wird Ihr Programmtext in

hohem Maße wiederverwendbar, vorausgesetzt, Sie machen sich bei der Strukturierung Ihrer Programme entsprechende Gedanken und zerlegen sie in sinnvoll aufgeteilte Klassen.

12.2.1 Mehrfachvererbung

Bisher haben wir eine Subklasse immer von genau einer Basisklasse erben lassen. Es gibt aber Situationen, in denen eine Klasse die Fähigkeiten von zwei oder noch mehr Basisklassen erben soll, um das gewünschte Ergebnis zu erzielen. Dieses Konzept, bei dem eine Klasse von mehreren Basisklassen erbt, wird *Mehrfachvererbung* genannt.

Möchte man eine Klasse von mehreren Basisklassen erben lassen, muss man die Basisklassen durch Kommata getrennt in die Klammern hinter dem Klassennamen schreiben:

```
class NeueKlasse(Basisklasse1, Basisklasse2, Basisklasse3,
...):
    # Definition von Methoden und Attributen
    pass
```

Wir werden die Mehrfachvererbung an einem einfachen Beispiel verdeutlichen. Angenommen, wir möchten eine Klasse für die Beschreibung von Hausbooten entwickeln, so könnten wir einfach jeweils eine Klasse für die Beschreibung eines Hauses und eine für die eines Bootes definieren, sodass wir durch Vererbung Spezialformen wie das Ferienhaus oder das Rennboot von jeweils einer der Klassen erben lassen könnten. [Dieses Beispiel ist zugegebenermaßen relativ praxisfern, eignet sich aber trotzdem gut, um das Konzept der Mehrfachvererbung zu veranschaulichen.]

Unsere Hausbootklasse soll die Eigenschaften von beiden Klassen, `Haus` und `Boot` erben.

Die beiden Klassen für das Haus und das Boot könnten in stark vereinfachter Form folgendermaßen aussehen:

```
class Haus(object):
    def __init__(self, anzahl_stockwerke, anzahl_zimmer,
                       flaeche, hausnummer):
        self.AnzahlStockwerke = anzahl_stockwerke
        self.AnzahlZimmer = anzahl_zimmer
        self.Flache = flaeche
        self.Hausnummer = hausnummer

        self.HaustuerOffen = False

    def oeffneHaustuer(self):
        self.HaustuerOffen = True

    def schliesseHaustuer(self):
        self.HaustuerOffen = False

class Boot(object):
    def __init__(self, laenge, tiefgang, motorleistung):
        self.Laenge = laenge
        self.Tiefgang = tiefgang
        self.Motorleistung = motorleistung

        self.MotorIstEingeschaltet = False
        self.AnkerGeworfen = True

    def starteMotor(self):
        self.MotorIstEingeschaltet = True

    def stoppeMotor(self):
        self.MotorIstEingeschaltet = False

    def ankerWerfen(self):
        self.AnkerGeworfen = True

    def ankerLichten(self):
        self.AnkerGeworfen = False
```

Die Klasse `Haus` kann sich einige grundlegende Eigenschaften eines Hauses merken und außerdem speichern, ob die Haustür gerade offen bzw. geschlossen ist. Außerdem bietet sie zum Öffnen und Schließen der Türe entsprechende Methoden an.

Mit der Klasse Boot kann man die Länge, den Tiefgang und die Motorleistung in PS speichern. Sie verfügt zusätzlich über Eigenschaften für den Status des Motors und des Ankers, die auch jeweils über Methoden gesetzt werden können.

Nun lassen wir unsere neue Klasse namens Hausboot von den Klassen Haus und Boot erben, wodurch sie alle Fähigkeiten von ihnen übernimmt. Da wir keine zusätzliche Funktionalität hinzufügen wollen, definieren wir nur einen Konstruktor für die Klasse Hausboot, der die Parameter an die Konstruktoren von Haus und Boot weitergibt:

```
class Hausboot(Haus, Boot):
    def __init__(self, anzahl_stockwerke, anzahl_zimmer,
                       flaeche, hausnummer,
                       laenge, tiefgang, motorleistung):

        Haus.__init__(self, anzahl_stockwerke,
anzahl_zimmer,
                      flaeche, hausnummer)
        Boot.__init__(self, laenge, tiefgang,
motorleistung)
```

Nun können wir eine Instanz der Klasse Hausboot erzeugen und zur Demonstration den Anker werfen und den Motor starten:

```
>>> mein_hausboot = Hausboot(2, 10, 200, 5, 20, 1.5,
1000)
>>> mein_hausboot.AnzahlStockwerke
2
>>> mein_hausboot.starteMotor()
>>> mein_hausboot.MotorGestartet
True
>>> mein_hausboot.AnkerGeworfen
False
>>> mein_hausboot.werfeAnker()
>>> mein_hausboot.Ankergeworfen
True
```

Wie das Beispiel zeigt, können wir die Instanz mein_hausboot problemlos wie ein Haus und wie ein Boot verwenden.

Mehrfachvererbung wird erst dann kniffelig, wenn einer Klasse gleichnamige Attribute oder Methoden von verschiedenen Basisklassen vererbt werden.

Was wäre beispielsweise passiert, wenn die Klasse Hausboot keinen eigenen Konstruktor definiert hätte, der die Konstruktoren beider Basisklassen aufruft? Wäre der Konstruktor der Basisklasse Haus oder der der Klasse Boot oder wären vielleicht beide aufgerufen worden?

Wenn in Python eine Klasse von mehreren Basisklassen gleichnamige Member erbt, wird nach der Reihenfolge entschieden, in der die Basisklassen angegeben werden: Es werden immer zuerst die Eigenschaften der weiter links stehenden Basisklasse vererbt.

Wenn wir also eine Klasse Hausboot2 definieren, die ebenfalls von Haus und Boot erbt und deren Klassenkörper ausschließlich aus einer pass-Anweisung besteht, würde Hausboot2 die __init__-Methode von Haus erben:

```
 class Hausboot2(Haus, Boot):
    pass

>>> mein_hausboot2 = Hausboot2()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    mein_hausboot2 = Hausboot2()
TypeError: __init__() takes exactly 5 arguments (1 given)
```

Die Fehlermeldung teilt uns mit, dass der Konstruktor von Hausboot2 genau fünf Parameter erwartet, was genau der Parameteranzahl des Konstruktors von Haus entspricht. Da die __init__-Methode von Boot nur vier Parameter benötigt, handelt es sich beim Konstruktor von Hausboot2 also um den der Haus-Klasse.

Mögliche Probleme der Mehrfachvererbung

Es ist kein Zufall, dass nur wenige Sprachen das Konzept der Mehrfachvererbung unterstützen, da Programme, die es verwenden, anfällig für schwierig auffindbare Fehler sind, weil gleichnamige Member auch dann überschrieben werden, wenn sie semantisch nichts miteinander zu tun haben.

Besonders kritisch wird es dann, wenn eine Klasse über Umwege mehrmals von derselben Basisklasse erbt. Betrachten wir einmal folgende vereinfachte Klassenhierarchie:

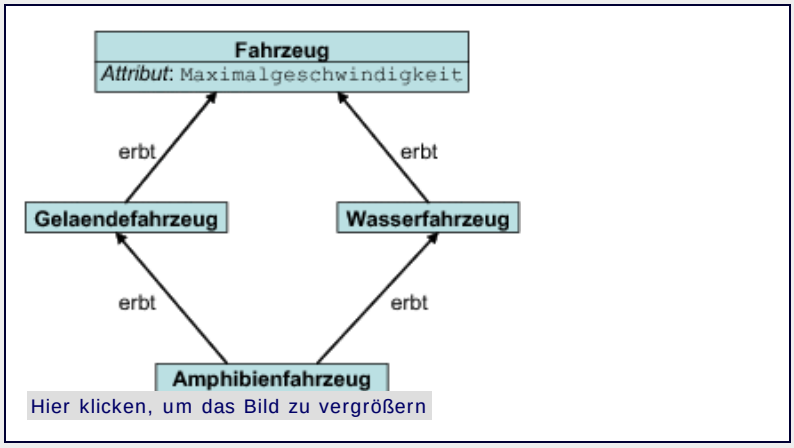


Abbildung 12.3 Amphibienfahrzeug erbt auf zwei Wegen von Fahrzeug

Die Klasse `Amphibienfahrzeug` hat exakt ein Attribut `Maximalgeschwindigkeit`, das sie entweder von `Geländefahrzeug` oder von `Wasserfahrzeug` erbt, je nachdem, in welche Reihenfolge die beiden Basisklassen bei der Definition von `Amphibienfahrzeug` angegeben wurden. Dies ist aber nicht sinnvoll, da sich die `Maximalgeschwindkeiten` zu Lande und zu Wasser in der Regel unterscheiden. Eine brauchbare Klasse zur Beschreibung von Amphibienfahrzeugen lässt sich also nicht durch die gezeigte Mehrfachvererbung definieren, wie es die Intuition raten würde.

Sie sollten in Ihren eigenen Programmen sehr genau darauf achten, dass Sie nur dann Mehrfachvererbungen einsetzen, wenn dadurch keine Konflikte entstehen können, die den Sinn der resultierenden Klasse entstellen – und nach Möglichkeit ganz auf sie verzichten.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

12.3 Magic Members

Es gibt in Python eine Reihe spezieller Methoden und Attribute, um Klassen besondere Fähigkeiten zu geben. Die Namen dieser Member beginnen und enden jeweils mit zwei Unterstrichen __. Im Laufe der letzten Kapitel haben Sie bereits zwei dieser sogenannten *Magic Members* kennengelernt: den Konstruktor namens `__init__` und den Destruktor namens `__del__`.

Der Umgang mit den Methoden und Attributen ist insofern »magisch«, als dass sie in der Regel nicht direkt mit ihrem Namen benutzt, sondern bei Bedarf implizit im Hintergrund verwendet werden. Der Konstruktor `__init__` wird beispielsweise immer dann aufgerufen, wenn ein neues Objekt einer Klasse erzeugt wird, auch wenn kein expliziter Aufruf mit zum Beispiel `Klassenname.__init__()` an der entsprechenden Stelle steht.

Mit vielen Magic Members lässt sich das Verhalten von Built-in Functions und Operatoren für die eigenen Klassen anpassen, sodass die Instanzen Ihrer Klassen beispielsweise sinnvoll mit den Vergleichsoperatoren < und > verglichen werden können.

Wir werden Ihnen im Folgenden eine Liste präsentieren, die häufig genutzte Magic Members mit ihrer Bedeutung auflistet. Aufgrund der großen Anzahl wird dabei bei vielen der besprochenen Methoden und Attribute auf Beispiele verzichtet. Wir bitten Sie, für genauere Informationen Pythons Online-Dokumentation zu konsultieren.

top

12.3.1 Allgemeine Magic Members

__init__(self[, ...])

Der Destruktor einer Klasse. Wird beim Erzeugen einer neuen Instanz aufgerufen. Näheres können Sie in Abschnitt 12.1.2, »Konstruktor, Destruktor und die Erzeugung von Attributen«, nachlesen.

__del__(self)

Der Destruktor einer Klasse. Wird beim Zerstören einer neuen Instanz aufgerufen. Näheres können Sie in Abschnitt 12.1.2, »Konstruktor, Destruktor und die Erzeugung von Attributen«, nachlesen.

__repr__(self)

Der Rückgabewert von `obj.__repr__` gibt an, was `repr(obj)` zurückgeben soll. Dies sollte nach Möglichkeit gültiger Python-Code sein, der beim Ausführen die Instanz `obj` erzeugt.

__str__(self)

Der Rückgabewert von `obj.__str__` gibt an, was `str(obj)` zurückgeben soll. Dies sollte nach Möglichkeit eine für den Menschen lesbare Repräsentation von `obj` sein.

Zugriff auf Attribute anpassen

Die Methoden in diesem Abschnitt dienen dazu, festzulegen, wie Python vorgehen soll, wenn die Attribute einer Instanz gelesen oder geschrieben werden. Da die Standardmechanismen in den meisten Fällen das gewünschte Resultat bewirken, werden Sie diese Methoden nur selten überschreiben.

__dict__

Jede Instanz besitzt ein Attribut namens `__dict__`, das die Member der Instanz in einem Dictionary speichert.

Die beiden folgenden Code-Zeilen produzieren also das gleiche Ergebnis, vorausgesetzt, `obj` ist eine Instanz einer Klasse, die ein Attribut »A« definiert:

```
>>> obj.A
"Der Wert des Attributs A"
>>> obj.__dict__["A"]
"Der Wert des Attributs A"
```

__getattr__(self, name)

Wird dann aufgerufen, wenn das Attribut mit dem Namen *name* gelesen wird, aber nicht existiert.

Die Methode `__getattr__` sollte entweder einen Wert zurückgeben, der für das Attribut gelten soll, oder einen `AttributeError` erzeugen.

__getattribute__(self, name)

Wird immer aufgerufen, wenn der Wert des Attributs mit dem Namen `name` gelesen wird, auch wenn das Attribut bereits existiert.

Implementiert eine Klasse sowohl `__getattr__` als auch `__getattribute__`, wird nur letztere Funktion beim Lesen von Attributen aufgerufen, es sei denn, `__getattribute__` ruft selbst `__getattr__` auf.

Wichtig

Greifen Sie innerhalb von `__getattribute__` niemals mit `self.attribut` auf die Attribute der Instanz zu, weil dies eine endlose Rekursion zur Folge hätte.

Benutzen Sie stattdessen immer `__getattribute__` der Basisklasse, zum Beispiel `object.__getattribute__(self, "attribut")`.

`__setattr__(self, name, value)`

Die Methode `__setattr__` wird immer dann aufgerufen, wenn der Wert eines Attributs per Zuweisung geändert oder ein neues Attribut erzeugt wird. Der Parameter *name* gibt dabei einen String an, der den Namen des zu verändernden Attributs enthält. Mit *value* wird der neue Wert übergeben.

Mit `__setattr__` lässt sich zum Beispiel festlegen, welche Attribute eine Instanz überhaupt haben darf, indem alle anderen Werte einfach ignoriert oder mit Fehlerausgaben quittiert werden.

Wichtig

Verwenden Sie innerhalb von `__setattr__` niemals eine Zuweisung der Form `self.attribut = wert`, um die Attribute auf bestimmte Werte zu setzen, da dies eine endlose Rekursion bewirken würde: Bei jeder Zuweisung würde `__setattr__` erneut aufgerufen.

Um Attribut-Werte mit `__setattr__` zu verändern, können Sie auf das Attribut `__dict__` zurückgreifen: `self.__dict__["attribut"] = wert`.

`__delattr__(self, name)`

Wird aufgerufen, wenn das Attribut mit dem Namen *name* per `del` gelöscht wird.

`__slots__`

Mit dem `__slots__`-Attribut können die Member einer Instanz in der Klasse genau definiert werden. Normalerweise ist es problemlos möglich, auch nach der Instanziierung neue Attribute und Methoden für eine Instanz zu erstellen bzw. Member zu löschen, wie das folgende Beispiel zeigt:

```
>>> class Test(object):
        def __init__(self):
            self.A = 1
            self.B = 2
>>> t = Test()
>>> t.A
1
>>> t.C = 1337
>>> t.C
1337
>>> del t.A
>>> t.A
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    t.A
AttributeError: 'Test' object has no attribute 'A'
```

Dieses Verhalten ist oft aus mehreren Gründen nicht erwünscht:

Das dynamische Erstellen und Löschen von Membern kann zu schwer lokalisierbaren Fehlern führen und das Kapselungsprinzip verletzen. Außerdem muss der Interpreter Aufwand treiben, um die Dynamik der Member zu gewährleisten. Gerade bei Klassen, die sehr oft instanziiert werden sollen, kann dies zu Speicher- und Geschwindigkeitsproblemen führen.

Deshalb kann mit `__slots__` angegeben werden, welche Member

eine Instanz einer Klasse haben darf. Man erzeugt zu diesem Zweck ein statisches Attribut namens __slot__, dem man eine Sequenz der Namen zuweist, die die Attribute und Methoden der Instanzen haben dürfen. Alle Versuche, auf andere Member als die mit __slots__ definierten zuzugreifen, führen dann zu Fehlern. Außerdem benutzt Python für solche Instanzen eine effizientere Technik, um die Attribute und Methoden zu speichern als bei »normalen« Klassen.

Im folgenden Beispiel darf die Klasse Test nur die Attribute namens A und B haben.

```
>>> class Test(object):
        __slots__ = ("A", "B")
        def __init__(self):
            self.A = 1
            self.B = 2
>>> t = Test()
>>> t.A
1
>>> t.C = 1337
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    t.C = 1337
AttributeError: 'Test' object has no attribute 'C'
>>> del t.A
>>> t.A
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    t.A
AttributeError: 'Test' object has no attribute 'A'
```

Wie Sie sehen, schlägt das Erstellen des neuen Attributs C mit einem Attribute Error fehl. Es ist allerdings immer noch möglich, bereits vorhandene Attribute per del zu löschen. Diese können allerdings auch wieder erzeugt werden, sofern sie in der __slots__-Liste stehen.

Wichtig

Eine __slots__-Definition lässt sich nicht auf Subklassen vererben.

Vergleichsoperatoren

Die folgenden *Magic Methods* dienen dazu, das Verhalten der Vergleichsoperatoren für die Klasse anzupassen. Man nennt diese Anpassung auch *Überladen* des Operators.

Um beispielsweise zwei Kontoklassen zu vergleichen, kann die Kontonummer herangezogen werden. Damit gibt es eine sinnvolle Interpretation für den Vergleich mit == bei Konten. Die *Magic Method* für Vergleiche mit == heißt __eq__ (von engl. *equal* = *gleich*) und erwartet als Parameter eine Instanz, mit der das Objekt verglichen werden soll, für das __eq__ aufgerufen wurde.

Der folgende Beispielcode erweitert unsere Konto-Klasse aus der Einführung zur Objektorientierung um die Fähigkeit, sinnvoll mit == verglichen zu werden:

```
class Konto(object):
    def __init__(self, inhaber, kontonummer, kontostand,
                 max_tagesumsatz=1500):
        self.Inhaber = inhaber
        self.Kontonummer = kontonummer
        self.Kontostand = kontostand
        self.MaxTagesumsatz = max_tagesumsatz
        self.UmsatzHeute = 0

    def __eq__(self, k2):
        return self.Kontonummer == k2.Kontonummer
```

Nun erzeugen wir drei Konten, wobei zwei die gleiche Kontonummer haben, und vergleichen sie mit dem ==-Operator. Das Szenario wird natürlich immer ein Wunschtraum für Donald Duck bleiben:

```
>>> konto1 = Konto("Dagobert Duck", 1337,
9999999999999999)
>>> konto2 = Konto("Donald Duck", 1337, 1.5)
>>> konto3 = Konto("Gustav Gans", 2674, "50000")
>>> konto1 == konto2
True
>>> konto1 == konto3
False
```

Die Anweisung `konto1 == konto2` wird intern von Python beim Ausführen durch `konto1.__eq__(konto2)` ersetzt.

Neben der `__eq__`-Methode gibt es eine Reihe weiterer Vergleichsmethoden, die jeweils einem Vergleichsoperator entsprechen. Alle diese Methoden erwarten neben `self` einen weiteren Parameter, der die Instanz referenzieren muss, mit der `self` verglichen werden soll.

Die nachfolgende Tabelle zeigt alle Vergleichsmethoden mit ihren Entsprechungen. Die Herkunftstabelle kann Ihnen unter Umständen helfen, sich die Methodennamen und ihre Bedeutung besser zu merken:

Methode	Operator	Herkunft
`__lt__(self, other)`	`<`	*Lower Than* (dt. kleiner als)
`__le__(self, other)`	`<=`	*Less or Equal* (dt. kleiner oder gleich)
`__eq__(self, other)`	`==`	*Equal* (dt. gleich)
`__ne__(self, other)`	`!=`	*Not Equal* (dt. ungleich)
`__gt__(self, other)`	`>`	*Greater Than* (dt. größer als)
`__ge__(self, other)`	`>=`	*Greater or Equal* (dt. größer oder gleich)

Tabelle 12.3 Die Magic Methods für Vergleiche

Es ist besonders für eher kleine Klassen lästig, immer alle dieser sechs Methoden zu implementieren, bloß damit zwei Instanzen verglichen werden können.

Alternativ kann die Methode `__cmp__` verwendet werden.

`__cmp__(self, other)`

Mit `__cmp__` kann eine Methode definiert werden, die es ermöglicht, die Vergleichsoperatoren für die implementierende Klasse zu verwenden, ohne dass dafür die sechs Methoden des vorhergehenden Abschnitts definiert werden müssen.

Die Methode `__cmp__` muss einen Zahlenwert zurückgeben, der angibt, ob *self* oder *other* den größeren Wert hat oder ob beide identisch sind. Die folgende Tabelle zeigt die möglichen Rückgabewerte und ihre Bedeutungen:

Rückgabewert	Bedeutung
Ganzzahl größer als 0	`self > other`
0	`self == other`
Ganzzahl kleiner als 0	`self < other`

Tabelle 12.4 Die Rückgabewerte von `__cmp__`

Wichtig
Wenn eine Klasse keine der Methoden `__cmp__`, `__eq__` oder `__ne__` implementiert, werden Instanzen der Klasse anhand ihrer Identität miteinander verglichen.

__hash__(self)

Die __hash__-Methode einer Instanz bestimmt, welchen Wert die Built-in Function hash für die Instanz zurückgeben soll. Die hash-Werte müssen Ganzzahlen sein und sind insbesondere für die Verwendung von Instanzen als Schlüssel für Dictionarys von Bedeutung.

Die einzige Bedingung für gültige hash-Werte ist, dass Objekte, die bei Vergleichen mit == als gleich angesehen werden, auch den gleichen hash-Wert besitzen.

__nonzero__(self)

Die __nonzero__-Methode sollte einen Wahrheitswert (True oder False) zurückgeben, der angibt, wie das Objekt in eine bool-Instanz umzuwandeln ist.

Ist __nonzero__ nicht implementiert, wird stattdessen der Rückgabewert von __len__ verwendet. Sind beide Methoden nicht vorhanden, werden alle Instanzen der betreffenden Klasse als True behandelt.

__unicode__(self)

Wie __str__, nur dass anstelle einer str-Instanz ein Objekt des Typs unicode zurückgegeben werden muss.

__call__(self[, args...])

Mit der __call__-Methode können die Instanzen einer Klasse wie Funktionen aufrufbar werden.

Das folgende Beispiel implementiert eine Klasse Potenz, die dazu dient, Potenzen zu berechnen. Welcher Exponent dabei verwendet werden soll, wird dem Konstruktor als Parameter übergeben. Durch die __call__-Methode können die Instanzen von Potenz wie Funktionen aufgerufen werden, um Potenzen zu berechnen:

```
class Potenz(object):
    def __init__(self, exponent):
        self.Exponent = exponent

    def __call__(self, basis):
        return basis ** self.Exponent
```

Nun kann bequem mit Potenzen gearbeitet werden:

```
>>> dreier_potenz = Potenz(3)
>>> dreier_potenz(2)
8
>>> dreier_potenz(5)
125
```

top

12.3.2 Datentypen emulieren

In Python entscheiden die Methoden, die ein Datentyp implementiert, zu welcher Kategorie von Datentypen er gehört. Deshalb ist es möglich, Ihre eigenen Datentypen wie beispielsweise numerische oder sequenzielle Datentypen »aussehen« zu lassen, indem sie die entsprechende Schnittstelle implementieren.

Sie werden im Folgenden die Methoden kennenlernen, die ein Datentyp implementieren muss, um ein numerischer Datentyp zu sein. Außerdem werden die Schnittstellen von Sequenzen und Mappings behandelt.

Numerische Datentypen emulieren

Ein numerischer Datentyp muss vor allem eine Reihe von Operatoren definieren.

Binäre Operatoren

Als Erstes gibt es da die sogenannten binären Operatoren, die zwei Operanden erwarten. Hierzu zählen unter anderem +, -, * und /.

Alle Methoden zum Überladen von binären Operatoren erwarten einen Parameter, der den zweiten Operanden referenziert. Ihr Rückgabewert muss eine neue Instanz sein, die das Ergebnis der Rechnung enthält.

Wenn Python einen Ausdruck auswertet, der binäre Operatoren enthält, werden intern automatisch die entsprechenden Methoden aufgerufen. Die folgenden beiden Befehle sind vollkommen gleichwertig, wobei die Klammern um die 1 aus syntaktischen Gründen notwendig sind:

```
>>> 1 + 2
3
>>> (1).__add__(2)
3
```

Als Beispiel werden wir eine kleine Klasse zum Verwalten von Längenangaben mit Einheiten implementieren, die die Operatoren für Addition und Subtraktion unterstützt.

Die Klasse wird intern alle Maße für die Berechnungen in Meter umwandeln. Ihre Definition sieht dann folgendermaßen aus:

```
class Laenge(object):
    Umrechnung = {"m" : 1, "cm" : 0.01, "mm" : 0.001,
                  "dm" : 10, "km" : 1000,
                  "ft" : 0.3048,   # Fuss
                  "in" : 0.0254,   # Zoll
                  "mi" : 1609344   # Meilen
                 }

    def __init__(self, zahlenwert, einheit):
        self.Zahlenwert = zahlenwert
        self.Einheit = einheit

    def __str__(self):
        return "%f%s" % (self.Zahlenwert, self.Einheit)

    def __add__(self, other):
        z = self.Zahlenwert *
Laenge.Umrechnung[self.Einheit]
        z += other.Zahlenwert *
Laenge.Umrechnung[other.Einheit]
        z /= Laenge.Umrechnung[self.Einheit]
        return Laenge(z, self.Einheit)

    def __sub__(self, other):
        z = self.Zahlenwert *
Laenge.Umrechnung[self.Einheit]
        z -= other.Zahlenwert *
Laenge.Umrechnung[other.Einheit]
        z /= Laenge.Umrechnung[self.Einheit]
        return Laenge(z, self.Einheit)
```

Das Dictionary Laenge.Umrechnung enthält Faktoren, mit denen geläufige Längenmaße in Meter umgerechnet werden. Die Methoden __add__ und __sub__ überladen jeweils den Operator für Addition + bzw. den für Subtraktion –, indem sie zuerst die Zahlenwerte beider Operanden gemäß ihrer Einheiten in Meter umwandeln, verrechnen und schließlich wieder in die Einheit des weiter links stehenden Operanden konvertieren.

In der nachstehenden Tabelle sind alle binären Operatoren und die entsprechenden Magic Methods aufgelistet:

Operator	Magic Method
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__div__(self, other)

//	__floordiv__(self, other)
**	__pow__(self, other[, modulo])
%	__mod__(self, other)
>>	__lshift__(self, other)
<<	__rshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Tabelle 12.5 Magic Methods für binäre Operatoren

Die Methoden sollten bei erfolgreicher Rechnung das Ergebnis zurückgeben. Ist es nicht möglich, den Operanden *other* zu verarbeiten, sollte `NotImplemented` zurückgegeben werden.

Binäre Operatoren mit umgekehrter Operandenreihenfolge

Wenn Python einen Ausdruck der Form `Operand1 Operator Operand2` wie beispielsweise `2 * "abc"` auswerten soll, wird zuerst versucht, eine passende Methode vom ersten Operanden zu benutzen. Existiert diese nicht oder gibt sie `NotImplemented` zurück, wird versucht, beim zweiten Operanden eine entsprechende Methode zu finden. Dies geschieht aber nur dann, wenn die beiden Operanden nicht vom gleichen Datentyp sind. Allerdings muss der zweite Operand eine spezielle Methode für vertauschte Operanden implementieren. Die folgende Tabelle listet alle dafür verfügbaren Methodennamen und die entsprechenden Operatoren auf:

Operator	Magic Method
+	__radd__(self, other)
–	__rsub__(self, other)
*	__rmul__(self, other)
/	__rdiv__(self, other)
//	__rfloordiv__(self, other)
**	__rpow__(self, other[, modulo])
%	__rmod__(self, other)
>>	__rlshift__(self, other)
<<	__rrshift__(self, other)
&	__rand__(self, other)
|	__ror__(self, other)
^	__rxor__(self, other)

Tabelle 12.6 Magic Methods für binäre Operatoren des rechten Operanden

Für nicht unterstützte Werte von *other* sollte auch hier `NotImplemented` zurückgegeben werden.

Erweiterte Zuweisungen

Neben den Operatoren selbst können auch die erweiterten Zuweisungen überladen werden. Bei einer erweiterten Zuweisung wird ein Gleichheitszeichen benutzt, dem der jeweilige Operator vorangestellt wird:

```
>>> a = 10
>>> a += 5
>>> a
15
```

Standardmäßig verwendet Python für solche Zuweisungen den Operator selbst, sodass `a += 5` intern wie `a = a + 5` ausgeführt wird. Diese Vorgehensweise hat für komplexe Datentypen wie beispielsweise Listen den Nachteil, dass immer eine komplett neue Liste erzeugt werden muss. Deshalb kann man gezielt die erweiterten Zuweisungen anpassen, um die Effizienz des Programms zu verbessern.

In der nachstehenden Tabelle stehen alle Operatoren für erweiterte Zuweisungen und die entsprechenden Methoden:

Operator	Magic Method
+=	__iadd__(self, other)
-=	__isub__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
**=	__ipow__(self, other[, modulo])
%=	__imod__(self, other)
>>=	__ilshift__(self, other)
<<=	__irshift__(self, other)
&=	__iand__(self, other)
|=	__ior__(self, other)
^=	__ixor__(self, other)

Tabelle 12.7 Methoden für die erweiterte Zuweisung

Unäre Operatoren

Mit den folgenden Methoden werden die unären Operatoren überladen. Unäre Operatoren erwarten im Gegensatz zu den binären Operatoren nur einen Parameter. Zu den unären Operatoren zählen die Vorzeichen + und –, die Built-in Function abs zur Bestimmung des absoluten Werts und die Tilde ~, um das Komplement eines Wertes zu berechnen:

Operator	Magic Method
+	__pos__(self)
–	__neg__(self)
abs	__abs__(self)
~	__invert__(self)

Tabelle 12.8 Magic Methods für die binären Operatoren

Bulit-In Functions und Umwandlung in andere Typen

Zu guter Letzt gibt es einen Satz von Magic Methods, mit dem bestimmt werden kann, welche Werte bestimmte Built-in Functions für sie zurückgeben sollen. Für diesen Zweck existieren die folgenden Methoden:

Built-in	Magic Method
complex	__complex__(self)
int	__int__(self)
long	__long__(self)
float	__float__(self)
oct	__oct__(self)
hex	__hex__(self)

Tabelle 12.9 Methoden zum Festlegen der Built-in-Rückgabewerte

__index__(self)

Wenn ein Datentyp außerdem als Index benutzt werden soll, wie er beispielsweise für das Slicing benötigt wird, muss er die parameterlose Methode __index__(self) überschreiben. Der Rückgabewert von __index__ muss eine Ganzzahl (int oder long) sein.

Container emulieren

Mithilfe der folgenden Methoden ist es möglich, eigene sequenzielle oder Mapping-Datentypen zu implementieren, die sich genauso wie Listen oder Dictionarys benutzen lassen.

Wenn Sie einen sequenziellen Datentyp entwickeln möchten, müssen alle Indizes `i` der Sequenz Ganzahlen sein und die Gleichung `0 <= i < N` erfüllen. `N` ist dabei als Länge der Sequenz definiert.

Alle Mapping-Datentypen sollten zusätzlich zu den nachfolgend besprochenen *Magic Methods* weitere Methoden implementieren, die in der nachstehenden Tabelle aufgelistet sind: [Wenn Ihnen die hier angegebenen Beschreibungen nicht ausführlich genug sind, können Sie sich noch einmal den Abschnitt 8.6.1, »Dictionary – dict«, ansehen.]

Methode	Bedeutung
`m.keys()`	Gibt eine Liste der Schlüssel von `m` zurück.
`m.values()`	Gibt eine Liste der Werte von `m` zurück.
`m.items()`	Gibt eine Liste der Schlüssel/Wert-Paare von `m` zurück.
`m.has_key(k)`	Prüft, ob der Schlüssel `k` in `m` existiert.
`m.get(k[, d])`	Wenn der Schlüssel `k` in `m` existiert, wird `m[k]` zurückgegeben, ansonsten `d`.
`m.clear()`	Entfernt alle Elemente aus `m`.
`m.setdefault(k[, x])`	Wenn der Schlüssel `k` in `m` existiert, wird `m[k]` zurückgegeben. Gibt es den Schlüssel `k` nicht in `m`, wird `m[k]` auf den Wert `x` gesetzt und `x` zurückgegeben.
`m.iterkeys()`	Gibt einen Iterator über die Schlüssel von `m` zurück.
`m.itervalues()`	Gibt einen Iterator über die Werte von `m` zurück.
`m.iteritems()`	Gibt einen Iterator über die Schlüssel/Wert-Paare von `m` zurück.
`m.pop(k[, d])`	Wenn der Schlüssel `k` in `m` existiert, wird `m[k]` zurückgegeben und danach mit `del` gelöscht. Gibt es den Schlüssel `k` nicht in `m`, so wird `d` zurückgegeben.
`m.popitem()`	Gibt ein willkürlich ausgewähltes Schlüssel/Wert-Paar von `m` zurück und entfernt es anschließend aus `m`.
`m.copy()`	Gibt eine Kopie von `m` zurück.
`m.update(b)`	Übernimmt alle Schlüssel/Wert-Paare von `b` in `m`. Vorhandene Einträge werden dabei überschrieben.

Tabelle 12.10 Nötige Methoden für Mapping-Typen

Nun folgt die Besprechung der *Magic Methods* für sequenzielle und Mapping-Datentypen:

`__len__(self)`

Liefert eine Ganzzahl zurück, die die Länge der Sequenz oder des Mapping-Objekts angibt.

`__getitem__(self, key)`

Wird aufgerufen, wenn mit dem `[]`-Operator ein Element des Containers gelesen wird. Der Parameter *key* gibt dabei den Index des gesuchten Elements an.

Ein sehr einfaches Beispiel gibt immer den Index selbst als Element des Containers zurück:

```
>>> class MeinContainer(object):
        def __getitem__(self, key):
            return key
>>> obj = MeinContainer()
>>> obj[10]
10
>>> obj[1337]
1337
```

Wenn der übergebene Index *key* ungültig ist, sollte `__getitem__` einen `IndexError` produzieren.

`__setitem__(self, key, value)`

Muss das Element mit dem Index *key* auf den Wert *value* setzen.

Diese Methode sollte nur dann implementiert werden, wenn der Datentyp das Verändern und Hinzufügen von Elementen unterstützen soll.

Bei ungültigen *key*-Werten sollte ein `IndexError` erzeugt werden.

`__delitem__(self, key)`

Muss das Element mit dem Index *key* aus dem Container entfernen.

Bei ungültigen *key*-Werten sollte ein `IndexError` erzeugt werden.

`__iter__(self)`

Muss einen Iterator über die Werte des sequenziellen Datentyps bzw. über die Schlüssel des Mapping-Typs zurückgeben.

Genaues zu Iteratoren können Sie in Abschnitt 13.5, »Iteratoren«, nachlesen.

`__contains__(self, item)`

Muss einen Wahrheitswert zurückgeben, der angibt, ob der sequenzielle Datentyp ein Element mit dem Wert von *item* enthält. Handelt es sich um einen Mapping-Typ, wird geprüft, ob es einen Schlüssel mit dem Wert von *item* gibt.

Diese Methode wird von den Operatoren `in` und `not in` benutzt.

Allerdings ist es nicht notwendig, `__contains__` zu implementieren, wenn bereits `__iter__` für den Typ definiert worden ist. Mit `__contains__` kann der Datentyp nur eine unter Umständen effizientere Prüfung anbieten, da nicht wie bei `__iter__` erst die Elemente der Sequenz durchlaufen werden müssen.

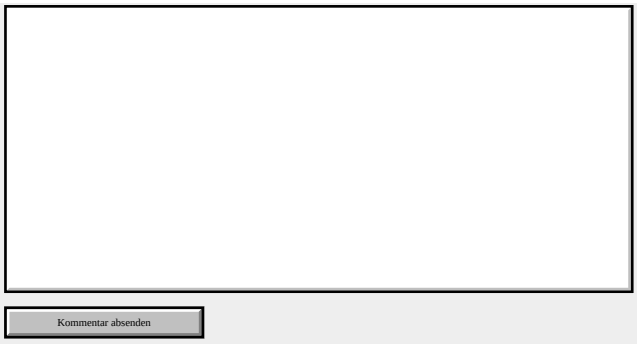
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

12.4 Objektphilosophie

Seitdem in Python 2.3 Datentypen und Klassen vereinigt wurden, ist Python von Grund auf objektorientiert. Das bedeutet, dass im Prinzip alles, mit dem Sie bei der Arbeit mit Python in Berührung kommen, eine Instanz irgendeiner Klasse ist. Von der einfachen Zahl bis zu den Klassen [Der Datentyp von Klassen-Instanzen sind sogenannte Metaklassen, deren Verwendung in diesem Buch nicht behandelt wird.] selbst hat dabei jedes Objekt seine eigenen Attribute und Methoden.

Insbesondere ist es möglich, von eingebauten Datentypen wie `list` oder `dict` zu erben.

Das folgende Beispiel zeigt eine Subklasse von `list`, die den Durchschnittswert ihrer Elemente berechnen kann. Sollte ein Element einen anderen Datentyp als `int` oder `float` haben, wird es einfach ignoriert:

```
class ListeMitDurchschnitt(list):
    def durchschnitt(self):
        summe, i = 0.0, 0
        for e in self:
            if type(e) in (int, float):
                summe += e
                i += 1
        return summe / i
```

Der Datentyp `ListeMitDurchschnitt` kann nun genau wie der Datentyp `list` verwendet werden:

```
>>> l = ListeMitDurchschnitt((2, 3, 4))
>>> l
[2, 3, 4]
```

```
>>> l.append(5)
>>> l.durchschnitt()
3.5
```

Durch die konsequente Objektorientierung werden Python-Programme noch leichter zu entwickeln und wiederzuverwenden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.2 List Comprehensions

Es ist ein häufig auftretendes Problem, dass man aus den Elementen einer bestehenden Liste eine neue Liste erstellen möchte, deren Elemente aus denen der alten Liste berechnet wurden. Bislang würden Sie dies entweder sehr umständlich in einer `for`-Schleife erledigen oder die Built-in Functions `map` und `filter` einsetzen. Letzteres ist zwar relativ kurz, bedarf jedoch einer Funktion, die auf jedes Element der Liste angewandt wird. Das ist umständlich und ineffizient.

Python unterstützt eine sehr viel flexiblere Syntax, die für gerade diesen Zweck geschaffen wurde: die sogenannten *List Comprehensions*. Die folgende List Comprehension erzeugt aus einer Liste mit ganzen Zahlen eine neue Liste, die die Quadrate dieser Zahlen enthält:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> [x**2 for x in lst]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Eine List Comprehension wird in eckige Klammern gefasst und

besteht zunächst aus einem Ausdruck, gefolgt von beliebig vielen for/in-Bereichen. Ein for/in-Bereich lehnt sich an die Syntax der for-Schleife an und gibt an, mit welchem Bezeichner über welche Liste iteriert wird – in diesem Fall mit dem Bezeichner x über die Liste lst. Der angegebene Bezeichner kann im Ausdruck zu Beginn der List Comprehension verwendet werden. Das Ergebnis einer List Comprehension ist eine neue Liste, die als Elemente die Ergebnisse des Ausdrucks in jedem Iterationsschritt enthält. Die Funktionsweise der obigen List Comprehension lässt sich folgendermaßen zusammenfassen:

Für jedes Element x der Liste lst, bilde das Quadrat von x, und füge das Ergebnis in die Ergebnisliste ein.

Dies ist die einfachste Form der List Comprehension. Der for/in-Bereich lässt sich um eine Fallunterscheidung erweitern, sodass nur bestimmte Elemente in die neue Liste übernommen werden. So könnten wir die obige List Comprehension beispielsweise dahingehend erweitern, dass nur die Quadrate gerader Zahlen gebildet werden:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> [x**2 for x in lst if x%2 == 0]
[4, 16, 36, 64]
```

Dazu wird der for/in-Bereich um das Schlüsselwort if erweitert, auf das eine Bedingung folgt. Nur wenn diese Bedingung True ergibt, wird das berechnete Element in die Ergebnisliste aufgenommen. Diese Form der List Comprehension lässt sich also folgendermaßen beschreiben:

Für jedes Element x der Liste lst – sofern es sich bei x um eine gerade Zahl handelt –, bilde das Quadrat von x, und füge das Ergebnis in die Ergebnisliste ein.

Als nächstes Beispiel soll eine List Comprehension dazu verwendet werden, zwei als Listen dargestellte Vektoren zu addieren. Die Addition zweier Vektoren erfolgt koordinatenweise, also in unserem Fall Element für Element:

```
>>> v1 = [1, 7, -5]
>>> v2 = [-9, 3, 12]
>>> [v1[i] + v2[i] for i in range(3)]
[-8, 10, 7]
```

Dazu wird eine von range erzeugte Liste von Indizes in der List Comprehension durchlaufen. In jedem Durchlauf werden die jeweiligen Koordinaten addiert und an die Ergebnisliste angehängt.

Es wurde bereits gesagt, dass eine List Comprehension beliebig viele for/in-Bereiche haben kann. Diese können wie verschachtelte for-Schleifen betrachtet werden. Im Folgenden möchten wir ein Beispiel besprechen, in dem diese Eigenschaft von Nutzen ist. Zunächst definieren wir zwei Listen:

```
>>> lst1 = ["A", "B", "C"]
>>> lst2 = ["D", "E", "F"]
```

Eine List Comprehension soll nun eine Liste erstellen, die alle möglichen Buchstabenkombinationen enthält, die gebildet werden können, indem man zunächst einen Buchstaben aus lst1 und dann einen aus lst2 wählt. Die Kombinationen sollen jeweils als Tupel in der Liste stehen:

```
>>> [(a,b) for a in lst1 for b in lst2]
[('A', 'D'), ('A', 'E'), ('A', 'F'), ('B', 'D'), ('B',
'E'), ('B',
'F'), ('C', 'D'), ('C', 'E'), ('C', 'F')]
```

Diese List Comprehension kann folgendermaßen beschrieben

werden:

Für jedes Element a der Liste `lst1`, gehe über alle Elemente b von `lst2`, und füge jeweils das Tupel `(a, b)` in die Ergebnisliste ein.

List Comprehensions bieten einen interessanten und eleganten Weg, um sehr komplexe Operationen platzsparend zu schreiben. Besonders möchten wir noch einmal auf die Effizienz von List Comprehensions hinweisen. So kann eine List Comprehension stets schneller ausgeführt werden als beispielsweise eine äquivalente `for`-Schleife.

Viele Probleme, in denen List Comprehensions zum Einsatz kommen, könnten auch durch die Built-in Functions `map`, `filter` oder durch eine Kombination der beiden gelöst werden, jedoch sind List Comprehensions zumeist besser lesbar und führen zu einem übersichtlicheren Quellcode.

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.3 Docstrings

In Abschnitt 5.3, »Kommentare«, wurde der sogenannte Blockkommentar eingeführt. Ein Blockkommentar wird folgendermaßen geschrieben:

```
"""
Dies ist ein Blockkommentar.
Er kann mehrere Zeilen umfassen.
"""
```

Der Name Blockkommentar wird den Möglichkeiten, die diese Notation bietet, jedoch nicht ganz gerecht. In der Python-Terminologie wird ein in drei doppelte oder einfache Hochkommata eingefasster Text *Docstring* genannt, kurz für »Documentation String«.

Docstrings sind dazu gedacht, Funktionen, Module oder Klassen zu beschreiben. Diese Beschreibungen können durch externe Tools oder beispielsweise die Built-in Function `help` gelesen und wiedergegeben werden. Auf diese Weise lassen sich sehr einfach Dokumentationen aus den – eigentlich programminternen –

Kommentaren erzeugen.

Die folgenden beiden Beispiele zeigen eine Klasse und eine Funktion jeweils mit einem Docstring dokumentiert. Beachten Sie, dass ein Docstring immer am Anfang des Funktions- bzw. Klassenkörper stehen muss, um als Docstring erkannt zu werden. Ein Docstring kann durchaus auch an anderen Stellen stehen, kann dann jedoch keiner Klasse oder Funktion zugeordnet werden und fungiert somit nur als Blockkommentar.

```
class MeineKlasse(object):
    """Beispiel fuer Docstrings.

    Diese Klasse zeigt, wie Docstrings verwendet
    werden.
    """
    pass

def MeineFunktion():
    """Diese Funktion macht nichts.

    Im Ernst, diese Funktion macht wirklich nichts.
    """
    pass
```

Um den Docstring programmintern verwenden zu können, besitzt jede Instanz ein Attribut namens __doc__, das ihren Docstring enthält. Beachten Sie, dass auch Funktionsobjekte und eingebundene Module Instanzen sind:

```
>>> print MeineKlasse.__doc__
Beispiel fuer Docstrings.
    Diese Klasse zeigt, wie Docstrings verwendet
    werden.

>>> print MeineFunktion.__doc__
Diese Funktion macht nichts.
    Im Ernst, diese Funktion macht wirklich nichts.
```

Auch ein Modul kann durch einen Docstring kommentiert werden. Der Docstring eines Moduls muss zu Beginn der entsprechenden Programmdatei stehen und ist ebenfalls über das Attribut __doc__ erreichbar. Beispielsweise kann folgendermaßen der Docstring des Moduls math der Standardbibliothek ausgelesen werden:

```
>>> import math
>>> math.__doc__
'This module is always available.  It provides access to
the\nmathematical functions defined by the C standard.'
```

Sobald Sie damit anfangen, größere Programme in Python zu realisieren, sollten Sie Funktionen, Methoden, Klassen und Module mit Docstrings versehen. Das hilft nicht nur beim Programmieren selbst, sondern auch beim späteren Erstellen einer Programmdokumentation.

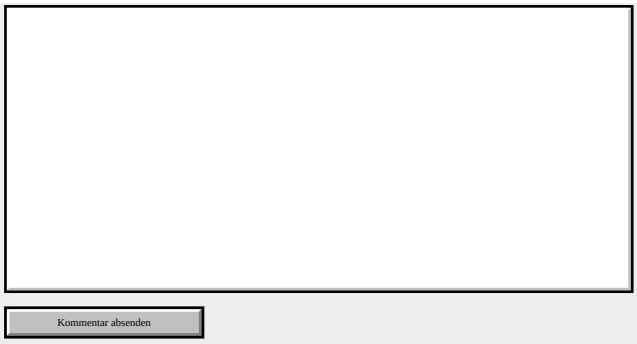
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.4 Generatoren

In diesem Abschnitt werden wir uns mit dem Konzept der Generatoren beschäftigen, die eine komfortable Möglichkeit anbieten, Reihen von Werten zu verarbeiten. Weil sich das noch sehr abstrakt anhört, wollen wir direkt mit einem Beispiel beginnen. Sie erinnern sich sicherlich noch an die Built-in Function range, die im Zusammenhang mit for-Schleifen eine wichtige Rolle spielt:

```
>>> for i in range(10):
        print i,
0 1 2 3 4 5 6 7 8 9
```

Wie wir bereits wissen, gibt range eine list-Instanz zurück, die in diesem Fall die ganzen Zahlen von 0 bis 9 enthält. Über diese Liste können wir dann mittels der for-Schleife iterieren und erhalten die Ausgabe aller Ziffern. Für kleine Listen ist dieses Vorgehen auch vollkommen angemessen, allerdings wird es höchst ineffizient, wenn man sehr große Listen durchlaufen möchte, weil erst die komplette Liste im Speicher aufgebaut

werden muss. Wir benötigen aber eigentlich pro Schleifendurchlauf nur eine einzige Zahl im Arbeitsspeicher, nämlich den aktuellen Zähler. Die gesamte Liste brauchen wir zu keiner Zeit.

Nun könnte man auf die Idee kommen, das Problem mithilfe einer `while`-Schleife zu umgehen, indem man den Zähler manuell verwaltet:

```
>>> i = 0
>>> while i < 10:
        print i,
        i += 1
```

Dieses Vorgehen, bei dem man um Pythons eigene `for`-Schleife »herumprogrammiert«, ist wenig elegant und widerspricht außerdem Pythons Grundsatz, klare und deshalb gut lesbare Programme zu schreiben. Allerdings würden weder Rechenzeit noch Speicherplatz für die Erzeugung der unnötigen Liste vergeudet.

Wir wünschen uns eine Technik, die die Lesbarkeit einer normalen `for`-Schleife mit der Effizienz der `while`-Schleife aus dem letzten Beispiel vereint.

An dieser Stelle kommen die sogenannten *Generatoren* ins Spiel. Ein Generator ist eine Funktion, die bei jedem Aufruf das nächste Element einer virtuellen Sequenz zurückgibt. Für unser Beispiel bräuchten wir also einen Generator, der nacheinander alle zehn Ziffern zurückgibt. Die Definition dieser auch *Generatorfunktionen* genannten Konstrukte ist der von normalen Funktionen sehr ähnlich. Der von uns benötigte Generator, wir nennen ihn `range_generator`, lässt sich folgendermaßen implementieren (wundern Sie sich bitte nicht über das »`yield`«, es wird im Anschluss erklärt):

```
def range_generator(max):
    i = 0
    while i < max:
        yield i
        i += 1
```

Sie werden nun sicherlich sagen, dass wir einfach die ungeschickte `while`-Schleife in eine Funktion verpackt haben und diese Lösung dadurch immer noch wenig elegant ist. Der Unterschied zu der vorhergehenden `while`-Schleife besteht aber darin, dass wir mithilfe von unserem neuen Generator namens `range_generator` elegante und gleichzeitig schnelle `for`-Schleifen schreiben können:

```
>>> for i in range_generator(10):
        print i,
0 1 2 3 4 5 6 7 8 9
```

Der Funktionsaufruf `range_generator(10)` gibt ein iterierbares Objekt (die `generator`-Instanz) zurück, das mit einer `for`-Schleife durchlaufen werden kann. Wir müssen also nur einmal mit der `while`-Notlösung arbeiten und können anschließend ganz normale `for`-Schleifen benutzen, wodurch unser Code sehr viel besser lesbar und schneller wird.

Der Knackpunkt bei Generatoren liegt in dem `yield`-Statement, mit dem wir die einzelnen Werte der virtuellen Sequenz zurückgeben. Die Syntax von `yield` unterscheidet sich dabei nicht von der des `return`-Statements und muss deshalb nicht weiter erläutert werden. Entscheidend ist, wie `yield` sich im Vergleich zu `return` auf die Verarbeitung des Programms auswirkt.

Wird in einer normalen Funktion während eines Programmlaufs ein `return` erreicht, wird der Kontrollfluss an die nächsthöhere Ebene zurückgegeben und der Funktionslauf beendet. Außerdem werden alle lokalen Variablen der Funktion wieder freigegeben.

Bei einem erneuten Aufruf der Funktion würde Python wieder ganz am Anfang der Funktion beginnen und die komplette Funktion erneut ausführen.

Im Gegensatz dazu werden beim Erreichen einer yield-Anweisung die aktuelle Position innerhalb der Generatorfunktion und ihre lokalen Variablen gespeichert, und es erfolgt ein Rücksprung in das aufrufende Programm mit dem hinter yield angegebenen Wert. Beim nächsten Iteratoraufruf macht Python dann hinter dem zuletzt ausgeführten yield weiter und kann wieder auf die alten lokalen Variablen, in dem Fall i, zugreifen. Erst wenn das Ende der Funktion erreicht wird, beginnen die endgültigen Aufräumarbeiten.

Weil die oben beschriebene Generatorfunktion range_generator äußerst oft gebraucht wird, ist sie bereits in Python unter dem Namen xrange vordefiniert. Die Funktion xrange erwartet die gleichen Parameter wie range und verhält sich auch sonst genauso – mit der einzigen Ausnahme, dass sie eben keine echte Liste erzeugt, sondern die Elemente nacheinander berechnet. Für Ihre zukünftigen for-Schleifen empfehlen wir Ihnen deshalb immer, xrange zu verwenden.

```
>>> for i in xrange(2, 5):
        print i, i*i, i**3, i**4
2 4 8 16
3 9 27 81
4 16 64 256
```

Generatoren sind sehr flexibel und können durchaus mehrere yield-Anweisungen enthalten:

```
def generator_mit_mehreren_yields():
    a = 10
    yield a
    yield a*2
    b = 5
    yield a+b
```

Auch dieser Generator kann mit einer for-Schleife durchlaufen werden:

```
>>> for i in generator_mit_mehreren_yields():
        print i,
10 20 15
```

Im ersten Iterationsschritt wird die lokale Variable a in der Generatorfunktion angelegt und ihr Wert dann mittels yield a an die Schleife übergeben. Beim nächsten Schleifendurchlauf wird dann bei yield a*2 weitergemacht, wobei die zurückgegebene 20 zeigt, dass der Wert von a tatsächlich zwischen den Aufrufen erhalten geblieben ist. Während des letzten Iterationsschritts erzeugen wir zusätzlich die lokale Variable b mit dem Wert 5 und geben die Summe von a und b an die Schleife weiter, wodurch die 15 ausgegeben wird. Da nun das Ende der Generatorfunktion erreicht ist, bricht die Schleife nach drei Durchläufen ab.

Es ist auch möglich, eine Generatorfunktion frühzeitig zu verlassen, wenn dies erforderlich sein sollte. Um dies zu erreichen, benutzt man das return-Statement ohne Rückgabewert. Der folgende Generator erzeugt abhängig vom Wert des optionalen Parameters auch_jungen eine Folge aus zwei Mädchennamen oder zwei Mädchen- und Jungennamen:

```
def namen(auch_jungen=True):
    yield "Sonja"
    yield "Lisa"
    if not auch_jungen:
        return
    yield "Florian"
    yield "Jan"
```

Mithilfe von der Built-in Function list können wir aus den Werten

des Generators eine Liste erstellen, die entweder nur "Sonja" und "Lisa" oder zusätzlich "Florian" und "Jan" enthält:

```
>>> list(namen())
['Sonja', 'Lisa', 'Florian', 'Jan']
>>> list(namen(False))
['Sonja', 'Lisa']
```

Generator Expressions

Sie erinnern sich sicherlich noch an die sogenannten List Comprehensions, mit denen Sie auf einfache Weise Listen erzeugen konnten. Mit solchen List Comprehensions konnten Sie beispielsweise eine Liste mit den ersten zehn Quadratzahlen erzeugen:

```
>>> [i*i for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Wenn wir nun die Summe dieser ersten zehn Quadratzahlen bestimmen wollten, könnten wir das mithilfe der Built-in-Function sum erreichen, indem wir schreiben:

```
>>> sum([i*i for i in range(1, 11)])
385
```

So weit, so gut. Allerdings wurde hier wieder eine nicht benötigte list-Instanz erzeugt, die Speicherplatz vergeudet.

Um auch in solchen Fällen nicht auf den Komfort von List Comprehensions verzichten zu müssen, wurden sogenannte *Generator Expressions* eingeführt. Generator Expressions sehen genauso aus wie die entsprechenden List Comprehensions, mit der Ausnahme, dass statt der eckigen Klammern [] die runden Klammern () als Begrenzung verwendet werden. Damit können wir das obige Beispiel speicherschonend mit einer Generator Expression formulieren:

```
>>> sum((i*i for i in range(1, 11)))
385
```

Die umschließenden runden Klammern können entfallen, wenn der Ausdruck sowieso schon geklammert ist. In unserem sum-Beispiel können wir also ein Klammerpaar entfernen:

```
>>> sum(i*i for i in range(1, 11))
385
```

Ein wichtiger Unterschied zu List Comprehensions betrifft die Seiteneffekte. Betrachten wir mal das folgende Beispiel, das List Comprehensions benutzt:

```
>>> x = "Ob ich wohl meinen Wert behalte?"
>>> sum([x*x for x in range(1, 11)])
385
>>> x
10
```

Wie Sie sehen, wurde der Wert der globalen Variablen x durch den Zähler der List Comprehension überschrieben. Benutzen wir einen Generator, um die Quadratzahlen zu erzeugen, ergibt sich folgendes Bild:

```
>>> x = "Ob ich wohl meinen Wert behalte?"
>>> sum(x*x for x in range(1, 11))
385
>>> x
'Ob ich wohl meinen Wert behalte?'
```

Hier wurde das globale x nicht angetastet und bei seinem

Anfangswert belassen. Sie sollten sich dieser möglichen Seiteneffekte bei List Comprehensions bewusst sein, gerade dann, wenn Sie eine vorhandene Generator Expression durch eine List Comprehension ersetzen möchten. [Dieses merkwürdige Verhalten kommt daher, dass die Python-Entwickler beim Einführen der List-Comprehension etwas unachtsam gewesen sind und solche Seiteneffekte zugelassen haben. Als man dies bemerkte, konnte man es nicht mehr ändern, da bereits geschriebene Programme durch eine entsprechende Änderung unter Umständen nicht mehr korrekt arbeiten würden. Erst mit Python 3000, das Ende 2008 erscheint, wird dieser Mangel beseitigt. Die später erst in die Sprache aufgenommenen Generator-Expressions wurden von Anfang an ohne gefährliche Seiteneffekte eingeführt.]

Generatoren können Ihnen helfen, Ihre Programme sowohl in der Lesbarkeit als auch hinsichtlich der Ausführungsgeschwindigkeit zu verbessern. Immer dann, wenn Sie es mit einer komplizierten und dadurch schlecht lesbaren `while`-Schleife zu tun haben, sollten Sie prüfen, ob ein Generator die Aufgabe nicht eleganter übernehmen kann.

Wir haben uns in diesem Kapitel auf die Definition von Generatoren und ihre Anwendung in der `for`-Schleife oder mit `list` beschränkt. Im folgenden Abschnitt werden Sie die Hintergründe und die technische Umsetzung kennenlernen, denn hinter den Generatoren und der `for`-Schleife steht das Konzept der Iteratoren.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.6 Interpreter im Interpreter

In bestimmten Fällen ist es nützlich, vom Benutzer eingegebenen oder anderweitig zur Laufzeit geladenen Python-Code aus einem Python-Programm heraus ausführen zu können. Stellen Sie sich einmal vor, Sie wollten ein Programm schreiben, das Wertetabellen für beliebige Funktionen mit einem ganzzahligen Parameter darstellen kann. Für ein solches Programm muss der Benutzer die Funktion festlegen können. Anstatt dafür eine eigene Sprache zu definieren und einen eigenen Parser und Compiler zu schreiben, bietet es sich an, Funktionsdefinitionen in Python-Syntax zu erlauben.

Mithilfe des exec-Statements können wir genau dies erreichen. Pythons exec-Anweisung erwartet einen String als Parameter, der den auszuführenden Code enthält. Alternativ kann auch ein geöffnetes Datei-Objekt an exec übergeben werden.

Um beispielsweise eine vom Benutzer eingegebene Funktion für die Ausgabe einer kleinen Wertetabelle zu benutzen, dient der folgende Code-Schnipsel:

```
print "Definieren Sie eine Funktion f mit einem
Parameter:"
definition = raw_input()
exec definition
for i in range(5):
    print "f(%d) = %f" % (i, f(i))
```

Ein Programmlauf könnte dann wie folgt aussehen:

```
Definieren Sie eine Funktion f mit einem Parameter:
def f(x): return x*x
f(0) = 0.000000
f(1) = 1.000000
f(2) = 4.000000
f(3) = 9.000000
f(4) = 16.000000
```

Wie Sie sehen, ist die Funktion `f`, die von dem Benutzer definiert wurde, nach dem Ausführen von exec im lokalen Namensraum unseres Programms verfügbar, denn wir können sie ganz normal aufrufen. Ebenso kann der Benutzer neue Variablen anlegen oder den Wert bereits bestehender Variablen auslesen, was allerdings ein Sicherheitsrisiko darstellt.

Um die Sicherheit zu erhöhen, kann man den mit exec ausgeführten Code in einem eigenen Namensraum »einsperren«. Alle neuen Variablen, Klassen und Funktionen werden in diesem gesonderten Namensraum abgelegt. Außerdem sind dem exec-Code nur noch die Variablen zugänglich, die in seinem Namensraum vorhanden sind. Ein Namensraum ist ein einfaches Dictionary, das den Referenznamen ihre Werte zuordnet. Um einem exec-Statement einen eigenen Namensraum zu geben, stellt man das Dictionary zusammen mit einem `in` hinten an:

```
>>> kontext = {"pi" : 3.1459}
>>> exec "print pi" in kontext
3.1459
```

Alle Referenzen, die innerhalb des exec-Codes definiert wurden, sind anschließend auch in dem übergebenen Kontext definiert. Damit können wir unser Einstiegsbeispiel gegen ungewollte Seiteneffekte absichern. Den Wert der Kreiszahl π wollen wir dem Benutzer auch für seine Funktionen zugänglich machen:

```
print "Definieren Sie eine Funktion f mit einem
Parameter:"
definition = raw_input()
kontext = {"pi" : 3.1459}
exec definition in kontext
for i in range(5):
    print "f(%d) = %f" % (i, kontext['f'](i))
```

Ein Beispiellauf, in dem der Benutzer eine Funktion für die Berechnung der Kreisfläche anhand des Kreisradius eingibt, sähe dann so aus:

```
Definieren Sie eine Funktion f mit einem Parameter:
def f(r): return pi * r*r
f(0) = 0.000000
f(1) = 3.145900
f(2) = 12.583600
f(3) = 28.313100
f(4) = 50.334400
```

Ausdrücke auswerten mit eval

Während mit exec beliebiger Python-Code ausgeführt werden kann, dient die Built-in Function eval dazu, Python-Ausdrücke auszuwerten und das Ergebnis zurückzugeben:

```
>>> eval("5 * 4")
20
```

Auch der von eval ausgewertete Ausdruck hat standardmäßig Zugriff auf alle Variablen des aktuellen Kontexts. Durch die beiden zusätzlichen Parameter globals und locals kann ein benutzerdefinierter Kontext festgelegt werden.

```
>>> x = 10
>>> eval("5 * x")
50
>>> eval("5 * x", {})
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    eval("5 * x", {})
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined
```

Beim ersten Aufruf von eval konnten wir auf die globale Variable x zugreifen, weil der Kontext einfach kopiert wurde. Der zweite Aufruf hingegen bekam ein leeres Dictionary als Kontext übergeben, weshalb der versuchte Zugriff auf x mit einer Exception quittiert wurde.

Die vollständige Schnittstelle von eval sieht folgendermaßen aus:

```
eval(expression[, globals[, locals]])
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.7 Geplante Sprachelemente

Die Sprache Python befindet sich in ständiger Entwicklung, und jede neue Version bringt neue Sprachelemente mit sich, die alten Python-Code unter Umständen inkompatibel zur neusten Version des Interpreters machen. Zwar geben sich die Entwickler Mühe, größtmögliche Kompatibilität zu wahren, doch ist durch das bloße Hinzufügen eines Schlüsselwortes schon derjenige Code inkompatibel geworden, der das neue Schlüsselwort als normalen Bezeichner verwendet.

Der Interpreter besitzt eine Art Modus, mit dem sich einige ausgewählte Sprachelemente der kommenden Python-Version bereits mit der aktuellen Version testen lassen. Dies soll den Wechsel von einer Version zur nächsten vereinfachen, da bereits gegen einige neue Features der nächsten Version getestet werden kann, bevor diese herausgegeben wird.

Zum Einbinden eines geplanten Features wird eine `import`-Anweisung verwendet:

```
from __future__ import sprachelement
```

Die Sprachelemente können verwendet werden, als wären sie in einem Modul namens __future__ gekapselt. Beachten Sie aber, dass Sie mit dem Modul __future__ nicht ganz so frei umgehen können, wie Sie das von anderen Modulen her gewohnt sind. Sie dürfen es beispielsweise nur am Anfang einer Programmdatei einbinden. Vor einer solchen import-Anweisung dürfen nur Kommentare, leere Zeilen oder andere *future imports* stehen.

Die folgende Anweisung erlaubt beispielsweise die Verwendung der sogenannten with-Anweisung. Das ist eine Anweisung, die in Python 2.6 eingeführt werden soll und die das Aufrufen von open- und close-Methoden bestimmter Instanzen, beispielsweise eines Dateiobjekts, automatisiert:

```
from __future__ import with_statement
```

Auf die with-Anweisung werden wir im folgenden Abschnitt näher eingehen.

Wenn das Modul __future__ selbst eingebunden wird, so enthält es unter anderem eine Liste namens all_feature_names, die die Namen aller Features enthält, die eingebunden werden können:

```
>>> import __future__
>>> __future__.all_feature_names
['nested_scopes', 'generators', 'division',
'absolute_import',
'with_statement']
```

Wir möchten hier nicht näher auf die einzelnen Features und deren Verwendung eingehen, da sie mitunter allzu speziell sind und teilweise aus älteren Python-Versionen stammen. Es ist jedoch immer interessant, ein wenig mit den geplanten Features herumzuspielen und sich selbst ein Bild davon zu machen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.8 Die with-Anweisung

Im vorherigen Abschnitt wurde die `with`-Anwendung angesprochen, mit der sich ein Codeblock in einem bestimmten Kontext ausführen lässt. Die `with`-Anweisung wird erst in Python 2.6 zum Sprachumfang gehören, kann aber mithilfe des Moduls `__future__` bereits in Python 2.5 verwendet werden.

Es gibt Operationen, die in einem bestimmten Kontext ausgeführt werden müssen und bei denen sichergestellt werden muss, dass der Kontext jederzeit korrekt deinitialisiert wird, beispielsweise auch, wenn eine Exception auftritt. Als Beispiel für einen solchen Kontext dient das Dateiobjekt. Es muss sichergestellt sein, dass die `close`-Methode des Dateiobjekts gerufen wird, selbst wenn zwischen dem Aufruf von open und dem der `close`-Methode des Dateiobjekts eine Exception geworfen wurde. Dazu ist mit den herkömmlichen Sprachelementen Pythons folgende `try/finally`-Anweisung nötig:

```
f = open("datei.txt", "r")
try:
    print f.read()
```

```
finally:
    f.close()
```

Zunächst wird eine Datei namens *datei.txt* zum Lesen geöffnet. Mithilfe der darauf folgenden `try/finally`-Anweisung wird sichergestellt, dass `f.close` in jedem Fall aufgerufen wird. Dieses Beispiel lässt sich mit der `with`-Anweisung folgendermaßen formulieren. Beachten Sie, dass die erste Zeile ab Python 2.6 nicht mehr benötigt wird.

```
from __future__ import with_statement
with open("programm.py", "r") as f:
    print f.read()
```

Die `with`-Anweisung besteht aus dem Schlüsselwort `with`, gefolgt von einer Instanz. Optional kann auf die Instanz das Schlüsselwort as und ein Bezeichner folgen. Dieser Bezeichner wird *Target* genannt, und seine Bedeutung hängt von der verwendeten Instanz ab. Im obigen Beispiel referenziert `f` das geöffnete Dateiobjekt. Um zu verstehen, was bei einer `with`-Anweisung genau passiert, soll im nächsten Beispiel eine eigene Klasse definiert werden, die sich mit der `with`-Anweisung verwenden lässt. Eine solche Klasse wird *Kontextmanager* genannt.

Die Klasse `MeinLogfile` ist dafür gedacht, eine rudimentäre Logdatei zu führen. Dazu implementiert sie die Funktion `eintrag`, die eine neue Zeile in die Logdatei schreibt. Die Klassendefinition sieht folgendermaßen aus:

```
class MeinLogfile(object):

    def __init__(self, logfile):
        self.logfile = logfile
        self.f = None

    def eintrag(self, text):
        self.f.write("==>%s\n" % text)

    def __enter__(self):
        self.f = open(self.logfile, "w")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.f.close()
```

Zu den beiden ersten Methoden der Klasse ist nicht viel zu sagen. Dem Konstruktor `__init__` wird der Dateiname der Logdatei übergeben, der intern im Attribut `self.logfile` gespeichert wird. Zusätzlich wird das Attribut `self.f` angelegt, das später das geöffnete Dateiobjekt referenzieren soll.

Die Methode `eintrag` hat die Aufgabe, den übergebenen Text in die Logdatei zu schreiben. Dazu ruft sie einfach die Methode `write` des Dateiobjekts auf. Beachten Sie, dass die Methode `eintrag` nur innerhalb einer `with`-Anweisung aufgerufen werden kann, da das Dateiobjekt erst in den folgenden Magic Functions geöffnet und geschlossen wird.

Die angesprochenen Magic Functions `__enter__` und `__exit__` sind das Herzstück der Klasse und müssen implementiert werden, wenn die Klasse im Zusammenhang mit `with` verwendet werden soll. Die Methode `__enter__` wird aufgerufen, wenn der Kontext aufgebaut, also bevor der Körper der `with`-Anweisung ausgeführt wird. Die Methode bekommt keine Parameter, gibt aber einen Wert zurück. Der Rückgabewert von `__enter__` wird später vom Target-Bezeichner referenziert, sofern einer angegeben wurde. Im Falle unserer Beispielklasse wird die Datei `self.logfile` zum Schreiben geöffnet und mit `return self` eine Referenz auf die eigene Instanz zurückgegeben.

Die zweite Magic Function `__exit__` wird aufgerufen, wenn der Kontext verlassen wird, also nachdem der Körper der `with`-Anweisung entweder vollständig durchlaufen ist oder durch eine Exception vorzeitig abgebrochen wurde. Im Falle der

Beispielklasse wird das geöffnete Dateiobjekt `self.f` geschlossen. Näheres zu den drei Parametern der Methode `__exit__` folgt weiter unten.

Die soeben erstellte Klasse `MeinLogfile` lässt sich folgendermaßen mit `with` verwenden:

```
from __future__ import with_statement
inst = MeinLogfile("logfile.txt")
with inst as log:
    log.eintrag("Hallo Welt")
    log.eintrag("Na, wie gehts?")
```

Zur Erklärung: Zunächst wird eine Instanz der Klasse `MeinLogfile` erstellt und dabei der Dateiname *logfile.txt* übergeben. Die `with`-Anweisung bewirkt als Erstes, dass die Methode `__enter__` der Instanz `inst` ausgeführt und ihr Rückgabewert durch `log` referenziert wird. Dann wird der Körper der `with`-Anweisung ausgeführt, in dem insgesamt zweimal die Methode `eintrag` aufgerufen und damit Text in die Logdatei geschrieben wird. Nachdem der Anweisungskörper ausgeführt worden ist, wird einmalig die Methode `__exit__` der Instanz `inst` aufgerufen.

Im Folgenden sollen die Magic Functions `__enter__` und `__exit__` vollständig erläutert werden.

`__enter__(self)`

Diese Magic Function wird einmalig zum Öffnen des Kontexts aufgerufen, bevor der Körper der `with`-Anweisung ausgeführt wird. Der Rückgabewert dieser Methode wird im Körper der `with`-Anweisung vom Target-Bezeichner referenziert.

`__exit__(self, exc_type, exc_value, traceback)`

Die Magic Function `__exit__` wird einmalig zum Schließen des Kontexts aufgerufen, nachdem der Körper der `with`-Anweisung ausgeführt worden ist. Die drei Parameter *exc_type*, *exc_value* und *traceback* spezifizieren Typ, Wert und Traceback-Objekt einer eventuell innerhalb des `with`-Anweisungskörpers geworfenen Exception. Wenn keine Exception geworfen wurde, referenzieren alle drei Parameter `None`. Wie mit einer geworfenen Exception weiter verfahren wird, kann mit dem Rückgabewert der Methode `__exit__` gesteuert werden: Gibt die Methode `True` zurück, wird die Exception unterdrückt. Bei einem Rückgabewert von `False` wird die Exception erneut geworfen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück	<top>	vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.9 Function Decorator

Aus dem Kapitel über objektorientierte Programmierung kennen Sie sicherlich noch die Built-in Function `staticmethod`, die folgendermaßen verwendet wurde:

```
class MeineKlasse(object):

    def methode():
        pass

    methode2 = staticmethod(methode)
```

Durch diese Schreibweise wird zunächst eine Methode angelegt und nachher durch die Built-in Function `staticmethod` modifiziert. Die angelegte Methode wird dann mit dem modifizierten Funktionsobjekt überschrieben.

Diese Art, `staticmethod` anzuwenden, ist zwar richtig und funktioniert, ist aber gleichzeitig auch unidiomatisch und nicht gerade gut lesbar. Aus diesem Grund unterstützt Python eine eigene Notation, um den obigen Code lesbarer zu gestalten. Das

folgende Beispiel ist zu dem vorherigen äquivalent:

```
class MeineKlasse(object):
    @staticmethod
    def methode():
        pass
```

Die Funktion, die die angelegte Methode modifizieren soll, wird nach einem @–Zeichen vor die Methodendefinition geschrieben. Eine solche Notation wird *Function Decorator* genannt. Allerdings sind Function Decorators nicht auf den Einsatz mit `staticmethod` beschränkt, vielmehr können Sie beliebige Decorators erstellen. Auf diese Weise kann eine Funktion durch bloßes Hinzufügen eines Decorators um eine gewisse Funktionalität erweitert werden.

Function Decorator können nicht nur auf Methoden angewendet werden, sondern genauso auf Funktionen. Zudem können sie ineinander verschachtelt werden, wie folgendes Beispiel zeigt:

```
@dec1
@dec2
def funktion():
    pass
```

Diese Funktionsdefinition ist äquivalent zu folgendem Code:

```
def funktion():
    pass
funktion = dec1(dec2(funktion))
```

Es erübrigt sich zu sagen, dass sowohl dec1 als auch dec2 implementiert werden müssen, bevor die Beispiele lauffähig sind.

Das jetzt folgende Beispiel soll einen interessanten Ansatz zum Cachen (dt. *Zwischenspeichern*) von Funktionsaufrufen zeigen, sodass die Ergebnisse von komplexen Berechnungen automatisch gespeichert werden. Diese können dann beim nächsten Funktionsaufruf mit den gleichen Parametern wiedergegeben werden, ohne die Berechnungen erneut durchführen zu müssen. Das Caching einer Funktion soll allein durch Angabe eines Function Decorators erledigt werden, also ohne in die Funktion selbst einzugreifen, und zudem mit beliebigen Funktionsschnittstellen, also beliebigen Funktionen arbeiten können. Dazu sehen wir uns zunächst einmal die Definition der Berechnungsfunktion an, die in diesem Fall die Fakultät einer ganzen Zahl berechnet, inklusive Function Decorator:

```
@CacheDecorator()
def fak(n):
    ergebnis = 1
    for i in xrange(2, n+1):
        ergebnis *= i
    return ergebnis
```

Die Berechnung einer Fakultät sollte Ihnen inzwischen geläufig sein. Was hier allerdings interessant ist, ist der Function Decorator, denn es handelt sich hierbei nicht um eine Funktion, sondern um eine Klasse namens `CacheDecorator`, die im Decorator instanziiert wird. Sie erinnern sich sicherlich, dass eine Klasse durch Implementieren der Magic Function `__call__` aufrufbar gemacht werden kann und sich damit wie ein Funktionsobjekt verhält. Wir müssen diesen Umweg gehen, da wir die Ergebnisse der Berechnungen so speichern müssen, dass sie auch in späteren Aufrufen des Decorators noch verfügbar sind. Das ist mit einer Funktion nicht möglich, wohl aber mit einer Klasse. Die Definition der Decorator-Klasse sieht folgendermaßen aus:

```
class CacheDecorator(object):
```

```
    def __init__(self):
        self.cache = {}
        self.func = None

    def cachedFunc(self, *args):
        if args not in self.cache:
            self.cache[args] = self.func(*args)
        return self.cache[args]

    def __call__(self, func):
        self.func = func
        return self.cachedFunc
```

Im Konstruktor der Klasse CacheDecorator wird ein leeres Dictionary für die zwischengespeicherten Werte angelegt. Neben dem Konstruktor ist unter anderem die Methode __call__ implementiert. Durch diese Methode werden Instanzen der Klasse aufrufbar, können also wie ein Funktionsobjekt verwendet werden. Um als Function Decorator verwendet werden zu können, muss die Methode __call__ ein Funktionsobjekt als Parameter akzeptieren und ein Funktionsobjekt zurückgeben, das fortan als veränderte Version der ursprünglich angelegten Funktion mit dieser assoziiert wird. In diesem Fall gibt __call__ das Funktionsobjekt der Methode cachedFunc zurück.

Die Methode cachedFunc soll also fortan anstelle der ursprünglich angelegten Funktion aufgerufen werden. Damit sie ihre Aufgabe erledigen kann, hat sie Zugriff auf das Funktionsobjekt der eigentlichen Funktion, das von dem Attribut self.func referenziert wird. Die Methode cachedFunc akzeptiert beliebig viele *positional arguments*, da sie später für beliebige Funktionen und damit beliebige Funktionsschnittstellen arbeiten muss. Diese Argumente sind innerhalb der Methode als Tupel verfügbar.

Jetzt wird geprüft, ob das Tupel mit den übergebenen Argumenten bereits als Schlüssel im Dictionary self.cache existiert. Wenn ja, wurde die Funktion bereits mit exakt den gleichen Argumenten aufgerufen und der im Cache gespeicherte Rückgabewert kann direkt zurückgegeben werden. Wenn der Schlüssel nicht vorhanden ist, wird die Berechnungsfunktion self.func mit den übergebenen Argumenten aufgerufen und das Ergebnis im Cache gespeichert. Anschließend wird es zurückgegeben.

Um zu testen, ob das Speichern der Werte funktioniert, wird das Beispiel um zwei Ausgaben erweitert, je nachdem, ob ein Ergebnis neu berechnet oder aus dem Cache geladen wurde. Und tatsächlich, es funktioniert:

```
>>> fak(10)
Ergebnis berechnet
3628800
>>> fak(20)
Ergebnis berechnet
2432902008176640000L
>>> fak(20)
Ergebnis geladen
2432902008176640000L
>>> fak(10)
Ergebnis geladen
3628800
```

Wie Sie sehen, wurden die ersten beiden Ergebnisse berechnet, während die letzten beiden aus dem internen Cache geladen wurden. Diese Form des Cachings könnte je nach Anwendungsbereich und Komplexität der Berechnung erhebliche Geschwindigkeitsvorteile bieten.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.10 assert

Mithilfe des Schlüsselworts `assert` lassen sich Konsistenzabfragen in ein Python-Programm integrieren. Durch das Schreiben einer `assert`-Anweisung legt der Programmierer eine Bedingung fest, die für die Ausführung des Programms essenziell ist und die bei Erreichen der `assert`-Anweisung zu jeder Zeit `True` ergeben muss. Wenn die Bedingung einer `assert`-Anweisung `False` ergibt, wird eine `AssertionError`-Exception geworfen. In der folgenden Sitzung im interaktiven Modus wurden mehrere `assert`-Anweisungen eingegeben:

```
>>> import math
>>> assert math.log(1) == 0
>>> assert math.sqrt(4) == 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert math.sqrt(9) == 3
>>>
```

Die `assert`-Anweisung ist damit ein wichtiges Hilfsmittel zum Aufspüren von Fehlern und ermöglicht es, den Programmlauf zu

beenden, wenn bestimmte Voraussetzungen nicht gegeben sind. Häufig prüft man an Schlüsselstellen im Programm mittels `assert`, ob alle Referenzen die erwarteten Werte referenzieren, um eventuelle Fehlberechnungen rechtzeitig und umfassend erkennen zu können.

Beachten Sie, dass `assert`-Anweisungen üblicherweise nur während der Entwicklung eines Programms benötigt werden und in einem fertigen Programm eher stören würden. Von daher werden `assert`-Anweisungen nur dann ausgeführt, wenn die globale Konstante `__debug__` `True` referenziert. Diese Konstante referenziert nur dann `False`, wenn der Interpreter mit der Kommandozeilenoption `-O` gestartet wurde. Wenn die Konstante `__debug__` `False` referenziert, werden `assert`-Anweisungen ignoriert und haben damit keinen Einfluss mehr auf die Laufzeit Ihres Programms.

Beachten Sie, dass Sie den Wert von `__debug__` im Programm selbst nicht verändern dürfen, sondern nur über die Kommandozeilenoption `-O` bestimmen können, ob `assert`-Anweisungen ausgeführt oder ignoriert werden sollen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.11 Weitere Aspekte der Syntax ▼

Das Thema dieses Abschnitts sollen kleinere Aspekte der Python-Syntax sein, die bisher vernachlässigt wurden. Allgemein gilt, dass die hier besprochenen Notationen keineswegs notwendig oder unumgänglich sind. Entscheiden Sie ganz nach Ihren Vorlieben, ob und in welchem Umfang Sie diese einsetzen möchten.

top

13.11.1 Umbrechen langer Zeilen ▼▲

Sicherlich haben Sie bereits einige eigene Python-Programme geschrieben und dabei ist die ein oder andere recht lange Quellcodezeile entstanden. Viele Programmierer beschränken die Länge ihrer Quellcodezeilen, damit beispielsweise mehrere Quellcodedateien nebeneinander auf den Bildschirm passen oder der Code auch auf Geräten mit einer festen Zeichenbreite angenehm zu lesen ist. Eine geläufige maximale Zeilenlänge ist 80 Zeichen. Doch welche Möglichkeiten bietet Python, überlange

Zeilen umzubrechen, sodass eine maximale Zeilenlänge eingehalten werden kann?

Sie wissen bereits, dass Sie Ihren Quellcode innerhalb von Klammern beliebig umbrechen dürfen, doch an vielen anderen Stellen sind Sie an die strengen syntaktischen Regeln von Python gebunden. Durch Einsatz der Backslash-Notation ist es möglich, Quellcode an nahezu beliebigen Stellen in eine neue Zeile umzubrechen:

```
>>> var \
... = \
... 10
>>> var
10
```

Grundsätzlich kann ein Backslash überall da stehen, wo auch ein Leerzeichen hätte stehen können. Somit ist auch ein Backslash innerhalb eines Strings möglich:

```
>>> "Hallo \
... Welt"
'Hallo Welt'
```

Beachten Sie dabei aber, dass eine Einrückung des umgebrochenen Teils des Strings Leerzeichen in den String schreibt. Aus diesem Grund sollten Sie folgende Variante, um einen String in mehrere Zeilen zu schreiben, vorziehen:

```
>>> "Hallo " \
... "Welt"
'Hallo Welt'
```

Allgemein kann die Backslash-Notation die Lesbarkeit des Quellcodes sowohl vermindern als auch, beispielsweise bei sehr langen Strings, erhöhen. Grundsätzlich sollten Sie nach Möglichkeit versuchen, lesbaren Code zu erzeugen, was unter anderem bedeutet, den Backslash nicht im Übermaß zu verwenden.

top

13.11.2 Zusammenfügen mehrerer Zeilen ▼▲

Genau so, wie Sie eine einzeilige Anweisung mithilfe des Backslashs auf mehrere Zeilen umbrechen können, können Sie mehrere einzeilige Anweisungen in eine Zeile zusammenfassen. Dazu werden die Anweisungen durch ein Semikolon voneinander getrennt:

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

Anweisungen, die aus einem Anweisungskopf und einem Anweisungskörper bestehen, können auch ohne Einsatz eines Semikolons in eine Zeile gefasst werden, sofern der Anweisungskörper selbst aus nicht mehr als einer Zeile besteht:

```
>>> x = True
>>> if x: print "Hallo Welt"
...
Hallo Welt
```

Sollte der Anweisungskörper mehrere Zeilen lang sein, so können diese selbstverständlich durch ein Semikolon zusammengefasst werden:

```
>>> x = True
>>> if x: print "Hallo"; print "Welt"
...
Hallo
```

```
Welt
```

Alle durch ein Semikolon zusammengefügten Anweisungen werden so behandelt, als wären sie gleich weit eingerückt. Allein ein Doppelpunkt vermag die Einrückungstiefe zu vergrößern. Aus diesem Grund gibt es im obigen Beispiel keine Möglichkeit, in derselben Zeile eine Anweisung zu schreiben, die nicht mehr im Körper der `if`-Anweisung steht.

Beachten Sie, dass beim Einsatz des Backslashs und vor allem des Semikolons schnell unleserlicher Code geschrieben wird. Verwenden Sie beide Notationen daher nur, wenn Sie meinen, dass es der Lesbarkeit und Übersichtlichkeit dienlich ist.

top

13.11.3 String conversions ▲

Python unterstützt eine spezielle Syntax, die es ermöglicht, eine String-Repräsentation des Wertes einer beliebigen Instanz zu erzeugen. Dazu wird ein entsprechendes Literal oder eine Referenz innerhalb von »umgekehrten Anführungszeichen« geschrieben. Diese Notation wird *string conversions* genannt.

```
>>> `12`
'12'
>>> `"Hallo Welt"`
"'Hallo Welt'"
>>> `[1,2,3]`
'[1, 2, 3]'
```

String conversions haben den gleichen Zweck wie die Built-in Function `repr` und werden in heutigen Programmen so gut wie nie verwendet. Der Grund besteht darin, dass das »umgekehrte Anführungszeichen« auf vielen Tastaturen, unter anderem der deutschen, eine Akzenttaste ist oder gar ganz fehlt. Aus diesem Grund sollten Sie stets die Built-in Functions `repr` bzw. `str` verwenden, wenn Sie eine Instanz in einen String konvertieren wollen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

14.2 Zufallszahlengenerator – random

Das Modul `random` der Standardbibliothek erzeugt Pseudozufallszahlen und bietet zudem einige zusätzliche Funktionen, um zufallsgesteuerte Operationen auf Basisdatentypen anzuwenden.

Beachten Sie, dass das Modul `random` keine echten Zufallszahlen erzeugen kann, sondern sogenannte Pseudozufallszahlen. Echte Zufallszahlen sind für einen Computer nicht berechenbar. Ein Generator für Pseudozufallszahlen wird mit einer ganzen Zahl initialisiert und erzeugt aufgrund dieser Basis eine deterministische, aber scheinbar zufällige Abfolge von Pseudozufallszahlen. Diese Zahlenfolge wiederholt sich dabei nach einer gewissen Anzahl von erzeugten Zufallszahlen. Im Falle des in Python standardmäßig verwendeten Algorithmus beträgt diese Periode $2^{19937} - 1$ Zahlen.

Bevor Sie die Beispiele dieses Abschnitts ausprobieren können, müssen Sie selbstverständlich das Modul `random` einbinden:

```
>>> import random
```

Steuerungsfunktionen

random.seed([x])

Initialisiert den Zufallszahlengenerator mit der Instanz *x*. Wenn es sich bei *x* um eine ganze Zahl handelt, wird der Zufallszahlengenerator direkt mit dieser Zahl, ansonsten mit dem Hash-Wert der übergebenen Instanz initialisiert.

Wenn kein Parameter übergeben wird, wird der Zufallszahlengenerator mit der aktuellen Systemzeit initialisiert. Auf diese Weise können die erzeugten Zahlen als quasi-zufällig angesehen werden.

Wenn der Zufallszahlengenerator zu unterschiedlichen Zeiten mit demselben Wert initialisiert wird, erzeugt er jeweils dieselbe Zahlenfolge.

random.getstate()

Die Funktion `getstate` gibt ein Tupel zurück, das den aktuellen Status des Zufallszahlengenerators beschreibt. Mithilfe der Funktion `setstate` lässt sich damit der Status des Generators speichern und zu einem späteren Zeitpunkt, beispielsweise nach zwischenzeitlicher Neuinitialisierung, wiederherstellen.

random.setstate()

Die Funktion `setstate` akzeptiert ein von `getstate` erzeugtes Tupel und überführt den Zufallszahlengenerator in den durch dieses Tupel beschriebenen Status.

```
>>> state = random.getstate()
>>> random.setstate(state)
```

random.jumpahead(n)

Ändert den Initialisierungszustand des Zufallszahlengenerators. Die ganze Zahl *n* wird dazu verwendet, den internen Zustand zu durchmischen. Beachten Sie, dass `jumpahead` auf einem deterministischen Algorithmus basiert, dass also ein Aufruf von `jumpahead` aus dem gleichen internen Zustand heraus und mit dem gleichen Parameter *n* immer zu dem gleichen neuen internen Zustand des Zufallszahlengenerators führen wird.

```
>>> random.seed(1234)
>>> random.jumpahead(567)
```

random.getrandbits(k)

Erzeugt eine ganze Zahl, deren Bitfolge aus *k* zufälligen Bits besteht. Das Ergebnis ist, unabhängig von der verwendeten Bitzahl, immer eine Instanz des Datentyps `long`.

```
>>> random.getrandbits(8)
149L
>>> random.getrandbits(8)
187L
```

Funktionen für ganze Zahlen

random.randrange([start,]stop[, step])

Gibt ein zufällig gewähltes Element der Liste zurück, die ein Aufruf der Built-in Function `range` mit gleichen Parametern erzeugen würde. Das heißt, es wird eine Zufallszahl *n* zwischen *start* und *stop* erzeugt, für die gilt: $start + n \cdot step$.

```
>>> random.randrange(0, 50, 2)
40
```

random.randint(a, b)

Erzeugt eine zufällige, ganze Zahl *n*, sodass gilt: $a \leq n \leq b$.

```
>>> random.randint(0, 10)
2
>>> random.randint(0, 10)
7
```

Funktionen für Sequenzen

random.choice(seq)

Gibt ein zufällig gewähltes Element der Sequenz *seq* zurück. Die übergebene Sequenz darf nicht leer sein.

```
>>> random.choice([1,2,3,4,5])
5
>>> random.choice([1,2,3,4,5])
2
```

Im Beispiel wurde der Einfachheit halber eine Liste mit ausschließlich numerischen Elementen verwendet. Dies muss nicht unbedingt sein, es darf ein beliebiger sequenzieller Datentyp mit beliebigen Elementen übergeben werden.

random.shuffle(x[, random])

Die Funktion `shuffle` bringt die Elemente der Sequenz *x* in eine zufällige Reihenfolge. Beachten Sie, dass diese Funktion nicht seiteneffektfrei ist, sondern die übergebene Sequenz an sich bearbeitet wird. Aus diesem Grund dürfen für *x* auch nur Instanzen veränderlicher sequenzieller Datentypen übergeben werden.

Als optionaler Parameter *random* kann ein Funktionsobjekt übergeben werden, das über die gleiche Schnittstelle verfügt wie die Funktion `random.random`, die später beschrieben wird. Durch Implementieren einer solchen Funktion ist es möglich, `shuffle` einen eigenen Zufallszahlengenerator vorzugeben.

```
>>> l = [1,2,3,4]
>>> random.shuffle(l)
>>> l
[1, 4, 3, 2]
```

random.sample(population, k)

Die Funktion `sample` bekommt eine Sequenz *population* und eine ganze Zahl *k* als Parameter übergeben. Das Ergebnis ist eine neue Liste mit *k* zufällig gewählten Elementen aus *population*. Auf diese Weise könnte beispielsweise eine gewisse Anzahl von Gewinnern aus einer Liste von Lotterieteilnehmern gezogen werden. Beachten Sie, dass auch die Reihenfolge der erzeugten Liste zufällig ist und die Ziehungen bei mehrmaligem Funktionsaufruf mit Wiederholungen durchgeführt werden.

```
>>> pop = [1,2,3,4,5,6,7,8,9,10]
>>> random.sample(pop, 3)
[7, 8, 5]
>>> random.sample(pop, 3)
[5, 9, 7]
```

Die Funktion `sample` kann insbesondere auch in Kombination mit der Built-in Function `xrange` verwendet werden:

```
>>> random.sample(xrange(10000000), 3)
[4571575, 2648561, 2009814]
```

Spezielle Verteilungen

random.random()

Gibt die nächste Zufallszahl zurück. Der Rückgabewert ist eine Gleitkommazahl zwischen `0.0` und `1.0`.

```
>>> random.random()
0.067300272273646655
>>> random.random()
0.52544342703734148
```

random.uniform(a, b)

Erzeugt eine gleichverteilte zufällige Gleitkommazahl *n*, sodass gilt: $a \leq n < b$.

```
>>> random.uniform(0.5, 0.6)
0.5618673220051662
```

random.betavariate(alpha, beta)

Erzeugt Zufallszahlen, die statistisch der Betaverteilung entsprechen. Die Parameter `alpha` und `beta` müssen numerische Werte größer als `-1` sein, und der Rückgabewert liegt zwischen `0` und `1`.

```
>>> random.betavariate(2.5, 1.0)
0.76494009914551264
```

random.expovariate(lambd)

Erzeugt Zufallszahlen, die statistisch der Exponentialverteilung entsprechen. Der Parameter `lambd` ist `1.0` geteilt durch das gewünschte arithmetische Mittel. Der Rückgabewert liegt zwischen `0` und positiv unendlich.

```
>>> random.expovariate(0.5)
0.85259287178065613
```

random.gammavariate(alpha, beta)

Erzeugt Zufallszahlen, die statistisch der Gammaverteilung entsprechen. Die Parameter *alpha* und *beta* müssen numerische Werte größer als `0` sein.

```
>>> random.gammavariate(1.3, 0.5)
1.1608977325106138
```

random.gauss(mu, sigma)

Erzeugt Zufallszahlen, die statistisch der Gaußverteilung entsprechen. Der Parameter *mu* entspricht dem arithmetischen Mittel und *sigma* der Standardabweichung.

```
>>> random.gauss(0.5, 1.9)
1.0084579933596225
```

random.lognormvariate(mu, sigma)

Erzeugt Zufallszahlen, die statistisch der logarithmischen Normalverteilung entsprechen. Der Parameter *mu* entspricht dem arithmetischen Mittel und *sigma* der Standardabweichung.

```
>>> random.lognormvariate(0.5, 1.9)
0.25625006871810202
```

random.normalvariate(mu, sigma)

Erzeugt Zufallszahlen, die statistisch der Normal- oder Gaußverteilung entsprechen. Der Parameter *mu* entspricht dem arithmetischen Mittel und *sigma* der Standardabweichung. Die Funktion ist damit äquivalent zu `gauss`.

```
>>> random.normalvariate(0.5, 1.9)
```

```
1.9176550196262139
```

random.vonmisesvariate(mu, kappa)

Erzeugt Zufallszahlen, die statistisch der von-Mises-Verteilung entsprechen. Der Parameter *mu* entspricht dem mittleren Winkel in Radiant und *kappa* dem Konzentrationsparameter, der größer oder gleich `0` sein muss.

```
>>> random.vonmisesvariate(0.5, 1.9)
2.0502913847498458
```

random.paretovariate(alpha)

Erzeugt Zufallszahlen, die statistisch der Pareto-Verteilung entsprechen.

```
>>> random.paretovariate(0.5)
43.528372368738189
```

random.weibullvariate(alpha, beta)

Erzeugt Zufallszahlen, die statistisch der Weibull-Verteilung entsprechen.

```
>>> random.weibullvariate(0.5, 1.9)
0.23610339261628124
```

Alternative Generatoren

random.SystemRandom([seed])

Das Modul `random` enthält zusätzlich zu den oben erläuterten Funktionen eine Klasse namens `SystemRandom`, die es ermöglicht, den Zufallszahlengenerator des Betriebssystems zu verwenden statt den Python-eigenen. Beachten Sie, dass diese Klasse nicht auf allen, aber auf den gängigsten Betriebssystemen existiert.

Beim Instanziieren der Klasse kann eine Zahl oder Instanz zur Initialisierung des Zufallszahlengenerators übergeben werden. Danach kann die Klasse `SystemRandom` wie das Modul `random` verwendet werden, da sie die meisten im Modul enthaltenen Funktionen als Methode implementiert.

Beachten Sie jedoch, dass nicht die komplette Funktionalität von `random` in `SystemRandom` zur Verfügung steht. So werden Aufrufe der Methoden `seed` und `jumpahead` ignoriert, während Aufrufe der Methoden `getstate` und `setstate` eine `NotImplementedError`-Exception werfen.

```
>>> sr = random.SystemRandom()
>>> sr.randint(1, 10)
9
```

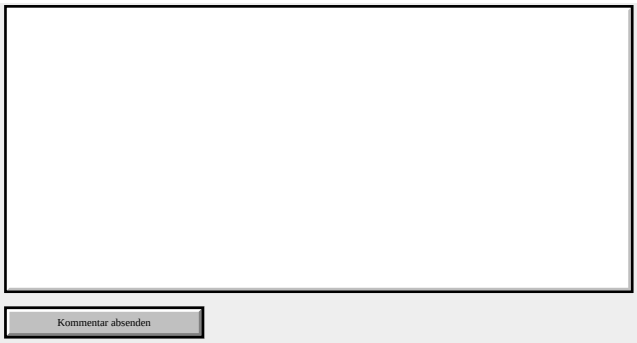
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

14.3 Präzise Dezimalzahlen – decimal

Sicherlich erinnern Sie sich noch an folgendes Beispiel, das zeigen sollte, dass der eingebaute Datentyp `float` nicht unendlich präzise ist:

```
>>> 0.9
0.90000000000000002
```

Das liegt daran, dass nicht jede Dezimalzahl durch das interne Speichermodell von `float` dargestellt werden kann, sondern nur mit einer gewissen Genauigkeit angenähert wird. Diese Einschränkung wird jedoch aus Gründen der Effizienz in Kauf genommen. Als wir über Gleitkommazahlen gesprochen haben, wurde Abhilfe durch ein Modul versprochen, und dieses Modul heißt `decimal`. Es muss aber noch einmal deutlich darauf hingewiesen werden, dass diese Abhilfe auf Kosten der Performance geht.

Das Modul `decimal` enthält im Wesentlichen den Datentyp `Decimal`, der Dezimalzahlen mit einer beliebigen Präzision speichern und verarbeiten kann. In diesem Abschnitt möchten wir Sie in die Verwendung des Datentyps einführen, die sich an die Verwendung der vorhandenen numerischen Datentypen anlehnt. Um die Beispiele auszuführen, müssen Sie den Datentyp zuerst einbinden:

```
>>> from decimal import Decimal
```

Hinweis

Das hier besprochene Modul `decimal` folgt in seiner Funktionsweise der *General Decimal Arithmetic Specification* von IBM. Aus diesem Grund ist es möglich, dass Ihnen ein ähnliches Modul bereits von einer anderen Programmiersprache her bekannt ist.

Es existieren beispielsweise Bibliotheken, die das decimal-Modul in gleicher oder abgewandelter Form für C, C++, Java oder Perl implementieren.

top

14.3.1 Verwendung des Datentyps

Es existiert kein Literal, mit dem Sie Instanzen des Datentyps Decimal direkt erzeugen könnten, wie es beispielsweise bei float der Fall ist. Um eine Decimal-Instanz mit einem bestimmten Wert zu erzeugen, müssen Sie den Datentyp explizit instanziieren. Der Wert kann dem Konstruktor in Form eines Strings übergeben werden.

```
>>> Decimal("0.9")
Decimal("0.9")
>>> Decimal("1.33e7")
Decimal("1.33E+7")
```

Dies ist die geläufigste Art, Decimal zu instanziieren. Es ist außerdem möglich, dem Konstruktor eine ganze Zahl oder ein Tupel zu übergeben:

```
>>> Decimal(123)
Decimal("123")
>>> Decimal((0, (3, 1, 4, 1), -3))
Decimal("3.141")
```

Im zweiten Fall bestimmt das erste Element des Tupels das Vorzeichen, wobei 0 für eine positive und 1 für eine negative Zahl steht. Das zweite Element muss ein weiteres Tupel sein, das alle Ziffern der Zahl enthält. Das dritte Element des Tupels entspricht dem Exponenten der zuvor angegebenen Zahl.

Beachten Sie, dass es ausdrücklich nicht möglich ist, bei der Instanziierung eine Gleitkommazahl direkt zu übergeben, da sich sonst die Ungenauigkeiten von float auf den Datentyp Decimal übertragen würden.

Sobald eine Decimal-Instanz erzeugt wurde, kann diese wie eine Instanz eines bekannten numerischen Datentyps verwendet werden. Das bedeutet insbesondere, dass alle von diesen Datentypen her bekannten Operatoren auch für Decimal definiert sind. Es ist zudem möglich, Decimal in Operationen mit anderen numerischen Datentypen zu verwenden. Kurzum: Decimal passt sich nahezu perfekt in die bestehende Welt der numerischen Datentypen ein.

```
>>> Decimal("0.9") * 5
Decimal("4.5")
>>> Decimal("0.9") / 10
Decimal("0.09")
>>> Decimal("0.9") % Decimal("1.0")
Decimal("0.9")
```

Eine Besonderheit des Datentyps ist es, abschließende Nullen beim Nachkommaanteil einer Dezimalzahl beizubehalten, obwohl diese eigentlich überflüssig sind. Das ist beispielsweise beim Rechnen mit Geldbeträgen von Nutzen:

```
>>> Decimal("2.50") + Decimal("4.20")
Decimal("6.70")
```

Ein Decimal-Wert lässt sich in einen Wert eines beliebigen anderen numerischen Datentyps überführen. Beachten Sie, dass solche Konvertierungen im Falle von Decimal in der Regel verlustbehaftet sind, der Wert also an Genauigkeit verliert.

```
>>> float(Decimal("1.337"))
1.337
>>> float(Decimal("0.9"))
0.90000000000000002
>>> int(Decimal("1.337"))
1
```

Diese Eigenschaft ermöglicht es, Decimal-Instanzen ganz selbstverständlich als Parameter von beispielsweise Built-in Functions oder Funktionen der Bibliothek math zu übergeben.

```
>>> import math
>>> math.sqrt(Decimal("2"))
1.4142135623730951
```

Beachten Sie dabei, dass von diesen Funktionen auch in einem solchen Fall niemals eine Decimal-Instanz zurückgegeben wird. Unter Verwendung des Moduls math laufen Sie also Gefahr, durch den float-Rückgabewert an Genauigkeit zu verlieren.

Diese Beschränkung lässt sich bei vielen mathematischen Operationen durch Verwendung der entsprechenden Operatoren umgehen, da diese das Ergebnis in jedem Fall als Decimal-Instanz zurückgeben. Als einzige mathematische Funktion ist, zusätzlich zu den Operatoren, für den Datentyp Decimal die Methode sqrt zum Ziehen der Quadratwurzel definiert:

```
>>> d = Decimal("9")
>>> d.sqrt()
Decimal("3")
```

Tipp

Das Programmieren mit dem Datentyp Decimal ist mit viel Schreibarbeit verbunden, da kein Literal für diesen Datentyp existiert. Viele Python-Programmierer behelfen sich damit, dem Datentyp einen kürzeren Namen zu verpassen:

```
>>> from decimal import Decimal as D >>> D("1.5e-7")
Decimal("1.5E-7")
```

top

14.3.2 Nichtnumerische Werte

Aus Abschnitt 8.3.2 kennen Sie bereits die Werte nan und inf des Datentyps float, die immer dann auftraten, wenn eine Berechnung nicht möglich war bzw. eine Zahl den Zahlenraum von float sprengte. Selbst konnten Sie diese Werte allerdings nicht vergeben.

Der Datentyp Decimal baut auf diesem Ansatz auf und ermöglicht es Ihnen zudem, Decimal-Instanzen mit einem solchen Zustand zu initialisieren. Folgende Werte sind möglich:

Wert	Bedeutung
Infinity, Inf	Positiv unendlich
-Infinity, -Inf	Negativ unendlich
NaN	Ungültiger Wert (»Not a Number«)
sNaN	Ungültiger Wert (»signaling Not a Number«) Der Unterschied zu NaN besteht darin, dass eine Exception geworfen wird, sobald versucht wird, mit sNaN weiterzurechnen. Rechenoperationen mit NaN funktionieren anstandslos, ergeben allerdings immer wieder NaN.

Tabelle 14.1 Nichtnumerische Werte des Datentyps Decimal

Diese nichtnumerischen Werte können wie Zahlen verwendet werden:

```
>>> Decimal("NaN") + Decimal("42.42")
Decimal("NaN")
>>> Decimal("Infinity") + Decimal("Infinity")
Decimal("Infinity")
>>> Decimal("sNaN") + Decimal("42.42")
Traceback (most recent call last):
[…]
decimal.InvalidOperation: sNaN
>>> Decimal("Inf") - Decimal("Inf")
Traceback (most recent call last):
[…]
decimal.InvalidOperation: -INF + INF
```

top

14.3.3 Das Context-Objekt

Eingangs wurde erwähnt, dass es der Datentyp `Decimal` erlaubt, Dezimalzahlen mit beliebiger Genauigkeit zu speichern. Die Genauigkeit, das heißt die Anzahl der Nachkommastellen ist eine von mehreren globalen Einstellungen, die innerhalb eines sogenannten `Context`-Objekts gekapselt werden.

Um auf den aktuellen Kontext der arithmetischen Operationen zugreifen zu können, existieren innerhalb des Moduls `decimal` die Funktionen `getcontext` und `setcontext`.

An dieser Stelle möchten wir nur auf drei Attribute des `Context`-Objekts eingehen, die die Berechnungen beeinflussen können:

prec

Das Attribut `prec` (für »precision«) ermöglicht es, die Genauigkeit der `Decimal`-Instanzen des aktuellen Kontextes zu bestimmen. Der Wert versteht sich als Anzahl der zu berechnenden Nachkommastellen und ist eine ganze Zahl.

```
>>> c = decimal.getcontext()
>>> c.prec = 3
>>> Decimal("1.23456789") * Decimal("2.3456789")
Decimal("2.90")
```

Emin, Emax

Die Attribute `Emin` und `Emax` ermöglichen es, die maximale bzw. minimale Größe des Exponenten festzulegen. Beide müssen eine ganze Zahl referenzieren. Wenn das Ergebnis einer Berechnung dieses Limit überschreitet, wird eine Exception geworfen.

```
>>> c = decimal.getcontext()
>>> c.Emax = 9
>>> Decimal("1e100") * Decimal("1e100")
Traceback (most recent call last):
[…]
decimal.Overflow: above Emax
```

Dieses Kapitel kann allenfalls als grundlegende Einführung in das Modul `decimal` verstanden werden, denn dieses Modul bietet noch viele weitere Möglichkeiten, Berechnungen anzustellen oder Ergebnisse dieser Berechnungen genau an die eigenen Bedürfnisse anzupassen. Sollte also Ihr Interesse an diesem Modul geweckt worden sein, fühlen Sie sich dazu ermutigt, insbesondere in der Python-Dokumentation nach weiteren Verwendungswegen zu forschen.

Beachten Sie aber, dass üblicherweise kein Bedarf an solch präzisen Berechnungen besteht, wie sie der Datentyp `Decimal` ermöglicht. Der Geschwindigkeitsvorteil von `float` wiegt in der Regel schwerer als der Genauigkeitsgewinn von `Decimal`.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

15.2 Reguläre Ausdrücke – re

Das Modul re der Standardbibliothek bietet umfangreiche Möglichkeiten zum Arbeiten mit sogenannten *regulären Ausdrücken* (engl. *regular expressions*). In einem solchen regulären Ausdruck wird durch eine spezielle Syntax ein Textmuster beschrieben, das dann auf verschiedene Texte oder Textfragmente angewendet werden kann. Grundsätzlich gibt es zwei große Anwendungsbereiche von regulären Ausdrücken.

Im ersten Bereich, beim sogenannten *Matching*, wird geprüft, ob ein Textabschnitt auf das Muster des regulären Ausdrucks passt oder nicht. Ein häufiges Beispiel für Matching wäre es zu testen, ob eine eingegebene E-Mail-Adresse syntaktisch gültig ist.

Die zweite Einsatzmöglichkeit von regulären Ausdrücken ist das sogenannte *Searching*, bei dem innerhalb eines größeren Textes nach Textfragmenten gesucht wird, die auf einen regulären Ausdruck passen. Es handelt sich dabei um eine eigene Disziplin, da dieses Verhalten vom Programmierer selbst nicht effizient durch Einsatz des Matchings implementiert werden kann. Ein Anwendungsbeispiel könnte der Syntax Highlighter Ihrer Python-Umgebung sein, der durch Searching nach speziellen Codeabschnitten wie Schlüsselwörtern oder Strings sucht, um diese grafisch hervorzuheben.

Ein regulärer Ausdruck ist in Python ein String, der die entsprechenden Regeln enthält. Im Gegensatz zu manch anderen Programmiersprachen existiert hier kein eigenes Literal zu diesem Zweck.

Sollten Sie sich mit regulären Ausdrücken bereits auskennen, sind Sie vielleicht gerade auf ein Problem aufmerksam geworden, denn der Backslash ist ein sehr wichtiges Zeichen zur Beschreibung regulärer Ausdrücke, und ausgerechnet dieses Zeichen trägt innerhalb eines Strings bereits Bedeutung. Normalerweise leitet ein Backslash eine Escape-Sequenz ein. Sie können nun entweder immer die Escape-Sequenz für einen Backslash ("\\") verwenden oder, was empfehlenswerter ist, auf Pythons Raw-Strings zurückgreifen, in denen keine Escape-Sequenzen möglich sind. Zur Erinnerung: Raw-Strings werden in Python durch ein vorangestelltes r gekennzeichnet:

```
r"\Hallo Welt"
```

Im Folgenden möchten wir Sie in die komplexe Syntax regulärer Ausdrücke einweihen. Allein zu diesem Thema sind bereits ganze Bücher erschienen, weswegen die Beschreibung hier vergleichsweise knapp, aber grundlegend ausfallen soll.

Es gibt verschiedene Notationen zur Beschreibung regulärer Ausdrücke. Python hält sich an die Syntax, die in der Programmiersprache Perl verwendet wird.

top

15.2.1 Syntax regulärer Ausdrücke

Grundsätzlich ist der String

```
r"python"
```

bereits ein regulärer Ausdruck. Dieser würde exakt auf den String "python" passen. Diese direkt angegebenen einzelnen Buchstaben werden *Zeichenliterale* genannt. Beachten Sie unbedingt, dass Zeichenliterale innerhalb regulärer Ausdrücke *case sensitive* sind, das heißt, dass der obige Ausdruck nicht auf den String "Python" passen würde.

In regulären Ausdrücken können eine ganze Reihe von Steuerungszeichen verwendet werden, die den Ausdruck flexibler und mächtiger machen. Diese sollen im Folgenden besprochen werden.

Beliebige Zeichen

Die einfachste Verallgemeinerung, die innerhalb eines regulären Ausdrucks verwendet werden kann, ist die Kennzeichnung eines beliebigen Zeichens durch einen Punkt. So passt der Ausdruck

```
r".ython"
```

sowohl auf "python", "Python" als auch auf "Jython", nicht jedoch auf "Blython", da es sich nur um ein einzelnes beliebiges Zeichen handelt. Ein durch einen Punkt gekennzeichnetes beliebiges Zeichen darf nicht weggelassen werden. Der obige Ausdruck würde demzufolge nicht auf "ython" passen.

Zeichenklassen

Abgesehen davon, ein Zeichen ausdrücklich als beliebig zu kennzeichnen, ist es auch möglich, eine Klasse von Zeichen vorzugeben, die an dieser Stelle vorkommen dürfen. Dazu werden die gültigen Zeichen in eckige Klammern an die entsprechende Position geschrieben:

```
r"[jp]ython"
```

Dieser reguläre Ausdruck arbeitet ähnlich wie der des letzten

Abschnitts, lässt jedoch nur die Buchstaben j und p als erstes Zeichen des Wortes zu. Damit passt der Ausdruck sowohl auf "jython" als auch auf "python". Der Ausdruck passt aber nicht auf "Python", "jpython" oder "ython". Um auch die jeweiligen Großbuchstaben im Wort zu erlauben, kann der Ausdruck folgendermaßen erweitert werden:

```
r"[jJpP]ython"
```

Innerhalb einer Zeichenklasse ist es ebenfalls möglich, ganze Bereiche von Zeichen zuzulassen. Dadurch wird folgende Syntax verwendet:

```
r"[A-Z]ython"
```

Dieser reguläre Ausdruck lässt jeden Großbuchstaben als Anfangsbuchstaben des Wortes durch, beispielsweise aber keinen Kleinbuchstaben und keine Zahl. Um mehrere Bereiche zuzulassen, werden diese ganz einfach hintereinander geschrieben:

```
r"[A-Ra-r]ython"
```

Dieser reguläre Ausdruck passt beispielsweise sowohl auf "Qython" als auch auf "qython", nicht aber auf "Sython" oder "3ython".

Auch Ziffernbereiche können als Zeichenklasse verwendet werden:

```
r"[0-9]ython"
```

Als letzte Möglichkeit, die eine Zeichengruppe bietet, können Zeichen oder Zeichenbereiche ausgeschlossen werden. Dazu wird zu Beginn der Zeichengruppe ein Zirkumflex (^) geschrieben. So erlaubt der reguläre Ausdruck

```
r"[^pP]ython"
```

jedes Zeichen, abgesehen von einem großen oder kleinen »P«. Demzufolge würden sowohl "Sython" als auch "wython" passen, während "Python" und "python" außen vor bleiben müssten.

Beachten Sie, dass es innerhalb einer Zeichenklasse, abgesehen vom Bindestrich und dem Zirkumflex, keine Zeichen mit spezieller Bedeutung gibt. Das heißt insbesondere, dass ein Punkt in einer Zeichenklasse tatsächlich das Zeichen . bedeutet und nicht etwa ein beliebiges Zeichen.

Quantoren

Bisher können wir in einem regulären Ausdruck bestimmte Regeln für einzelne Zeichen aufstellen. Wir würden allerdings vor einem Problem stehen, wenn wir an einer bestimmten Stelle des Wortes eine gewisse Anzahl oder gar beliebig viele dieser Zeichen erlauben wollten. Für diesen Zweck werden sogenannte *Quantoren* eingesetzt. Das sind spezielle Zeichen, die hinter ein einzelnes Zeichenliteral oder eine Zeichenklasse geschrieben werden und kennzeichnen, wie oft diese auftreten dürfen. Die folgende Tabelle listet alle Quantoren auf und erläutert kurz ihre Bedeutung. Danach werden wir Beispiele für die Verwendung von Quantoren bringen.

Quantor	Bedeutung
?	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse darf entweder keinmal oder einmal vorkommen.
	Das vorangegangene Zeichen bzw. die

*	vorangegangene Zeichenklasse darf beliebig oft hintereinander vorkommen, das heißt unter anderem, dass sie auch weggelassen werden kann.
+	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse darf beliebig oft hintereinander vorkommen, mindestens aber einmal. Sie darf also nicht weggelassen werden.

Tabelle 15.2 Quantoren in regulären Ausdrücken

Die folgenden drei Beispiele zeigen einen regulären Ausdruck mit je einem Quantor. Nachfolgend soll besprochen werden, wie sich diese auf die Bedeutung des Ausdrucks auswirken.

- `r"P[Yy]?thon"`

  Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes ein höchstens einmaliges Auftreten des großen oder kleinen »Y«. Damit passt der Ausdruck auf die Wörter `"Python"` und `"Pthon"`, beispielsweise jedoch nicht auf `"Pyython"`.

- `r"P[Yy]*thon"`

  Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes ein beliebig häufiges Auftreten des großen oder kleinen »Y«. Damit passt der Ausdruck auf die Wörter `"Python"`, `"Pthon"` und `"PyyYYYyython"`, beispielsweise jedoch nicht auf `"Pzthon"`.

- `r"P[Yy]+thon"`

  Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes ein mindestens einmaliges Auftreten des großen oder kleinen »Y«. Damit passt der Ausdruck auf die Wörter `"Python"`, `"PYthon"` und `"PyyYYYyython"`, beispielsweise jedoch nicht auf `"Pthon"`.

Neben diesen allgemeinen Quantoren gibt es eine Syntax, die es ermöglicht, exakt anzugeben, wie viele Wiederholungen einer Zeichengruppe erlaubt sind. Dabei werden die Unter- und Obergrenzen für Wiederholungen in geschweifte Klammern hinter das entsprechende Zeichen bzw. die entsprechende Zeichengruppe geschrieben. Die folgende Tabelle listet die Möglichkeiten der Notation auf:

Quantor	Bedeutung
`{anz}`	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse muss exakt `anz`-mal vorkommen.
`{min,}`	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse muss mindestens `min`-mal vorkommen.
`{,max}`	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse darf maximal `max`-mal vorkommen.
`{min,max}`	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse muss mindestens `min`-mal und darf maximal `max`-mal vorkommen.

Tabelle 15.3 Quantoren in regulären Ausdrücken

Auch für diese Quantoren möchten wir das bisherige Beispiel abändern und untersuchen, was sie für Auswirkungen haben.

- `r"P[Yy]{2}thon"`

  Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes exakt zwei jeweils große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter `"Pyython"` oder

"PYython", beispielsweise jedoch nicht auf "Pyyython".

- `r"P[Yy]{2,}thon"`

  Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes mindestens zwei jeweils große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter "Pyython", "PYython" und "PyyYYYyython", beispielsweise jedoch nicht auf "Python".

- `r"P[Yy]{,2}thon"`

  Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes maximal zwei jeweils große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter "Python", "Pthon" und "PYYthon", beispielsweise jedoch nicht auf "Pyyython".

- `r"P[Yy]{1,2}thon"`

  Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes mindestens ein und maximal zwei große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter "Python" oder "PYython", beispielsweise jedoch nicht auf "Pthon" oder "PYYYthon".

Vordefinierte Zeichenklassen

Damit nicht bei jedem regulären Ausdruck das Rad neu erfunden werden muss, existiert eine Reihe von vordefinierten Zeichenklassen, die beispielsweise alle Ziffern oder alle alphanumerischen Zeichen umfassen. Diese Zeichenklassen werden bei der Arbeit mit regulären Ausdrücken sehr häufig benötigt und können deswegen durch einen speziellen Code abgekürzt werden. Jeder dieser Codes beginnt mit einem Backslash. Die folgende Tabelle listet alle vordefinierten Zeichenklassen mit ihren Bedeutungen auf.

Zeichenklasse	Bedeutung
`\d`	Passt auf alle Zeichen, die Ziffern des Dezimalsystems sind. Äquivalent zu `[0-9]`.
`\D`	Passt auf alle Zeichen, die nicht Ziffern des Dezimalsystems sind. Äquivalent zu `[^0-9]`.
`\s`	Passt auf alle Whitespace-Zeichen. Äquivalent zu `[ \t\n\r\f\v]`.
`\S`	Passt auf alle Zeichen, die kein Whitespace sind. Äquivalent zu `[^ \t\n\r\f\v]`.
`\w`	Passt auf alle alphanumerischen Zeichen und den Unterstrich. Äquivalent zu `[a-zA-z0-9_]`.
`\W`	Passt auf alle Zeichen, die nicht alphanumerisch und kein Unterstrich sind. Äquivalent zu `[^a-zA-Z0-9_]`.

Tabelle 15.4 Vordefinierte Zeichenklassen in regulären Ausdrücken

Diese vordefinierten Zeichenklassen können wie ein normales Zeichen im regulären Ausdruck verwendet werden. So passt der Ausdruck

```
r"P\w*th\dn"
```

auf die Wörter "Pyth0n" oder "P_th1n", beispielsweise jedoch nicht auf "Python".

Beachten Sie, dass die üblichen Escape-Sequenzen, die innerhalb eines Strings verwendet werden können, auch innerhalb eines regulären Ausdrucks – selbst wenn er in einem Raw-String geschrieben wird – ihre Bedeutung behalten und nicht mit den hier vorgestellten Zeichenklassen interferieren. Gebräuchlich sind

hier vor allem \n, \t, \r oder \\, insbesondere aber auch \x.

Zudem ist es mit dem Backslash möglich, einem Sonderzeichen die spezielle Bedeutung zu nehmen, die es innerhalb eines regulären Ausdrucks trägt. Auf diese Weise kann zum Beispiel mit den Zeichen * oder + gearbeitet werden, ohne dass diese als Quantoren angesehen werden. So passt der folgende reguläre Ausdruck

```
r"\*Py\.\.\.on\*"
```

allein auf den String "*Py...on*".

Weitere Sonderzeichen

Für gewisse Einsatzgebiete wird es unbedingt verlangt, Regeln aufstellen zu können, die über die bloße Zeichenebene hinausgehen. So wäre es beispielsweise interessant, einen regulären Ausdruck zu erschaffen, der nur passt, wenn sich das Wort am Ende oder Anfang einer Textzeile befindet. Für solche und ähnliche Fälle gibt es einen bestimmten Satz an zusätzlichen Sonderzeichen, die genau so angewendet werden können wie die vordefinierten Zeichenklassen.

Die folgende Tabelle listet alle zusätzlichen Sonderzeichen auf und gibt zu jedem eine kurze Erklärung. In der Tabelle finden Sie einige Anmerkungen zu sogenannten Flags. Das sind Einstellungen, die entweder aktiviert oder deaktiviert werden können und die Auswertung eines regulären Ausdrucks beeinflussen. Näheres dazu, wie Sie diese Einstellungen setzen können, erfahren Sie im Laufe dieses Kapitels.

Sonderzeichen	Bedeutung
\A	Passt nur am Anfang eines Strings.
\b	Passt nur am Anfang oder Ende eines Wortes. Ein Wort kann aus allen Zeichen der Klasse \w bestehen und wird durch ein Zeichen der Klasse \s begrenzt.
\B	Passt nur, wenn es sich nicht um den Anfang oder das Ende eines Wortes handelt.
\Z	Passt nur am Ende eines Strings.
^	Passt nur am Anfang eines Strings. Beachten Sie, dass das Zeichen ^ zwei Bedeutungen hat und innerhalb einer Zeichenklasse die aufgelisteten Zeichen ausschließt. Wenn das MULTILINE-Flag gesetzt wurde, passt ^ auch direkt nach jedem Newline-Zeichen innerhalb des Strings.
$	Passt nur am Ende eines Strings. Wenn das MULTILINE-Flag gesetzt wurde, passt $ auch direkt vor jedem Newline-Zeichen innerhalb des Strings.

Tabelle 15.5 Vordefinierte Zeichenklassen in regulären Ausdrücken

Im konkreten Beispiel passt also der reguläre Ausdruck

```
r"\APython\Z"
```

nur bei dem String "Python", nicht jedoch bei den Strings "abcPythonabc" oder "Pythonabc".

Die hier besprochenen Beispiele beziehen sich hauptsächlich auf das Matching von regulären Ausdrücken, weswegen Ihnen die

Bedeutung dieser Sonderzeichen möglicherweise noch nicht ersichtlich ist. Diese Sonderzeichen sind aber gerade beim Searching von unerlässlicher Wichtigkeit. Stellen Sie sich einmal vor, Sie würden in einem Text nach allen Vorkommen einer bestimmten Zeichenkette am Zeilenanfang suchen wollen. Dies wäre nur durch Einsatz des Sonderzeichens ^ möglich.

Genügsame Quantoren

Wir haben bereits die Quantoren ?, * und + besprochen. Diese werden in der Terminologie regulärer Ausdrücke als »gefräßig« (engl. *greedy*) bezeichnet. Diese Klassifizierung ist nur beim Searching von Bedeutung. Betrachten Sie dazu einmal folgenden regulären Ausdruck:

```
r"Py.*on"
```

Dieser Ausdruck passt auf jeden Teilstring, der mit Py beginnt und mit on endet. Dazwischen können beliebig viele, nicht näher spezifizierte Zeichen stehen. Behalten Sie im Hinterkopf, dass wir uns beim Searching befinden, der Ausdruck also dazu verwendet werden soll, aus einem längeren String verschiedene Teilstrings zu isolieren, die auf den regulären Ausdruck passen.

Nun möchten wir den regulären Ausdruck gedanklich auf den folgenden String anwenden:

```
"Python Python Python"
```

Sie meinen, dass drei Ergebnisse gefunden werden? Irrtum, es handelt sich um exakt ein Ergebnis, nämlich den Teilstring "Python Python Python". Zur Erklärung: Es wurde der »gefräßige« Quantor * eingesetzt. Ein solcher gefräßiger Quantor hat die Ambition, die maximal mögliche Anzahl Zeichen zu »verschlingen«. Beim Searching wird also, solange die »gefräßigen« Quantoren eingesetzt werden, stets der größtmögliche passende String gefunden.

Dieses Verhalten kann umgekehrt werden, sodass immer der kleinstmögliche passende String gefunden wird. Dazu kann an jeden Quantor ein Fragezeichen angefügt werden. Dadurch wird der Quantor »genügsam« (engl. *non-greedy*). Angenommen, das Searching auf dem obigen String wäre mit dem regulären Ausdruck

```
r"Py.*?on"
```

durchgeführt worden, so wäre als Ergebnis tatsächlich dreimal der Teilstring "Python" gefunden worden. Dies funktioniert für die Quantoren ?, *, + und {}.

Gruppen

Ein Teil eines regulären Ausdrucks kann durch runde Klammern zu einer sogenannten *Gruppe* zusammengefasst werden. Eine solche Gruppierung hat im Wesentlichen drei Vorteile:

- Eine Gruppe kann als Einheit betrachtet und als solche natürlich auch mit einem Quantor versehen werden. Auf diese Weise lässt sich beispielsweise das mehrmalige Auftreten einer bestimmten Zeichenkette erlauben:

  ```
  r"( ?Python)+ ist gut"
  ```

  In diesem Ausdruck existiert eine Gruppe um den Teilausdruck r" ?Python". Dieser Teilausdruck passt auf den String "Python" mit einem optionalen Leerzeichen zu Beginn. Die gesamte Gruppe kann nun beliebig oft

vorkommen, womit der obige reguläre Ausdruck sowohl auf "Python ist gut" als auch auf "Python Python Python ist gut" passt.

Beachten Sie das Leerzeichen zu Beginn der Gruppe, um die Funktionsweise des Ausdrucks zu verstehen.

- Der zweite Vorteil einer Gruppe ist der, dass man auf sie zugreifen kann, nachdem das Searching bzw. Matching durchgeführt wurde. Das heißt, man könnte beispielsweise überprüfen, ob eine eingegebene URL gültig ist, und gleichzeitig Subdomain, Domain und TLD herausfiltern.

  Näheres dazu, wie der Zugriff auf Gruppen funktioniert, erfahren Sie in Abschnitt 15.2.2, »Verwendung des Moduls«.

- Es gibt Gruppen, die in einem regulären Ausdruck häufiger gebraucht werden. Um diese nicht jedes Mal erneut schreiben zu müssen, werden Gruppen mit 1 beginnend durchnummeriert und können dann anhand ihres Index referenziert werden. Eine solche Referenz besteht aus einem Backslash, gefolgt von dem Index der jeweiligen Gruppe, und passt auf den gleichen Teilstring, auf den die Gruppe gepasst hat. So passt der reguläre Ausdruck

```
r"(Python) \1"
```

  auf "Python Python".

Alternativen

Eine weitere Möglichkeit, die die Syntax regulärer Ausdrücke vorsieht, sind sogenannte *Alternativen*. Im Prinzip handelt es sich dabei um nichts anderes als um eine ODER-Verknüpfung zweier Zeichen oder Zeichengruppen, wie Sie sie bereits von dem Operator or her kennen. Diese Verknüpfung wird durch den senkrechten Strich, auch *Pipe* genannt, durchgeführt.

```
r"P(ython|eter)"
```

Dieser reguläre Ausdruck passt sowohl auf den String "Python" als auch auf "Peter". Durch die Gruppe kann später ausgelesen werden, welche der beiden Alternativen aufgetreten ist.

Extensions

Damit wäre die Syntax regulärer Ausdrücke beschrieben. Zusätzlich zu dieser mehr oder weniger standardisierten Syntax erlaubt Python die Verwendung sogenannter *Extensions*. Eine Extension ist folgendermaßen aufgebaut:

```
(?...)
```

Die drei Punkte werden durch eine Kennung der gewünschten Extension und weitere extension-spezifische Angaben ersetzt. Diese Syntax wurde gewählt, da eine öffnende Klammer, gefolgt von einem Fragezeichen, keine syntaktisch sinnvolle Bedeutung hat und demzufolge »frei« war. Beachten Sie aber, dass eine Extension in der Regel keine neue Gruppe erzeugt, auch wenn die runden Klammern dies nahelegen. Nachfolgend möchten wir näher auf die Extensions eingehen, die in Pythons regulären Ausdrücken verwendet werden können.

(?iLmsux)

Diese Extension erlaubt es, ein oder mehrere Flags für den gesamten regulären Ausdruck zu setzen. Der Begriff *Flag* ist

bereits verwendet worden und beschreibt eine bestimmte Einstellung, die entweder aktiviert oder deaktiviert werden kann. Ein Flag kann entweder im regulären Ausdruck selbst, eben durch diese Extension, oder durch einen Parameter der Funktion `re.compile` gesetzt werden. Im Zusammenhang mit dieser Funktion werden wir näher darauf eingehen, welche Flags wofür stehen. Das Flag `i` macht den regulären Ausdruck beispielsweise *case insensitive*:

```
r"(?i)P"
```

Dieser Ausdruck passt sowohl auf "P" als auch auf "p".

(?:...)

Diese Extension kann wie normale runde Klammern verwendet werden, erzeugt dabei aber keine Gruppe. Das heißt, auf einen durch diese Extension eingeklammerten Teilausdruck kann später nicht zugegriffen werden. Ansonsten ist diese Syntax äquivalent zu runden Klammern:

```
r"(?:abc|def)"
```

(?P<name>...)

Diese Extension erzeugt eine Gruppe mit dem angegebenen Namen. Das Besondere an einer solchen benannten Gruppe ist, dass sie nicht allein über ihren Index, sondern auch über ihren Namen referenziert werden kann. Der Name muss ein gültiger Bezeichner sein:

```
r"(?P<hallowelt>abc|def)"
```

(?P=name)

Passt auf all das, auf das die bereits definierte Gruppe mit dem Namen *name* gepasst hat. Diese Extension erlaubt es also, eine benannte Gruppe zu referenzieren.

```
r"(?P<py>[Pp]ython) ist, wie (?P=py) sein sollte"
```

Dieser reguläre Ausdruck passt auf den String "Python ist, wie Python sein sollte".

(?#...)

Diese Extension stellt einen Kommentar dar. Der Inhalt der Klammern wird schlicht ignoriert:

```
r"Py(?#lalala)thon"
```

(?=...)

Passt nur dann, wenn der reguläre Ausdruck ... als Nächstes passt. Diese Extension greift also vor, ohne in der Auswertung des Ausdrucks tatsächlich voranzuschreiten.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?!...)

Passt nur dann, wenn der reguläre Ausdruck ... als Nächstes nicht passt. Diese Extension ist das Gegenstück zu der vorherigen.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?<=…)

Passt nur, wenn der reguläre Ausdruck … zuvor gepasst hat. Diese Extension greift also auf bereits ausgewertete Teile des Strings zurück, ohne die Auswertung selbst zurückzuwerfen.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?<!…)

Passt nur, wenn der reguläre Ausdruck … zuvor nicht gepasst hat. Diese Extension ist damit das Gegenstück zu der vorherigen.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?(id/name)yes-pattern|no-pattern)

Diese, recht kompliziert anmutende Extension kann in einem regulären Ausdruck als eine Art Fallunterscheidung verwendet werden. Abhängig davon, ob eine Gruppe mit dem angegebenen Index bzw. dem angegebenen Namen auf einen Teilstring gepasst hat, wird entweder (im positiven Fall) auf das *yes-pattern* oder (im negativen Fall) auf das *no-pattern* getestet. Das *no-pattern* wird durch einen senkrechten Strich vom *yes-pattern* getrennt, kann aber auch weggelassen werden. Vielleicht ist der Sinn dieser Extension noch nicht ganz klar geworden, deshalb folgendes Beispiel:

```
r"(?P<klammer>\()?Python(?(klammer)\))"
```

In diesem Ausdruck wird zunächst eine Gruppe namens `klammer` erstellt, die maximal einmal vorkommen darf und aus einer öffnenden, runden Klammer besteht. Danach folgt die Zeichenkette `Python`, und schlussendlich wird durch die Extension eine schließende Klammer gefordert, sofern zuvor eine öffnende aufgetreten ist, also sofern die Gruppe `klammer` zuvor gepasst hat.

Damit passt der reguläre Ausdruck auf die Strings `"Python"` und `"(Python)"`, beispielsweise aber nicht auf `"(Python"`.

Damit wäre den syntaktischen Regeln für reguläre Ausdrücke Genüge getan. Auch wenn dieser Abschnitt möglicherweise etwas trocken und theoretisch war, ist es durchaus wichtig, sich mit regulären Ausdrücken auseinanderzusetzen, denn in vielen Fällen ist der Einsatz regulärer Ausdrücke besonders elegant.

In den folgenden Abschnitten möchten wir über die praktische Anwendung regulärer Ausdrücke in Python reden. Dazu gehört zunächst einmal die Verwendung des Moduls `re`. Danach werden wir jeweils ein kleines Beispielprojekt zum Matching bzw. Searching bringen.

top

15.2.2 Verwendung des Moduls

Nachdem Sie in die unendlichen Weiten der regulären Ausdrücke eingeführt wurden, werden wir uns hier um ihre konkrete Verwendung in Python kümmern. Die Beispiele dieses Abschnitts werden im interaktiven Modus durchgeführt und setzen voraus, dass das Modul `re` eingebunden wurde:

```
>>> import re
```

Flags

Im vorherigen Abschnitt wurden mehrfach die sogenannten *Flags* angesprochen. Das sind bestimmte Einstellungen, die die

Auswertung eines regulären Ausdrucks beeinflussen. Flags können entweder im Ausdruck selbst durch eine Extension oder als Parameter einer der im Modul re verfügbaren Funktionen angegeben werden. Sie beeinflussen nur den Ausdruck, der aktuell verarbeitet wird, und verbleiben nicht nachhaltig im System. Jedes Flag ist als Konstante im Modul re enthalten und kann über eine Lang- oder eine Kurzversion seines Namens angesprochen werden. Die folgende Tabelle listet alle Flags auf und erläutert ihre Bedeutung.

Alias	Name	Bedeutung
re.I	re.IGNORECASE	Macht die Auswertung des regulären Ausdrucks *case insensitive*, das heißt, dass die Zeichengruppe [A-Z] sowohl auf Groß- als auch auf Kleinbuchstaben passen würde.
re.L	re.LOCALE	Gibt an, dass bestimmte vordefinierte Zeichenklassen von der aktuellen Lokalisierung abhängig gemacht werden sollen. Das betrifft die Gruppen \w, \W, \b, \B, \s und \S.
re.M	re.MULTILINE	Wenn dieses Flag gesetzt wurde, passt ^ sowohl zu Beginn des Strings als auch nach jedem Newline-Zeichen und $ vor jedem Newline-Zeichen. Normalerweise passen ^ und $ nur am Anfang bzw. am Ende des Strings.
re.S	re.DOTALL	Wenn dieses Flag gesetzt wurde, passt das Sonderzeichen . tatsächlich auf jedes Zeichen. Normalerweise passt der Punkt auf jedes Zeichen außer auf das Newline-Zeichen \n.
re.U	re.UNICODE	Wenn dieses Flag gesetzt wurde, passen sich die vordefinierten Zeichenklassen dem Unicode-Standard an. Das heißt, dass dann auch Nicht-ASCII-Zeichen als Buchstabe oder Ziffer eingestuft werden.
re.X	re.VERBOSE	Das Setzen dieses Flags erlaubt es Ihnen, einen regulären Ausdruck zu formatieren. Wenn es gesetzt wurde, werden Whitespace-Zeichen wie Leerzeichen, Tabulatoren oder Newline-Zeichen ignoriert, solange sie nicht durch einen Backslash eingeleitet werden. Zudem leitet ein #-Zeichen einen Kommentar ein. Das heißt, alles hinter diesem Zeichen bis zu einem Newline-Zeichen wird ignoriert.

Tabelle 15.6 Flags

Funktionen

Neben den Flags sind im Modul re noch einige Funktionen enthalten, die im Folgenden besprochen werden sollen.

re.compile(pattern[, flags])

Kompiliert den regulären Ausdruck *pattern* zu einem Regular-Expression-Objekt, im Folgenden *RE-Objekt* genannt. Bei mehreren Operationen auf demselben regulären Ausdruck lohnt es sich, diesen zu kompilieren, da diese Operationen dann wesentlich schneller durchgeführt werden können. Zum Durchführen der Operationen bietet das RE-Objekt im Wesentlichen die gleiche Funktionalität wie das Modul re.

Um die Auswertung des Ausdrucks zu beeinflussen, können ein oder mehrere Flags angegeben werden. Wenn es sich um mehrere handelt, müssen diese durch das bitweise ODER |

getrennt werden.

```
>>> c1 = re.compile(r"P[yY]thon")
>>> c2 = re.compile(r"P[y]thon", re.I)
>>> c3 = re.compile(r"P[y]thon", re.I | re.S)
```

Die Angabe von Flags ist bei den meisten Funktionen des Moduls `re` über den Parameter *flags* möglich. Wir werden darauf in Zukunft nicht mehr eingehen.

Näheres zum RE-Objekt folgt im nächsten Abschnitt.

re.search(pattern, string[, flags])

Durchsucht den String *string* nach einem Teilstring, auf den der reguläre Ausdruck *pattern* passt. Der erste gefundene Teilstring wird in Form eines sogenannten *Match-Objekts* zurückgegeben. Näheres zur Verwendung des Match-Objekts erfahren Sie im entsprechenden Abschnitt.

Wenn kein Ergebnis gefunden wurde, gibt die Funktion `None` zurück.

```
>>> re.search(r"P[Yy]thon", "Nimm doch Python")
<_sre.SRE_Match object at 0xb7bd7f00>
```

re.match(pattern, string[, flags])

Wenn null oder mehr Zeichen am Anfang des Strings *string* auf den regulären Ausdruck *pattern* passen, wird diese Übereinstimmung in Form eines Match-Objekts zurückgegeben. Wenn keine Übereinstimmung gefunden wurde, wird `None` zurückgegeben.

```
>>> print re.match(r"P[Yy]thon", "PYYthon")
None
>>> re.match(r"P[Yy]thon", "PYthon")
<_sre.SRE_Match object at 0xb7bd7f00>
```

re.split(pattern, string[, maxsplit])

Der String *string* wird nach Übereinstimmungen mit dem regulären Ausdruck *pattern* durchsucht. Alle passenden Teilstrings werden als Trennzeichen angesehen, und die dazwischenliegenden Teile werden als Liste von Strings zurückgegeben.

```
>>> re.split(r"\s", "Python Python Python")
['Python', 'Python', 'Python']
```

Eventuell vorkommende Gruppen innerhalb des regulären Ausdrucks werden ebenfalls als Elemente dieser Liste zurückgegeben:

```
>>> re.split(r"\s(.*?)\s", "Python oder Python und
Python")
['Python', 'oder', 'Python', 'und', 'Python']
```

In diesem regulären Ausdruck werden alle von zwei Whitespaces umgebenen Wörter als Trennzeichen behandelt.

Wenn der Parameter `maxsplit` angegeben wurde und ungleich `0` ist, wird der String maximal `maxsplit`-mal unterteilt. Der Reststring wird als letztes Element der Liste zurückgegeben.

re.findall(pattern, string[, flags])

Sucht im String *string* nach Übereinstimmungen mit dem regulären Ausdruck *pattern*. Alle gefundenen, nicht überlappenden Übereinstimmungen werden in Form einer Liste von Strings

zurückgegeben:

```
>>> re.findall(r"P[Yy]thon", "Python oder PYthon und
Python")
['Python', 'PYthon', 'Python']
```

Wenn *pattern* ein oder mehrere Gruppen enthält, werden diese anstelle der übereinstimmenden Teilstrings in die Ergebnisliste geschrieben.

```
>>> re.findall(r"P([Yy])thon", "Python oder PYthon und
Python")
['y', 'Y', 'y']
>>> re.findall(r"P([Yy])th(.)n", "Python oder PYthon und
Python")
[('y', 'o'), ('Y', 'o'), ('y', 'o')]
```

Bei mehreren Gruppen handelt es sich um eine Liste von Tupeln.

re.finditer(pattern, string[, flags])

Sucht im String *string* nach Übereinstimmungen mit dem regulären Ausdruck *pattern*. Das Ergebnis ist ein Iterator, der über alle gefundenen, nicht überlappenden Übereinstimmungen jeweils als Match-Objekt iteriert.

re.sub(pattern, repl, string[, flags])

Die Funktion *sub* sucht im String *string* nach nicht überlappenden Übereinstimmungen mit dem regulären Ausdruck *pattern*. Es wird eine Kopie des Strings *string* zurückgegeben, in dem alle passenden Teilstrings durch den String *repl* ersetzt wurden:

```
>>> re.sub(r"[Jj]a[Vv]a","Python", "Java oder java und
jaVa")
'Python oder Python und Python'
```

Statt eines Strings kann für *repl* auch ein Funktionsobjekt übergeben werden. Dieses wird für jede gefundene Übereinstimmung aufgerufen und bekommt das jeweilige Match-Objekt als einzigen Parameter. Der übereinstimmende Teilstring wird durch den Rückgabewert der Funktion ersetzt.

Es ist möglich, durch die Schreibweisen \g<name> oder \g<index> Gruppen des regulären Ausdrucks zu referenzieren:

```
>>> re.sub(r"([Jj]ava)","Python statt \g<1>", "Nimm doch
Java")
'Nimm doch Python statt Java'
```

Durch den optionalen Parameter *count* kann die maximale Anzahl an Ersetzungen festgelegt werden, die vorgenommen werden dürfen.

re.subn(pattern, repl, string[, flags])

Funktioniert ähnlich wie sub, mit dem Unterschied, dass ein Tupel zurückgegeben wird, in dem zum einen der neue String und zum anderen die Anzahl der vorgenommenen Ersetzungen stehen:

```
>>> re.subn(r"([Jj]ava)","Python statt \g<1>", "Nimm doch
Java")
('Nimm doch Python statt Java', 1)
```

re.escape(string)

Wandelt alle nicht-alphanumerischen Zeichen von *string* in ihre entsprechende Escape-Sequenz um und gibt das Ergebnis als String zurück. Diese Funktion ist besonders dann sinnvoll, wenn man einen String in einen regulären Ausdruck einbetten möchte,

aber nicht sicher sein kann, ob Sonderzeichen, beispielsweise ein Punkt, enthalten sind.

```
>>> re.escape("Funktioniert das wirklich? ... (ja!)")
'Funktioniert\\ das\\ wirklich\\?\\ \\.\\.\\.\\
\\(ja\\!\\)'
```

Beachten Sie, dass die Escape-Sequenzen im Stringliteral jeweils durch einen doppelten Backslash eingeleitet werden. Das liegt daran, dass das Ergebnis als String und nicht als Raw-String zurückgegeben wird.

Das Regular-Expression-Objekt

Ein Regular-Expression-Objekt, im Folgenden RE-Objekt genannt, wird erzeugt, wenn ein regulärer Ausdruck kompiliert wurde. Das Kompilieren eines regulären Ausdrucks ist sinnvoll, wenn mehrere Operationen mit ihm durchgeführt werden sollen. Diese können dann zusammengenommen wesentlich schneller durchgeführt werden, als wenn man die Funktionen `match` oder `search` direkt aufrufen würde.

Damit Searching- und Matching-Operationen mit einem kompilierten regulären Ausdruck durchgeführt werden können, besitzt das RE-Objekt eine Funktionalität, die deckungsgleich ist mit der des `re`-Moduls. Das bedeutet, dass für das RE-Objekt größtenteils die Funktionen des `re`-Moduls als Methoden implementiert sind, selbstverständlich mit gewissen Änderungen der Schnittstelle.

Wir werden hier nicht genau auf die Funktionsweise der Methoden eingehen, sondern nur einen Vergleich zu den Funktionen des `re`-Moduls ziehen. Dennoch ist es aufgrund der Änderungen bei den Schnittstellen wichtig, alle Methoden zu behandeln. Die Beispiele verstehen sich in folgendem Kontext:

```
>>> import re
>>> c = re.compile(r"P[Yy]th.n")
```

Das bedeutet: Es existiert ein RE-Objekt namens `c`, dem der reguläre Ausdruck `r"P[Yy]th.n"` zugrunde liegt.

c.match(string[, pos[, endpos]])

Äquivalent zur Funktion `re.match`. Die optionalen Parameter *pos* und *endpos* geben, wenn sie ungleich `0` sind, zwei Indizes an, zwischen denen das Matching durchgeführt werden soll. Wenn sie nicht angegeben wurden, wird das Matching auf dem gesamten String durchgeführt.

```
>>> print c.match("Pythoon")
None
>>> c.match("Python")
<_sre.SRE_Match object at 0xb7c49e58>
```

c.search(string[, pos[, endpos]])

Äquivalent zur Funktion `re.search`. Die optionalen Parameter *pos* und *endpos* haben dieselbe Bedeutung wie bei der Methode `match`.

```
>>> c.search("Dies ist Python")
<_sre.SRE_Match object at 0xb7c49e58>
```

c.split(string[, maxsplit])

Äquivalent zur Funktion `re.split`.

```
>>> c.split("halloweltPythonhallowelt")
```

```
['hallowelt', 'hallowelt']
```

c.findall(string[, pos[, endpos]])

Äquivalent zur Funktion `re.findall`. Die optionalen Parameter *pos* und *endpos* haben dieselbe Bedeutung wie bei der Methode `match`.

```
>>> c.findall("Python Python Python")
['Python', 'Python', 'Python']
```

c.finditer(string[, pos[, endpos]])

Äquivalent zur Funktion `re.finditer`. Die optionalen Parameter *pos* und *endpos* haben dieselbe Bedeutung wie bei der Methode `match`.

c.sub(repl, string[, count])

Äquivalent zur Funktion `re.sub`.

c.subn(repl, string[, count])

Äquivalent zur Funktion `re.subn`.

Neben diesen Methoden enthält das RE-Objekt drei Attribute, die das Arbeiten mit dem Objekt erleichtern.

c.flags

Das Attribut `flags` ist eine ganze Zahl und enthält alle gesetzten Flags. Beachten Sie, dass Flags selbst auch ganze Zahlen sind und eine Kombination von Flags durch ihr bitweises ODER repräsentiert wird. Die zu setzenden Flags werden beim Erzeugen des RE-Objekts der Funktion `re.compile` übergeben. Wenn kein Flag gesetzt wurde, ist der Wert des Attributs `0`.

```
>>> c.flags
0
```

Um zu testen, ob ein bestimmtes Flag gesetzt ist, kann das bitweise UND verwendet werden:

```
>>> c1 = re.compile(r"P[Yy]th.n", re.I)
>>> c1.flags
2
>>> c1.flags & re.I
2
>>> c1.flags & re.M
0
```

Das bitweise UND zwischen dem Attribut `flags` und einem nicht gesetzten Flag ergibt immer `0`.

c.groupindex

Das Attribut `groupindex` ist ein Dictionary, das alle Namen benannter Gruppen als Schlüssel enthält und die Indizes dieser Gruppen als Werte. Eine benannte Gruppe wird durch die Extension `(?P<name>…)` erzeugt.

```
>>> c2 = re.compile(r"(?P<gruppe1>P[Yy])(?
P<gruppe2>th.n)")
>>> c2.groupindex
{'gruppe1': 1, 'gruppe2': 2}
```

c.pattern

Das Attribut `pattern` ist ein String und enthält den regulären

Ausdruck, der dem RE-Objekt zugrunde liegt.

```
>>> c.pattern
'P[Yy]th.n'
```

Das Match-Objekt

Nachdem wir das RE-Objekt besprochen haben, wenden wir uns einem wesentlich interessanteren Objekt zu, dem Match-Objekt. Eine solche Instanz wird zurückgegeben, wenn eine Match- oder Search-Operation Übereinstimmungen gefunden hat. Das Match-Objekt enthält nähere Details zu diesen gefundenen Übereinstimmungen.

Die Beispiele in diesem Unterkapitel verstehen sich in folgendem Kontext:

```
>>> import re
>>> c = re.compile(r"(P[Yy])(th.n)")
```

Das Match-Objekt verfügt über folgende Methoden:

m.expand(template)

Die Methode expand erlaubt es, den String *template* mit Informationen zu füllen, die aus der Matching- bzw. Searching-Operation stammen. So können über \g<index> und \g<name> die Teilstrings eingefügt werden, die auf die jeweiligen Gruppen gepasst haben. Beachten Sie unbedingt, dass Sie *template* wegen der Backslashs als Raw-String angeben sollten.

```
>>> m = c.match("Python")
>>> m.expand(r"Hallo \g<1> Welt \g<2>")
'Hallo Py Welt thon'
```

m.group([group1, ...])

Die Methode group erlaubt einen komfortablen Zugriff auf die Teilstrings, die auf die verschiedenen Gruppen des regulären Ausdrucks gepasst haben. Wenn nur ein Argument übergeben wurde, ist der Rückgabewert ein String, ansonsten ein Tupel von Strings. Wenn eine Gruppe auf keinen Teilstring gepasst hat, wird für diese None zurückgegeben. Ein Index von 0 gibt alle Gruppen zurück.

```
>>> m = c.match("Python")
>>> m.group(0)
'Python'
>>> m.group(1)
'Py'
>>> m.group(1, 2)
('Py', 'thon')
```

m.groups([default])

Gibt ein Tupel zurück, das alle Teilstrings enthält, die auf eine der im regulären Ausdruck enthaltenen Gruppen gepasst haben. Der optionale Parameter *default* erlaubt es, den Wert festzulegen, der in das Tupel geschrieben wird, wenn auf eine Gruppe kein Teilstring gepasst hat. Der Parameter ist mit None vorbelegt.

```
>>> m = c.match("Python")
>>> m.groups()
('Py', 'thon')
```

m.groupdict([default])

Gibt ein Dictionary zurück, das die Namen aller benannten Gruppen als Schlüssel und die jeweils passenden Teilstrings als

Werte enthält. Der Parameter *default* hat die gleiche Bedeutung wie bei der Methode groups.

```
>>> c2 = re.compile(r"(?P<gruppe>P[Yy])(th.n)")
>>> m2 = c2.match("Python")
>>> m2.groupdict()
{'gruppe': 'Py'}
```

m.start([group]), end([group])

Gibt den Start- bzw. Endindex des Teilstrings zurück, der auf die Gruppe *group* gepasst hat. Der optionale Parameter *group* ist mit `0` vorbelegt.

```
m = c.match("Python")
>>> m.start(2)
2
>>> m.end(2)
6
```

m.span([group])

Gibt das Tupel `(start(group), end(group))` zurück.

```
>>> m = c.match("Python")
>>> m.span(2)
(2, 6)
```

Neben den soeben beschriebenen Methoden besitzt das Match-Objekt sechs Attribute, die im Folgenden beschrieben werden sollen.

m.pos, m.endpos

Die Methoden `match` und `search` des RE-Objekts besitzen zwei Parameter namens *pos* und *endpos*. Die Attribute `pos` und `endpos` des Match-Objekts erlauben den Zugriff auf die dort zuletzt übergebenen Werte.

m.lastindex

Der Index der Gruppe, die bei der Auswertung als Letzte auf einen Teilstring gepasst hat, oder `None`, wenn keine Gruppe gepasst hat.

m.lastgroup

Der Name der symbolischen Gruppe, die bei der Auswertung als Letzte auf einen Teilstring gepasst hat, oder `None`, wenn keine Gruppe gepasst hat.

m.re

Der ursprüngliche reguläre Ausdruck als String.

m.string

Der String, der der `match`- bzw. `search`-Methode des RE-Objekts zuletzt übergeben wurde.

top

15.2.3 Ein einfaches Beispielprogramm – Searching

Bisher wurde sowohl die Syntax regulärer Ausdrücke als auch deren Verwendung durch das Modul `re` der Standardbibliothek besprochen. Eigentlich ist die Thematik damit erschöpfend behandelt, doch wir möchten an dieser Stelle zwei kleine Beispielprojekte vorstellen, die stark auf reguläre Ausdrücke

setzen, um auch einer praxisorientierten Einführung gerecht zu werden. Zunächst soll in diesem relativ einfach gehaltenen Programm das Searching und im nächsten, etwas komplexeren Beispiel das Matching erklärt werden.

Mithilfe des Searchings können Muster innerhalb eines längeren Textes gefunden und herausgefiltert werden. In unserem Beispielprogramm soll das Searching dazu genutzt werden, alle Links aus einer beliebigen HTML-Datei mitsamt Beschreibung herauszulesen. Dazu müssen wir uns zunächst den Aufbau eines HTML-Links vergegenwärtigen:

```
<a href="URL">Beschreibung</a>
```

Dazu ist zu sagen, dass HTML nicht zwischen Groß- und Kleinschreibung unterscheidet, wir den regulären Ausdruck also mit dem IGNORECASE-Flag verwenden sollten. Des Weiteren handelt es sich bei dem obigen Beispiel um die einfachste Form eines HTML-Links, denn neben der URL und der Beschreibung können noch weitere Angaben getätigt werden.

Der folgende reguläre Ausdruck passt sowohl auf den oben beschriebenen als auch auf weitere, komplexere HTML-Links:

```
r"<a.*href=[\"\']([^\"\']*)[\"\'].*>(.*?)</a>"
```

Wichtig ist, dass der reguläre Ausdruck zwei Gruppen enthält, jeweils für die URL und die Beschreibung, sodass diese beiden Angaben später bequem ausgelesen werden können. Außerdem sollten Sie unbedingt beachten, dass innerhalb dieser Gruppen »genügsame« Quantoren eingesetzt wurden, da sonst mehrere Links fälschlicherweise zu einem zusammengefasst werden könnten.

Doch nun zum Beispielprogramm:

```
import re

f = open("test.html", "r")
html = f.read()
f.close()

it = re.finditer(r"<[a].*href=[\"\'](.*?)[\"\'].*>(.*?
)</[a]>",
                 html, re.I)
for m in it:
    print "Name: %s, Link: %s" % (m.group(2), m.group(1))
```

Zunächst wird eine beliebige HTML-Datei, in diesem Fall *test.html*, geöffnet und mithilfe der Methode read des Dateiobjekts ausgelesen. Danach wird die Funktion finditer des Moduls re aufgerufen, um alle Übereinstimmungen mit dem vorhin besprochenen regulären Ausdruck im HTML-Code zu finden. Das Ergebnis wird als Iterator zurückgegeben und von it referenziert.

Schlussendlich wird über it iteriert. In jedem Iterationsschritt ist die aktuelle Übereinstimmung als Match-Objekt m verfügbar. Jetzt werden nur noch die Teilstrings ausgegeben, die auf die beiden Gruppen des regulären Ausdrucks gepasst haben.

Sie können das Programm mit beliebigen HTML-Seiten testen. Besuchen Sie dazu im Internet eine möglichst komplexe Website, beispielsweise die eines Nachrichtenmagazins, und speichern Sie diese als HTML-Datei ab. Sie werden sehen, dass das Beispielprogramm auch hier die enthaltenen Links findet.

Das hier vorgestellte Programm schreit geradezu danach, erweitert zu werden. Beispielsweise könnten neben Links noch andere Teile des HTML-Codes, wie enthaltene Bilder oder Überschriften, ausgelesen werden.

top

15.2.4 Ein komplexeres Beispielprogramm – Matching

Es ist allgemein – und besonders im Web – ein häufiges Problem, eingegebene Formulardaten zu validieren und die wichtigen Informationen aus den Eingaben herauszufiltern. Dies ist selbstverständlich auch mit normalen String-Operationen möglich, doch mutiert der Code bei solchen Versuchen schnell zu einem unförmigen Batzen von Irgendwas. Das angesprochene Problem lässt sich durch reguläre Ausdrücke sehr elegant und nur mit geringem Quellcodeaufwand lösen. Unser Beispielprogramm soll aus einer Art elektronischer Visitenkarte alle relevanten Informationen auslesen und maschinenlesbar aufbereiten. Die Visitenkarte ist in einer Textdatei in folgendem Format gespeichert:

```
Name: Max Mustermann
Addr: Musterstr 123
      12345 Musterhausen
Tel:  +49 1234 56789
```

Das Programm soll nun diese Textdatei einlesen, die enthaltenen Informationen extrahieren und zu einem solchen Dictionary aufbereiten:

```
{
'Tel': ('+49', '1234', '56789'),
'Name': ('Max', 'Mustermann'),
'Addr': ('Musterstr', '123', '12345', 'Musterhausen')
}
```

In der Textdatei soll dabei immer nur ein Datensatz stehen.

Zunächst einmal möchten wir etwas detaillierter auf die Funktionsweise des Beispielprogramms eingehen. Die Visitenkarte besteht aus verschiedenen Informationen, denen immer eine Überschrift bzw. Kategorie gegeben wurde (»Name«, »Addr« und »Tel«). Die Kategorie von der Information zu trennen ist keine komplizierte Angelegenheit, da der Doppelpunkt innerhalb der Informationen nicht vorkommt und somit in jeder Zeile einzigartig ist. Ein Problem ist die dritte Zeile, da hier keine explizite Überschrift gegeben ist. In einem solchen Fall wird die Zeile an die Information der vorherigen Überschrift angehängt. Auf diese Weise lässt sich ein Dictionary erzeugen, das die Überschriften auf die jeweiligen Informationen mappt.

Die Informationen werden allerdings zeilenweise aus der Datei ausgelesen. Das ist nicht optimal, da wir die Daten ausdrücklich maschinenlesbar einlesen wollten, das heißt insbesondere nach Einzelinformationen getrennt. Für diese Arbeit bieten sich reguläre Ausdrücke förmlich an.

Kommen wir zur konkreten Implementierung. Dazu schreiben wir zunächst eine Funktion, die die Daten zeilenweise einliest und zu einem Dictionary aufbereitet:

```
def leseDatei(datei):
    d = {}
    f = open(datei)
    for zeile in f:
        if ":" in zeile:
            key, d[key] = (s.strip() for s in
zeile.split(":"))
        elif "key" in locals():
            d[key] += "\n%s" % zeile.strip()
    f.close()
    return d
```

Die Funktion `leseDatei` bekommt den String `datei` mit einer Pfadangabe übergeben. Innerhalb der Funktion wird die Datei zeilenweise eingelesen. Jede Zeile wird anhand des Doppelpunktes in die beiden Teile »Überschrift« und »Information« aufgeteilt und, durch Einsatz der Methode `strip`, von überflüssigen Leerzeichen befreit. Danach werden Überschrift und Information in das Dictionary `d` geschrieben und die jeweils aktuelle Überschrift

zusätzlich durch key referenziert.

Wenn in einer Zeile kein Doppelpunkt vorkommt, wurde die Information auf mehrere Zeilen umgelegt. Das bedeutet für uns, dass wir zunächst auch die Methode strip auf den kompletten Zeileninhalt anwenden und sie dann unter der Überschrift key an den bereits bestehenden Wert im Dictionary anhängen. Damit dieses durchgeführt werden kann, muss die Referenz key selbstverständlich existieren. Da diese erst innerhalb der if-Anweisung angelegt wird, wird vorausgesetzt, dass eine Zeile mit Doppelpunkt vor einer Zeile ohne Doppelpunkt kommen muss. Obwohl es keine syntaktisch sinnvolle Datei gibt, in der das nicht gilt, überprüfen wir im elif-Zweig explizit, ob die Referenz key existiert.

Das Resultat dieser Funktion ist ein Dictionary mit den Überschriften als Schlüssel und den dazugehörigen Informationen (in Form von Strings) als Werte. Die zweite Funktion des Beispiels soll die Daten mithilfe regulärer Ausdrücke analysieren und dann als Tupel im Dictionary ablegen. Dazu erzeugen wir zunächst ein Dictionary namens regexp, das für jede Überschrift einen regulären Ausdruck bereitstellt, der verwendet werden kann, um die Information zu validieren:

```
regexp = {
         "Name" : r"([A-Za-z]+)\s([A-Za-z]+)",
         "Addr" : r"([A-Za-z]+)\s(\d+)\s*(\d{5})\s([A-Za-
z]+)",
         "Tel"  : r"(\+\d{2})\s(\d{4})\s(\d{3,})"
         }
```

Diese regulären Ausdrücke verfügen über mehrere Gruppen, um das Aufteilen der Information in die verschiedenen Einzelinformationen zu erleichtern.

Die Funktion, mit der die Daten analysiert werden, sieht folgendermaßen aus:

```
def analysiereDaten(daten, regexp):
    for key in daten:
        m = re.match(regexp[key], daten[key])
        if not m:
            return False
        daten[key] = m.groups()
    return True
```

Die Funktion analysiereDaten bekommt zwei Dictionarys als Parameter übergeben: zum einen das soeben erstellte Dictionary regexp und zum anderen das, das von der vorherigen Funktion erstellt wurde und die eingelesenen Daten enthält.

Die Funktion iteriert in einer for-Schleife über das Dictionary daten und wendet, jeweils passend zur aktuellen Überschrift, mithilfe der Funktion re.match den regulären Ausdruck auf den eingelesenen String an. Das zurückgegebene Match-Objekt wird durch m referenziert.

Nachfolgend wird getestet, ob re.match None zurückgegeben hat. Ist das der Fall, gibt die Funktion analysiereDaten ihrerseits False zurück. Andernfalls wird der aktuelle Wert des Dictionarys daten mit den Teilstrings überschrieben, die auf die einzelnen Gruppen der regulären Ausdrücke gepasst haben. Die Methode group des Match-Objekts gibt ein Tupel von Strings zurück. Nach dem Durchlaufen der Funktion analysiereDaten enthält das Dictionary die gewünschten Daten in aufbereiteter Form.

Zu guter Letzt fehlt noch der Code, der den Anstoß zum Einlesen und Aufbereiten der Daten gibt:

```
daten = leseDatei("id.txt")
if analysiereDaten(daten, regexp):
    print daten
else:
    print "Die Angaben sind fehlerhaft"
```

Je nachdem, welchen Wahrheitswert die Funktion `analysiereDaten` zurückgegeben hat, werden die aufbereiteten Daten oder eine Fehlermeldung ausgegeben.

Hoffentlich haben Ihnen die beiden Beispiele geholfen, einen praxisbezogenen Einstieg in die Welt der regulären Ausdrücke zu finden. Bleibt noch zu sagen, dass das dargebotene Programm zwar funktioniert, aber nicht perfekt ist. Fühlen Sie sich dazu ermutigt, es zu erweitern oder anzupassen. So erlauben die regulären Ausdrücke beispielsweise noch keine Umlaute oder Interpunktionszeichen im Straßennamen. Sie könnten beispielsweise auch Visitenkarte und Programm um die Angabe einer E-Mail-Adresse erweitern.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

15.3 Lokalisierung von Programmen – gettext

Das Modul gettext der Standardbibliothek ist bei der *Internationalisierung* und *Lokalisierung* von Python-Programmen von Nutzen. Mit Internationalisierung, auch als »I18N« abgekürzt, wird der Vorgang bezeichnet, die Benutzerschnittstelle eines Programms so zu abstrahieren, dass sie sehr leicht an fremde sprachliche oder kulturelle Umgebungen angepasst werden kann. Als Lokalisierung, was auch mit »L10N« abgekürzt wird, wird dann das Anpassen des Programms an die Gegebenheiten eines bestimmten Landes oder einer Region bezeichnet. Beachten Sie, dass sich das Modul gettext dabei auf die Übersetzung von Strings beschränkt. Andere Unterschiede, wie beispielsweise Datumsformate oder Währungssymbole, werden nicht berücksichtigt.

Das Modul gettext lehnt sich an die *GNU gettext API* an, die als Teil des GNU-Projekts weit verbreitet ist und die rein sprachliche Anpassung eines Programms gewährleistet. Das Modul erlaubt es, eine möglichst genaue Nachbildung der *GNU gettext API* zu verwenden. Zudem ist eine gegenüber der *GNU gettext API* etwas abstraktere, objektorientierte Schnittstelle vorhanden, auf die wir uns in diesem Abschnitt beziehen werden.

Zunächst ein paar Worte dazu, wie die Lokalisierung eines Programms vonstattengeht. Der Programmierer schreibt sein Programm, in dem die Benutzerführung vorzugsweise in englischer Sprache stattfindet. Zur Internationalisierung des Programms wird jeder String, der ausgegeben werden soll, durch eine sogenannte *Wrapper-Funktion* geschickt. Das ist eine Funktion, die den nicht-lokalisierten englischen String als Parameter übergeben bekommt und die passende Übersetzung zurückgibt.

Intern verwendet gettext zur Übersetzung verschiedene Sprachkompilate. Das sind Binärdateien, die die Übersetzung des

Programms in jeweils eine bestimmte Sprache enthalten. Diese Binärdateien werden aufgrund ihrer Dateiendung *.mo-Dateien* genannt. Wie diese Dateien erzeugt werden, wird unter anderem Inhalt des nächsten Abschnitts sein.

top

15.3.1 Beispiel für die Verwendung von gettext ▲

Das Modul `gettext` spielt immer dann eine Rolle, wenn ein Programm veröffentlicht und somit einer großen Gruppe von Anwendern zugänglich gemacht wird. Dabei ist es besonders bei Open-Source-Projekten üblich, dass das Programm ursprünglich nur in einer oder zwei Sprachen veröffentlicht wird und weitere Übersetzungen später von den Anwendern erstellt und an den Autor geschickt werden. Damit dies funktioniert, sollte der Programmierer jedoch zumindest die Übersetzbarkeit seines Programms gewährleisten.

An dieser Stelle soll die Verwendung von `gettext` an einem kleinen Beispielprogramm gezeigt werden. Dabei kann selbstverständlich nicht die vollständige Funktionalität von `gettext` zum Tragen kommen. Der Quellcode des Beispielprogramms sieht folgendermaßen aus:

```
import gettext
import random

trans = gettext.translation("meinprogramm", "locale", ["de"])
trans.install()

werte = []
while True:
    w = raw_input(_("Please enter a value: "))
    if not w:
        break
    werte.append(w)
print _("The random choice is: %s") % random.choice(werte)
```

Das Programm selbst ist unspektakulär, es liest so lange Strings vom Benutzer ein, bis einer dieser Strings leer ist, der Benutzer also ↵ gedrückt hat, ohne eine Eingabe zu tätigen. Dann wählt das Programm zufällig einen dieser Strings und gibt ihn aus. Mit diesem Programm könnte also beispielsweise eine zufällig gewählte Person einer Gruppe für den nächsten Samstagabend zum Fahrer ernannt werden. Beachten Sie aber besonders, dass die Interaktion mit dem Benutzer ausschließlich auf Englisch geschieht, jeder String, der ausgegeben wird, aber vorher durch eine Funktion namens `_` geschickt wird. Beachten Sie außerdem bei der `print`-Ausgabe, dass die Funktion `_` um den mit einem Platzhalter behafteten String geschrieben wurde, also bevor der Platzhalter durch dynamischen Inhalt ersetzt wurde. Das ist wichtig, da sonst keine Übersetzung erfolgen kann.

Der eigentlich interessante Teil des Programms sind die beiden Zeilen nach den `import`-Anweisungen:

```
trans = gettext.translation("meinprogramm", "locale", ["de"])
trans.install()
```

Hier wird ein sogenanntes Translation-Objekt erstellt. Das ist eine Instanz, die die Übersetzung aller Strings in eine bestimmte Sprache gewährleistet. Um ein solches Objekt zu erstellen, wird die Funktion `gettext.translation` aufgerufen. Diese bekommt einen frei wählbaren Namen, die sogenannte Domain, als ersten Parameter. Der zweite Parameter ist das Unterverzeichnis, in dem sich die Übersetzungen befinden, und der dritte Parameter ist schließlich eine Liste von Sprachen. Das Translation-Objekt übersetzt nun in die erste Sprache aus der Liste, für die ein Sprachkompilat gefunden werden kann.

Durch Aufruf der Methode `install` des Translation-Objekts installiert dieses seine interne Übersetzungsmethode als Funktion `_` im lokalen Namensraum. Damit werden alle Strings, mit denen die Funktion `_` aufgerufen wird, in die Sprache übersetzt, für die das Translation-

Objekt steht, sofern denn eine Übersetzung verfügbar ist.

Erstellen des Sprachkompilats

Zum Erstellen des Sprachkompilats muss zunächst eine Liste aller zu übersetzenden Strings erstellt werden. Das sind all jene, die vor der Ausgabe durch die Funktion _ geschickt werden. Da es eine unzumutbare Arbeit wäre, diese Liste von Hand anzufertigen, ist in Python ein Programm namens `pygettext.py` [Unter Windows finden Sie das Programm *pygettext.py* im Unterordner *Tools/i18n* Ihrer Python-Installation. Unter Unix-ähnlichen Betriebssystemen wie beispielsweise Linux sollte sich die Programmdatei im Systempfad befinden und direkt ausführbar sein.] im Lieferumfang enthalten, das genau dies für Sie erledigt. Das Programm erstellt eine sogenannte *.po-Datei*. Das ist eine für Menschen lesbare Variante des .mo-Dateiformats. Diese .po-Datei wird dann von den Übersetzern in verschiedene Sprachen übersetzt. Dies kann von Hand geschehen oder durch Einsatz diverser Tools, die für diesen Zweck existieren. Die für unser Beispielprogramm erzeugte .po-Datei sieht folgendermaßen aus:

```
[…]

#: main.py:9
msgid "Please enter a value: "
msgstr "Bitte geben Sie einen Wert ein: "

#: main.py:13
msgid "The random choice is: %s"
msgstr "Die Zufallswahl ist: %s"
```

Anstelle der Auslassungszeichen enthält die Datei Informationen wie etwa den Autor, die verwendete Software oder das Encoding der Datei.

Eine übersetzte .po-Datei kann durch das Programm `msgfmt.py` [Unter Windows finden Sie das Programm *msgfmt.py* im Unterordner *Tools/i18n* Ihrer Python-Installation. Unter Unix-ähnlichen Betriebssystemen wie beispielsweise Linux sollte sich die Programmdatei im Systempfad befinden und direkt ausführbar sein.] , das ebenfalls im Lieferumfang von Python vorhanden ist, in das binäre .mo-Format kompiliert werden. Ein fertiges Sprachkompilat muss sich in folgendem Ordner befinden, damit es von `gettext` als solches gefunden wird:

Programmverzeichnis/Unterordner/LC_MESSAGES/Sprache/Domain.mo

Der Name des Verzeichnisses *Unterordner* wird beim Aufruf der Funktion `gettext.translate` angegeben und war in unserem Beispiel `locale`. Dieses Verzeichnis muss ein Unterverzeichnis namens `LC_MESSAGES` haben, in dem für jede vorhandene Sprache ein weiteres Unterverzeichnis existieren muss. Das Sprachkompilat selbst muss die im Programm angegebene Domain als Namen haben.

In unserem Beispielprogramm muss das Sprachkompilat also in folgendem Verzeichnis liegen:

Programmverzeichnis/locale/LC_MESSAGES/de/meinprogramm.mo

Wenn das Sprachkompilat nicht vorhanden ist, wird beim Aufruf der Funktion `gettext.translate` eine entsprechende Exception geworfen:

```
Traceback (most recent call last):
  […]
IOError: [Errno 2] No translation file found for domain:
'meinprogramm'
```

Wenn das Sprachkompilat an seinem Platz ist, werden Sie beim Ausführen des Programms feststellen, dass alle Strings ins Deutsche übersetzt wurden:

```
Bitte geben Sie einen Wert ein: Donald Duck
Bitte geben Sie einen Wert ein: Daisy Duck
Bitte geben Sie einen Wert ein: Onkel Dagobert
Bitte geben Sie einen Wert ein:
Die Zufallswahl ist: Donald Duck
```

Mit dem Modul gettext ist es also möglich, ein Programm ohne großen Aufwand in mehrere Sprachen zu übersetzen und darüber hinaus zu einem späteren Zeitpunkt weitere Sprachen hinzuzufügen, ohne große Änderungen am Programm selbst durchführen zu müssen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

15.4 Hash-Funktionen – hashlib

Das Modul `hashlib` der Standardbibliothek implementiert die gängigsten sogenannten *Hash-Funktionen*. Ganz allgemein sind das sehr komplexe Algorithmen, die aus einem Parameter, zumeist einem String, einen sogenannten *Hash-Wert* berechnen. Wozu kann ein solcher Hash-Wert verwendet werden?

Nun, stellen Sie sich einmal vor, Sie würden eine Foren-Software entwickeln, die später für eine Community im Internet eingesetzt werden soll. Bevor ein Benutzer Beiträge im Forum verfassen darf, muss er sich mit seinem Benutzernamen und dem dazu passenden Passwort anmelden. Natürlich ist es im Sinne des Forenbetreibers und vor allem des Benutzers selbst, dass das Passwort nicht in falsche Hände gerät. Es stellt sich also die Frage, wie die Anmeldeprozedur möglichst sicher gestaltet werden kann.

Die intuitivste Möglichkeit wäre es, Benutzername und Passwort im Klartext an die Foren-Software zu übermitteln. Dort werden diese beiden Informationen mit den Anmeldedaten aller Benutzer verglichen, und bei einem Treffer wird der Zugang zum Forum ermöglicht.

Würde eine solche Software die Anmeldeprozedur tatsächlich so durchführen, müssten Benutzername und Passwort im Klartext in der internen Datenbank des Forums gespeichert werden. Das ist

beim Benutzernamen kein größeres Problem, da es sich dabei im Allgemeinen um eine öffentliche Information handelt. Doch das Passwort im Klartext in einer solchen Datenbank zu speichern wäre grob fahrlässig. Stellen Sie sich einmal vor, ein Angreifer würde über eine Sicherheitslücke in einem anderen Teil der Software Zugriff auf die Datenbank erlangen. Er wäre sofort im Besitz aller Passwörter der angemeldeten Benutzer. Das wird besonders dann brisant, wenn man bedenkt, dass viele Leute das gleiche Passwort für mehrere Benutzerkonten verwenden.

Wünschenswert wäre es also, die Korrektheit eines Passworts mit an Sicherheit grenzender Wahrscheinlichkeit zu ermitteln, ohne Referenzpasswörter im Klartext speichern zu müssen. Und genau hier kommen Hash-Funktionen ins Spiel. Eine Hash-Funktion bekommt einen Parameter übergeben und errechnet daraus eine Art Prüfsumme, den sogenannten *Hash-Wert*. Wenn sich jetzt also ein neuer Benutzer bei der Foren-Software anmeldet und sein Passwort wählt, wird dieses nicht im Klartext in die Datenbank eingetragen, sondern es wird der Hash-Wert des Passworts gespeichert.

Beim Einloggen schickt der Benutzer sein Passwort an den Server. Dieser errechnet dann den Hash-Wert des übertragenen Passworts und vergleicht ihn mit den gespeicherten Hash-Werten.

Damit eine solche Anmeldeprozedur funktioniert und ein potenzieller Angreifer auch mit Zugriff auf die Datenbank keine Passwörter errechnen kann, müssen Hash-Funktionen einige Bedingungen erfüllen:

- Eine Hash-Funktion stellt eine *Einwegkodierung* dar. Das heißt, dass die Berechnung des Hash-Wertes nicht umkehrbar ist, man also aus einem Hash-Wert nicht auf den ursprünglichen Parameter schließen kann.
- Bei Hash-Funktionen treten grundsätzlich sogenannte *Kollisionen* auf, das sind zwei verschiedene Parameter, die denselben Hash-Wert ergeben. Ein wesentlicher Schritt zum Knacken einer Hash-Funktion ist es, solche Kollisionen berechnen zu können. Eine Hash-Funktion sollte also die Berechnung von Kollisionen so stark erschweren, dass sie nur unter extrem hohem Zeitaufwand zu bestimmen wären.
- Eine Hash-Funktion sollte möglichst willkürlich sein, sodass man nicht aufgrund eines ähnlichen Hash-Wertes darauf schließen kann, dass man in der Nähe des gesuchten Passworts ist. Sobald der Parameter der Hash-Funktion minimal verändert wird, sollte ein völlig verschiedener Hash-Wert berechnet werden.
- Zu guter Letzt sollte eine Hash-Funktion selbstverständlich sehr schnell zu berechnen sein. Außerdem müssen sich die entstehenden Hash-Werte untereinander sehr effizient vergleichen lassen.

Das Anwendungsfeld von Hash-Funktionen ist weit gefächert. So werden sie, abgesehen von dem obigen Passwortbeispiel, unter anderem auch zum Vergleichen großer Dateien verwendet. Anstatt diese Dateien untereinander Byte für Byte zu vergleichen, werden ihre Hash-Werte berechnet und verglichen. Mit den Hash-Werten lässt sich sagen, ob die Dateien mit Sicherheit verschieden oder mit großer Wahrscheinlichkeit identisch sind.

Beachten Sie, dass die Wahrscheinlichkeit einer Kollision bei den im Modul `hashlib` implementierten Verfahren sehr gering, aber theoretisch immer noch vorhanden ist.

top

15.4.1 Verwendung des Moduls ▼▲

Zunächst enthält das Modul `hashlib` eine Reihe von Klassen, die jeweils einen Hash-Algorithmus implementieren:

Klasse	Algorithmus	Beschreibung
hashlib.md5	MD5	*Message-Digest Algorithm 5* Erzeugt aus einem beliebigen String einen 128-Bit-Hash-Wert.
hashlib.sha1	SHA-1	*Secure Hash Algorithm 1* Erzeugt aus einem beliebigen String einen 160-Bit-Hash-Wert. Beachten Sie, dass der SHA-1-Algorithmus bereits ansatzweise geknackt wurde, also nicht mehr zum Speichern von Passwörtern verwendet werden sollte.
hashlib.sha224	SHA-224	*Secure Hash Algorithm 224* Erzeugt aus einem beliebigen String einen 224-Bit-Hash-Wert.
hashlib.sha256	SHA-256	*Secure Hash Algorithm 256* Erzeugt aus einem beliebigen String einen 256-Bit-Hash-Wert.
hashlib.sha384	SHA-384	*Secure Hash Algorithm 384* Erzeugt aus einem beliebigen String einen 384-Bit-Hash-Wert.
hashlib.sha512	SHA-512	*Secure Hash Algorithm 512* Erzeugt aus einem beliebigen String einen 512-Bit-Hash-Wert.

Tabelle 15.7 Unterstützte Hash-Funktionen

Die Verwendung dieser Klassen ist identisch. Deshalb wird sie hier exemplarisch an der Klasse `md5` gezeigt.

Beim Instanziieren der Klasse `md5` wird der String übergeben, dessen Hash-Wert berechnet werden soll.

```
>>> import hashlib
>>> m = hashlib.md5("Hallo Welt")
```

Durch Aufruf der Methode `digest` wird der berechnete Hash-Wert als Bytefolge zurückgegeben. Beachten Sie, dass der zurückgegebene String durchaus nicht-druckbare Zeichen enthalten kann.

```
>>> m.digest()
'\\7*2\xc9\xaet\x8aL\x04\x0e\xba\xdcQ\xa8)'
```

Durch Aufruf der Methode `hexdigest` wird der berechnete Hash-Wert als String zurückgegeben, der eine Folge von zweistelligen Hexadezimalzahlen enthält. Diese Hexadezimalzahlen repräsentieren jeweils ein Byte des Hash-Wertes. Der zurückgegebene String enthält ausschließlich druckbare Zeichen.

```
>>> m.hexdigest()
'5c372a32c9ae748a4c040ebadc51a829'
```

top

15.4.2 Beispiel

Das folgende kleine Beispielprogramm verwendet das Modul `hashlib`, um einen Passwortschutz zu realisieren. Das Passwort soll dabei nicht als Klartext im Quelltext gespeichert werden, sondern als Hash-Wert. Dadurch ist gewährleistet, dass die Passwörter nicht einsehbar sind, selbst wenn jemand in den Besitz

der Hash-Werte kommen sollte. Auch anmeldepflichtige Internetportale wie beispielsweise Foren speichern die Passwörter der Benutzer als Hash-Wert.

```
import hashlib

pwhash = "578127b714de227824ab105689da0ed2"

m = hashlib.md5(raw_input("Ihr Passwort bitte: "))
if pwhash == m.hexdigest():
    print "Zugriff erlaubt"
else:
    print "Zugriff verweigert"
```

Das Programm liest ein Passwort vom Benutzer ein, errechnet den MD5-Hash-Wert dieses Passworts und vergleicht ihn mit dem gespeicherten Hash-Wert. Der vorher berechnete Hash-Wert pwhash ist in diesem Fall im Programm vorgegeben. Unter normalen Umständen stünde er mit anderen Hash-Werten in einer Datenbank oder wäre in einer Datei gespeichert. Wenn beide Werte übereinstimmen, wird symbolisch »Zugriff erlaubt« ausgegeben. Das Passwort für dieses Programm lautet »Mein Passwort«.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

16.2 Komfortable Datumsfunktionen – datetime

Das Modul `datetime` ist im Vergleich zum `time`-Modul wesentlich abstrakter und durch seine eigenen Zeit- und Datumstypen auch wesentlich angenehmer zu benutzen.

Das Modul unterscheidet zwei Arten von Datums- und Zeitobjekten: die sogenannten *naiven* und die *bewussten* Objekte.

Ein *naives* Objekt kümmert sich nicht darum, auf welche Zeitzone sich sein Wert bezieht, und enthält auch keine Informationen darüber, wohingegen ein *bewusstes* Objekt mit Informationen zu seiner Zeitzone verknüpft ist. Ihre Programme können selbst entscheiden, ob die von ihnen benutzten Objekte *naiv* oder *bewusst* sind.

Wie das genau funktioniert, wird hier nicht näher thematisiert. Weitere Informationen darüber finden Sie in der Python-Dokumentation.

Konstanten des Moduls datetime

Es gibt zwei Konstanten, die das `datetime`-Modul definiert, um den Wertebereich für die Jahreszahlen zu definieren:

datetime.MINYEAR

Der minimal mögliche Wert für eine Jahreszahl. Der Wert ist in der Regel 1.

datetime.MAXYEAR

Der maximal mögliche Wert für eine Jahreszahl. Der Wert ist in der Regel 9999.

Die fünf Datentypen von datetime

Das Modul datetime definiert fünf eigene Datentypen für den Umgang mit Datum und Zeit. Alle diese Datentypen sind *immutable*.

datetime.date

Ein Datentyp zum Speichern von Datumsangaben. Alle Instanzen dieses Datentyps sind prinzipiell *naiv*, kümmern sich also nicht um die Gegebenheiten der lokalen Zeitzone.

datetime.time

Mit datetime.time können Zeitpunkte an einem Tag gespeichert werden. Dabei wird idealisiert angenommen, dass jeder Tag 24*60*60 Sekunden umfasst und dass es keine Schaltsekunden gibt.

datetime.datetime

Die Kombination aus datetime.date und datetime.time zum Speichern von ganzen Zeitpunkten, die sowohl ein Datum als auch eine Uhrzeit umfassen. Der Datentyp datetime.datetime ist der wichtigste des Moduls.

datetime.timedelta

Es ist möglich, Differenzen zwischen datetime.date- und auch datetime.date time-Instanzen zu bilden. Die Ergebnisse solcher Subtraktionen sind dann date time.timedelta-Objekte.

datetime.tzinfo

Dieser Typ wird benötigt, um mit Zeitzonen umzugehen. Dafür muss das Programm eine Subklasse von datetime.tzinfo erzeugen und bestimmte Methoden überschreiben.

Aus Platzgründen werden wir diesen Datentyp nicht behandeln. Allerdings finden Sie in der Python-Dokumentation ein gutes Beispiel für die Implementation einer datetime.tzinfo-Klasse.

top

16.2.1 datetime.date

Hier werden wir die Attribute und Methoden des Datentyps datetime.date behandeln.

Konstruktoren der Klasse datetime.date

Es gibt drei Konstruktoren für datetime.date-Instanzen:

datetime.date(year, month, day)

Erzeugt eine neue Instanz des Datentyps datetime.date, die den durch die Parameter festgelegten Tag repräsentiert. Dabei müssen die Parameter folgenden Bedingungen genügen:

- datetime.MINYEAR <= *year* <= datetime.MAXYEAR
- 1 <= *month* <= 12
- 1 <= *day* <= (Anzahl der Tage des übergebenen Monats)

```
>>> geburtstag = datetime.date(1987, 11, 3)
>>> geburtstag
datetime.date(1987, 11, 3)
```

datetime.date.today()

Erzeugt eine neue datetime.date-Instanz, die den aktuellen Tag repräsentiert:

```
>>> datetime.date.today()
datetime.date(2007, 9, 19)
```

datetime.date.fromtimestamp(timestamp)

Erzeugt ein neues datetime.date-Objekt, das das Datum des übergebenen Unix-Timestamps speichert.

Klassenmember von datetime.date

datetime.date.min

Ein Klassenattribut, das den frühesten Tag beinhaltet, der durch den date time.date-Typ abgebildet werden kann. Wie das folgende Beispiel zeigt, ist dies der 1. Januar im Jahr 1:

```
>>> datetime.date.min
datetime.date(1, 1, 1)
```

datetime.date.max

Das Klassenattribut datetime.date.max speichert eine datetime.date-Instanz, die den spätesten Tag repräsentiert, der von datetime.date verwaltet werden kann: den 31.12. im Jahr 9999.

```
>>> datetime.date.max
datetime.date(9999, 12, 31)
```

Operatoren für datetime.date-Instanzen

Sie können Differenzen zwischen zwei datetime.date-Instanzen bilden. Das Ergebnis einer solchen Subtraktion ist ein datetime.timedelta-Objekt:

```
>>> datetime.date(1987, 7, 26) - datetime.date(1987, 1,
9)
datetime.timedelta(198)
```

In dem Beispiel liegen die beiden Zeitpunkte 198 Tage auseinander.

Es ist auch möglich, zu einer datetime.date-Instanz ein datetime.timedelta-Objekt zu addieren oder es davon abzuziehen. In diesem Fall ist das Ergebnis ein datetime.date-Objekt:

```
>>> datetime.date(1987, 1, 9) + datetime.timedelta(198)
datetime.date(1987, 7, 26)
```

Außerdem können datetime.date-Instanzen mit den Vergleichsoperatoren < und > verglichen werden. Dabei wird das Datum als »kleiner« betrachtet, das in der Zeit weiter in Richtung Vergangenheit liegt:

```
>>> datetime.date(1987, 1, 9) < datetime.date(1987, 7,
26)
True
```

Die Attribute und Methoden von datetime.date-Instanzen

Im Folgenden sei *d* eine datetime.date-Instanz.

d.year

Speichert das Jahr des Datums. Dieses Attribut kann nur gelesen werden.

d.month

Speichert den Monat des Datums. Dieses Attribut kann nur gelesen werden.

d.day

Speichert den Tag des Datums. Dieses Attribut kann nur gelesen werden.

d.replace(year, month, day)

Erzeugt ein neues Datum, dessen Attribute den übergebenen Parametern entsprechen. Fehlt eine Angabe, wird das entsprechende Attribut von *d* verwendet:

```
>>> d = datetime.date(1987, 7, 26)
>>> d.replace(month=11, day=3)
datetime.date(1987, 11, 3)
```

d.timetuple()

Gibt eine `time.struct_time`-Instanz [Siehe dazu Abschnitt 16.1, »Elementare Zeitfunktionen – time«.] zurück, die das Datum von *d* repräsentiert. Die Elemente für die Uhrzeit werden dabei auf `0` und das `tm_isdst`-Attribut wird auf `-1` gesetzt:

```
>>> d = datetime.date(2007, 7, 6)
>>> d.timetuple()
(2007, 7, 6, 0, 0, 0, 4, 187, -1)
```

d.weekday()

Gibt den Wochentag als Zahl zurück, wobei Montag als `0` und Sonntag als `6` angegeben werden.

d.isoweek()

Gibt den Wochentag als Zahl zurück, wobei Montag den Wert `0` und Sonntag den Wert `7` ergibt.

Siehe dazu auch `d.isocalendar()`.

d.isocalendar()

Gibt ein Tupel zurück, das drei Elemente enthält: `(ISO year, ISO week number, ISO weekday)`.

Die Angaben in dem Tupel erfolgen dabei im Format des sogenannten *ISO-Kalenders*, der eine Variante des gregorianischen Kalenders ist. Im ISO-Kalender wird ein Jahr in 52 oder 53 Wochen geteilt. Jede der Wochen beginnt mit einem Montag und endet mit einem Sonntag. Die erste Woche eines Jahres, deren Donnerstag in diesem Jahr liegt, erhält im ISO-Kalender die Wochennummer 1.

Die drei Elemente des zurückgegebenen Tupels bedeuten das Folgende: `(Jahr, Wochennummer, Tagesnummer)`.

Beispielsweise wird der 01.01.2008 ein Dienstag sein, weshalb der 31.12.2007 der erste Tag im Jahr 2008 des ISO-Kalenders ist:

```
>>> d = datetime.date(2007, 12, 31)
>>> d.isocalendar()
(2008, 1, 1)
```

d.isoformat()

Gibt einen String zurück, der den von *d* repräsentierten Tag im ISO-8601-Format enthält. Dieses Standardformat sieht folgendermaßen aus: YYYY-MM-DD, wobei die »Y« für die Ziffern der Jahreszahl, die »M« für die Ziffern der Monatszahl und die »D« für die Ziffern des Tages im Monat stehen.

```
>>> d = datetime.date(2007, 6, 18)
>>> d.isoformat()
'2007-06-18'
```

Achtung

Die Methode `isoformat` hat nichts mit dem ISO-Kalender zu tun, den die Methoden `isoweekday` und `isocalendar` verwenden.

d.ctime()

Gibt einen String in einem 24-Zeichen-Format aus, der den von *d* gespeicherten Tag repräsentiert. Die Platzhalter für Stunde, Minute und Sekunde werden dabei auf `"00"` gesetzt:

```
>>> d = datetime.date(2007, 10, 23)
>>> d.ctime()
'Tue Oct 23 00:00:00 2007'
```

d.strftime(format)

Gibt den von d repräsentierten Tag formatiert aus, wobei der Parameter *format* die Beschreibung des gewünschten Ausgabeformats enthält.

Nähere Informationen können Sie in Abschnitt 16.1, »Elementare Zeitfunktionen – time«, unter `time.strftime` nachschlagen.

top

16.2.2 datetime.time ▼▲

In diesem Abschnitt werden wir uns mit den Methoden und Attributen des Datentyps `datettime.time` beschäftigen.

Objekte des Typs `datetime.time` dienen dazu, Tageszeiten anhand von Stunde, Minute, Sekunde und auch Mikrosekunde zu verwalten.

In dem Attribut `tzinfo` können `datetime.time`-Instanzen Informationen zur lokalen Zeitzone speichern und ihre Werte damit an die Lokalzeit anpassen. Dadurch ist es möglich, sowohl *naive* als auch *bewusste* `datettime.time`-Instanzen zu erzeugen.

Konstruktor von datetime.time

Ein neues `datetime.time`-Objekt kann mit dem folgenden Konstruktor erzeugt werden:

datetime.time([hour[, minute[, second[, microsecond[, tzinfo]]]]])

Die vier ersten Parameter legen den Zeitpunkt fest und müssen folgende Bedingungen erfüllen, wobei nur Ganzzahlen zugelassen sind:

- `0 <= hour < 24`
- `0 <= minute < 60`

- `0 <= second < 60`
- `0 <= microsecond < 1000000`

Die Standardbelegung für *hour*, *minute*, *second* und *microsecond* ist der Wert `0`.

Für den letzten Parameter namens *tzinfo* können Informationen über die lokale Zeitzone in Form einer `datetime.tztime`-Instanz übergeben werden. Wie das genau geht, können Sie der Python-Dokumentation entnehmen.

Klassenmember von datetime.time

Der Datentyp `datetime.time` verfügt über folgende Klassenmember:

datetime.time.min

Der früheste Zeitpunkt, der gespeichert werden kann, in der Regel date `time.time(0, 0, 0, 0)`.

datetime.time.max

Der späteste darstellbare Zeitpunkt, in der Regel `datetime.time(23, 59, 59, 999999)`.

datetime.time.resolution

Der minimale Unterschied zwischen zwei unterschiedlichen `datetime.time`-Objekten. Diese kleinstmögliche Zeiteinheit wird auch *Auflösung* genannt.

Attribute und Methoden von datetime.time-Instanzen

Nachfolgend wird angenommen, dass `t` eine Instanz des Datentyps date `time.time` ist.

t.hour

Speichert den Stundenanteil des Zeitpunkts *t*.

t.minute

Speichert den Minutenanteil des Zeitpunkts *t*.

t.second

Speichert den Sekundenanteil des Zeitpunkts *t*.

t.microsecond

Speichert den Mikrosekundenanteil (Mikrosekunde = Millionstelsekunde) des Zeitpunkts *t*.

t.tzinfo

Information zur lokalen Zeitzone. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.

t.replace([hour[, minute[, second[, microsecond[,tzinfo]]]]])

Analog zur `datetimte.date.replace`-Methode.

Eignet sich sehr gut dazu, aus einem *naiven* ein *bewusstes* `time`-Objekt zu machen, ohne dass die Daten-Member angepasst werden. Dazu ruft man `replace` mit *tzinfo* als einzigem Parameter auf.

t.isoformat()

Gibt einen String zurück, der den Zeitpunkt `t` im ISO-8601-Format enthält. Das Format ist folgendermaßen aufgebaut, wobei die »H« für die Ziffern der Stunde, die »M« für die Ziffern der Minute, die »S« für die Ziffern der Sekunden und die »m« für die Ziffern der Mikrosekunden stehen: »HH:MM:SS:mmmmmm«.

Ist das `microseconds`-Attribut von `t` gleich `0`, entfallen die Mikrosekunden, und das Format verkürzt sich auf »HH:MM:SS«.

t.strftime(format)

Erzeugt einen String, der den Zeitpunkt `t` nach der Formatbeschreibung in *format* enthält. Näheres dazu können Sie unter `time.strftime` nachlesen.

t.utcoffset()

Wenn `t` ein *bewusstes* Objekt ist, also `t.tzinfo` nicht den Wert `None` hat, gibt `t.utcoffset` den Wert zurück, der von `t.tzinfo.utcoffset(None)` erzeugt wird.

Dies sollte die Verschiebung der Lokalzeit relativ zur UTC in Sekunden sein.

t.tzname()

Gibt den Namen der Zeitzone zurück, wenn `t.tzinfo` nicht den Wert `None` hat. Ist `t.tzinfo` gleich `None`, wird stattdessen `None` zurückgegeben.

(Der Wert wird dadurch bestimmt, dass intern `t.tzinfo.tzname(None)` aufgerufen wird.)

top

16.2.3 datetime.datetime

In den meisten Fällen werden die Fähigkeiten der Datentypen `datetime.date` und `datetime.time` jeweils einzeln nicht ausreichen, um Zeitpunkte zu verwalten, da Zeitangaben in der Regel aus einem Datum und der Uhrzeit an dem jeweiligen Tag bestehen.

Der Datentyp `datetime.datetime` ist genau das, was sein Name vermuten lässt: ein Typ zum Speichern von Datums- und Uhrzeitangaben. Er vereint dazu die Fähigkeiten von `datetime.date` und `datetime.time` in einem Datentyp.

Konstruktoren von datetime.datetime

Es gibt acht Konstruktoren, um neue `datetime.datetime`-Instanzen zu erzeugen:

datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])

Die Parameter haben die gleiche Bedeutung wie die gleichnamigen Elemente der Konstruktoren von `datetime.date` und `datetime.time`, weshalb hier auf eine Wiederholung verzichtet wird.

```
>>> bescherung = datetime.datetime(2007, 12, 24, 18, 30)
>>> bescherung
datetime.datetime(2007, 12, 24, 18, 30)
```

datetime.datetime.today()

Erzeugt eine `datetime.datetime`-Instanz, die die aktuelle Lokalzeit speichert. Das `tzinfo`-Attribut wird dabei immer auf `None` gesetzt.

```
>>> datetime.datetime.today()
datetime.datetime(2007, 9, 19, 18, 46, 29, 87000)
```

Achtung

Auch wenn der Name der Methode `today` (dt. *heute*) darauf schließen lassen könnte, dass nur die Attribute für das Datum und nicht die für die Zeit gesetzt werden, erzeugt `datetime.today` ein `datetime.datetime`-Objekt, das auch die Uhrzeit enthält.

datetime.now([tz])

Erzeugt eine `datetime.datetime`-Instanz mit dem aktuellen Datum und der aktuellen Zeit. Wird die Methode ohne Parameter aufgerufen, erzeugt sie das gleiche Ergebnis wie `datetime.datetime.today`.

Mit dem optionalen Parameter *tz* können Informationen zur Lokalzeit übergeben werden. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.

datetime.utcnow()

Gibt die aktuelle koordinierte Weltzeit (UTC) zurück, wobei das `tzinfo`-Attribut der resultierenden `datetime.datetime`-Instanz den Wert `None` hat.

datetime.fromtimestamp(timestamp[, tz])

Erzeugt eine `datetime.datetime`-Instanz, die den gleichen Zeitpunkt wie der für `timestamp` übergebene Unix-Zeitstempel repräsentiert.

Wird für *tz* kein Wert oder `None` übergeben, ist der Rückgabewert ein *naives* Zeitobjekt.

Wie mit dem Parameter *tz* Informationen zur Zeitzone übergeben werden, können Sie in der Python-Dokumentation nachlesen.

datetime.utcfromtimestamp(timestamp)

Wandelt den übergebenen Unix-Timestamp in ein `datetime.datetime`-Objekt um, das die koordinierte Weltzeit (UTC) speichert. Der Unix-Zeitstempel wird dabei als lokale Zeit interpretiert. Deshalb wird bei der Umwandlung nach UTC die Zeitverschiebung berücksichtigt:

```
>>> import time
>>> t = time.time()
>>> datetime.datetime.fromtimestamp(t)
datetime.datetime(2007, 9, 21, 1, 48, 13, 718000)
>>> datetime.datetime.utcfromtimestamp(t)
datetime.datetime(2007, 9, 20, 23, 48, 13, 718000)
```

Wie Sie sehen, liegen die von `fromtimestamp` und `utcfromtimestamp` gelieferten `datetime.datetime`-Objekte um genau zwei Stunden auseinander. Dies rührt daher, dass das Beispiel auf einem Computer mit deutscher Lokalzeit (UTC+1) während der Sommerzeit (noch einmal eine Stunde mehr nach vorne) ausgeführt wurde.

datetime.combine(date, time)

Erzeugt ein `datetime.datetime`-Objekt, das aus der Kombination von *date* und *time* hervorgeht. Der Parameter *date* muss eine `datetime.date`-Instanz enthalten, und der Parameter *time* muss auf ein `datetime.time`-Objekt verweisen.

Alternativ kann für *date* auch ein datetime.datetime-Objekt übergeben werden. In diesem Fall wird die in *date* enthaltene Uhrzeit ignoriert und nur das Datum betrachtet.

datetime.strptime(date_string, format)

Interpretiert den String, der als Parameter *date_string* übergeben wurde, gemäß der Formatbeschreibung aus *format* als Zeitinformation und gibt ein entsprechendes datetime.datetime-Objekt zurück.

Für die Formatbeschreibung gelten die gleichen Regeln wie bei time.strftime.

Operatoren für datetime.datetime

Der Datentyp datetime.datetime überlädt die Operatoren für die Subtraktion und Addition, sodass mit Zeitangaben gerechnet werden kann.

Dabei sind folgende Summen und Differenzen möglich, wobei d1 und d2 jeweils datetime.datetime-Instanzen sind und t ein datetime.timedelta-Objekt referenziert:

Ausdruck	Hinweise
d2 = d1 + t	Der von d2 beschriebene Zeitpunkt ergibt sich dadurch, dass in der Zeit von d1 aus um die von t beschriebene Zeitspanne in die Zukunft oder die Vergangenheit gegangen wird, je nachdem, ob der Wert von t positiv oder negativ ist. Das datetime.datetime-Objekt d2 übernimmt außerdem das tzinfo-Attribut von d1.
d2 = d1 – t	Wie bei der Addition, außer dass nun bei positivem t in Richtung Vergangenheit und bei negativem t in Richtung Zukunft gegangen wird.
t = d1 – d2	Das datetime.timedelta-Objekt t beschreibt den zeitlichen Abstand zwischen den Zeitpunkten d1 und d2. Dabei wird t so gewählt, dass d1 = d2 + t gilt. Diese Operation kann nur durchgeführt werden, wenn d1 und d2 *bewusst* oder beide *naiv* sind. Ist dies nicht der Fall, wird ein TypeError erzeugt. Die Details zu *naiven* und *bewussten* Zeitobjekten entnehmen Sie bitte der Python-Dokumentation.

Tabelle 16.3 Rechnen mit datetime.datetime

Es ist auch möglich, zwei datetime.datetime-Instanzen mit den Vergleichsoperatoren < und > zu vergleichen. Dabei gilt das Zeitobjekt als »kleiner«, das in der Zeit weiter in Richtung Vergangenheit liegt.

Beispiele für die Verwendung dieser Operatoren können Sie im Kapitel über datetime.date nachlesen, da die Verwendung für datetime.datetime analog erfolgt.

Statische und dynamische Attribute von datetime.datetime

Der Datentyp datetime.datetime besitzt die gleichen Member, die auch date time.date und datetime.time besitzen: min, max, resoultion, year, month, day, hour, minute, second und microsecond.

Die Bedeutung der einzelnen Member können Sie in den Kapiteln zu date time.date und datetime.time nachlesen.

Methoden von datetime.datetime-Instanzen

Im Folgenden wird angenommen, dass d eine Instanz des Datentyps date time.datetime ist.

d.date()

Gibt ein datetime.date-Objekt zurück, das die gleichen year-, month- und day-Attribute wie d hat.

d.time()

Gibt ein datetime.time-Objekt zurück, das die gleichen hour-, minute-, second- und microsecond-Attribute wie d hat.

d.timetz()

Wie d.time, aber es wird zusätzlich das tzinfo-Attribut mitkopiert.

d.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]])

Erzeugt eine neue datetime.datetime-Instanz, die aus *d* dadurch hervorgeht, dass die Attribute, die der replace-Methode übergeben wurden, durch die neuen Werte ersetzt werden.

d.utcoffset()

Wenn *d* ein *bewusstes* Objekt ist, also d.tzinfo nicht den Wert None hat, gibt d.utcoffset den Wert zurück, der von d.tzinfo.utcoffset(None) erzeugt wird. Dies sollte die Verschiebung der Lokalzeit relativ zur UTC in Sekunden sein.

d.tzname()

Gibt den Namen der Zeitzone zurück, wenn d.tzinfo nicht den Wert None hat. Ist d.tzinfo gleich None, wird stattdessen None zurückgegeben.

(Der Wert wird dadurch bestimmt, dass intern d.tzinfo.tzname(None) aufgerufen wird.)

d.timetuple()

Gibt ein time.struct_time-Objekt zurück, das den von d beschriebenen Zeitpunkt enthält.

d.utctimetuple()

Wenn d ein *naives* Zeitobjekt ist, also wenn d.tzinfo den Wert None hat, verhält sich d.utctimetuple genau wie d.timetuple.

Ist d ein *bewusstes* Zeitobjekt, wird sein Wert erst in die globale Weltzeit umgerechnet und dann als time.struct_time-Instanz zurückgegeben.

d.weekday()

Gibt den Wochentag als Zahl zurück, wobei Montag als 0 und Sonntag als 6 betrachtet wird.

d.isoweekday()

Gibt den Wochentag als Zahl zurück, wobei Montag den Wert 1 und Sonntag den Wert 7 ergibt.

d.isocalendar()

Gibt ein Tupel mit drei Elementen zurück, das den von d beschriebenen Tag als Datum im ISO-Kalender ausdrückt.

Näheres dazu finden Sie unter der Methode isocalendar des

Datentyps `date` `time.date`.

d.isoformat()

Gibt den von d beschriebenen Zeitpunkt im ISO-8601-Format zurück. Das Format ist folgendermaßen aufgebaut:

YYYY-MM-DDTHH:MM:SS.mmmmmm

Die »Y« stehen für die Ziffern der Jahreszahl, die »M« für die Ziffern der Monatszahl und die »D« für die Ziffern des Tages. Das große »T« ist ein Trennzeichen, das zwischen Datums- und Zeitangabe steht. In der Zeitangabe stehen die »H« für die Ziffern der Stunde, die »M« für die Ziffern der Minute und die »S« für die Ziffern der Sekunden.

Ist das `microseconds`-Attribut von d von `0` verschieden, werden die Mikrosekunden, durch einen Doppelpunkt abgetrennt, an das Ende des Strings geschrieben. Ansonsten entfällt der Mikrosekundenteil inklusive Doppelpunkt.

d.ctime()

Gibt einen String zurück, der den von d repräsentierten Zeitpunkt beschreibt:

```
>>> datetime.datetime(1987, 07, 26, 10, 15, 00).ctime()
'Sun Jul 26 10:15:00 1987'
```

d.strftime()

Erzeugt einen String, der den von *d* beschriebenen Zeitpunkt formatiert enthält.

Genaueres können Sie unter `time.strftime` nachlesen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Verarbeitung in elektronischen Systemen.

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

17.2 Umgang mit Pfaden – os.path

Verschiedene Plattformen – verschiedene Pfadnamenskonventionen. Während beispielsweise Windows-Betriebssysteme bei absoluten Pfadnamen das Laufwerk erwarten, auf das sich der Pfad bezieht, wird unter Unix ein einfacher Slash vorangestellt. Außerdem unterscheiden sich auch die Trennzeichen für einzelne Ordner innerhalb des Pfadnamens, denn Microsoft hat sich im Gegensatz zur Unix-Welt, in der der Slash üblich ist, für den Backslash entschieden.

Als Programmierer für plattformübergreifende Software stehen Sie nun vor dem Problem, dass Ihre Programme mit diesen verschiedenen Konventionen und auch denen dritter Betriebssysteme zurechtkommen müssen.

Damit dafür keine programmtechnischen Verrenkungen notwendig werden, wurde das Modul os.path entwickelt, mit dem Sie Pfadnamen komfortabel verwenden können.

Sie können das Modul auf zwei verschiedene Arten nutzen:

- Sie importieren erst os und greifen dann über os.path darauf zu.
- Sie importieren os.path direkt.

Bevor wir mit der Beschreibung der Funktionen dieses Moduls beginnen, möchten wir Sie drauf hinweisen, dass unter Windows nicht alle Funktionen korrekt mit UNC-Pfadnamen [Uniform/Universal Naming Convention (UNC) ist ein Standard, um Ressourcen in einem Netzwerk anzusprechen.] umgehen können. Nur für splitunc und ismount wird garantiert, dass sie mit solchen Pfaden richtig verfahren können.

os.path.abspath(path)

Gibt zu einem relativen Pfad den dazugehörigen absoluten und normalisierten Pfad (siehe dazu os.normpath) zurück. Das folgende Beispiel verdeutlicht die Arbeitsweise:

```
>>> os.path.abspath(".")
'Z:\\beispiele\\os'
```

In diesem Fall haben wir mithilfe des relativen Pfads "." auf das aktuelle Verzeichnis herausgefunden, dass unser Script unter 'Z:\\beispiele\\os' gespeichert ist.

os.path.basename(path)

Gibt den sogenannten *Basisnamen* des Pfads zurück. Der Basisname eines Pfads ist der Teil hinter dem letzten Ordnertrennzeichen, wie zum Beispiel \ oder /. Diese Funktion eignet sich sehr gut, um den Dateinamen aus einem vollständigen Pfad zu extrahieren:

```
>>> os.path.basename(r"C:\Windows\System32\ntoskrnl.exe")
'ntoskrnl.exe'
```

Wichtig

Diese Funktion unterscheidet sich von dem Unix-Kommando *basename* dadurch, dass sie einen leeren String zurückgibt, wenn der String mit einem Ordnertrennzeichen endet:

```
>>> os.path.basename(r"/usr/lib/compiz/") ''
```

Im Gegensatz dazu sieht die Ausgabe des gleichnamigen Unix-Kommandos so aus:

```
$ basename /usr/lib/compiz/ compiz
```

os.path.commonprefix(list)

Gibt einen möglichst langen String zurück, mit dem alle Elemente der als Parameter übergebenen Pfadliste *list* beginnen:

```
>>>
os.path.commonprefix([r"C:\Windows\System32\ntoskrnl.exe",
                      r"C:\Windows\System\TAPI.dll",
                      r"C:\Windows\system32\drivers"])
'C:\\Windows\\'
```

Es ist aber nicht garantiert, dass der resultierende String auch ein gültiger und existierender Pfad ist, da die Pfade als einfache Strings betrachtet werden.

os.path.dirname(path)

Gibt den Ordnerpfad zurück, den *path* enthält:

```
>>> os.path.dirname(r"C:\Windows\System\TAPI.dll")
'C:\\Windows\\System'
```

Genau wie bei `os.path.basename` müssen Sie auch hier das abweichende Verhalten bei Pfaden beachten, die mit einem Ordnertrennzeichen enden:

```
>>> os.path.dirname(r"/usr/lib/compiz")
'/usr/lib'
>>> os.path.dirname(r"/usr/lib/compiz/")
'/usr/lib/compiz'
```

os.path.exists(path)

Gibt `True` zurück, wenn der angegebene Pfad auf eine existierende Datei oder ein vorhandenes Verzeichnis verweist, ansonsten `False`.

os.path.getatime(path)

Gibt den Unix-Zeitstempel des letzten Zugriffs auf den übergebenen Pfad zurück. Kann auf die übergebene Datei oder den Ordner nicht zugegriffen werden oder ist sie bzw. er nicht vorhanden, führt dies zu einem `os.error`.

Unix-Zeitstempel sind Ganzzahlen, die die Sekunden seit Beginn der Unix-Epoche, also dem 01.01.1970, angeben.

os.path.getmtime(path)

Gibt einen Unix-Zeitstempel zurück, der angibt, wann die Datei oder der Ordner unter *path* zum letzten Mal verändert wurde. Existiert der übergebene Pfad nicht im Dateisystem, wird `os.error` geworfen.

Unix-Zeitstempel sind Zahlen, die die Sekunden seit Beginn der Unix-Epoche, also dem 01.01.1970 um 00:00 Uhr, angeben.

os.path.getsize(path)

Gibt die Größe der unter *path* zu findenden Datei in Bytes zurück. Der Rückgabewert ist dabei immer eine `long`-Instanz.

os.path.isabs(path)

Der Rückgabewert ist `True`, wenn es sich bei *path* um eine absolute Pfadangabe handelt, sonst `False`.

os.path.isfile(path)

Gibt `True` zurück, wenn *path* auf eine Datei verweist, sonst `False`. Die Funktion folgt dabei gegebenenfalls symbolischen Links.

os.path.isdir(path)

Wenn der übergebene Pfad auf einen Ordner verweist, wird `True` zurückgegeben, ansonsten `False`.

os.path.islink(path)

Gibt `True` zurück, wenn unter *path* ein symbolischer Link zu finden ist, sonst `False`.

os.path.join(path1[, path2[, ...]])

Fügt die übergebenen Pfadangaben zu einem einzigen Pfad

zusammen, indem sie verkettet werden:

```
>>> os.path.join(r"C:\Windows", r"System\ntoskrnl.exe")
'C:\\Windows\\System\\ntoskrnl.exe'
```

Wird ein absoluter Pfad als zweites oder späteres Argument übergeben, ignoriert `os.path.join` alle übergebenen Pfade vor dem absoluten:

```
>>> os.path.join(r"Das\wird\ignoriert", r"C:\Windows",
r"System\ntoskrnl.exe")
'C:\\Windows\\System\\ntoskrnl.exe'
```

os.path.normcase(path)

Auf Betriebssystemen, die bei Pfaden nicht hinsichtlich Groß- und Kleinschreibung unterscheiden (z. B. Windows), werden alle Großbuchstaben durch ihre kleinen Entsprechungen ersetzt. Außerdem werden unter Windows alle Slashs durch Backslashs ausgetauscht:

```
>>> os.path.normcase(r"C:\Windows/System32/ntoskrnl.exe")
'c:\\windows\\system32\\ntoskrnl.exe'
```

Unter Unix wird der übergebene Pfad ohne Änderung zurückgegeben.

os.path.realpath(path)

Gibt einen zu *path* äquivalenten Pfad zurück, der keine Umwege über symbolische Links enthält.

os.path.split(path)

Teilt den übergebenen Pfad in den Namen des Ordners oder der Datei, die er beschreibt, und den Pfad zu dem direkt übergeordneten Verzeichnis und gibt ein Tupel zurück, das die beiden Teile enthält:

```
>>> os.path.split(r"C:\Windows\System32\ntoskrnl.exe")
('C:\\Windows\System32', 'ntoskrnl.exe')
```

Wichtig

Wenn der Pfad mit einem Slash oder Backslash endet, ist das zweite Element des Tupels ein leerer String:

```
>>> os.path.split("/home/revelation/")
('/home/revelation', '')
```

os.path.splitdrive(path)

Teilt den übergebenen Pfad in die Laufwerksangabe und den Rest, sofern die Plattform Laufwerksangaben unterstützt:

```
>>>
os.path.splitdrive(r"C:\Windows/System32/ntoskrnl.exe")
('C:', '\\Windows/System32/ntoskrnl.exe')
```

os.path.splitext(path)

Teilt den *path* in den Pfad zu der Datei und die Dateiendung. Beide Elemente werden in einem Tupel zurückgegeben:

```
>>> os.path.splitext(r"C:\Windows\System32\Notepad.exe")
```

```
('C:\\Windows\\System32\\Notepad', '.exe')
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

17.3 Zugriff auf die Laufzeitumgebung – sys

Das Modul sys der Standardbibliothek stellt Konstanten und Funktionen zur Verfügung, die sich auf den Python-Interpreter selbst beziehen oder eng mit diesem zusammenhängen. So kann über das Modul sys beispielsweise die Versionsnummer des Interpreters oder des Betriebssystems abgefragt werden. Das Modul stellt dem Programmierer eine Reihe von Informationen zur Verfügung, die mitunter sehr nützlich sein können. Es lohnt sich also, sich einen Überblick über die Funktionalität von sys zu verschaffen, allein schon, um einen Begriff davon zu bekommen, an welche Informationen Sie durch dieses Modul gelangen können.

Um die Beispiele des Kapitels ausführen zu können, muss zuvor das Modul sys eingebunden werden:

```
>>> import sys
```

top

17.3.1 Konstanten

Das Modul sys enthält eine ganze Reihe von Konstanten, die mitunter sehr nützliche Informationen bereitstellen. Die wichtigsten dieser Konstanten sollen im Folgenden erklärt werden.

sys.argv

Die Liste sys.argv enthält die Kommandozeilenparameter, mit denen das Python-Programm aufgerufen wurde. sys.argv[0] ist der Name des Programms selbst. Im interaktiven Modus hat sys.argv die Länge 0. Bei dem Programmaufruf

```
programm.py -bla 0 -blubb abc
```

würde sys.argv folgende Liste referenzieren:

```
['programm.py', '-bla', '0', '-blubb', 'abc']
```

Verwenden Sie das Modul optparse, wenn Sie Kommandozeilenparameter komfortabel verwalten möchten.

sys.byteorder

Diese Konstante spezifiziert die Byte-Order des aktuellen Systems. Der Wert ist entweder "big" für ein *Big-Endian*-System, bei dem das signifikanteste Byte an erster Stelle gespeichert wird, oder "little" für ein *Little-Endian*-System, bei dem das am wenigsten signifikante Byte zuerst gespeichert wird.

sys.executable

Dies ist ein String, der den vollen Pfad zur ausführbaren Datei des Python-Interpreters angibt.

```
>>> sys.executable
'C:\\Python25\\pythonw.exe'
```

sys.hexversion

Diese Konstante enthält die Versionsnummer des Python-Interpreters als ganze Zahl. Wenn sie durch Aufruf der Built-in Function hex als Hexadezimalzahl geschrieben wird, wird der Aufbau der Zahl deutlich:

```
>>> hex(sys.hexversion)
'0x20501f0'
```

In diesem Fall wurde Python 2.5.1 verwendet. Es ist garantiert, dass hexversion mit jeder Python-Version immer größer wird, dass man also mit den Operatoren < und > testen kann, ob die verwendete Version des Interpreters aktueller ist als eine bestimmte, die für die Ausführung des Programms mindestens vorausgesetzt wird.

sys.maxint

Diese Konstante enthält den größtmöglichen Wert, der in einer Instanz des Datentyps int gespeichert werden kann. Dieser Wert hängt von dem zugrunde liegenden System ab, ist aber mindestens $2^{31} - 1$ (32-Bit). Der kleinstmögliche Wert entspricht -sys.maxint -1.

sys.maxunicode

Diese Konstante enthält den größtmöglichen Zeichencode, den ein Unicode-Zeichen haben kann. Dieser Wert hängt davon ab, welche Unicode-Darstellung intern verwendet wird.

sys.modules

Das Dictionary `sys.modules` enthält die Namen aller momentan eingebundenen Module als Schlüssel und die dazugehörigen Namespaces als jeweiligen Wert.

sys.path

Die Liste `sys.path` enthält eine Reihe von Pfadangaben, die beim Einbinden eines Moduls der Reihe nach vom Interpreter durchsucht werden. Das zuerst gefundene Modul mit dem in einer `import`-Anweisung angegebenen Namen wird eingebunden.

Es steht dem Programmierer frei, die Liste so zu modifizieren, dass das Einbinden eines Moduls nach seinen Wünschen erfolgt.

```
>>> import sys
>>> sys.path
['C:Python25\\Lib\\idlelib',
'C:\\WINDOWS\\system32\\python25.zip',
'C:\\Python25\\DLLs',
'C:\\Python25\\lib',
'C:\\Python25\\lib\\plat-win',
'C:Python25\\lib\\lib-tk',
'C:\\Python25',
'C:\\Python25\\lib\\site-packages']
```

sys.platform

Dieser String enthält eine Kennung des zugrunde liegenden Betriebssystems. Der Wert ist beispielsweise `"win32"` für Windows oder `"linux2"` für Linux.

Diese Kennung könnte beispielsweise dazu verwendet werden, um plattformspezifische Pfade an `sys.path` anzuhängen.

sys.stdin, sys.stdout, sys.stderr

Dies sind die Dateiobjekte, die für Ein- und Ausgaben des Interpreters verwendet werden. Dabei steht `sys.stdin` für *standard input* und entspricht dem Dateiobjekt, aus dem die Benutzereingaben beim Aufruf von `input` oder `raw_input` gelesen werden. In das Dateiobjekt `sys.stdout` (*standard output)* werden alle Ausgaben des Python-Programms geschrieben, während Ausgaben des Interpreters, beispielsweise Tracebacks, in `sys.stderr` (*standard error*) geschrieben werden.

Das Überschreiben dieser vorbelegten Dateiobjekte mit eigenen Dateiobjekten erlaubt es, Ein- und Ausgaben auf andere Streams, beispielsweise in eine Datei, umzulenken. Beachten Sie dabei, dass `sys.stdin` stets ein vollwertiges Dateiobjekt sein muss, während für `sys.stdout` und `sys.stderr` eine Instanz reicht, die eine Methode `write` implementiert.

Die ursprünglichen Streams von `sys.stdin`, `sys.stdout` und `sys.stderr` werden in `sys.__stdin__`, `sys.__stdout__` und `sys.__stderr__` gespeichert, sodass sie jederzeit wiederhergestellt werden können.

sys.version

Ein String, der die Versionsnummer des Python-Interpreters und einige weitere Informationen, wie beispielsweise das Datum seiner Kompilierung und den verwendeten Compiler, enthält.

```
>>> sys.version
'2.5.1 (r251:54863, Apr 19 2007, 11:03:39) \n[GCC 4.1.2]'
```

Beachten Sie, dass es bei sys.version im Gegensatz zu sys.hexversion nicht garantiert ist, dass die Versionsnummern mit den Operatoren > und < sinnvoll miteinander verglichen werden können.

sys.version_info

Ein Tupel, das die einzelnen Komponenten der Versionsnummer des Interpreters enthält.

```
>>> sys.version_info
(2, 5, 1, 'final', 0)
```

top

17.3.2 Exceptions ▼▲

Das Modul sys enthält einige Funktionen, die speziell dazu gedacht sind, Zugriff auf geworfene Exceptions zu erhalten oder anderweitig mit Exceptions zu arbeiten.

Näheres dazu, wie Sie das in diesem Kapitel angesprochene *Traceback-Objekt* verwenden können, erfahren Sie in Abschnitt 21.6.

sys.exc_info()

Diese Funktion ermöglicht es, Zugriff auf eine momentan abgefangene Exception zu erlangen. Momentan abgefangen bedeutet, dass sich der Kontrollfluss innerhalb eines except-Zweiges einer try/except-Anweisung befinden muss, damit diese Funktion einen sinnvollen Wert zurückgibt.

Die Funktion exc_info gibt ein Tupel zurück, das drei Werte enthält: den Exception-Typ, die geworfene Instanz des Exception-Typs und das entsprechende Traceback-Objekt.

Beachten Sie, dass die Informationen über die aktuell abgefangene Exception nicht erhalten bleiben, sondern nur innerhalb des except-Zweiges verwendbar sind. Sollten Sie Informationen über die zuletzt geworfene Exception außerhalb eines except-Zweiges benötigen, sollten Sie last_type, last_value oder last_traceback verwenden.

sys.exc_clear()

Löscht die Informationen über die aktuell abgefangene Exception. Dies geschieht am Ende eines except-Zweiges automatisch.

sys.last_type, sys.last_value, sys.last_traceback

Erlauben es, Zugriff auf die zuletzt geworfene Exception zu erlangen. Die drei Informationen entsprechen denen, die von sys.exc_info zurückgegeben werden.

Beachten Sie, dass diese Konstanten auch außerhalb eines except-Zweiges Gültigkeit haben, da sie stets Informationen über die zuletzt geworfene Exception enthalten.

sys.tracebacklimit

Diese ganze Zahl kennzeichnet die maximale Tiefe, bis zu der ein Traceback Informationen über die Funktionshierarchie liefern soll. Initial ist dieser Wert auf 1000 gesetzt. Ein Wert von 0 veranlasst, dass ein Traceback nur aus dem Exception-Typ und der Fehlermeldung besteht.

top

17.3.3 Hooks ▼▲

Das Modul sys erlaubt den Zugriff auf sogenannte *Hooks* (dt. *Haken*). Das sind Funktionen, die bei gewissen Aktionen des Python-Interpreters aufgerufen werden. Durch Überschreiben dieser Funktionen kann sich der Programmierer in den Interpreter »einhaken« und so die Funktionsweise des Interpreters verändern.

sys.displayhook(value)

Diese Funktion wird immer dann aufgerufen, wenn das Ergebnis eines Ausdrucks im interaktiven Modus ausgegeben werden soll, also beispielsweise in der folgenden Situation:

```
>>> 42
42
```

Durch Überschreiben von sys.displayhook mit einer eigenen Funktion lässt sich dieses Verhalten ändern. Im folgenden Beispiel möchten wir erreichen, dass bei einem eingegebenen Ausdruck nicht das Ergebnis selbst, sondern die Identität des Ausdrucks ausgegeben wird:

```
>>> def f(value):
...     print id(value)
...
>>> sys.displayhook = f
>>> 42
134536524
>>> 97 + 32
134537456
>>> "Hallo Welt"
3083420560
```

Beachten Sie, dass sys.displayhook nicht aufgerufen wird, wenn eine Ausgabe mittels print getätigt wird: [Das wäre auch sehr ungünstig, da wir im Hook selbst ja eine print-Ausgabe tätigen. Würde eine print-Ausgabe wieder den Hook aufrufen, befänden wir uns in einer endlosen Rekursion.]

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

Das ursprüngliche Funktionsobjekt von sys.displayhook können Sie über sys.__displayhook__ erreichen und somit die ursprüngliche Funktionsweise wiederherstellen:

```
>>> sys.displayhook = sys.__displayhook__
```

sys.excepthook(type, value, traceback)

Diese Funktion wird immer dann aufgerufen, wenn eine nicht abgefangene Exception auftritt. Sie ist dafür verantwortlich, den Traceback auszugeben. Durch Überschreiben dieser Funktion mit einem eigenen Funktionsobjekt lässt sich zum Beispiel die Ausgabe eines Tracebacks verändern.

Die drei Parameter der Funktion entsprechen denen, die von sys.exc_info zurückgegeben werden, und enthalten Informationen über die Exception.

Im folgenden Beispiel möchten wir einen Hook einrichten, damit bei einer nicht abgefangenen Exception kein dröger Traceback mehr, sondern ein hämischer Kommentar ausgegeben wird:

```
>>> def f(type, value, traceback):
...     print "gnahahaha: '%s'" % value
...
>>> sys.excepthook = f
>>> abc
gnahahaha: 'name 'abc' is not defined'
```

Das ursprüngliche Funktionsobjekt von sys.excepthook können

Sie über sys.__excepthook__ erreichen und somit die ursprüngliche Funktionsweise wiederherstellen.

top

17.3.4 Sonstige Funktionen

Neben den bereits besprochenen Konstanten sowie den exception- bzw. hook-bezogenen Funktionen stellt das Modul sys einige weitere Funktionen bereit, um an Informationen über den Interpreter oder das Betriebssystem zu gelangen oder mit dem System zu interagieren.

sys.exit([arg])

Wirft eine SystemExit-Exception. Diese hat, sofern sie nicht abgefangen wird, zur Folge, dass das Programm ohne Traceback-Ausgabe beendet wird.

Als optionaler Parameter arg kann, wenn es sich um eine ganze Zahl handelt, ein *Exit Code* ans Betriebssystem übergeben werden. Ein Exit Code von 0 steht im Allgemeinen für ein erfolgreiches Beenden des Programms, und ein Exit Code ungleich 0 steht für einen Programmabbruch aufgrund eines Fehlers.

Wenn eine andere Instanz für arg übergeben wurde, beispielsweise ein String, wird diese nach sys.stderr ausgegeben, bevor das Programm mit dem Exit Code 0 beendet wird.

sys.getrefcount(object)

Gibt den aktuellen *Reference Count* für die übergebene Instanz object zurück. Der Reference Count ist eine ganze Zahl und entspricht der Anzahl von Referenzen, die auf eine Instanz bestehen. Wenn eine Instanz einen Reference Count von 0 hat, wird sie vom Garbage Collector entsorgt.

Beachten Sie, dass es dem Interpreter bei Instanzen unveränderlicher Datentypen frei steht, eine neue Instanz zu erzeugen oder eine bereits bestehende neu zu referenzieren. Aus diesem Grund kann es vorkommen, dass zum Beispiel Instanzen ganzer Zahlen einen hohen Reference Count haben.

sys.getrecursionlimit(), setrecursionlimit(limit)

Mit diesen Funktionen kann die maximale Rekursionstiefe ausgelesen oder verändert werden. Die maximale Rekursionstiefe ist mit 1000 vorbelegt und bricht endlos rekursive Funktionsaufrufe ab, bevor diese zu einem Speicherüberlauf führen können.

sys.getwindowsversion()

Erlaubt es, die Details über die Version des aktuell verwendeten Windows-Betriebssystems auszulesen. Die Funktion gibt ein Tupel zurück, dessen erste drei Elemente ganze Zahlen sind und die Versionsnummer beschreiben. Das vierte Element ist ebenfalls eine ganze Zahl und beschreibt die verwendete Plattform. Folgende Werte sind hier gültig:

Plattform	Bedeutung
0	Windows 3.1 (32-Bit)
1	Windows 95/98/ME
2	Windows NT/2000/XP/2003/Vista
3	Windows CE

Tabelle 17.3 Windows-Plattformen

Das letzte Element des Tupels ist ein String, der weiterführende Informationen enthält.

```
>>> sys.getwindowsversion()
(5, 1, 2600, 2, 'Service Pack 2')
```

Unter anderen Betriebssystemen als Microsoft Windows ist die Funktion `sys.getwindowsversion` nicht verfügbar.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

17.4 Informationen über das System – platform

Das Modul `platform` der Standardbibliothek stellt Informationen über das Betriebssystem bzw. die zugrunde liegende Hardware bereit. Diese Informationen sind teilweise deckungsgleich mit denen, auf die über das Modul sys zugegriffen werden kann. Aus diesem Grund werden wir hier nur die wichtigsten Funktionen erläutern.

top

17.4.1 Funktionen

platform.machine()

Gibt die Prozessorarchitektur des PCs als String zurück. Bei aktuellen Prozessoren ist dies `i686`.

platform.node()

Gibt den Netzwerknamen des PCs als String zurück.

platform.processor()

Gibt einen String zurück, der den Typ und den Hersteller des Prozessors enthält.

platform.system()

Gibt einen String zurück, der den Namen des Betriebssystems, beispielsweise also »Linux« oder »Windows«, enthält.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

17.6 Kopieren von Instanzen – copy

Wie Sie bereits wissen, wird in Python bei einer Zuweisung nur eine neue Referenz auf ein und dieselbe Instanz erzeugt, anstatt eine Kopie der Instanz zu erzeugen. Im folgenden Beispiel verweisen `s` und `t` auf dieselbe Liste, wie der Vergleich mit `is` offenbart:

```
>>> s = [1, 2, 3]
>>> t = s
>>> t is s
True
```

Dieses Vorgehen ist nicht immer erwünscht, weil Änderungen an der von `s` referenzierten Liste auch `t` über Seiteneffekte betreffen und umgekehrt.

Wenn beispielsweise eine Methode einer Klasse eine Liste zurückgibt, die auch innerhalb der Klasse verwendet wird, kann die Liste auch über die zurückgegebene Referenz verändert

werden, womit das Kapselungsprinzip verletzt wäre:

```
class MeineKlasse(object):
    def __init__(self):
        self.__Liste = [1, 2, 3]

    def getListe(self):
        return self.__Liste

    def zeigeListe(self):
        print self.__Liste
```

Wenn wir uns nun mittels der getListe-Methode eine Referenz auf die Liste zurückgeben lassen, können wir über einen Seiteneffekt das private Attribut __Liste der Instanz verändern:

```
>>> instanz = MeineKlasse()
>>> liste = instanz.getListe()
>>> liste.append(1337)
>>> instanz.zeigeListe()
[1, 2, 3, 1337]
```

Um dies zu verhindern, sollte die Methode getListe anstelle der Liste selbst eine Kopie derselben zurückgeben.

An dieser Stelle kommt das Modul copy ins Spiel, das dazu gedacht ist, echte Kopien einer Instanz zu erzeugen. Für diesen Zweck bietet copy zwei Funktionen an: copy.copy und copy.deepcopy. Beide Methoden erwarten als Parameter die zu kopierende Instanz und geben eine Referenz auf eine Kopie von ihr zurück: [Natürlich kann eine Liste auch per Slicing kopiert werden. Das Modul copy erlaubt aber das Kopieren beliebiger Instanzen.]

```
>>> import copy
>>> s = [1, 2, 3]
>>> t = copy.copy(s)
>>> t
[1, 2, 3]
>>> t is s
False
```

Das Beispiel zeigt, dass t zwar die gleichen Elemente wie s enthält, aber trotzdem nicht auf dieselbe Instanz wie s referenziert, sodass der Vergleich mit is negativ ausfällt.

Der Unterschied zwischen copy.copy und copy.deepcopy besteht darin, wie mit Referenzen umgegangen wird, die die zu kopierenden Instanzen enthalten. Die Funktion copy.copy erzeugt zwar eine neue Liste, aber die Referenzen innerhalb der Liste verweisen trotzdem auf dieselben Elemente. Mit copy.deepcopy hingegen wird die Instanz selbst kopiert und anschließend rekursiv auch alle von ihr referenzierten Instanzen.

Wir veranschaulichen diesen Unterschied anhand einer Liste, die eine weitere Liste enthält:

```
>>> liste = [1, [2, 3]]
>>> liste2 = copy.copy(liste)
>>> liste2.append(4)
>>> liste2
[1, [2, 3], 4]
>>> liste
[1, [2, 3]]
```

Wie erwartet verändert sich beim Anhängen des neuen Elements 4 an liste2 nicht die von liste referenzierte Instanz. Wenn wir aber die innere Liste [2, 3] verändern, betrifft dies sowohl liste als auch liste2:

```
>>> liste2[1].append(1337)
>>> liste2
[1, [2, 3, 1337], 4]
>>> liste
[1, [2, 3, 1337]]
```

Der is-Operator zeigt uns den Grund für dieses Verhalten: Bei liste[1] und liste2[1] handelt es sich um dieselbe Instanz:

```
>>> liste[1] is liste2[1]
True
```

Arbeiten wir stattdessen mit copy.deepcopy, wird die Liste inklusive aller enthaltenen Elemente kopiert:

```
>>> liste = [1, [2, 3]]
>>> liste2 = copy.deepcopy(liste)
>>> liste2[1].append(4)
>>> liste2
[1, [2, 3, 4]]
>>> liste
[1, [2, 3]]
>>> liste[1] is liste2[1]
False
```

Sowohl die Manipulation von liste2[1] als auch der is-Operator zeigen, dass es sich bei liste2[1] und liste[1] um verschiedene Instanzen handelt.

Es gibt allerdings Datentypen, die sowohl von copy.copy als auch von copy.deepcopy nicht wirklich kopiert, sondern nur ein weiteres Mal referenziert werden. Dazu zählen unter anderem Modul-Objekte, Methoden, file-Objekte, socket-Instanzen und traceback-Instanzen.

Hinweis

Beim Kopieren einer Instanz mithilfe des copy-Moduls wird das Objekt ein weiteres Mal im Speicher erzeugt. Dies kostet erheblich mehr Speicherplatz und Rechenzeit als eine einfache Zuweisung. Deshalb sollten Sie copy wirklich nur dann benutzen, wenn Sie tatsächlich eine echte Kopie brauchen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

17.7 Zugriff auf das Dateisystem – shutil

Das Modul `shutil` ist als Ergänzung zu `os` und `os.path` anzusehen und definiert abstrakte Funktionen, die insbesondere das Kopieren und Entfernen von Dateien betreffen, ohne dass man die dazu erforderlichen plattformabhängigen Programme wie beispielsweise copy unter Windows oder cp auf Unix-Maschinen kennen muss.

Folgende Funktionen werden von `shutil` implementiert, wobei die Parameter *src* und *dst* jeweils Strings sind, die den Pfad der Quell- bzw. der Zieldatei enthalten:

shutil.copyfile(src, dst)

Kopiert die Datei unter *src* nach *dst*. Wenn die Datei unter *dst* bereits existiert, wird sie überschrieben.

Dabei muss der Pfad *dst* schreibbar sein. Ansonsten wird ein `IOError` geworfen.

shutil.copyfileobj(fsrc, fdst[, length])

Kopiert den Inhalt des zum Lesen geöffneten Dateiobjekts *fsrc* in das zum Schreiben geöffnete *fdst*-Objekt.

Mit dem optionalen Parameter *length* kann dabei die zu verwendende Zwischenspeichergröße in Bytes angegeben werden. Ist *length* positiv, wird die *fsrc* portionsweise ausgelesen und nach *fdst* geschrieben, während bei negativem *length* zuerst der gesamte Inhalt von *fsrc* in den Speicher gelesen und dann in einem Rutsch nach *fdst* geschrieben wird. Standardmäßig wird ein positiver Wert für *length* verwendet, den das System wählt.

shutil.copymode(src, dst)

Kopiert die Zugriffsrechte vom Pfad *src* auf den Pfad *dst*. Dabei bleiben der Inhalt von *dst* sowie der Besitzer und die Gruppe unangetastet.

Beide Pfade müssen bereits im Dateisystem existieren.

shutil.copystat(src, dst)

Wie `shutil.copymode`, aber es werden zusätzlich die Zeiten für den letzten Zugriff in die letzte Modifikation kopiert.

shutil.copy(src, dst)

Kopiert die Datei unter dem Pfad *src* nach *dst*. Der Parameter *dst* kann dabei einen Pfad zu einer Datei enthalten, die dann erzeugt oder überschrieben wird. Verweist *dst* auf einen Ordner, wird eine neue Datei mit dem Dateinamen von *src* im Ordner *dst* erzeugt oder gegebenenfalls überschrieben.

Die Zugriffsrechte werden dabei mitkopiert.

shutil.copy2(src, dst)

Genau wie `shutil.copy`, aber es werden zusätzlich die Zeiten des letzten Zugriffs und der letzten Änderung kopiert.

shutil.copytree(src, dst[, symlinks])

Kopiert die gesamte Verzeichnisstruktur unter *src* nach *dst*. Der Pfad *dst* darf dabei nicht auf einen bereits existierenden Ordner verweisen, und es werden alle fehlenden Verzeichnisse des Pfads *dst* erzeugt.

Die Rechte der erzeugten Ordner und Dateien werden mittels `shutil.copystat` gesetzt, und Dateien werden mit `shutil.copy2` kopiert.

Mit dem optionalen Parameter *symlinks* wird angegeben, wie mit symbolischen Links verfahren werden soll. Hat *symlinks* den Wert `False` oder wird *symlinks* nicht angegeben, werden die verlinkten Dateien oder Ordner selbst in die kopierte Verzeichnisstruktur eingefügt. Bei einem *symlinks*-Wert von `True` werden nur die Links kopiert.

shutil.rmtree(src[, ignore_errors[, onerror]])

Löscht die gesamte Verzeichnisstruktur unter *src*. Für *ignore_errors* kann ein Wahrheitswert übergeben werden, der angibt, ob beim Löschen auftretende Fehler ignoriert oder von der Funktion, die für *onerror* übergeben wurde, behandelt werden sollen. Wird *ignore_errors* nicht angegeben, ruft jeder auftretende Fehler eine `Exception` hervor.

Wenn Sie *onerror* angeben, muss es eine Funktion sein, die drei

Parameter erwartet:

- `function` – eine Referenz auf die Funktion, die den Fehler verursacht hat. Dies können `os.listdir`, `os.remove` oder `os.rmdir` sein.
- `path` – der Pfad, für den der Fehler auftrat.
- `Excinfo` – der Rückgabewert von `sys.exc_info` im Kontext des Fehlers.

Achtung

Exceptions, die von der Funktion `onerror` geworfen werden, werden nicht abgefangen.

shutil.move(src, dst)

Verschiebt rekursiv die Datei oder den Ordner von *src* nach *dst*.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

17.8 Das Programmende – atexit

Mit dem Modul `atexit` lassen sich Funktionen registrieren, die nach Programmende aufgerufen werden sollen. Dies kann nützlich sein, um Daten zu sichern, Netzwerkverbindungen zu trennen oder sonstige Aufräumarbeiten durchzuführen.

Zu diesem Zweck implementiert `atexit` eine Funktion namens `register`, die als Parameter eine Referenz auf die Funktion erwartet, die am Programmende aufgerufen werden soll.

Im folgenden Beispiel wird eine einfache Funktion registriert, die eine Nachricht auf dem Bildschirm ausgibt:

```
import atexit
print "Programm gestartet"

def amEnde():
    print "Programm beendet"

atexit.register(amEnde)
```

Ein Programmlauf erzeugt nachstehende Ausgabe:

```
Programm gestartet
Programm beendet
```

Als zusätzliche Argumente können der `register`-Funktion beliebig viele Parameter übergeben werden, die beim Aufruf der registrierten Funktion an diese weitergereicht werden. Es können positionsbezogene Parameter und Schlüsselwortparameter gemischt werden.

Das folgende Beispiel lässt den Benutzer so lange neue Zeilen eintippen, bis dieser den String `"exit"` eingibt. Alle Eingaben werden in einer Liste verwaltet, die am Ende des Programms mit einer durch `atexit.register` registrierten Funktion in einer Datei gesichert werden:

```
import atexit
eingaben = []

def sichereEingaben(liste):
    open("eingaben.txt", "w").writelines("\n".join(liste))

atexit.register(sichereEingaben, eingaben)

while True:
    zeile = raw_input()
    if zeile == "exit":
        break
    eingaben += [zeile]
```

Es ist auch möglich, mehrere Funktionen per `atexit.register` zu registrieren, indem `atexit.register` für jede dieser Funktionen aufgerufen wird. Diese werden dann am Programmende nacheinander aufgerufen.

Achtung

Es kann vorkommen, dass die von `atexit` registrierten Funktionen nicht aufgerufen werden: zum einen, wenn das Programm nicht normal, sondern durch eine nicht behandelte Ausnahme abgestürzt ist, oder zum anderen, wenn es durch ein Systemsignal direkt beendet wurde.

Zum anderen kann es Probleme geben, wenn das Programm selbst oder ein Modul die Funktion `sys.exitfunc` überschreibt. Wenn Sie selbst Module entwickeln, sollten Sie immer `atexit.register` anstelle von `sys.exitfunc` benutzen, um zu verhindern, dass Ihr Modul die Aufräumarbeiten des einbindenden Programms behindert.

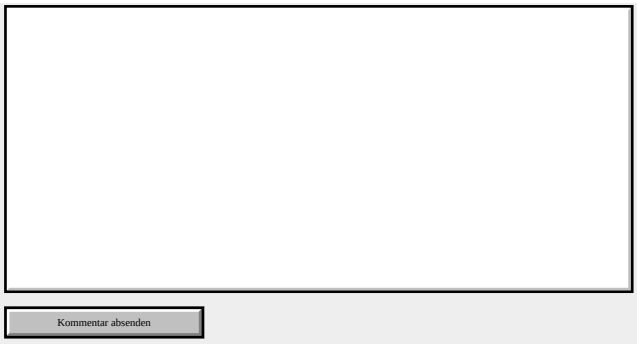
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

top

18.2 Die Thread-Unterstützung in Python

Python bietet zwei Module für den Umgang mit Threads an: `thread` und `threading`.

Das erste Modul namens `thread` ist die einfachere Variante und sieht jeden Thread als Funktion. Mit `threading` wird ein objektorientierter Ansatz implementiert, bei dem jeder Thread ein eigenes Objekt darstellt.

Wir werden uns mit beiden Ansätzen beschäftigen, wobei wir mit dem einfacheren Modul `thread` beginnen werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

top

18.3 Das Modul thread

Das Modul `thread` kann einzelne Funktionen in einem separaten Thread ausführen. Dazu dient die Funktion `thread.start_new_thread`, die mindestens zwei Parameter erwartet:

thread.start_new_thread(function, args[, kwargs])

Der Parameter *function* muss dabei eine Referenz auf die Funktion enthalten, die ausgeführt werden soll. Mit *args* muss eine `tuple`-Instanz übergeben werden, die die Parameter für *function* enthält.

Mit dem optionalen Parameter *kwargs* kann ein Dictionary übergeben werden, das zusätzliche Schlüsselwortparameter für die Funktion *function* bereitstellt.

Als Rückgabewert gibt `thread.start_new_thread` eine Zahl zurück, die den erzeugten Thread eindeutig identifiziert.

Nachdem *function* verlassen wurde, wird der Thread automatisch gelöscht.

Parallele Berechnung von Pi

Als Beispiel für das Multithreading werden wir eine Funktion entwickeln, mit der die Kreiszahl π mithilfe des Wallis'schen Produkts berechnet werden kann, das der englische Mathematiker John Wallis (1616 – 1703) im Jahre 1655 entdeckte:

Im Zähler stehen dabei immer gerade Zahlen, die sich bei jedem zweiten Faktor um 2 erhöhen. Der Nenner enthält nur ungerade Zahlen, die sich mit Ausnahme des ersten Faktors ebenfalls alle zwei Faktoren um 2 erhöhen.

Die Funktion naehere_pi_an, die als Parameter die Anzahl der zu berücksichtigenden Faktoren erhält, kann damit folgendermaßen definiert werden:

```
def naehere_pi_an(n):
    pi_halbe = 1
    zaehler, nenner = 2.0, 1.0

    for i in xrange(n):
        pi_halbe *= zaehler / nenner
        if i % 2:
            zaehler += 2
        else:
            nenner += 2

    print "Annaeherung mit %d Faktoren: %.16f" % (n,
2*pi_halbe)
```

Wenn für n der Wert 1000 übergeben wird, erzeugt die Funktion folgende Ausgabe, bei der nur die ersten beiden Nachkommastellen korrekt sind:

```
>>> naehere_pi_an(1000)
Annaeherung mit 1000 Faktoren: 3.140023818600586200
```

Wirklich brauchbare Näherungen werden erst für recht große n erzielt, was aber auch mit wesentlich mehr Rechenzeit bezahlt werden muss. Beispielsweise benötigte ein Aufruf mit n = 10000000 auf unserem Testrechner ca. sieben Sekunden.

Im nächsten Programm werden wir mithilfe von thread.start_new_thread mehrere Threads erzeugen, die die Funktion naehere_pi_an für verschiedene n aufrufen.

```
import thread

thread.start_new_thread(naehere_pi_an, (10000000,))
thread.start_new_thread(naehere_pi_an, (10000,))
thread.start_new_thread(naehere_pi_an, (99999999,))
thread.start_new_thread(naehere_pi_an, (123456789,))
thread.start_new_thread(naehere_pi_an, (,), {"n" : 1337})

while True:
    pass
```

Die Endlosschleife am Ende des Programms ist deshalb notwendig, damit der Thread des Hauptprogramms auf die anderen Threads wartet und nicht sofort beendet wird. Es ist nämlich so, dass alle Threads eines Programms sofort abgebrochen werden, wenn das Hauptprogramm sein Ende erreicht hat.

Eine Endlosschleife für diesen Zweck zu benutzen ist natürlich sehr unschön, weil sie Rechenleistung sinnlos vergeudet und das Programm mit Strg + C beendet werden muss. Wir werden erst bei dem Modul threading bessere Methoden kennenlernen, um einen Thread auf das Ende eines anderen warten zu lassen.

Das Interessante an diesem Programm ist die Reihenfolge der Ausgabe, die nicht mit der Reihenfolge der Aufrufe übereinstimmt:

```
Annaeherung mit 1337 Faktoren: 3.1427668611489281
Annaeherung mit 10000 Faktoren: 3.1414355935898644
Annaeherung mit 100000 Faktoren: 3.1415769458226377
Annaeherung mit 1234569 Faktoren: 3.1415939259321926
Annaeherung mit 11111111 Faktoren: 3.1415927949601699
```

Je größer das übergebene n war, desto länger musste auf die Ausgabe der dazugehörigen Annäherung von π gewartet werden, ganz egal, wann die Funktion gestartet wurde. Offensichtlich liefen alle Berechnungen parallel ab, wie wir es erwartet hatten.

Im letzten Beispiel hatte jeder Thread seine eigenen Variablen und musste keine Daten mit anderen Threads austauschen. Im nächsten Abschnitt werden wir uns mit dem Datenaustausch zwischen Threads beschäftigen.

top

18.3.1 Datenaustausch zwischen Threads – locking

Threads haben gegenüber Prozessen den Vorteil, dass sie sich dieselben globalen Variablen teilen und deshalb sehr einfach Daten austauschen können. Trotzdem gibt es ein paar Stolperfallen, die beim Zugriff auf dieselbe Variable durch mehrere Threads beachtet werden müssen.

Würde man beispielsweise unser vorhergehendes Beispiel um einen Zähler erweitern, der die Anzahl der zurzeit aktiven Threads enthält, damit das Programm nach dem Beenden aller Berechnungen von selbst terminiert, könnte man ganz naiv folgende Implementation vorschlagen:

```
import thread

anzahl_threads = 0

def naehere_pi_an(n):
    global anzahl_threads
    anzahl_threads += 1

    # hier wurde der Berechnungscode zur Übersicht
ausgelassen

    anzahl_threads -= 1

thread.start_new_thread(naehere_pi_an, (10000000,))
thread.start_new_thread(naehere_pi_an, (10000,))
thread.start_new_thread(naehere_pi_an, (99999999,))
thread.start_new_thread(naehere_pi_an, (123456789,))
thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while anzahl_threads > 0:
    pass
```

Dieses Programm hat zwei schwerwiegende Fehler. Erstens funktioniert es nicht, weil die `while`-Schleife erreicht wird, bevor überhaupt ein Thread gestartet werden konnte. Dies liegt einfach daran, dass die Zeitscheibe des Hauptprogramms nach den Aufrufen von `thread.start_new_thread` noch nicht aufgebraucht war und deshalb die Schleife zu laufen beginnt, bevor auch nur ein einziger Thread seine Arbeit aufgenommen hat.

Aber selbst, wenn dieses Problem bereits gelöst wäre, kann sich das Programm unter Umständen fehlerhaft verhalten. Die Gefahr lauert in den beiden Zeilen, die den Wert der globalen Variable `anzahl_threads` verändern:

Es ist theoretisch möglich, dass das Zeitfenster eines Threads genau während der Veränderung von `anzahl_threads` endet, denn Zuweisungen bestehen intern aus mehreren Schritten. Zuerst muss der Wert von `anzahl_threads` gelesen werden, dann muss eine neue Instanz mit dem um eins vergrößerten bzw. verringerten Wert erzeugt werden, die im letzten Schritt mit der Referenz `anzahl_threads` verknüpft wird.

Wenn ein Thread *A* nun beim Erhöhen von `anzahl_threads` während der Erzeugung der neuen Instanz schlafen gelegt wird, könnte ein anderer Thread *B* aktiviert werden, der ebenfalls `anzahl_threads` erhöhen möchte. Weil aber der Thread *A* seinen neuen Wert von `anzahl_threads` noch nicht berechnet und auch nicht mit der Referenz verknüpft hat, würde der neu aktivierte Thread *B* den alten Wert von `anzahl_threads` lesen und erhöhen. Wird dann später der Thread *A* wieder aktiv, erhöht er den schon vorher eingelesenen Wert um eins und weist ihn `anzahl_threads` zu. Das Ende vom Lied wäre ein um eins zu kleiner Wert von `anzahl_threads`, wodurch die Schleife im Hauptprogramm endlos laufen würde.

Die folgende Tabelle soll das beschriebene Szenario veranschaulichen:

Zeitfenster	Thread A	Thread B

1	Wert von `anzahl_threads` einlesen. Beispielsweise 2.	*schläft*
--------- Zeitfenster von *A* endet, und der Thread *B* wird aktiviert. ----------		
2	*schläft*	Wert von `anzahl_threads` einlesen. In diesem Fall 2. Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 3. Die neue Instanz an die Referenz `anzahl_threads` knüpfen. Damit verweist `anzahl_threads` auf den Wert 3.
--------- Zeitfenster von *B* endet, und der Thread *A* wird aktiviert. ----------		
3	Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 3. Die neue Instanz an die Referenz `anzahl_threads` knüpfen. Damit verweist `anzahl_threads` auf den Wert 3.	*schläft*

Tabelle 18.1 Problemszenario beim gleichzeitigen Zugriff auf eine globale Variable

Im Beispiel wurde `anzahl_threads` also nur um 1 erhöht, obwohl zwei neue Threads gestartet wurden.

Um solche Probleme zu vermeiden, kann ein Programm Stellen markieren, die nicht parallel in mehreren Threads laufen dürfen. Man bezeichnet solche Stellen auch als *Critical Sections* (dt. *kritische Abschnitte*).

Critical Sections werden durch sogenannte *Lock-Objekte* (von engl. *to lock = sperren*) realisiert. Mithilfe der parameterlosen Funktion `thread.allocate_lock` kann ein neues Lock-Objekt erzeugt werden:

```
lock_objekt = thread.allocate_lock()
```

Lock-Objekte haben die beiden wichtigen Methoden `acquire` und `release`, die jeweils beim Betreten bzw. beim Verlassen einer Critical Section aufgerufen werden müssen. Wenn die `acquire`-Methode eines Lock-Objekts aufgerufen wurde, ist es *gesperrt*. Ruft ein Thread die `acquire`-Methode eines gesperrten Lock-Objekts auf, muss er so lange warten, bis das Lock-Objekts wieder mit `release` freigegeben worden ist. Durch diese Technik wird verhindert, dass eine Critical Section von mehreren Threads gleichzeitig ausgeführt werden kann.

Wir können unser Beispielprogramm folgendermaßen um Critical Sections erweitern, wobei wir außerdem einen Schalter namens `thread_gestartet` einfügen, damit das Hauptprogramm mindestens so lange wartet, bis die Threads gestartet worden sind. Der Zugriff auf die Variablen `anzahl_threads` und `thread_gestartet` wird durch das Lock-Objekt `lock` gesichert:

```
import thread

anzahl_threads = 0
thread_gestartet = False

lock = thread.allocate_lock()

def naehere_pi_an(n):
    global anzahl_threads, thread_gestartet

    lock.acquire()
    anzahl_threads += 1
    thread_gestartet = True
    lock.release()

    # hier wurde der Berechnungscode zur Übersicht
ausgelassen

    lock.acquire()
    anzahl_threads -= 1
    lock.release()

thread.start_new_thread(naehere_pi_an, (100000,))
thread.start_new_thread(naehere_pi_an, (10000,))
thread.start_new_thread(naehere_pi_an, (11111111,))
thread.start_new_thread(naehere_pi_an, (1234569,))
thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while not thread_gestartet:
    pass

while anzahl_threads > 0:
    pass
```

Am Anfang des Programms wird der Schalter `thread_gestartet` auf `False` gesetzt, und mittels `thread.allocate_lock()` wird ein neues Lock-Objekt erzeugt. Innerhalb von `naehere_pi_an` gibt es dann eine Critical Section, in der `anzahl_threads` an die Anzahl der laufenden Threads angepasst bzw. die Variable `thread_gestartet` auf `True` gesetzt wird.

Die erste `while`-Schleife des Hauptprogramms sorgt nun dafür, dass auf jeden Fall so lange gewartet wird, bis ein Thread gestartet worden ist und den Wert von `thread_gestartet` auf `True` gesetzt hat. Die zweite Schleife sorgt wie gehabt dafür, dass das Programm so lange läuft, wie noch Threads ausgeführt werden.

Um die Wirkungsweise eines Lock-Objekts zu verdeutlichen, zeigt Ihnen die folgende Tabelle, wie unser Problemszenario durch die Critical Sections gelöst wird:

Zeitfenster	Thread A	Thread B
1	Das Lock-Objekt mit `lock.acquire()` sperren. Wert von `anzahl_threads` einlesen. Beispielsweise 2.	*schläft*
--------- Zeitfenster von *A* endet, und der Thread *B* wird aktiviert. ----------		
2	*schläft*	`lock.acquire` wird aufgerufen, aber das Lock-Objekt ist bereits gesperrt. Deshalb wird *B* schlafen gelegt.
--- *B* wurde durch `lock.acquire` schlafen gelegt. *A* wird weiter ausgeführt. ----		
3	Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 3. Die neue Instanz an die Referenz `anzahl_threads` knüpfen. Damit verweist `anzahl_threads` auf den Wert 3.	*schläft*

	Das Lock-Objekt wird mittels `lock.release()` wieder freigegeben.	
--------- Zeitfenster von *A* endet, und der Thread *B* wird aktiviert. ----------		
4	*schläft*	Das Lock-Objekt wird automatisch gesperrt, da *B* `lock.acquire` aufgerufen hat. Wert von `anzahl_threads` einlesen. In diesem Fall 3. Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 4. Die neue Instanz an die Referenz `anzahl_threads` knüpfen. Damit verweist `anzahl_threads` auf den Wert 4. Das Lock-Objekt wird mit `lock. release()` wieder freigegeben.

Tabelle 18.2 Lösung des »anzahl_threads«-Problems mit einem Lock-Objekt

Sie sollten darauf achten, dass Sie in Ihren eigenen Programmen alle Stellen, in denen Probleme durch Zugriffe von mehreren Threads vorkommen können, durch Critical Sections schützen.

Unzureichend abgesicherte Programme mit mehreren Threads können sehr schwer reproduzierbare und lokalisierbare Fehler produzieren. Die Herausforderung beim Umgang mit Threads besteht deshalb darin, solche Probleme zu umgehen.

Achtung

Wenn Sie mehrere Lock-Objekte verwenden, kann es passieren, dass sich ein Programm in einem sogenannten *Deadlock* aufhängt, weil zwei gelockte Threads gegenseitig aufeinander warten.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

18.4 Das Modul threading

Mit dem Modul `threading` wird eine objektorientierte Schnittstelle für Threads angeboten.

Jeder Thread ist dabei eine Instanz einer Klasse, die von `threading.Thread` erbt. Da die Klasse selbst ein Teil des globalen Namensraums ist, eignen sich ihre statischen Member sehr gut, um Daten zwischen den Threads auszutauschen. Natürlich muss auch hier der Zugriff auf die von mehreren Threads genutzten Variablen durch Critical Sections gesichert werden.

Wir wollen ein Programm schreiben, das in mehreren Threads parallel prüfen kann, ob vom Benutzer eingegebene Zahlen Primzahlen [Eine Primzahl ist eine natürliche Zahl, die genau zwei Teiler besitzt. Die ersten sechs Primzahlen sind demnach 2, 3, 5, 7, 11 und 13.] sind. Zu diesem Zweck definieren wir eine Klasse `PrimzahlThread`, die von `threading.Thread` erbt und als Parameter für den Konstruktor die zu überprüfende Zahl erwartet.

Die Klasse `threading.Thread` besitzt eine Methode namens `start`, die den Thread ausführt. Was genau ausgeführt werden soll, bestimmt die `run`-Methode, die wir mit unserer Primzahlberechnung überschreiben. Im ersten Schritt soll der Benutzer in einer Eingabeaufforderung Zahlen eingeben können, die dann überprüft werden. Ist die Überprüfung abgeschlossen, wird das Ergebnis auf dem Bildschirm ausgegeben. Das Programm inklusive der Klasse `PrimzahlThread` sieht dann folgendermaßen aus: [Der verwendete Algorithmus für die Primzahlprüfung ist sehr primitiv und dient hier nur als Beispiel für irgendeine rechenintensive Funktion.]

```
import threading
```

```
class PrimzahlThread(threading.Thread):
    def __init__(self, zahl):
        threading.Thread.__init__(self)
        self.Zahl = zahl

    def run(self):
        i = 2
        while i*i < self.Zahl:
            if self.Zahl % i == 0:
                print "%d ist nicht prim, da %d = %d * %d"
% (
                    self.Zahl, self.Zahl, i, self.Zahl /
i)
                return
            i += 1
        print "%d ist prim" % self.Zahl

meine_threads = []

while 1:
    eingabe = raw_input("> ")
    if eingabe == "ende":
        break

    thread = PrimzahlThread(long(eingabe))
    meine_threads.append(thread)
    thread.start()

for t in meine_threads:
    t.join()
```

Innerhalb der Schleife wird die Eingabe vom Benutzer eingelesen, und es wird geprüft, ob es sich um das Schlüsselwort "ende" zum Beenden des Programms handelt. Wurde etwas anderes als "ende" eingegeben, wird eine neue Instanz der Klasse `PrimzahlThread` mit der Benutzereingabe als Parameter erzeugt und mit der `start`-Methode gestartet.

Das Programm verwaltet außerdem eine Liste namens `meine_threads`, in der alle Threads gespeichert werden. Nach dem Verlassen der Eingabeschleife wird über `meine_threads` iteriert und für jeden Thread die `join`-Methode aufgerufen. Mit `join` wird dafür gesorgt, dass das Hauptprogramm so lange wartet, bis alle gestarteten Threads beendet worden sind, denn `join` unterbricht die Programmausführung so lange, bis der Thread, für den es aufgerufen wurde, terminiert wurde.

Diese Methode, auf das Ende aller Threads zu warten, ist wesentlich eleganter als die im letzten Abschnitt verwendete Endlosschleife, da mit `join` keine Rechenzeit verschwendet und das Programm automatisch beendet wird, sobald kein Thread mehr läuft.

Ein Programmlauf könnte dann so aussehen, wobei die teils verzögerten Ausgaben zeigen, dass tatsächlich parallel gerechnet wurde:

```
> 737373737373737
> 5672435793
5672435793 ist nicht prim, da 5672435793 = 3 * 1890811931
> 909091
909091 ist prim
> 10000000000037
> 5643257
5643257 ist nicht prim, da 5643257 = 23 * 245359
> 4567
4567 ist prim
10000000000037 ist prim
737373737373737 ist prim
> ende
```

top

18.4.1 Locking im threading-Modul ▼▲

Genau wie das Modul `thread` bietet auch `threading` Methoden an, um den Zugriff auf Variablen abzusichern, die in mehreren Threads verwendet werden. Die dazu benutzten Lock-Objekte lassen sich dabei genauso wie die von `thread.allocate_lock` zurückgegebenen Objekte verwenden.

Um den Umgang mit Lock-Objekten zu zeigen, werden wir das Primzahlprogramm des letzten Abschnitts verbessern. Eine

Schwachstelle des Programms bestand darin, dass, während der Benutzer gerade die nächste Zahl zur Prüfung eingibt, ein Thread im Hintergrund seine Arbeit beendet hat und sein Ergebnis auf den Bildschirm schreibt. Dadurch verliert der Benutzer unter Umständen die Übersicht, was er schon eingegeben hat, und es sieht äußerst unschön aus, wie das folgende Beispiel zeigt:

```
> 10000000000037
> 5610000000000037 ist prim
547
56547 ist nicht prim, da 56547 = 3 * 18849
> ende
```

In diesem Fall hat der Benutzer die Zahl 10000000000037 auf ihre Primzahleigenschaft hin untersuchen wollen. Unglücklicherweise wurde der Thread, der die Überprüfung übernahm, genau dann fertig, als der Benutzer bereits die ersten beiden Ziffern 56 der nächsten zu prüfenden Zahl, 56547, eingegeben hatte. Dies führte zu einer hässlichen »Zerstückelung« der Eingabe und sollte vermieden werden.

Wir werden zu diesem Zweck die Klasse `PrimzahlThread` mit einem statischen Attribut namens `Ergebnis` versehen, das in einem Dictionary die Ergebnisse der Berechnungen speichert. Dabei wird jeder zu prüfenden Zahl der Status bzw. das Ergebnis der Berechnung zugewiesen, wobei der Wert `"in Arbeit"` dafür steht, dass aktuell noch gerechnet wird, und der String `"prim"` anzeigt, dass es sich bei der Zahl um eine Primzahl handelt. Für Nicht-Primzahlen werden wir das gefundene Teilerprodukt in dem Dictionary speichern. Eine Momentaufnahme von `PrimzahlThread.Ergebnis` könnte dann folgendermaßen aussehen:

```
{
    737373737373737 : "in Arbeit",
    5672435793 : "3 * 1890811931",
    909091 : "prim",
    10000000000037 : "in Arbeit",
    5643257 :  "23 * 245359"
}
```

In dem Beispiel befinden sich die Zahlen 737373737373737 und 10000000000037 noch in der Prüfung, während für 909091 bereits nachgewiesen werden konnte, dass sie eine Primzahl ist. 5672435793 und 5643257 sind keine Primzahlen, da sie sich über die angegebenen Produkte berechnen lassen.

In dem neuen Programm wird der Benutzer wie bisher Zahlen eingeben und das Programm durch die Eingabe von `"ende"` terminieren können. Zusätzlich wird es einen Befehl `"status"` geben, der den aktuellen Berechnungsstand, eben den Inhalt von `PrimzahlThread.Ergebnis`, ausgibt.

Da die Threads zum Setzen der jeweiligen Ergebnisse alle `Primzahl Thread.Ergebnis` verändern müssen, ist es notwendig, den Zugriff auf das Dictionary mittels einer Critical Section abzusichern. Das dazu erforderliche Lock-Objekt speichern wir in der statischen Variable `PrimzahlThread.ErgebnisLock`. Das neue Programm sieht damit wie folgt aus:

```
import threading

class PrimzahlThread(threading.Thread):
    Ergebnis = {}
    ErgebnisLock = threading.Lock()

    def __init__(self, zahl):
        threading.Thread.__init__(self)
        self.Zahl = zahl

        PrimzahlThread.ErgebnisLock.acquire()
        PrimzahlThread.Ergebnis[zahl] = "in Arbeit"
        PrimzahlThread.ErgebnisLock.release()

    def run(self):
        i = 2
        while i*i < self.Zahl + 1:
            if self.Zahl % i == 0:
                ergebnis = "%d * %d" % (i, self.Zahl / i)

                PrimzahlThread.ErgebnisLock.acquire()
```

```
                PrimzahlThread.Ergebnis[self.Zahl] =
ergebnis
                PrimzahlThread.ErgebnisLock.release()

                return
            i += 1

        PrimzahlThread.ErgebnisLock.acquire()
        PrimzahlThread.Ergebnis[self.Zahl] = "prim"
        PrimzahlThread.ErgebnisLock.release()

meine_threads = []

while 1:
    eingabe = raw_input("> ")
    if eingabe == "ende":
        break

    elif eingabe == "status":
        print "-------- AktuellerStatus --------"
        PrimzahlThread.ErgebnisLock.acquire()
        for z, e in PrimzahlThread.Ergebnis.iteritems():
            print "%d: %s" % (z, e)
        PrimzahlThread.ErgebnisLock.release()
        print "-----------------------------------"

    elif long(eingabe) not in PrimzahlThread.Ergebnis:
        thread = PrimzahlThread(long(eingabe))
        meine_threads.append(thread)
        thread.start()

for t in meine_threads:
    t.join()
```

Wie Sie sehen, sind alle schreibenden Zugriffe auf `PrimzahlThread.Ergebnis` durch die Aufrufe von `acquire` und `release` umgeben, wodurch das Dictionary gefahrlos in verschiedenen Threads verändert werden kann. Da sich ein Dictionary außerdem nicht verändern darf, während darüber iteriert wird, muss auch die Statusausgabe durch eine Critical Section gesichert werden.

In der Schleife für die Verarbeitung der Benutzerdaten ist neben der Ausgabe des aktuellen Status noch eine Abfrage hinzugekommen, die verhindert, dass dieselbe Zahl unnötigerweise mehr als einmal überprüft wird.

Ein Beispiellauf des Programms könnte dann so aussehen:

```
> 10000000000037
> 5643257
> 909091
> 737373737373737
> 56547
> status
-------- Aktueller Status --------
5643257: 5643257 * 245359
909091: prim
737373737373737: in Arbeit
10000000000037: in Arbeit
56547: 56547 * 18849
-----------------------------------
> status
-------- Aktueller Status --------
5643257: 5643257 * 245359
909091: prim
737373737373737: in Arbeit
10000000000037: prim
56547: 56547 * 18849
-----------------------------------
> status
--------- Aktueller Status --------
5643257: 5643257 * 245359
909091: prim
737373737373737: prim
10000000000037: prim
56547: 56547 * 18849
-----------------------------------
> ende
```

Mit dieser Version des Programms werden die angesprochenen Probleme zufriedenstellend beseitigt. Allerdings kann immer noch ein kleiner Schönheitsfehler auftreten: Wenn der Benutzer sehr viele, sehr große Zahlen eingibt, kann es passieren, dass das Programm eine lange Zeit rechnet, bevor das erste Ergebnis erzielt wird. Das rührt daher, dass sich die Threads gegenseitig ausbremsen, weil zwar alle Threads gleichzeitig ausgeführt werden, aber durch ihre große Anzahl für den einzelnen Thread nur wenig Rechenleistung übrig bleibt.

Um auch diese Unschönheit zu beseitigen, werden wir im nächsten Abschnitt eine Technik kennenlernen, mit der wir die

Anzahl der Threads sinnvoll begrenzen können.

18.4.2 Worker-Threads und Queues ▼▲

In unseren bisherigen Programmen haben wir immer für jede Aufgabe einen neuen Thread gestartet, sodass es theoretisch beliebig viele Threads geben konnte. Wie am Ende des letzten Abschnitts angemerkt wurde, kann dies zu Geschwindigkeitsproblemen führen, wenn sehr viele Threads gleichzeitig laufen.

Dies lässt sich an einem Beispiel veranschaulichen: Wären wir ein Unternehmen, das für seine Kunden Zahlen daraufhin untersucht, ob sie Primzahlen sind, könnten wir uns unser Vorgehen so vorstellen, dass wir für jede Zahl, die wir überprüfen möchten, einen separaten Mathematiker einstellen, der mit den nötigen Berechnungen betraut wird. Hat der Mathematiker sein Werk vollendet, gibt er uns als Arbeitgeber Rückmeldung über das Ergebnis und wird entlassen.

In einem realen Unternehmen ist es nicht denkbar, für jede neue Aufgabe einen neuen Arbeiter einzustellen und diesen nach der Fertigstellung seiner Tätigkeit wieder zu entlassen. Vielmehr gibt es eine relativ konstante Anzahl von Arbeitern, denen die Aufgaben zugeteilt werden. Damit auch in diesem Modell eine beliebige Anzahl von Berechnungen durchgeführt werden kann, gibt es in unserer Firma einen Briefkasten, in den die Kunden die zu prüfenden Zahlen einwerfen können. Die Arbeiter holen sich dann selbstständig neue Aufgaben aus dem Briefkasten, sobald sie ihre vorherige Arbeit vollendet haben. Ist der Briefkasten einmal leer, warten die Arbeiter so lange, bis neue Zahlen eingeworfen werden.

In der Programmierung sprich man statt von Arbeitern von sogenannten *Worker-Threads* (von engl. *to work* = *arbeiten*). Der Briefkasten wird *Queue* (dt. *Warteschlange*) genannt.

Python hat ein eigenes Modul namens `Queue`, um mit Warteschlangen zu arbeiten. Der Konstruktor von `Queue` erwartet eine ganze Zahl als Parameter, die angibt, wie viele Elemente maximal in der Warteschlange stehen können. Ist der Parameter kleiner oder gleich `0`, ist die Länge der Queue nicht begrenzt.

Queue-Instanzen verfügen im Wesentlichen über drei wichtige Methoden: `put`, `get` und `task_done`.

Mit der `put`-Methode können neue Aufträge in die Warteschlage eingestellt werden. Sie wird in unserem Beispiel vom Hauptprogramm benutzt werden, um neue Zahlen in den »Briefkasten« zu werfen.

Die Methode `get` liefert die nächste Aufgabe der Queue. Befindet sich gerade kein Arbeitsauftrag in der Warteschlange, blockiert `get` den Thread so lange, bis der nächste Auftrag verfügbar ist.

Hat ein Thread die Prüfung einer Zahl abgeschlossen, muss er dies der Queue mitteilen, indem er `task_done` aufruft. Die Warteschlange kümmert sich dabei selbstständig darum, dass das fertig verarbeitete Element entfernt wird.

Das folgende Beispiel wird fünf Worker-Threads einsetzen, die sich alle eine Queue teilen:

```
import threading
import Queue

class Mathematiker(threading.Thread):
    Ergebnis = {}
    ErgebnisLock = threading.Lock()

    Briefkasten = Queue.Queue()

    def run(self):
        while True:
            zahl = Mathematiker.Briefkasten.get()
```

```
            ergebnis = self.istPrimzahl(zahl)

            Mathematiker.ErgebnisLock.acquire()
            Mathematiker.Ergebnis[zahl] = ergebnis
            Mathematiker.ErgebnisLock.release()

            Mathematiker.Briefkasten.task_done()

    def istPrimzahl(self, zahl):
        i = 2
        while i*i < zahl + 1:
            if zahl % i == 0:
                return "%d * %d" % (zahl, zahl / i)

            i += 1

        return "prim"


meine_threads = [Mathematiker() for i in range(5)]
for thread in meine_threads:
    thread.setDaemon(True)
    thread.start()

while True:
    eingabe = raw_input("> ")
    if eingabe == "ende":
        break

    elif eingabe == "status":
        print "-------- AktuellerStatus --------"
        Mathematiker.ErgebnisLock.acquire()
        for z, e in Mathematiker.Ergebnis.iteritems():
            print "%d: %s" % (z, e)
        Mathematiker.ErgebnisLock.release()
        print "---------------------------------"

    elif long(eingabe) not in Mathematiker.Ergebnis:
        Mathematiker.ErgebnisLock.acquire()
        Mathematiker.Ergebnis[long(eingabe)] = "in Arbeit"
        Mathematiker.ErgebnisLock.release()

        Mathematiker.Briefkasten.put(long(eingabe))

Mathematiker.Briefkasten.join()
```

Die neben dem Einbau der Queue wichtigen Änderungen im Vergleich zum letzten Programm sind zum einen die run-Methode, die jetzt in einer Endlosschleife immer wieder neue Zahlen aus dem Briefkasten nimmt und mit der `istPrimzahl`-Methode überprüft, und zum anderen die Initialisierung und der Abschluss des Programms. Zu Anfang werden die fünf Worker-Threads in einer List Comprehension erzeugt und in der `for`-Schleife gestartet. Durch den Aufruf von `thread.setDaemon(True)` werden die Threads als sogenannte *Dämon-Threads* markiert. Der wesentliche Unterschied zwischen Dämon-Threads und normalen Threads besteht darin, dass ein Programm beendet wird, wenn nur noch Dämon-Threads laufen. Bei normalen Threads kann das Programm so lange laufen, bis auch der letzte Thread beendet worden ist.

Im Beispiel benötigen wir die Dämon-Threads deshalb, weil wir am Ende des Programms nicht wie bisher auf die Terminierung jedes Threads warten, sondern die `join`-Methode der Queue aufrufen. Mit dieser Methode `join` wird der Hauptprogramm-Thread so lange unterbrochen, bis alle noch in der Warteschlange stehenden Zahlen verarbeitet worden sind. Ist die Warteschlange leer, wird das Programm inklusive aller Worker-Threads beendet. Dass die Worker-Threads dabei nicht den Programmabbruch behindern können, wird durch `setDaemon` sichergestellt.

Falls Sie sich wundern, warum wir die Zugriffe auf die Queue nicht durch Critical Sections abgesichert haben, obwohl von allen Threads auf `Mathematiker.Brief kasten` zugegriffen wird, wundern Sie sich zu Recht: Normalerweise wäre es erforderlich, jedes Mal ein Lock-Objekt zu sperren und wieder zu entsperren. Allerdings nimmt uns das `Queue`-Modul von Python diese lästige Arbeit ab, wodurch die Arbeit mit Wartschlangen wesentlich komfortabler wird.

Wir werden uns jetzt noch zwei Klassen zuwenden, die für sehr spezielle Zwecke im Zusammenhang mit Threads dienen.

top

18.4.3 Ereignisse definieren – threading.Event ▼▲

Mit der Klasse threading.Event können sogenannte *Ereignisse* (engl. *events*) definiert werden, um Threads bis zum Eintritt eines bestimmten Ereignisses zu unterbrechen.

Ein Thread, der die wait-Methode eines frisch erzeugten threading.Event-Objekts aufruft, wird so lange unterbrochen, bis ein anderer Thread das Event mit set auslöst.

Ausführliche Informationen über threading.Event finden Sie in der Python-Dokumentation.

top

18.4.4 Eine Funktion zeitlich versetzt ausführen – threading.Timer ▲

Das threading-Modul bietet eine praktische Klasse namens threading.Timer, um Funktionen nach dem Verstreichen einer gewissen Zeit aufzurufen.

threading.Timer(interval, function, args=[], kwargs={})

Der Parameter *interval* des Konstruktors gibt die Zeit in Sekunden an, die gewartet werden soll, bis die für *function* übergebene Funktion aufgerufen werden soll. Dabei können für *interval* sowohl Ganzzahlen aus auch float-Instanzen übergeben werden. Für *args* und *kwargs* kann eine Liste bzw. ein Dictionary übergeben werden, das die Parameter enthält, mit denen *function* aufgerufen werden soll.

Wir werden threading.Timer im nächsten Beispiel verwenden, um exemplarisch einen Wecker zu programmieren:

```
>>> import time, threading
>>> def wecker(gestellt):
        print "RIIIIIIIING!!!"
        print "Der Wecker wurde um %s Uhr gestellt." %
gestellt
        print "Es ist nun %s Uhr" %
time.strftime("%H:%M:%S")
>>> timer = threading.Timer(30, wecker,
                            [time.strftime("%H:%M:%S")])
>>> timer.start()
```

(30 Sekunden später)

```
>>> RIIIIIIIING!!!
Der Wecker wurde um 03:11:26 Uhr gestellt.
Es ist nun 03:11:58 Uhr
```

Mit der Methode start beginnt der Timer zu laufen und ruft dann – wie man der vorhergehenden Ausgabe entnehmen kann – nach der festgelegten Zeitspanne die übergebene Funktion auf. Die Differenz von 2 Sekunden rührt daher, dass zwischen dem Erstellen des Timer-Objekts und dem Aufrufen der start-Methode 2 Sekunden vergangen sind.

Nachdem die start-Methode aufgerufen wurde, kann der Timer außerdem mit der parameterlosen cancel-Methode wieder abgebrochen werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr

Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

19.2 XML

Das Modul `xml` der Standardbibliothek erlaubt es, XML-Dateien einzulesen. XML (kurz für »Extensible Markup Language«) ist eine standardisierte Beschreibungssprache, die es ermöglicht, komplexe, hierarchisch aufgebaute Datenstrukturen in einem lesbaren Textformat abzuspeichern. XML kann daher sehr gut zum Datenaustausch bzw. zur Datenspeicherung verwendet werden.

Besonders in der Welt des Internets finden sich viele auf XML basierende Beschreibungssprachen, wie beispielsweise XHTML, RSS, MathML oder SVG.

An dieser Stelle soll eine kurze Einführung in XML gegeben werden. Dazu dient folgende, einfache XML-Datei, die eine Möglichkeit aufzeigt, wie der Inhalt eines Python-Dictionarys dauerhaft abgespeichert werden könnte:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
    <eintrag>
        <schluessel>Hallo</schluessel>
        <wert>0</wert>
    </eintrag>
    <eintrag>
        <schluessel>Welt</schluessel>
        <wert>1</wert>
    </eintrag>
</dictionary>
```

Die erste Zeile der Datei ist die sogenannte *XML-Deklaration*. Diese optionale Angabe kennzeichnet die verwendete Version von XML und vor allem, was viel wichtiger ist, das Encoding, in dem die Datei gespeichert wurde. Durch Angabe des Encodings, in diesem Fall UTF-8, können auch Umlaute und andere

Sonderzeichen korrekt verarbeitet werden.

Abgesehen von der XML-Deklaration besteht ein XML-Dokument aus sogenannten *Tags*. Tags können wie Klammern geöffnet und geschlossen werden und stellen damit eine Art Gruppe dar, die weitere Tags enthalten kann. Jedes Tag hat einen Namen, den sogenannten *Tag-Namen*. Um ein Tag zu öffnen, wird dieser Tag-Name in spitze Klammern geschrieben. Ein schließendes Tag besteht aus dem Tag-Namen, der zusammen mit einem Slash ebenfalls in spitze Klammern geschrieben wird. Das folgende Beispiel zeigt ein öffnendes Tag, direkt gefolgt von dem entsprechenden schließenden Tag.

```
<wert></wert>
```

Innerhalb eines Tags können sowohl Text als auch weitere Tags stehen. Auf diese Weise kann eine hierarchische Struktur erstellt werden, die dazu in der Lage ist, auch komplexe Datensätze abzubilden.

Zudem können bei einem Tag sogenannte *Attribute* angegeben werden. Dazu soll das vorherige Beispiel dahingehend erweitert werden, dass der Datentyp der Schlüssel und Werte des abzubildenden Dictionarys als Attribut des jeweiligen `schluessel`- bzw. `wert`-Tags gespeichert werden kann.

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
    <eintrag>
        <schluessel typ="str">Hallo</schluessel>
        <wert typ="int">0</wert>
    </eintrag>
    <eintrag>
        <schluessel typ="str">Welt</schluessel>
        <wert typ="int">1</wert>
    </eintrag>
</dictionary>
```

Ein Attribut stellt im Prinzip ein Schlüssel/Wert-Paar dar. Im Beispiel wurde jedem `schluessel`- und `wert`-Tag ein Attribut `typ` verpasst, über das der Datentyp des Schlüssels bzw. des Werts angegeben werden kann. Beachten Sie, dass der Wert eines XML-Attributs stets in Anführungszeichen zu schreiben ist.

Zum Einlesen von XML-Dateien stellt Python, wie die meisten anderen Programmiersprachen oder XML-Bibliotheken auch, zwei sogenannte *Parser* zur Verfügung. Der Begriff des Parsers ist nicht auf XML beschränkt, sondern bezeichnet ganz allgemein ein Programm, das eine Syntaxanalyse bestimmter Daten eines speziellen Formats leistet. Die beiden im Modul `xml` enthaltenen Parser heißen `dom` und `sax` und implementieren zwei unterschiedliche Herangehensweisen an das XML-Dokument. Aus diesem Grund ist es sinnvoll, beide getrennt und ausführlich zu besprechen, was in den nächsten beiden Kapiteln geschehen soll. Das Thema des dritten Unterkapitels soll eine relativ neue Herangehensweise an XML-Daten namens `ElementTree` sein.

Hinweis

Eine Besonderheit bei XML-Tags stellen sogenannte *körperlose Tags* dar. Solche Tags spielen in den Beispielen, die in diesem Buch vorgestellt werden, keine Rolle, sind jedoch in einigen Fällen durchaus sinnvoll. Ein körperloses Tag sieht folgendermaßen aus:

```
<tag attr="wert" />
```

Ein körperloses Tag ist öffnendes und schließendes Tag zugleich und darf demzufolge nur über Attribute verfügen. Ein solches Tag kann keinen Text oder weitere Tags enthalten. Von einem XML-Parser wird ein körperloses Tag behandelt, als stünde `<tag attr="wert"></tag>` in der XML-Datei.

top

19.2.1 DOM – Document Object Model ▼▲

Das *Document Object Model*, kurz DOM, ist eine Schnittstelle, die vom *World Wide Web Consortium* (W3C) standardisiert wurde und es ermöglicht, auf einzelne Elemente einer XML-Datei zuzugreifen und diese zu modifizieren. Dazu wird die Datei vollständig eingelesen und zu einer baumartigen Struktur aufbereitet. Jedes Tag wird durch eine Klasse repräsentiert, den sogenannten *Knoten* (engl. *node*). Durch Methoden und Attribute dieser Klasse können die enthaltenen Informationen ausgelesen oder verändert werden.

Das DOM ist vor allem dann interessant, wenn ein wahlfreier Zugriff auf die XML-Daten möglich sein muss. Unter einem *wahlfreien Zugriff* versteht man den punktuellen Zugriff auf verschiedene, voneinander unabhängige Teile des Datensatzes. Das Gegenteil des wahlfreien Zugriffs wäre das sequenzielle Einlesen der XML-Datei.

Da die Datei stets vollständig eingelesen wird, ist die Verwendung von DOM sehr speicherintensiv. Im Gegensatz dazu liest das Konkurrenzmodell SAX immer nur kleine Teile der XML-Daten ein und stellt sie sofort zur Weiterverarbeitung zur Verfügung. Diese Herangehensweise benötigt weniger Arbeitsspeicher und erlaubt es, Teile der gespeicherten Daten bereits zu verwenden, beispielsweise anzuzeigen, während die Datei selbst noch nicht vollständig eingelesen ist. Ein wahlfreier Zugriff auf die XML-Daten und die Manipulation selbiger ist mit SAX allerdings nicht möglich.

Jetzt möchten wir darauf zu sprechen kommen, wie die XML-Daten bei Verwendung eines DOM-Parsers aufbereitet werden. Betrachten Sie dazu noch einmal unser vorheriges Beispiel einer XML-Datei:

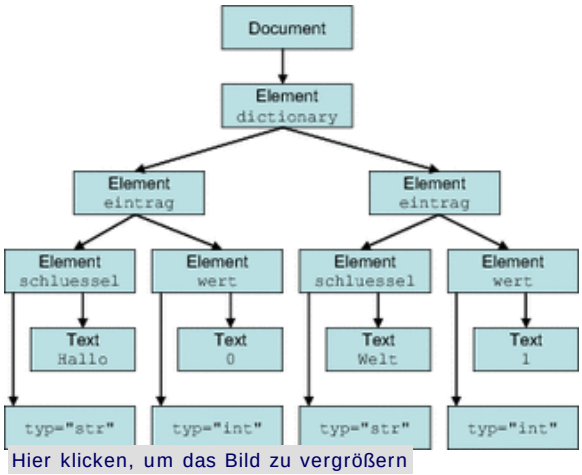
```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
    <eintrag>
        <schluessel typ="str">Hallo</schluessel>
        <wert typ="int">0</wert>
    </eintrag>
    <eintrag>
        <schluessel typ="str">Welt</schluessel>
        <wert typ="int">1</wert>
    </eintrag>
</dictionary>
```

Unter Verwendung eines DOM-Parsers werden die XML-Daten zu einem sogenannten *Baum* aufbereitet. Ein Baum besteht aus einzelnen Klassen, den sogenannten Knoten. Jede dieser Knotenklassen enthält verschiedene Referenzen auf benachbarte Knoten, nämlich:

- sein *Elternelement* (engl. *parent*). Das ist der Knoten, der im Baum direkt über diesem Knoten steht.
- seine *Kindelemente* (engl. *children*). Das sind alle Knoten, die im Baum direkt unter diesem Knoten stehen.
- seine *Geschwisterelemente* (engl. *siblings*). Das sind alle Knoten, die im Baum direkt neben diesem Knoten stehen und dasselbe Elternelement haben.

Somit enthält jeder Knoten des Baumes Referenzen zu allen umliegenden, auch verwandten Knoten. Auf diese Weise lässt sich der Baum vollständig durchlaufen und verarbeiten.

Die aus dem obigen Beispiel erzeugte Baumstruktur sieht folgendermaßen aus:



Hier klicken, um das Bild zu vergrößern

Abbildung 19.1 Vom DOM-Parser erzeugter Baum

Dabei handelt es sich bei `Document`, `Element` und `Text` um die grundlegenden Knotenklassen, aus denen ein DOM-Baum aufgebaut ist. Die `Document`-Instanz ist einmalig und entspricht der *Wurzel* des Baumes (engl. *root*). Sie enthält eine Referenz auf alle Tags erster Ordnung, wie in diesem Fall beispielsweise das Tag `dictionary`. Diesem Knoten sind mehrere Instanzen der Klasse `Element` untergeordnet, die jeweils ein `eintrag`-Tag repräsentieren. Durch Attribute dieser Klasse können Informationen wie der Tag-Name, enthaltene XML-Attribute oder Ähnliches abgerufen werden.

Beachten Sie zum einen, dass in Abbildung 19.1 aus Gründen der Übersichtlichkeit keine Geschwisterbeziehungen eingezeichnet wurden, und zum anderen, dass die Attribute der Elemente `schluessel` und `wert` keine eigenständigen Instanzen einer Knotenklasse sind, sondern Teil des Elementknotens.

Neben den Klassen `Document` und `Element` existieren Instanzen einer weiteren Klasse namens `Text`. Diese Instanzen enthalten Text, der innerhalb eines Tags geschrieben wurde.

Abgesehen von den hier aufgelisteten Klassen gibt es noch weitere Knotenklassen, die allerdings nur in Spezialfällen im Baum vorkommen. So existiert beispielsweise die Klasse `Comment` für ein Kommentar-Tag in der XML-Datei. Wir möchten uns in diesem Kapitel auf das Wesentliche, das heißt auf die Klassen `Document`, `Element` und `Text`, beschränken.

Beispiel

An dieser Stelle soll die Verwendung von DOM an einem einfachen Beispiel gezeigt werden. Dazu rufen wir uns erneut unsere Beispieldatei ins Gedächtnis, deren Zweck es war, den Inhalt eines Python-Dictionarys abzubilden:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
    <eintrag>
        <schluessel typ="str">Hallo</schluessel>
        <wert typ="int">0</wert>
    </eintrag>
</dictionary>
```

Die Datei besteht aus einem Tag erster Ordnung namens `dictionary`, in dem mehrere `eintrag`-Tags vorkommen dürfen. Jedes `eintrag`-Tag enthält zwei untergeordnete Tags namens `schluessel` und `wert`, die gemeinsam jeweils ein Schlüssel/Wert-Paar des Dictionarys repräsentieren. Der Datentyp des Schlüssels bzw. des Wertes wird über das Attribut `typ` festgelegt, das bei den Tags `schluessel` und `wert` vorkommen muss.

Das Beispielprogramm soll dazu in der Lage sein, eine solche XML-Datei einzulesen und das entsprechende Dictionary daraus zu rekonstruieren. Im Folgenden soll der Quelltext des Beispielprogramms besprochen werden.

```
import xml.dom.minidom as dom

def _knoten_auslesen(knoten):
    return eval("%s('%s')" % (knoten.getAttribute("typ"),
knoten.firstChild.data.strip()))
```

In der ersten Zeile wird der DOM-Parser eingebunden und unter dem Namensraum dom verfügbar gemacht. Für dieses Beispiel wurde der Parser xml.dom.minidom eingebunden, der eine grundlegende und simple Implementation darstellt, die in den meisten Fällen genügen sollte. Abgesehen von dem Minidom-Parser existieren noch weitere spezielle DOM-Parser im Paket xml.dom.

Danach wird die Funktion _knoten_auslesen definiert. Der Funktionsname beginnt mit einem Unterstrich, da es sich um eine Hilfsfunktion handelt, die für sich allein keinen Wert darstellt. Die Aufgabe der Funktion _knoten_auslesen ist es, aus einer Element-Instanz das Attribut typ auszulesen und den im Element enthaltenen Text in den angegebenen Datentyp zu konvertieren. Dazu wird dynamisch ein String erzeugt, der beispielsweise für den Typ int und den Text "123" zu "int('123')" wird. Dieser String wird mittels eval interpretiert und zurückgegeben. Beachten Sie, dass aus Gründen der Übersichtlichkeit alle Konsistenzprüfungen weggelassen wurden. In einem normalen Programm sollte in der Funktion _knoten_auslesen beispielsweise geprüft werden, ob ein Attribut typ überhaupt existiert oder ob der dort angegebene Datentyp gültig ist.

Das Auslesen eines XML-Attributs geschieht über die Methode getAttribute einer Element-Instanz. Um den vom Tag umschlossenen Text auszulesen, wird über das Attribut firstChild das erste Kindelement der übergebenen Element-Instanz angesprochen. Dabei handelt es sich um die jeweilige Text-Instanz. Über das Attribut data dieser Text-Instanz kann der enthaltene Text ausgelesen werden.

Beachten Sie beim Arbeiten mit Text-Instanzen, dass der DOM-Standard vorsieht, dass Whitespace-Zeichen, auch wenn sie nur aus Formatierungsgründen in der XML-Datei stehen, später im Baum wiederzufinden sind. Aus diesem Grund müssen wir eventuell vorkommende Whitespace-Zeichen durch Aufruf der String-Methode strip entfernen.

```
def lade_dict(dateiname):
    d = {}
    baum = dom.parse(dateiname)

    for eintrag in baum.firstChild.childNodes:
        if eintrag.nodeName == "eintrag":
            schluessel = wert = None

            for knoten in eintrag.childNodes:
                if knoten.nodeName == "schluessel":
                    schluessel = _knoten_auslesen(knoten)
                elif knoten.nodeName == "wert":
                    wert = _knoten_auslesen(knoten)

            d[schluessel] = wert
    return d
```

Danach wird die Hauptfunktion lade_dict definiert. Die Aufgabe dieser Funktion ist es, eine XML-Datei, deren Dateinamen sie übergeben bekommt, zu öffnen, die enthaltenen Informationen zu extrahieren, in das Dictionary d zu schreiben und das entstandene Dictionary zurückzugeben.

Zunächst wird durch Aufruf der Funktion parse des minidom-Parsers das XML-Dokument eingelesen und zu einer Baumstruktur aufbereitet. Die Referenz baum referenziert jetzt eine Instanz der Klasse Document, über die auf alle Elemente des Dokuments zugegriffen werden kann. Alternativ hätte auch die Methode parse String des minidom-Parsers aufgerufen werden können, wenn die XML-Daten in Form eines Strings vorliegen würden.

Dann soll über alle eintrag-Tags iteriert und das jeweilige Schlüssel/Wert-Paar ins Dictionary d eingefügt werden. Dazu nutzen wir die Attribute der Klasse Node, von der sowohl Document als auch Element abgeleitet sind. Von der Document-Instanz baum aus erreichen wir über das Attribut baum.firstChild das erste Kindelement, also die Element-Instanz, die das dictionary-Tag repräsentiert. Genau genommen interessieren wir uns jedoch auch nicht für das dictionary-Tag, sondern für alle diesem Tag direkt untergeordneten Elemente. Diese können wir über das Attribut childNodes erreichen, das eine Liste aller Kindelemente bereitstellt. Über diese Liste wird in einer for-Schleife iteriert.

Innerhalb der for-Schleife wird zunächst geprüft, ob es sich tatsächlich um den Knoten eines eintrag-Tags handelt. Dazu wird das Attribut nodeName verwendet, das jede Node-Instanz, also jeder Knoten besitzt. Beachten Sie wie bereits gesagt, dass der DOM-Standard vorschreibt, auch Whitespaces, die zur Formatierung der XML-Datei eingesetzt wurden, in Form von Text-Instanzen im DOM-Baum einzutragen. Diese Text-Instanzen werden hier ebenfalls herausgefiltert, ihr nodeName-Wert ist "#text". Zudem werden zwei Referenzen namens schluessel und wert angelegt, die später zum Aufbau des Dictionarys verwendet werden.

In der darauf folgenden for-Schleife wird über alle Kindelemente des eintrag-Tags iteriert. Je nachdem, ob es sich bei dem aktuellen Kindelement um ein schluessel- oder ein wert-Tag handelt, wird das Ergebnis des Funktionsaufrufs von _knoten_auslesen dem Namen schluessel bzw. wert zugewiesen. Nachdem die innere Schleife durchlaufen ist, werden Schlüssel und Wert ins Dictionary d eingetragen.

Beachten Sie unbedingt, dass wir in diesem Beispiel davon ausgehen, dass die XML-Datei exakt unseren Ansprüchen entspricht. In einem wirklichen Programm sollten Sie grundsätzlich davon ausgehen, dass auch fehlerhafte Angaben vorkommen, und diese entsprechend behandeln. Auch der sorglose Umgang mit dem Attribut typ (direktes Übergeben nach eval) sollte in einem fertigen Programm so nicht vorkommen.

Dieses Beispiel sollte einen kurzen Überblick über die Verwendung des DOM-Baumes bieten. Im Folgenden werden die Klassen Node, Document, Element und Text besprochen, aus denen ein DOM-Baum zusammengesetzt ist.

Die Klasse Node

Die Klasse Node ist die Basisklasse aller im DOM-Baum verwendeten Klassen. Das bedeutet, dass die in dieser Klasse enthaltene Funktionalität an allen Knoten des Baumes verfügbar ist. In der Klasse Node sind vor allem Attribute und Methoden enthalten, die es ermöglichen, Zugriff auf verwandte Knoten, das heißt Kinder, Geschwister oder den Elternknoten, zu erlangen.

Im Folgenden sollen die wichtigsten Attribute der Klasse Node beschrieben werden. Dabei soll *n* eine Instanz der Klasse Node sein.

n.nodeType

Kennzeichnet den Typ des Knotens. Das Attribut referenziert eine ganze Zahl, die mit folgenden symbolischen Konstanten verglichen werden kann:

Konstante	Beschreibung
Node.ELEMENT_NODE	Bei dem Knoten handelt es sich um eine Element-Instanz.
Node.TEXT_NODE	Bei dem Knoten handelt es sich um eine Text-Instanz.
	Bei dem Knoten handelt es sich um eine

`Node.DOCUMENT_NODE`	`Document`-Instanz.

Tabelle 19.2 Konstanten zur Beschreibung eines Knotentyps

n.parentNode

Referenziert das Elternelement des Knotens *n*. Wenn es sich bei dem Knoten um die `Document`-Instanz handelt, referenziert dieses Attribut `None`.

n.previousSibling

Referenziert das Geschwisterelement, das in der Reihenfolge vor dem Knoten *n* steht, oder `None`, wenn dieser Knoten das erste Kind von `parentNode` ist.

n.nextSibling

Referenziert das Geschwisterelement, das in der Reihenfolge hinter dem Knoten *n* steht, oder `None`, wenn dieser Knoten das letzte Kind von `parentNode` ist.

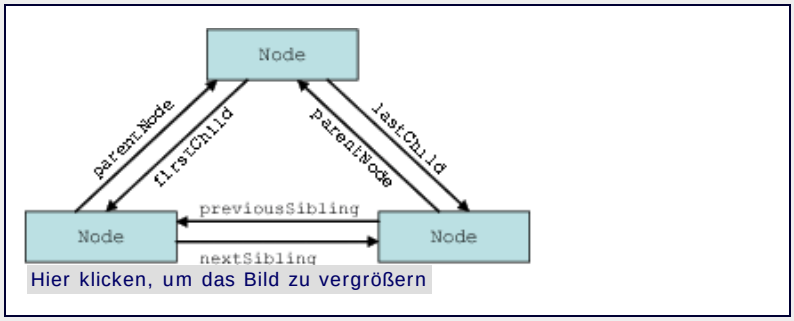
n.firstChild

Referenziert das erste Kindelement des Knotens *n* oder `None`, wenn keine untergeordneten Knoten existieren.

n.lastChild

Referenziert das letzte Kindelement des Knotens *n* oder `None`, wenn keine untergeordneten Knoten existieren.

Abbildung 19.2 soll die hier vorgestellten Attribute anhand der Beziehung von drei Knoten eines Baumes verdeutlichen:



Hier klicken, um das Bild zu vergrößern

Abbildung 19.2 Verwandtschaftsbeziehungen dreier Knoten

n.childNodes

Referenziert eine Liste aller Kinder des Knotens *n*.

Dieser Auflistung der wichtigsten Attribute der Klasse `Node` folgen die wichtigsten Methoden dieser Klasse.

n.hasChildNodes()

Gibt `True` zurück, wenn der Knoten *n* über Kinder verfügt, andernfalls `False`.

n.appendChild(newChild)

Fügt die `Node`-Instanz *newChild* als Kindelement an das Ende der Liste aller Kinder von *n* ein. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.insertBefore(newChild, refChild)

Node

Fügt die -Instanz *newChild* als Kindelement des aktuellen Knotens vor dem Kindelement *refChild* in die Liste aller Kinder von *n* ein. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.removeChild(oldChild)

Löscht das angegebene Kindelement *oldChild*. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.replaceChild(newChild, oldChild)

Ersetzt das Kindelement *oldChild* durch *newChild*. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.writexml(writer[, indent[, addindent[, newl]]])

Schreibt die Node-Instanz *n* mitsamt all ihren Kindelementen als XML in das geöffnete Dateiobjekt *writer*. Beachten Sie, dass diese Methode auch an die Klasse Document weitervererbt wird. Wenn sie für eine Document-Instanz aufgerufen wird, kann der vollständige DOM-Baum als XML-Datei gespeichert werden.

Die optionalen Parameter *indent*, *addindent* und *newl* (allesamt Strings) können verwendet werden, um die Ausgabe der XML-Daten zu formatieren. Dabei steht *indent* für die Zeichen, die zur Einrückung der gesamten Ausgabe verwendet werden, *addindent* für die Zeichen, die zur Einrückung tieferer Ebenen verwendet werden, und *newl* für das zu verwendende Newline-Zeichen.

Wenn die Methode auf einer Document-Instanz aufgerufen wird, kann ein zusätzlicher, optionaler Schlüsselwortparameter encoding angegeben werden. Das hier als String übergebene Encoding wird in die XML-Deklaration eingetragen und zum Speichern der Datei verwendet.

n.toxml([encoding])

Ähnlich wie writexml, gibt die XML-Daten jedoch als String zurück. Optional kann über den Parameter *encoding* ein Encoding angegeben werden, das in die XML-Deklaration geschrieben und im zurückgegebenen String verwendet wird.

n.toprettyxml([indent[, newl]])

Ähnlich wie toxml, gibt die XML-Daten jedoch in einem formatierten String zurück. Um die Formatierung der Daten zu verändern, können das Einrückungszeichen (*indent*, üblicherweise \t) und das zu verwendende Newline-Zeichen (*newl*, üblicherweise \n) angegeben werden.

Ein Encoding kann wie bei der Methode writexml angegeben werden.

Die Klasse Document

Ein von einem DOM-Parser erzeugter Baum enthält als Wurzelelement eine Instanz der Klasse Document. Dies ist die Instanz, die bei einem Aufruf der Funktion parse zurückgegeben wird und alle weiteren Elemente des Baumes direkt oder indirekt referenziert. Eine Document-Instanz verwaltet dabei immer ein vollständiges XML-Dokument.

Die Klasse Document erbt von der Basisklasse Node.

Nachfolgend sollen die wichtigsten Methoden und Attribute der Klasse Document erläutert werden. Dabei sei *d* eine Instanz der Klasse Document.

d.documentElement

Dieses Attribut referenziert die Element-Instanz des ersten Tags des XML-Dokuments *d*. Beachten Sie, dass ein wohlgeformtes XML-Dokument über genau ein Wurzel-Tag verfügt. Sollten mehrere sogenannte *toplevel-Tags* vorkommen, kann auf diese über ihre Geschwisterbeziehung zu `documentElement` zugegriffen werden.

d.createElement(tagName)

Erzeugt einen neuen Elementknoten mit dem Tag-Namen *tagName*. Die Funktion gibt eine Instanz der Klasse `Element` zurück. Beachten Sie, dass der Knoten zwar erzeugt, aber nicht automatisch in den Baum eingefügt wird. Dazu können beispielsweise die Methoden `insertBefore` oder `appendChild` der Klasse `Node` verwendet werden.

d.createTextNode(data)

Erzeugt einen neuen Textknoten mit dem Inhalt *data*. Die Funktion gibt eine Instanz der Klasse `Text` zurück. Für diese Methode gilt ebenfalls der Hinweis, dass der erzeugte Knoten nicht automatisch in den DOM-Baum eingefügt wird.

d.getElementsByTagName(tagName)

Gibt eine Liste zurück, in der alle `Element`-Instanzen enthalten sind, die Tags mit dem Tag-Namen *tagName* repräsentieren.

Die Klasse Element

Die Klasse `Element` repräsentiert ein Tag im DOM-Baum. Sie erbt von der Basisklasse `Node`.

Im Folgenden sollen die wichtigsten Attribute und Methoden der Klasse `Element` erläutert werden. Dabei sei *e* eine Instanz der Klasse `Element`.

e.tagName

Dieses Attribut referenziert den Tag-Namen des von *e* repräsentierten Tags.

e.getElementsByTagName(tagName)

Äquivalent zu `Document.getElementsByTagName`. Beachten Sie, dass diese Methode nur nach direkt oder indirekt untergeordneten Elementen mit dem Tag-Namen *tagName* sucht.

e.hasAttribute(name)

Gibt `True` zurück, wenn das Element *e* ein Attribut mit dem Schlüssel *name* besitzt, andernfalls `False`.

e.getAttribute(name)

Gibt den Wert des Attributs mit dem Schlüssel *name* zurück. Sollte kein Attribut *name* existieren, wird ein leerer String zurückgegeben.

e.removeAttribute(name)

Löscht das Attribut mit dem Schlüssel *name*. Beachten Sie, dass keine Exception geworfen wird, wenn kein Attribut mit dem Schlüssel *name* existiert.

e.setAttribute(name, value)

Erzeugt ein neues Attribut mit dem Schlüssel *name* und dem Wert *value* oder überschreibt ein bereits bestehendes Attribut.

Die Klasse Text

Die Klasse `Text` erbt von `Node` und fügt ein einziges Attribut hinzu:

t.data

Das Attribut `data` referenziert den String, den die `Text`-Instanz *t* repräsentiert.

Schreiben einer XML-Datei

Im vorangegangenen Beispiel wurde gezeigt, wie die in einer XML-Datei enthaltenen Daten zu einem Baum aufbereitet werden können. Zudem haben Sie soeben einige Methoden der Knotenklassen des Baums kennengelernt, die dazu in der Lage sind, den Baum zu modifizieren. Der nächste logische Schritt ist es, den modifizierten Baum wieder als XML-Datei abzuspeichern.

In diesem Abschnitt wird ein Beispielprogramm besprochen, das den umgekehrten Weg des ersten Beispiels geht. Das heißt, es erzeugt aus einem Dictionary einen DOM-Baum und speichert diesen als XML-Datei ab. Diese XML-Datei soll so aufgebaut sein, dass sie von dem vorherigen Beispielprogramm wieder ausgelesen werden kann.

Das Schreiben der XML-Datei soll durch eine Funktion `schreibe_dict` erfolgen, die das zu schreibende Dictionary `d` und den Dateinamen der Ausgabedatei als Parameter übergeben bekommt. Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
import xml.dom.minidom as dom

def _erstelle_eintrag(schluessel, wert):
    tag_eintrag = dom.Element("eintrag")
    tag_schluessel = dom.Element("schluessel")
    tag_wert = dom.Element("wert")

    tag_schluessel.setAttribute("typ",
type(schluessel).__name__)
    tag_wert.setAttribute("typ", type(wert).__name__)

    text = dom.Text()
    text.data = str(schluessel)
    tag_schluessel.appendChild(text)

    text = dom.Text()
    text.data = str(wert)
    tag_wert.appendChild(text)

    tag_eintrag.appendChild(tag_schluessel)
    tag_eintrag.appendChild(tag_wert)
    return tag_eintrag
```

Auch hier wird, ähnlich wie beim vorherigen Beispiel, zuerst eine Hilfsfunktion angelegt, die einen Schlüssel und einen Wert übergeben bekommt und daraus eine `Element`-Instanz erzeugt, die das entsprechende `eintrag`-Tag repräsentiert. Die Funktion an sich bedarf eigentlich keiner weiteren Erläuterung. Einzig erwähnenswert ist folgender Ausdruck:

```
type(schluessel).__name__
```

Dieser Ausdruck ermittelt den Namen des Datentyps der von `schluessel` referenzierten Instanz. Das wäre beispielsweise `"int"` für ganze Zahlen oder `"str"` für Strings. Jetzt folgt die Hauptfunktion des Beispielprogramms:

```
def schreibe_dict(d, dateiname):
    baum = dom.Document()
    tag_dict = dom.Element("dictionary")
    baum.appendChild(tag_dict)

    for schluessel, wert in d.iteritems():
        tag_eintrag = _erstelle_eintrag(schluessel, wert)
        tag_dict.appendChild(tag_eintrag)

    f = open(dateiname, "w")
    baum.writexml(f, "", "\t", "\n")
    f.close()
```

Im Funktionskörper wird zunächst eine neue Instanz der Klasse `Document` angelegt. Diese Instanz soll die Wurzel des DOM-Baums werden, den wir im Laufe der Funktion erzeugen, und wird von `baum` referenziert. Danach wird das oberste Element, das `dictionary`-Tag, erzeugt und an die `Document`-Instanz als Kindelement angefügt. Das `dictionary`-Tag wird von `tag_dict` referenziert.

Danach werden in einer `for`-Schleife alle Schlüssel/Wert-Paare des Dictionarys `d` durchlaufen. In jedem Schleifendurchlauf wird eine neue `Element`-Instanz für das jeweilige `eintrag`-Tag mithilfe der Funktion `_erstelle_eintrag` erzeugt. Danach wird die erzeugte `Element`-Instanz als Kindelement des `dictionary`-Tags in den DOM-Baum eingefügt.

Am Ende der Funktion wird die Datei `dateiname` zum Schreiben geöffnet und die XML-Daten mittels der Methode `writexml` hineingeschrieben.

Die hier vorgestellte Funktion `schreibe_dict` arbeitet perfekt mit der Funktion `lade_dict` des vorherigen Beispiels zusammen. Das bedeutet, dass eine von `schreibe_dict` erzeugte XML-Datei problemlos von `lade_dict` wieder eingelesen werden kann.

Damit wäre das Konzept des Document Object Model umrissen und anhand zweier grundlegender Beispiele erklärt. Beachten Sie, dass hier nicht alle Möglichkeiten von DOM angesprochen wurden. Fühlen Sie sich also dazu ermutigt, weiter zu recherchieren und auszuprobieren, wenn Sie weitere Details zu speziellen Features des DOM erfahren möchten.

top

19.2.2 SAX – Simple API for XML ▼▲

Nachdem wir uns im letzten Abschnitt ausführlich der DOM-Herangehensweise an XML-Dateien gewidmet haben, möchten wir nun einen zweiten Weg vorstellen, diese Dateien zu verarbeiten. Die *Simple API for XML*, kurz *SAX,* baut im Gegensatz zu DOM kein vollständiges Abbild der XML-Datei im Speicher auf, sondern liest die Datei fortlaufend ein und setzt den Programmierer durch Aufrufen bestimmter Funktionen davon in Kenntnis, dass beispielsweise ein öffnendes oder schließendes Tag gelesen wurde. Diese Herangehensweise hat neben der Speichereffizienz noch einen weiteren Vorteil: Beim Laden von sehr großen XML-Dateien können bereits eingelesene Teile weiterverarbeitet werden, obwohl die Datei noch nicht vollständig eingelesen worden ist.

Allerdings sind mit der Verwendung von SAX auch einige Nachteile verbunden. So ist beispielsweise kein wahlfreier Zugriff auf einzelne Elemente der XML-Daten möglich, wie es beispielsweise bei DOM praktiziert werden kann. Außerdem sieht SAX keine Möglichkeit vor, die XML-Daten komfortabel zu verändern oder wieder zu speichern. Doch nun zur Funktionsweise von SAX.

Das Einlesen einer XML-Datei durch einen SAX-Parser, in der SAX-Terminologie auch *Reader* genannt, geschieht event-gesteuert. Das bedeutet, dass der Programmierer beim Erstellen des Readers verschiedene sogenannte *Callback-Funktionen* einrichten und mit einem bestimmten Event verknüpfen muss. Wenn beim Einlesen der XML-Datei durch den Reader dann das besagte Event auftritt, wird die damit verknüpfte Callback-Funktion aufgerufen und somit der Code ausgeführt, den der Programmierer für diesen Zweck vorgesehen hat. Ein Event könnte beispielsweise das Auffinden eines öffnenden Tags sein.

Man könnte also sagen, dass der SAX-Reader nur die Infrastruktur zum Einlesen der XML-Datei bereitstellt. Ob und in welcher Form die gelesenen Daten aufbereitet werden, entscheidet allein der Programmierer. Damit bietet SAX wesentlich mehr Flexibilität als DOM, auf Kosten eines mitunter höheren Aufwandes selbstverständlich.

Beispiel

Die Verwendung von SAX soll direkt an einem einfachen Beispiel gezeigt werden. Dazu dient uns das bereits bekannte Szenario: Ein Python-Dictionary wurde in einer XML-Datei abgespeichert und soll durch das Programm eingelesen und wieder in ein Dictionary verwandelt werden. Die Daten liegen im folgenden Format vor:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
    <eintrag>
        <schluessel typ="str">Hallo</schluessel>
        <wert typ="int">0</wert>
    </eintrag>
</dictionary>
```

Zum Einlesen dieser Datei dient folgendes Programm, das einen SAX-Reader verwendet:

```
import xml.sax as sax

class DictHandler(sax.handler.ContentHandler):

    def __init__(self):
        self.ergebnis = {}
        self.schluessel =""
        self.wert = ""
        self.aktiv = None
        self.typ = None

    def startElement(self, name, attrs):
        if name == "eintrag":
            self.schluessel = ""
            self.wert = ""
        elif name == "schluessel" or name == "wert":
            self.aktiv = name
            self.typ = eval(attrs["typ"])

    def endElement(self, name):
        if name == "eintrag":
            self.ergebnis[self.schluessel] =
self.typ(self.wert)
        elif name == "schluessel" or name == "wert":
            self.aktiv = None

    def characters(self, content):
        if self.aktiv == "schluessel":
            self.schluessel += content
        elif self.aktiv == "wert":
            self.wert += content
```

Zunächst wird die Klasse `DictHandler` angelegt, in der alle interessanten Callback-Funktionen, auch *Callback-Handler* genannt, in Form von Methoden implementiert werden. Die Klasse muss von der Basisklasse `sax.handler.Content Handler` abgeleitet werden.

Ein Nachteil des SAX-Modells ist es, dass man nach jedem Schritt den aktuellen Status speichern muss, damit beim nächsten Aufruf einer der Callback-Funktionen klar ist, ob der eingelesene Text beispielsweise innerhalb eines `schluessel`- oder eines `wert`-Tags gelesen wurde. Aus diesem Grund legen wir im Konstruktor der Klasse einige Attribute an:

- `self.ergebnis` für das resultierende Dictionary,
- `self.schluessel` für den Inhalt des aktuell bearbeiteten Schlüssels,
- `self.wert` für den Inhalt des aktuell bearbeiteten Wertes,
- `self.aktiv` für den Tag-Namen des Tags, das zuletzt eingelesen wurde, und
- `self.typ` für den Datentyp, der im Attribut `typ` eines `schluessel`- oder `wert`-Tags steht.

Zuerst implementieren wir die Methode `startElement`, die immer dann aufgerufen wird, wenn ein öffnendes Tag eingelesen wurde. Die Methode bekommt den Tag-Namen und die enthaltenen Attribute als Parameter übergeben. In dieser Methode wird zunächst ausgelesen, um welches öffnende Tag es sich handelt.

Im Falle eines schluessel- oder wert-Tags wird self.name entsprechend angepasst und das Attribut typ des Tags ausgelesen.

Die Methode endElement wird aufgerufen, wenn ein schließendes Tag eingelesen wurde. Auch sie bekommt den Tag-Namen als Parameter übergeben. Im Falle eines schließenden eintrag-Tags fügen wir das eingelesene Schlüssel/Wert-Paar, das aus self.schluessel und self.wert besteht, in das Dictionary self.ergeb nis ein. Wenn ein schließendes schluessel- oder wert-Tag gefunden wurde, wird das Attribut self.aktiv wieder auf None gesetzt, sodass keine weiteren Zeichen mehr verarbeitet werden.

Die letzte Methode characters wird aufgerufen, wenn Zeichen eingelesen wurden, die nicht zu einem Tag gehören. Beachten Sie, dass der SAX-Reader nicht garantiert, dass eine zusammenhängende Zeichenfolge auch in einem einzelnen Aufruf von characters resultiert. Je nachdem, welchen Namen das zuletzt eingelesene Tag hatte, werden die gelesenen Zeichen an self.schluessel oder self.wert angehängt.

Schlussendlich fehlt noch die Hauptfunktion lade_dict des Beispielprogramms, in der der SAX-Parser erzeugt und gestartet wird.

```
def lade_dict(dateiname):
    handler = DictHandler()
    parser = sax.make_parser()
    parser.setContentHandler(handler)
    parser.parse(dateiname)
    return handler.ergebnis
```

Im Funktionskörper wird die Klasse DictHandler instanziiert, und durch die Funktion make_parser des Moduls xml.sax wird ein SAX-Parser erzeugt. Dann wird die Methode setContentHandler des Parsers aufgerufen, um die DictHand ler-Instanz mit den enthaltenen Callback-Handlern anzumelden. Zum Schluss wird der Parsing-Prozess durch die Methode parse eingeleitet.

Die Klasse ContentHandler

Die Klasse ContentHandler dient als Basisklasse und implementiert alle SAX-Callback-Handler als Methoden. Um einen SAX-Parser einsetzen zu können, muss eine eigene Klasse erstellt werden, die von ContentHandler erbt und die benötigten Callback-Handler überschreibt. Eine Instanz einer von ContentHandler abgeleiteten Klasse wird von der Methode setContentHandler des SAX-Parsers erwartet.

Es folgt eine Auflistung der wichtigsten Callback-Handler, die in einer von ContentHandler abgeleiteten Klasse überschrieben werden können. Dabei sei *c* eine Instanz der Klasse ContentHandler.

c.startDocument()

Wird einmalig aufgerufen, wenn der SAX-Parser damit beginnt, ein XML-Dokument einzulesen.

c.endDocument()

Wird einmalig aufgerufen, wenn der SAX-Parser ein XML-Dokument vollständig eingelesen hat.

c.startElement(name, attrs)

Wird aufgerufen, wenn ein öffnendes Tag eingelesen wurde. Die Methode bekommt weitere Informationen über das Tag in Form von zwei Parametern übergeben: den Tag-Namen (*name*) und die im Tag angegebenen Attribute (*attrs*) als Attributes-Instanz. Auf eine solche Instanz kann wie auf ein Dictionary zugegriffen werden, um einzelne Attribute abzufragen.

c.endElement(name)

Wird aufgerufen, wenn ein schließendes Tag mit dem Tag-Namen *name* eingelesen wurde.

c.characters(content)

Wird aufgerufen, wenn ein Textabschnitt eingelesen wurde. Beachten Sie, dass es dem SAX-Parser freisteht, den gesamten Textabschnitt in einem Event zu verarbeiten oder auf mehrere Events aufzuteilen.

Über den Parameter *content* können Sie auf den gelesenen Text zugreifen.

c.ignorableWhitespace(whitespace)

Wird aufgerufen, wenn Whitespace-Zeichen eingelesen wurden. Diese könnten von Bedeutung sein, sind jedoch in den meisten Fällen allein aus Gründen der Formatierung vorhanden und können ignoriert werden. Beachten Sie, dass der SAX-Parser auch hier eine Folge von mehreren Whitespace-Zeichen auf mehrere Events aufteilen kann.

Über den Parameter *whitespace* können Sie auf die gelesenen Zeichen zugreifen.

So viel zur DOM- und SAX-Implementierung in Python. Diese Kapitel sollten nicht als DOM- bzw. SAX-Referenz verstanden werden, sondern als projektorientierte Einführung in die Thematik. Bedenken Sie, dass XML, aber auch DOM und SAX standardisiert sind bzw. De-facto-Standards darstellen. Es existieren DOM- und SAX-Implementierungen für so gut wie jede nennenswerte Programmiersprache, und dementsprechend einfach sollte es sein, weitere Informationen zu diesen Themen zu finden.

Im nun folgenden Abschnitt möchten wir uns einer dritten Herangehensweise an XML namens `ElementTree` widmen.

19.2.3 ElementTree

Seit Python 2.5 ist im Modul `xml.etree.ElementTree` der Standardbibliothek der Datentyp `ElementTree` enthalten, der in einer gewissen Konkurrenz zu DOM steht. Der Datentyp `ElementTree` speichert ein XML-Dokument und stellt außerordentlich komfortable Möglichkeiten zur Verfügung, sich in diesem Dokument zu bewegen und Daten auszulesen. Im Gegensatz zu DOM ist `ElementTree` nicht für mehrere Sprachen verfügbar oder gar standardisiert, weswegen es spezielle Sprachfeatures von Python, beispielsweise Iteratoren, nutzen kann und sich somit perfekt in die Sprache Python integriert.

Auch eine `ElementTree`-Instanz kann, ähnlich wie bei DOM, als Baum betrachtet werden. Dieser Baum besteht aus Instanzen der Klasse `Element`, die jeweils ein Tag repräsentieren. Attribute und Textinhalt der Tags werden ebenfalls in der jeweiligen `Element`-Instanz gespeichert.

Im Folgenden sollen zunächst die im Modul `xml.etree.ElementTree` enthaltenen Funktionen und danach die Klassen `ElementTree` und `Element` erläutert werden.

Der Inhalt des Moduls ElementTree

In diesem Abschnitt sollen die wichtigsten Funktionen besprochen werden, die im Modul `xml.etree.ElementTree` enthalten sind. Mit diesen Funktionen ist es beispielsweise möglich, eine XML-Datei einzulesen und zu einer `ElementTree`-Instanz aufzubereiten.

ElementTree.parse(source[, parser])

Liest die XML-Datei *source* ein und gibt die aufbereiteten XML-Daten in Form einer ElementTree-Instanz zurück. Für den Parameter *source* kann sowohl ein Dateiname als auch ein geöffnetes Dateiobjekt übergeben werden. Durch Angabe des optionalen Parameters *parser* kann ein eigener XML-Parser verwendet werden. Ein solcher Parser muss von der Klasse TreeBuilder abgeleitet werden, was wir an dieser Stelle nicht näher erläutern möchten. Der Standardparser ist die Klasse XMLTreeBuilder.

ElementTree.tostring(element[, encoding])

Schreibt die Element-Instanz *element* mit all ihren Unterelementen als XML in einen String und gibt diesen zurück. Durch den optionalen Parameter *encoding* kann das Encoding des resultierenden Strings festgelegt werden.

Die Klasse ElementTree

Die Klasse ElementTree repräsentiert ein XML-Dokument und enthält damit den vollständigen Baum, der daraus aufgebaut worden ist. Eine Instanz der Klasse ElementTree stellt die folgenden Methoden bereit. Im Folgenden sei *et* eine Instanz der Klasse ElementTree.

et.getiterator([tag])

Die Methode getiterator gibt einen Iterator zurück, der alle Elemente des Baums inklusive des Wurzelelements durchläuft. Die Elemente werden dabei in der Reihenfolge durchlaufen, in der ihre öffnenden Tags in der XML-Datei vorkommen. Wenn der optionale Parameter *tag* angegeben wurde, durchläuft der zurückgegebene Iterator alle Elemente des Baums mit dem Tag-Namen *tag*.

et.getroot()

Gibt die Element-Instanz des Wurzelelements zurück.

et.write(file[, encoding])

Speichert den vollständigen Baum als XML-Datei *file* ab. Dabei kann für *file* sowohl ein Dateiname als auch ein zum Schreiben geöffnetes Dateiobjekt übergeben werden. Über den optionalen Parameter *encoding* kann das Encoding der geschriebenen Daten festgelegt werden.

Die Klasse Element

Die Klasse Element repräsentiert ein Tag des XML-Dokuments im ElementTree-Baum. Dafür kann eine Element-Instanz über beliebig viele Kindelemente verfügen.

Die Klasse Element erbt alle Eigenschaften einer Liste. Es ist also möglich, auf Kindelemente wie bei einer Liste mit ihrem Index zuzugreifen. Außerdem können insbesondere die Methoden append, insert, items, keys und remove einer Liste verwendet werden. Im Folgenden sei *e* eine Instanz der Klasse Element.

e.find(path)

Gibt das erste direkte Kindelement von *e* mit dem Tag-Namen *path* zurück. Statt eines einzelnen Tag-Namens kann für *path*, wie der Name bereits andeutet, auch ein Pfad übergeben werden. So würde ein Aufruf von find mit einem Parameter *path* von "element1/element2" das erste Kindelement namens element2 des ersten direkten Kindelements mit dem Tag-Namen element1 zurückgeben. Auch das Wildcard-Zeichen * kann verwendet werden, um einen beliebigen Tag-Namen zu kennzeichnen.

e.findall(path)

Wie `find`, gibt aber eine Liste aller passenden `Element`-Instanzen zurück statt nur des zuerst gefundenen Elements.

e.findtext(path[, default])

Wenn für *path* ein leerer String übergeben wird, gibt die Methode `findtext` den Text als String zurück, den *e* enthält. Ansonsten kann der Parameter *path* wie bei den Methoden `find` und `findall` verwendet werden. Wenn eine Element-Instanz keinen Text enthält, wird `None` zurückgegeben. Sollte dies nicht Ihren Wünschen entsprechen, können Sie über den Parameter *default* festlegen, was in diesen Fällen stattdessen zurückgegeben werden soll.

Beachten Sie, dass auch Whitespace-Zeichen wie beispielsweise ein Zeilenumbruch zum Text einer `Element`-Instanz zählen.

e.get(key[, default])

Mithilfe der Methode `get` kann auf den Wert des Attributs *key* der Element-Instanz *e* zugegriffen werden. Wenn kein Attribut mit dem Schlüssel *key* vorhanden ist, wird *default* zurückgegeben. Der Parameter *default* ist mit `None` vorbelegt.

e.set(key, value)

Durch Aufruf der Methode `set` wird ein neues Attribut mit dem Schlüssel *key* und dem Wert *value* im Element *e* angelegt.

e.getiterator([tag])

Die Methode `getiterator` hat die gleiche Bedeutung wie die gleichnamige Methode der Klasse `ElementTree`, allerdings nur für alle Kindelemente von *e*.

e.getchildren()

Gibt eine Liste aller Kindelemente zurück.

Neben den soeben besprochenen Methoden verfügen alle `Element`-Instanzen über die folgenden Attribute:

e.attrib

Das Attribut `attrib` referenziert ein Dictionary, das alle in der `Element`-Instanz *e* vorhandenen XML-Attribute als Schlüssel/Wert-Paare enthält.

e.tag

Das Attribut `tag` enthält den Tag-Namen des Elements *e*.

e.text

Das Attribut `text` enthält den Text, der in der XML-Datei zwischen dem öffnenden Tag der `Element`-Instanz *e* und dem öffnenden Tag des ersten Kindelements steht. Wenn kein Kindelement existiert, enthält das Attribut `text` den vollständigen im Körper des Tags enthaltenen Text.

e.tail

Das Attribut `tail` enthält den Text, der in der XML-Datei zwischen dem schließenden Tag der `Element`-Instanz *e* und dem nächsten öffnenden oder schließenden Tag steht.

Beispiel

Als Beispiel für die Verwendung des Datentyps ElementTree soll das Beispielprogramm der vorherigen Abschnitte an diesen Datentyp angepasst werden und somit seine Stärken demonstrieren. Das Beispielprogramm soll eine XML-Datei des folgenden Formats einlesen und zu einem Dictionary aufbereiten:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
    <eintrag>
        <schluessel typ="str">Hallo</schluessel>
        <wert typ="int">0</wert>
    </eintrag>
</dictionary>
```

Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
import xml.etree.ElementTree as ElementTree

def _lese_text(element):
    typ = element.get("typ", "str")
    return eval("%s('%s')" % (typ, element.text))

def lade_dict(dateiname):
    d = {}
    baum = ElementTree.parse(dateiname)
    tag_dict = baum.getroot()
    for eintrag in tag_dict.getchildren():
        tag_schluessel = eintrag.find("schluessel")
        tag_wert = eintrag.find("wert")
        d[_lese_text(tag_schluessel)] =
_lese_text(tag_wert)
    return d
```

Zunächst wird die Funktion _lese_text implementiert, die aus der Element-Instanz eines schluessel- oder wert-Tags das Attribut typ ausliest und den vom jeweiligen Tag umschlossenen Text in den durch typ angegebenen Datentyp konvertiert. Dazu wird die Built-in Function eval wie bei den Beispielen der vorherigen Kapitel verwendet. Der Inhalt des Tags wird dann als Instanz des passenden Datentyps zurückgegeben.

Die Hauptfunktion des Beispielprogramms lade_dict bekommt den Dateinamen einer XML-Datei übergeben und soll die darin enthaltenen Daten zu einem Python-Dictionary aufbereiten. Dazu wird die XML-Datei zunächst mithilfe der Funktion parse des Moduls xml.etree.ElementTree zu einem Baum aufbereitet. Danach wird der Referenz tag_dict das Wurzelelement des Baums zugewiesen, um auf diesem weiter zu operieren.

In der nun folgenden Schleife wird über alle Kindelemente des Wurzelelements, also über alle eintrag-Tags, iteriert. In jedem Iterationsschritt werden die ersten Kindelemente mit den Tag-Namen schluessel und wert gesucht und den Referenzen tag_schluessel und tag_wert zugewiesen. Am Ende des Schleifenkörpers werden die Element-Instanzen der jeweiligen schluessel- oder wert-Tags durch die Funktion _lese_text geschleust und der im Tagkörper enthaltene Text damit in eine Instanz des korrekten Datentyps konvertiert. Die resultierenden Instanzen werden als Schlüssel bzw. als Wert in das Dictionary d eingetragen. Schlussendlich wird das erzeugte Dictionary d zurückgegeben.

So viel zum Datentyp ElementTree. Wir behalten die grobe Richtung Datenspeicherung bei und werden uns im nächsten Abschnitt um das Thema Datenbanken kümmern.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr

Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

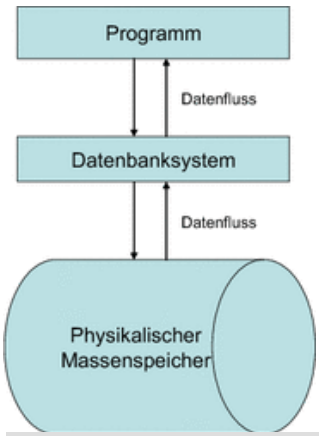
Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

19.3 Datenbanken

Je mehr Daten ein Programm verwalten muss und je komplexer die Struktur dieser Daten wird, desto größer wird der programmtechnische Aufwand für die dauerhafte Speicherung und Verwaltung der Daten. Außerdem gibt es eine ganze Reihe von Aufgaben wie das Lesen, Schreiben oder Aktualisieren, die in fast jedem Programm gebraucht werden, aber immer wieder neu implementiert werden müssten.

Abhilfe für diese Problematik wurde dadurch geschaffen, dass man eine Abstraktionsschicht zwischen dem benutzenden Programm und dem physikalischen Massenspeicher einzog, die sogenannte *Datenbank*. Dabei erfolgt die Kommunikation zwischen Benutzerprogramm und Datenbank über eine vereinheitlichte Schnittstelle.



Hier klicken, um das Bild zu vergrößern

Abbildung 19.3 Die Datenbankschnittstelle

Das Datenbanksystem nimmt dabei Abfragen, sogenannte *Queries*, entgegen und gibt alle Datensätze zurück, die den Bedingungen der

Abfragen genügen.

Wir werden uns in diesem Kapitel ausschließlich mit sogenannten *relationalen Datenbanken* beschäftigen, die einen Datenbestand in Tabellen organisieren. [Der Attribut »relational« geht auf den Begriff der ***Relation*** aus der Mathematik zurück. Vereinfacht gesagt ist eine Relation eine Zuordnung von Elementen zweier oder mehrerer Mengen.] Für die Abfragen in relationalen Datenbanken wurde eine eigene Sprache entwickelt, deren Name *SQL* (*Structured Query Language*, dt. *strukturierte Abfragesprache*) ist. SQL ist zu komplex, um es in diesem Kapitel erschöpfend zu beschreiben. Wir werden hier nur auf grundlegende Befehle von SQL eingehen, die nötig sind, um das Prinzip von Datenbanken und deren Anwendung in Python zu verdeutlichen.

SQL ist standardisiert und wird von eigentlich allen Datenbanksystemen unterstützt. Dabei ist zu beachten, dass die Systeme immer nur Teilmengen der Sprache implementieren und teilweise geringfügig abändern. Aus diesem Grund werden wir Sie hier in das SQL einführen, das von SQLite, der Standarddatenbank in Python, genutzt wird. Im Abschnitt über die Nutzung des Datenbankservers MySQL gehen wir dann auf die kleinen Unterschiede ein.

Neben der Abfragesprache SQL ist in Python auch die Schnittstelle der Datenbankmodule standardisiert. Dies hat für den Programmierer den angenehmen Nebeneffekt, dass sein Code mit minimalen Anpassungen auf allen Datenbanksystemen lauffähig ist, die diesen Standard implementieren. Die genaue Definition dieser sogenannten *Python Database API Specification* können Sie im Internet unter der Adresse http://www.python.org/dev/peps/pep-0249/ nachlesen.

Bevor wir uns aber eingehend mit der Abfragesprache SQL selbst beschäftigen, werden wir eine kleine Beispieldatenbank erarbeiten und uns überlegen, welche Operationen man überhaupt ausführen könnte. Anschließend werden wir dieses Beispiel mithilfe von SQLite implementieren und dabei auf Teile der Abfragesprache SQL und die Verwendung in Python-Programmen eingehen.

Stellen wir uns vor, wir müssten das Lager eines Computerversands verwalten. Wir wären dafür verantwortlich, dass die gelieferten Teile an der richtigen Stelle im Lager aufbewahrt werden, wobei für jede Komponente der Lieferant, der Lieferzeitpunkt und die Nummer des Fachs im Lager gespeichert werden soll. Für Kunden, die bei dem Versand ihre Rechner bestellen, werden die entsprechenden Teile reserviert, und diese sind dann für andere Kunden nicht mehr verfügbar. Außerdem sollen wir Listen mit allen Kunden und Lieferanten der Firma bereitstellen.

Um ein Datenbankmodell für dieses Szenario zu erstellen, legen wir zuerst eine Tabelle namens »Lager« an, die alle im Lager befindlichen Komponenten enthält. Wir gehen der Einfachheit halber davon aus, dass unser Lager in 100 Fächer eingeteilt ist, die fortlaufend nummeriert sind. Dabei kann jedes Fach nur ein einzelnes Computerteil aufnehmen. Eine entsprechende Tabelle mit ein paar Beispieldatensätzen für das Lager könnte dann wie folgt aussehen, wenn wir zusätzlich den Lieferanten und den Reservierungsstatus speichern wollen:

Fachnummer	Seriennummer	Komponente	Lieferant	Reserviert
1	26071987	Grafikkarte Typ 1	FC	0
2	19870109	Prozessor Typ 13	LPE	57
10	06198823	Netzteil Typ 3	FC	0
25	11198703	LED-Lüfter	FC	57
26	19880105	Festplatte 128 GB	LPE	12

Tabelle 19.3 Tabelle »Lager« für den Lagerbestand

Die Spalte »Lieferant« enthält dabei das Kürzel der liefernden Firma, und das Feld »Reserviert« ist auf »0« gesetzt, wenn der betreffende Artikel noch nicht von einem Kunden reserviert wurde. Ansonsten enthält das Feld die Kundenummer des reservierenden Kunden. In der Tabelle werden nur die belegten Fächer gespeichert, weshalb alle Fächer, für die kein Eintrag existiert, mit neuen Teilen gefüllt werden können.

Die ausführlichen Informationen zu Lieferanten und Kunden werden in zwei weiteren Tabellen namens »Lieferanten« und »Kunden« abgelegt:

Kurzname	Name	Telefonnummer
FC	FiboComputing Inc.	011235813
LPE	LettgenPetersErnesti	026741337
GC	Golden Computers	016180339

Tabelle 19.4 Tabelle »Lieferanten«

Kundennummer	Name	Anschrift
12	Heinz Elhurg	Turnhallenstr. 1, 3763 Sporthausen
57	Markus Altbert	Kämperweg 24, 2463 Duisschloss
64	Steve Apple	Podmacstr 2, 7467 Iwarhausen

Tabelle 19.5 Tabelle »Kunden«

Damit wir als Lagerverwalter von dieser Datenbank profitieren können, müssen wir die Möglichkeit haben, den Datenbestand zu manipulieren. Wir brauchen Routinen, um neue Kunden und Lieferanten hinzuzufügen, ihre Daten beispielsweise bei einem Umzug zu aktualisieren oder sie auf Wunsch aus unserer Datenbank zu entfernen. Auch in die Tabelle »Lager« müssen neue Einträge eingefügt und alte gelöscht oder angepasst werden können. Um die Datenbank aktuell zu halten, benötigen wir also Funktionen zum *Hinzufügen* und *Löschen*.

Wirklich nützlich wird die Datenbank aber erst, wenn wir die enthaltenen Daten nach bestimmten Kriterien abfragen können. Im einfachsten Fall könnten wir beispielsweise einfach nur eine Liste aller Kunden oder Lieferanten anfordern oder uns informieren wollen, welche Fächer zurzeit belegt sind. Uns könnte aber auch interessieren, ob und wenn ja, welche Artikel der Kunde mit dem Namen »Markus Altbert« reserviert hat und wo diese gelagert werden oder welche Komponenten wir von dem Lieferanten mit der Telefonnummer »011235813« nachbestellen müssen, weil sie nicht mehr vorhanden oder bereits reserviert sind. Bei diesen Operationen werden immer Datensätze nach bestimmten Kriterien *ausgewählt* und an das aufrufende Benutzerprogramm zurückgegeben.

Nach dieser theoretischen Vorbereitung werden wir uns der Implementation des Beispiels in einer SQLite-Datenbank zuwenden.

top

19.3.1 Pythons eingebaute Datenbank – sqlite3

SQLite ist ein sehr einfaches Datenbanksystem, das seine Daten in normalen Dateien abspeichert. Trotzdem ist es extrem schnell und auch für verhältnismäßig große Datenmengen geeignet.

In Python muss man das Modul `sqlite3` importieren, um mit der Datenbank zu arbeiten. Anschließend muss man eine Verbindung zu der Datenbank aufbauen, indem man die `connect`-Funktion, die ein `Connection`-Objekt zu der Datenbank zurückgibt, aufruft und ihr den Dateinamen für die Datenbank übergibt:

```
import sqlite3
connection = sqlite3.connect("lagerverwaltung.db")
```

Die Dateiendung kann frei gewählt werden und hat keinerlei Einfluss

auf die Funktionsweise der Datenbank. Obiger Code führt dazu, dass die Datenbank, die in der Datei *lagerverwaltung.db* im selben Verzeichnis wie das Programm liegt, eingelesen und mit dem Connection-Objekt connection verbunden wird. Wenn es noch keine Datei mit dem Namen *lagerverwaltung.db* gibt, so wird eine leere Datenbank erzeugt und die Datei angelegt.

Oft benötigt man eine Datenbank nur während des Programmlaufs, um Daten zu verwalten oder zu ordnen, ohne dass diese dauerhaft auf der Festplatte gespeichert werden müssen. Zu diesem Zweck gibt es die Möglichkeit, eine Datenbank im Arbeitsspeicher zu erzeugen, indem man anstatt eines Dateinamens den String ":memory:" an die connect-Methode übergibt:

```
>>> connection = sqlite3.connect(":memory:")
```

Um mit der verbundenen Datenbank arbeiten zu können, werden sogenannte *Cursor* (dt. *Positionsanzeigen*) benötigt. Einen Cursor kann man sich ähnlich wie den blinkenden Strich in Textverarbeitungsprogrammen als aktuelle Bearbeitungsposition innerhalb der Datenbank vorstellen. Erst mit solchen Cursorn können wir Datensätze verändern oder abfragen, wobei es zu einer Datenbankverbindung beliebig viele Cursor geben kann. Ein neuer Cursor kann mithilfe der cursor-Methode des Connection-Objekts erzeugt werden:

```
cursor = connection.cursor()
```

Neue Tabellen anlegen

Nun können wir endlich unser erstes SQL-Statement an die Datenbank schicken, um unsere Tabellen anzulegen. Für das Anlegen unserer Tabelle »Lager« sähe das SQL-Statement folgendermaßen aus:

```
CREATE TABLE lager (
    fachnummer INTEGER, seriennummer INTEGER, komponente TEXT,
    lieferant TEXT, reserviert INTEGER
)
```

Alle großgeschriebenen Wörter sind Bestandteile der Sprache SQL. Es ist allerdings so, dass SQL nicht zwischen Groß- und Kleinschreibung unterscheidet und wir deshalb auch alles hätten kleinschreiben können. Wegen der besseren Lesbarkeit werden wir SQL-Schlüsselwörter immer komplett groß- und von uns vergebene Namen durchgängig kleinschreiben.

Die Zeichenketten INTEGER und TEXT hinter den Spaltennamen geben den Datentyp an, der in den Spalten gespeichert werden soll. Sinnvollerweise werden die Spalten fachnummer, seriennummer und reserviert als Ganzzahlen und die Spalten komponente und lieferant als Zeichenketten definiert. SQLite kennt mehrere solcher Datentypen, in die Python-Datentypen beim Schreiben der Datenbank automatisch umgewandelt werden, wie es die folgende Tabelle zeigt:

Python-Datentyp	SQLite-Datentyp
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (UTF8-Kodiert)	TEXT
unicode	TEXT
Buffer	BLOB

Tabelle 19.6 So konvertiert SQLite beim Schreiben der Daten.

Es ist auch möglich, andere Datentypen in SQLite-Datenbanken abzulegen, wenn entsprechende Konvertierungsfunktionen definiert wurden. Wie das genau erreicht werden kann, wird später behandelt.

Nun können wir das SQL-Statement mithilfe der execute-Methode des Cursor-Objekts an die SQLite-Datenbank senden:

```
cursor.execute("""CREATE TABLE lager (
    fachnummer INTEGER, seriennummer INTEGER,
    komponente TEXT, lieferant TEXT, reserviert INTEGER)""")
```

Die Tabellen für die Lieferanten und Kunden erzeugen wir auf die gleiche Weise:

```
cursor.execute("""CREATE TABLE lieferanten (
    kurzname TEXT, name TEXT, telefonnummer TEXT)""")

cursor.execute("""CREATE TABLE kunden (
    kundennummer INTEGER, name TEXT, anschrift TEXT)""")
```

Daten in die Tabellen einfügen

Als Nächstes werden wir die noch leeren Tabellen mit unseren Beispieldaten füllen. Zum Einfügen neuer Datensätze in eine bestehende Tabelle dient das INSERT-Statement, das für den ersten Beispieldatensatz folgendermaßen aussieht:

```
INSERT INTO lager VALUES (
    1, 26071987, 'Grafikkarte Typ 1', 'FC', 0
)
```

Innerhalb der Klammern hinter VALUES stehen die Werte für jede einzelne Spalte in der gleichen Reihenfolge, wie auch die Spalten selbst definiert wurden. Wie bei allen anderen Datenbankabfragen auch können wir per execute unser Statement abschicken:

```
cursor.execute("""INSERT INTO lager VALUES (
    1, 26071987, 'Grafikkarte Typ 1', 'FC', 0)""")
```

Beim Einfügen von Datensätzen müssen Sie allerdings beachten, dass die neuen Daten nicht sofort nach Ausführen eines INSERT-Statements in die Datenbank daten geschrieben werden, sondern vorerst nur im Arbeitsspeicher liegen. Um sicherzugehen, dass die Daten wirklich auf der Festplatte landen und damit dauerhaft gespeichert sind, muss man die commit-Methode des Connection-Objekts aufrufen: [Dies ist deshalb notwendig, damit die Datenbank transaktionssicher ist. Transaktionen sind Ketten von Operationen, die vollständig ausgeführt werden müssen, damit die Konsistenz der Datenbank erhalten bleibt. Stellen Sie sich einmal vor, bei einer Bank würde während einer Überweisung zwar das Geld von Ihrem Konto abgebucht, jedoch aufgrund eines Fehlers nicht dem Empfänger gutgeschrieben werden. Mit der Methode rollback können alle Operationen seit dem letzten commit-Aufruf wieder rückgängig gemacht werden, um solche Probleme zu vermeiden.]

```
connection.commit()
```

In der Regel werden die Daten, die wir in die Datenbank einfügen wollen, nicht schon vor dem Programmlauf bekannt sein und deshalb auch nicht in Form von String-Konstanten im Quellcode stehen. Stattdessen werden es Benutzereingaben oder Berechnungsergebnisse sein, die wir dann als Python-Instanzen im Speicher haben. Auf den ersten Blick scheint für solche Fälle der Formatierungsoperator % für Strings ein geeignetes Mittel zu sein, und die letzte INSERT-Anweisung hätte auch folgendermaßen zusammengebaut werden können:

```
>>> werte = (1, 26071987, "Grafikkarte Typ 1", "FC", 0)
>>> "INSERT INTO lager VALUES (%d, %d, '%s', '%s', %d)" %
werte
'INSERT INTO lager VALUES (1, 26071987, 'Grafikkarte Typ 1',
'FC', 0)'
```

Diese auf den ersten Blick sehr elegante Methode entpuppt sich bei genauer Betrachtung aber als gefährliche Sicherheitslücke. Betrachten wir einmal folgende INSERT-Anweisung, die einen neuen Lieferanten in die Tabelle »Lieferanten« einfügen soll:

```
>>> werte = ("DR", "Danger Electronics",
             "666'); Hier kann Schadcode stehen")
>>> "INSERT INTO lieferanten VALUES ('%s', '%s', '%s')" %
```

```
werte
'INSERT INTO lager VALUES ('DR', 'Danger Electronics', '666');
Hier kann Schadcode stehen')'
```

Wie Sie sehen, haben wir dadurch, dass der Wert für die Telefonnummer den String "');" enthält, die SQL-Abfrage verunstaltet, sodass der Versuch, sie auszuführen, zu einem Fehler führen und damit unser Programm zum Absturz bringen würde. Durch den außerdem enthaltenen Text "Hier kann Schadcode stehen" wird angedeutet, dass es unter Umständen sogar möglich ist, eine Abfrage so zu manipulieren, dass wieder gültiger SQL-Code dabei herauskommt, wobei jedoch eine andere Operation als beabsichtigt (zum Beispiel das Auslesen von Benutzerdaten) ausgeführt wird. [Man nennt diese Form des Angriffs auf verwundbare Programme auch ***SQL Injection***.]

Verwenden Sie deshalb niemals die String-Formatierung zur Übergabe von Parametern in SQL-Abfragen!

Um sichere Parameterübergaben durchzuführen, schreibt man in den Query-String an die Stelle, an der der Parameter stehen soll, ein Fragezeichen und übergibt der execute-Methode ein Tupel mit den entsprechenden Werten als zweiten Parameter:

```
werte = ("DR", "Danger Electronics",
         "666'); Hier kann Schadcode stehen")
sql = "INSERT INTO lieferanten VALUES (?, ?, ?)"
cursor.execute(sql, werte)
```

In diesem Fall kümmert sich SQLite darum, dass die übergebenen Werte korrekt umgewandelt werden und es zu keinen Sicherheitslücken durch böswillige Parameter kommen kann.

Analog zur String-Formatierung gibt es auch hier die Möglichkeit, den übergebenen Parametern Namen zu geben und statt der tuple-Instanz mit einem Dictionary zu arbeiten. Dazu wird im Query-String statt des Fragezeichens ein Doppelpunkt, gefolgt von dem symbolischen Namen des Parameters, geschrieben und das passende Dictionary als zweiter Parameter an execute übergeben:

```
werte = {"kurz" : "DR", "name" : "Danger Electronics",
         "telefon" : "123456"}
sql = "INSERT INTO lieferanten VALUES (:kurz, :name,
:telefon)"
cursor.execute(sql, werte)
```

Mit diesem Wissen können wir unsere Tabellen elegant und sicher mit Daten füllen:

```
for row in ((1, "2607871987", "Grafikkarte Typ 1", "FC", 0),
            (2, "19870109", "Prozessor Typ 13", "LPE", 57),
            (10, "06198823", "Netzteil Typ 3", "FC", 0),
            (25, "11198703", u"LED-Lüfter", "FC", 57),
            (26, "19880105", "Festplatte 128 GB", "LPE", 12)):
    cursor.execute("INSERT INTO lager VALUES (?,?,?,?,?)",
row)
```

Ihnen ist bestimmt aufgefallen, dass der String "LED-Lüfter" als einziger durch das führende »u« in eine unicode-Instanz überführt worden ist. Das ist deshalb notwendig, weil SQLite nur mit UTF-8 kodierte Strings richtig verarbeiten kann. Wie Sie aber seit Abschnitt 8.5.3, »Strings – str, unicode«, über String-Kodierung wissen, werden meist standardmäßig andere Kodierungsverfahren verwendet, was zu Fehlern beim Datenbankzugriff führen würde. Durch die Umwandlung in eine unicode-Instanz sind wir in jedem Fall auf der sicheren Seite. Zusätzlich könnten Sie den Umlaut ü auch durch die Escape-Sequenz \xfc kodieren, was zusätzlich den Vorteil hätte, dass Ihr Quellcode nur aus ASCII-Zeichen bestünde und somit unabhängig von der Lokalisierung des Betriebssystems überall lauffähig ist.

Generell ist es problemlos möglich, an alle im Modul sqlite3 enthaltenen Funktionen Unicode-Strings zu übergeben.

Strukturen wie die obige for-Schleife, die die gleiche Datenbankoperation sehr oft für jeweils andere Daten durchführen, kommen sehr häufig vor und bieten großes Optimierungspotenzial. Aus diesem Grund haben cursor-Instanzen zusätzlich noch die Methode executemany, die als zweiten Parameter eine Sequenz oder

ein anderes iterierbares Objekt erwartet, das die Daten für die einzelnen Operationen enthält. Wir nutzen executemany, um unsere Tabellen »Lieferanten« und »Kunden« mit Daten zu füllen:

```
lieferanten = (("FC", "FiboComputing Inc.", "011235813"),
               ("LPE", "LettgenPetersErnesti", "026741337"),
               ("GC", "Golden Computers", "016180339"))
cursor.executemany("INSERT INTO lieferanten VALUES (?,?,?)",
                   lieferanten)

kunden = ((12, "Heinz Elhurg",
           "Turnhallenstr. 1, 3763 Sporthausen"),
          (57, "Markus Altbert",
           u"K\xe4mperweg 24, 2463 Duisschloss"),
          (64, "Steve Apple",
           "Podmacstr 2, 7467 Iwarhausen"))
cursor.executemany("INSERT INTO kunden VALUES (?,?,?)",
kunden)
```

Nun haben wir gelernt, wie man Datenbanken und Tabellen anlegt und diese mit Daten füllt. Im nächsten Schritt wollen wir uns mit dem Abfragen von Daten beschäftigen.

Daten abfragen

Um Daten aus der Datenbank abzufragen, dient das SELECT-Statement. SELECT erwartet als Parameter durch Kommata getrennt die Spalten, die uns von den Datensätzen interessieren, und den Tabellennamen der Tabelle, aus der wir abfragen wollen. Standardmäßig werden alle Zeilen aus der abgefragten Tabelle zurückgegeben. Mit einer WHERE-Klausel können wir nur bestimmte Datensätze auswählen, indem wir Bedingungen für die Auswahl angeben. Stark vereinfacht ist ein SELECT-Statement folgendermaßen aufgebaut:

```
SELECT <spaltenliste> FROM <tabellenname> [WHERE <bedingung>]
```

Wie durch die eckigen Klammern angedeutet wird, ist die WHERE-Klausel optional und kann entfallen.

Wenn wir beispielsweise alle belegten Fachnummern und die dazugehörigen Komponenten abfragen wollen, können wir das mit dem folgenden Statement tun:

```
SELECT fachnummer, komponente FROM lager
```

Auch bei Datenabfragen benutzen wir die execute-Methode des Cursor-Objekts, um der Datenbank unser Anliegen mitzuteilen. Anschließend können wir uns mittels cursor.fetchall alle Datensätze zurückgeben lassen, die unsere Abfrage ergeben hat:

```
>>> cursor.execute("SELECT fachnummer, komponente FROM lager")
>>> cursor.fetchall()
[(1, u'Grafikkarte Typ 1'), (2, u'Prozessor Typ 13'),
 (10, u'Netzteil Typ 3'), (25, u'LED-L\xfcfter'),
 (26, u'Festplatte 128 GB')]
```

Der Rückgabewert von fetchall ist eine Liste, die für jeden Datensatz ein Tupel mit den Werten der angeforderten Spalten enthält. [Standardmäßig werden bei der Abfrage alle TEXT-Spalten als unicode-Instanzen zurückgegeben. Wie Sie dieses Verhalten anpassen können, werden wir später behandeln.]

Mit einer passenden WHERE-Klausel können wir die Auswahl auf die Computerteile beschränken, die noch nicht reserviert sind:

```
>>> cursor.execute("""
    SELECT fachnummer, komponente FROM lager WHERE reserviert=0
""")
>>> cursor.fetchall()
[(1, u'Grafikkarte Typ 1'), (10, u'Netzteil Typ 3')]
```

Es können auch mehrere Bedingungen mittels logischer Operatoren wie AND und OR zusammengefasst werden. Damit könnten wir beispielsweise ermitteln, welche Artikel, die von der Firma »FiboComputing Inc.« geliefert wurden, schon reserviert worden sind:

```
>>> cursor.execute("""
    SELECT fachnummer, komponente FROM lager
    WHERE reserviert!=0 AND lieferant='FC'
""")
>>> cursor.fetchall()
[(25, u'LED-L\xfcfter')]
```

Da es lästig ist, immer die auszuwählenden Spaltennamen anzugeben, und man sehr oft Abfragen über alle Spalten machen möchte, gibt es dafür eine verkürzte Schreibweise, bei der die Spaltenliste durch ein Sternchen ersetzt wird:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[(12, u'Heinz Elhurg', u'Turnhallenstr. 1, 3763 Sporthausen'),
(57, u'Markus Altbert', u'K\xe4mperweg 24, 2463 Duisschloss'),
(64, u'Steve Apple', u'Podmacstr 2, 7467 Iwarhausen')]
```

Die Reihenfolge der Spaltenwerte richtet sich danach, in welcher Reihenfolge die Spalten der Tabelle mit `CREATE` definiert wurden.

Als letzte Ergänzung zum `SELECT`-Statement wollen wir uns mit den Abfragen über mehrere Tabellen, den sogenannten *Joins* (dt. *Verbindungen*), beschäftigen. Wir möchten zum Beispiel abfragen, welche Komponenten des Lieferanten mit der Telefonnummer `011235813` zurzeit im Lager vorhanden sind und in welchen Fächern sie liegen.

Eine Abfrage über mehrere Tabellen unterscheidet sich von einfachen Abfragen dadurch, dass anstelle des einfachen Tabellennamens eine durch Kommata getrennte Liste angegeben wird, die alle an der Abfrage beteiligten Tabellen enthält. Wenn auf Spalten, zum Beispiel in der `WHERE`-Bedingung, verwiesen wird, muss der jeweilige Tabellenname mit angegeben werden. Das gilt auch für die auszuwählenden Spalten direkt hinter `SELECT`. Unsere Beispielabfrage betrifft nur die Tabellen »Lager« und »Lieferanten« und lässt sich als Join folgendermaßen formulieren:

```
SELECT lager.fachnummer, lager.komponente, lieferanten.name
FROM lager, lieferanten
WHERE lieferanten.telefonnummer='011235813' AND
      lager.lieferant=lieferanten.kurzname
```

Man kann sich die Verarbeitung eines solchen Joins so vorstellen, dass die Datenbank jede Zeile der Tabelle »Lager« mit jeder Zeile der Tabelle »Lieferanten« zu neuen Datensätzen verknüpft und aus der dadurch entstehenden Liste alle Zeilen zurückgibt, bei denen die Spalte `lieferanten.telefonnummer` den Wert `'011235813'` hat und die Spalten `lager.lieferant` und `lieferanten.kurzname` übereinstimmen.

Führen wir die Abfrage mit SQLite aus, erhalten wir die erwartete Ausgabe:

```
>>> sql = """
SELECT lager.fachnummer, lager.komponente, lieferanten.name
FROM lager, lieferanten
WHERE lieferanten.telefonnummer='011235813' AND
      lager.lieferant=lieferanten.kurzname"""
>>> cursor.execute(sql)
>>> cursor.fetchall()
[(1, u'Grafikkarte Typ 1', u'FiboComputing Inc.'),
 (10, u'Netzteil Typ 3', u'FiboComputing Inc.'),
 (25, u'LED-L\xfcfter', u'FiboComputing Inc.')]
```

Bis hierher haben wir nach einer Abfrage immer mit `cursor.fetchall` direkt alle Ergebnisse der Abfrage aus der Datenbank geladen und dann gesammelt ausgegeben. Diese Methode eignet sich allerdings nur für relativ kleine Datenmengen, da erstens das Programm so lange warten muss, wie die Datenbank noch alle Ergebnisse ermittelt und zurückgibt, und zweitens das Resultat komplett als Liste im Speicher gehalten wird. Dass dies bei sehr umfangreichen Ergebnissen eine Verschwendung von Speicherplatz darstellt, bedarf keiner weiteren Erklärung. Aus diesem Grund gibt es die Möglichkeit, die Daten zeilenweise, also immer in kleinen Portionen, abzufragen. Wir erreichen durch dieses Vorgehen, dass wir nicht mehr auf die Berechnung der kompletten Ergebnismenge warten müssen, sondern schon währenddessen mit der Verarbeitung beginnen können. Außerdem müssen nicht mehr alle Datensätze zeitgleich im Arbeitsspeicher verfügbar sein.

Mit der Methode fetchone der cursor-Klasse können wir jeweils ein Ergebnis-Tupel anfordern. Wurden bereits alle Datensätze der letzten Abfrage ausgelesen, gibt fetchone None zurück. Damit lassen sich auch große Datenmengen speichereffizient auslesen, auch wenn unser Beispiel mangels einer großen Datenbank nur drei Zeilen ermittelt:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> row = cursor.fetchone()
>>> while row:
        print row
        row = cursor.fetchone()
(12, u'Heinz Elhurg', u'Turnhallenstr. 1, 3763 Sporthausen')
(57, u'Markus Altbert', u'K\xe4mperweg 24, 2463 Duisschloss')
(64, u'Steve Apple', u'Podmacstr 2, 7467 Iwarhausen')
```

Diese Methode führt durch die while-Schleife zu etwas holprigem Code und wird deshalb seltener eingesetzt. Eine wesentlich elegantere Methode bietet die Iterator-Schnittstelle der cursor-Klasse, die es uns erlaubt, wie bei einer Liste mithilfe von for über die Ergebniszeilen zu iterieren:

```
>>> for row in cursor:
        print row
(12, u'Heinz Elhurg', u'Turnhallenstr. 1, 3763 Sporthausen')
(57, u'Markus Altbert', u'K\xe4mperweg 24, 2463 Duisschloss')
(64, u'Steve Apple', u'Podmacstr 2, 7467 Iwarhausen')
```

Aufgrund des wesentlich besser lesbaren Programmtextes ist die Iterator-Methode für solche Anwendungen der Methode fetchone vorzuziehen. Sie sollten fetchone nur dann benutzen, wenn Sie gezielt jede Ergebniszeile separat und auf eine andere Weise verarbeiten wollen.

Der Umgang mit Datentypen bei SQLite

Wie Ihnen sicherlich schon aufgefallen ist, gibt SQLite beim Abfragen von Daten für TEXT-Spalten immer unicode-Instanzen zurück, auch wenn beim Schreiben der Daten ursprünglich eine str-Instanz an die Datenbank übergeben wurde. Aus dem einleitenden Teil dieses Abschnitts kennen Sie bereits das Schema, nach dem SQLite Daten beim Schreiben der Datenbank konvertiert. Die entsprechende Rückübersetzung von SQLite-Datentypen zu Python-Datentypen wird durch folgende Tabelle beschrieben:

SQLite-Datentyp	Python-Datentyp
NULL	None
INTEGER	int oder long, je nachdem, ob der Wertebereich von int ausreicht oder nicht.
REAL	float
TEXT	unicode
BLOB	buffer

Tabelle 19.7 Typumwandlung beim Lesen von SQLite-Datenbanken

Im Wesentlichen wirft diese Tabelle nur zwei Fragen auf: Wie speichere ich andere Datentypen, beispielsweise Listen oder meine eigenen Klassen, in der Datenbank, wenn doch nur diese Typen unterstützt werden? Und wie kann ich erreichen, dass ich für TEXT-Spalten anstatt unicode- wieder str-Instanzen erhalte?

Wir werden zuerst die zweite Frage beantworten.

Connection.text_factory

Jede von sqlite3.connect erzeugte Connection-Instanz hat ein Attribut text_factory, das eine Referenz auf eine Funktion enthält, die immer dann aufgerufen wird, wenn TEXT-Spalten ausgelesen werden. Im Ergebnistupel der Datenbankabfrage steht dann der Rückgabewert dieser Funktion. Standardmäßig ist das text_factory-Attribut auf die Built-in Function unicode gesetzt, was auch erklärt, warum immer unicode-Instanzen zurückgegeben werden:

```
>>> connection = sqlite3.connect("lagerverwaltung.db")
>>> connection.text_factory
<type 'unicode'>
```

Um unser Ziel zu erreichen, `str`-Instanzen für `TEXT`-Spalten zu erhalten, können wir eine eigene `text_factory`-Funktion angeben. Diese Funktionen müssen einen Parameter erwarten und den konvertierten Wert zurückgeben. Im Falle der `str`-Instanzen brauchen wir uns noch nicht einmal eine eigene Funktion dafür zu schreiben, da die Built-in Function `str` bereits alle Kriterien erfüllt. Damit können wir unsere Kundendaten folgendermaßen direkt als Byte-Strings ermitteln:

```
>>> connection.text_factory = str
>>> cursor = connection.cursor()
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[(12, 'Heinz Elhurg', 'Turnhallenstr. 1, 3763 Sporthausen'),
 (57, 'Markus Altbert', 'K\xc3\xa4mperweg 24, 2463
Duisschloss'),
 (64, 'Steve Apple', 'Podmacstr 2, 7467 Iwarhausen')]
```

Beachten Sie hierbei, dass `sqlite3` alle Sonderzeichen mit UTF-8 kodiert.

Connection.row_factory

Ein ähnliches Attribut wie `text_factory` für `TEXT`-Spalten existiert auch für ganze Zeilen. In dem Attribut `row_factory` kann eine Referenz auf eine Funktion gespeichert werden, die Zeilen für das Benutzerprogramm aufbereitet. Standardmäßig wird die Funktion `tuple` benutzt. Wir wollen beispielhaft eine Funktion implementieren, die uns auf die Spaltenwerte eines Datensatzes über die Namen der jeweiligen Spalten zugreifen lässt. Das Ergebnis soll dann folgendermaßen aussehen:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[{'anschrift': 'Turnhallenstr. 1, 3763 Sporthausen',
'kundennummer': 12, 'name': 'Heinz Elhurg'},
 {'anschrift': 'K\xc3\xa4mperweg 24, 2463 Duisschloss',
 'kundennummer': 57, 'name': 'Markus Altbert'},
 {'anschrift': 'Podmacstr 2, 7467 Iwarhausen', 'kundennummer':
64,
 'name': 'Steve Apple'}]
```

Um dies zu bewerkstelligen, benötigen wir noch das Attribut `description` der `Cursor`-Klasse, das uns Informationen zu den Spaltennamen der letzten Abfrage liefert. `description` enthält dabei eine Sequenz, die für jede Spalte ein Tupel mit sieben Elementen bereitstellt, von denen uns aber nur das erste, nämlich der Spaltenname, interessiert:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.description
(('kundennummer', None, None, None, None, None, None),
 ('name', None, None, None, None, None, None),
 ('anschrift', None, None, None, None, None, None))
```

Die `row_factory`-Funktionen erhalten als Parameter eine Referenz auf den Cursor, der für die Abfrage verwendet wurde, und die Ergebniszeile als Tupel.

Mit diesem Wissen können wir unsere `row_factory`-Funktion namens `zeilen_dict` wie folgt implementieren:

```
def zeilen_dict(cursor, zeile):
    ergebnis = {}
    for spaltennr, spalte in enumerate(cursor.description):
        ergebnis[spalte[0]] = zeile[spaltennr]
    return ergebnis
```

Zur Erinnerung: `enumerate` erzeugt einen Iterator, der für jedes Element der übergebenen Sequenz ein Tupel zurückgibt, das den Index des Elements in der Sequenz und seinen Wert enthält.

In der Praxis arbeitet unsere `row_factory` wie folgt:

```
>>> connection.row_factory = zeilen_dict
```

```
>>> cursor = connection.cursor()
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[{'anschrift': 'Turnhallenstr. 1, 3763 Sporthausen',
'kundennummer': 12, 'name': 'Heinz Elburg'},
 {'anschrift': 'K\xc3\xa4mperweg 24, 2463 Duisschloss',
 'kundennummer': 57, 'name': 'Markus Altbert'},
 {'anschrift': 'Podmacstr 2, 7467 Iwarhausen', 'kundennummer':
64,
 'name': 'Steve Apple'}]
```

Pythons `sqlite3`-Modul liefert schon eine erweiterte `row_factory` namens `sqlite3.Row` mit, die die Zeilen ihn ähnlicher Weise verarbeitet wie unsere `zeilen_dict`-Funktion. Da `sqlite3.Row` sehr stark optimiert ist und außerdem der Zugriff auf die Spaltenwerte über deren Namen unabhängig von Groß- und Kleinschreibung erfolgen kann, sollten Sie die eingebaute Funktion unserem Beispiel vorziehen und nur dann eine eigene `row_factory` implementieren, wenn Sie etwas ganz anderes erreichen möchten.

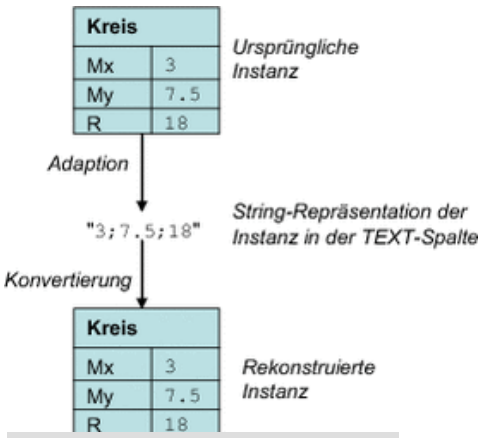
Nach diesem kleinen Ausflug zu den `factory`-Funktionen wenden wir uns endlich der ersten unserer beiden Fragen zu: Wie können wir beliebige Datentypen in SQLite-Datenbanken speichern?

Adapter und Konvertierer

Wie Sie bereits wissen, unterstützt SQLite nur eine beschränkte Menge von Datentypen. Als Folge davon müssen wir alle anderen Datentypen, die wir in der Datenbank ablegen möchten, durch die vorhandenen abbilden. Aufgrund ihrer Flexibilität eignen sich die `TEXT`-Spalten am besten, um beliebige Daten aufzunehmen, weshalb wir uns im Folgenden auf diese beschränken.

Analog zur String-Kodierung, bei der wir `unicode`-Instanzen mittels ihrer `encode`-Methode in gleichwertige `str`-Instanzen umformen und die ursprünglichen Unicode-Daten mithilfe der `decode`-Methode wiederherstellen konnten, brauchen wir nun Operationen, um beliebige Datentypen erst in Strings zu transformieren und anschließend die Ursprungsdaten wieder aus dem String zu extrahieren.

Das Umwandeln von beliebigen Datentypen in einen String wird *Adaption* genannt, und die Rückgewinnung der Daten aus diesem String heißt *Konvertierung*. Abbildung 19.4 veranschaulicht diesen Zusammenhang am Beispiel der Klasse `Kreis`, die als Attribute die Koordinaten des Kreismittelpunktes `Mx` und `My` sowie die Länge des Radius `R` besitzt:



Hier klicken, um das Bild zu vergrößern

Abbildung 19.4 Schema der Adaption und Konvertierung

Eine entsprechende Kreis-Klasse lässt sich folgendermaßen definieren:

```
class Kreis(object):
    def __init__(self, mx, my, r):
        self.Mx = mx
        self.My = my
        self.R = r
```

Nun müssen wir eine Adapterfunktion erstellen, die aus unseren Kreis-Instanzen Strings macht. Die Umwandlung nehmen wir so vor, dass wir einen String erstellen, der durch Semikola getrennt die drei

Attribute des Kreises enthält:

```
def kreisadapter(k):
    return "%f;%f;%f" % (k.Mx, k.My, k.R)
```

Damit die Datenbank weiß, dass wir die Kreise mit dieser Funktion adaptieren möchten, muss sie registriert und mit dem Datentyp `Kreis` verknüpft werden. Dies geschieht durch den Aufruf der `sqlite3.register_adapter`-Methode, die als ersten Parameter den zu adaptierenden Datentyp und als zweiten Parameter die Adapterfunktion erwartet:

```
>>> sqlite3.register_adapter(Kreis, kreisadapter)
```

Durch diese Schritte ist es uns möglich, Kreise in `TEXT`-Spalten abzulegen. Wirklich nützlich wird das Ganze aber erst dann, wenn beim Auslesen auch automatisch wieder `Kreis`-Instanzen generiert werden.

Deshalb müssen wir noch die Umkehrfunktion von `kreisadapter`, den Konverter, definieren, der aus dem String die ursprüngliche `Kreis`-Instanz wiederherstellt. In unserem Beispiel erweist sich das als sehr einfach:

```
def kreiskonverter(string):
    mx, my, r = string.split(";")
    return Kreis(float(mx), float(my), float(r))
```

Genau wie der Adapter muss auch die Konverterfunktion bei SQLite registriert werden, was mit der Methode `sqlite3.register_converter()` erreicht wird:

```
>>> sqlite3.register_converter("KREIS", kreiskonverter)
```

Anders als `register_adapter` erwartet `register_convert` dabei einen String als ersten Parameter, der dem zu konvertierenden Datentyp einen Namen innerhalb von SQLite zuweist. Dadurch haben wir einen neuen SQLite-Datentyp namens `KREIS` definiert, den wir genau wie die eingebauten Typen für die Spalten unserer Tabellen verwenden können. Allerdings müssen wir SQLite beim Verbinden zu der Datenbank mitteilen, dass wir von uns definierte Typen verwenden möchten. Dazu übergeben wir der `connect`-Methode einen entsprechenden Wert als Schlüsselwortparameter `detect_types`:

```
>>> connection = sqlite3.connect(":memory:",
detect_types=sqlite3.PARSE_DECLTYPES)
```

Nachfolgend wird die Definition und Verwendung unseres neuen Datentyps `kreis` in einem Miniprogramm demonstriert:

```
import sqlite3

class Kreis(object):
    def __init__(self, mx, my, r):
        self.Mx = mx
        self.My = my
        self.R = r

def kreisadapter(k):
    return "%f;%f;%f" % (k.Mx, k.My, k.R)

def kreiskonverter(string):
    mx, my, r = string.split(";")
    return Kreis(float(mx), float(my), float(r))

# Adapter und Konverter registrieren
sqlite3.register_adapter(Kreis, kreisadapter)
sqlite3.register_converter("KREIS", kreiskonverter)

# Hier wird eine Beispieldatenbank im Arbeitsspeicher mit
# einer einspaltigen Tabelle für Kreise definiert
connection = sqlite3.connect(":memory:",
                             detect_types=sqlite3.PARSE_DECLTYPES)
cursor = connection.cursor()
cursor.execute("CREATE TABLE kreis_tabelle(k KREIS)")

# Kreis in die Datenbank schreiben
kreis = Kreis(1, 2.5, 3)
```

```
cursor.execute("INSERT INTO kreis_tabelle VALUES (?)",
(kreis,))

# Kreis wieder auslesen
cursor.execute("SELECT * FROM kreis_tabelle")

gelesener_kreis = cursor.fetchall()[0][0]
print type(gelesener_kreis)
print gelesener_kreis.Mx, gelesener_kreis.My,
gelesener_kreis.R
```

Die Ausgabe dieses Programms ergibt sich wie folgt und zeigt, dass `gelesener_kreis` tatsächlich eine Instanz unserer `Kreis`-Klasse mit den korrekten Attributen ist:

```
<class '__main__.Kreis'>
1.0 2.5 3.0
```

Einschränkungen

Das Datenbanksystem SQLite ist in bestimmten Punkten eingeschränkt. Beispielsweise wird eine Datenbank beim Verändern oder Hinzufügen von Datensätzen für Lesezugriffe gesperrt, was besonders bei Webanwendungen sehr hinderlich ist: In der Regel werden mehrere Besucher eine Internetseite gleichzeitig aufrufen, und wenn jemand beispielsweise einen neuen Foreneintrag erstellt, wollen die anderen Besucher nicht länger auf die Anzeige der Seite warten müssen.

Deshalb gibt es andere Systeme, die auch mit den Anforderungen größerer Projekte zurechtkommen. Wir werden uns im nächsten Abschnitt mit dem verbreiteten Datenbankserver namens MySQL beschäftigen.

top

19.3.2 MySQLdb

MySQL ist ein Datenbankserver, der speziell für die schnelle Verwaltung sehr großer Datenmengen konzipiert wurde. Das System wird von der schwedischen Firma *MySQL AB* entwickelt und ist als quelloffene Software unter der *GNU General Public License* (GPL) verfügbar. Sie können sich die Software von der Homepage des Herstellers kostenlos herunterladen: http://www.mysql.com.

Für Internetanwendungen hat sich MySQL zum De-facto-Standard entwickelt, sodass heute praktisch jedes Hosting-Paket auch MySQL-Datenbanken enthält. Weil Python sich besonders für die Entwicklung von Webanwendungen eignet, haben wir uns entschlossen, an dieser Stelle die MySQL-Datenbank zu beschreiben, auch wenn sie nicht Teil der Python-Standardbibliothek ist.

Im Gegensatz zu SQLite, das sämtliche Funktionen der Datenbank in den Python-Interpreter kompiliert, wählt MySQL einen *Client/Server-Ansatz*. Das bedeutet, dass sich ein eigenes Programm, der sogenannte Server, um die Verwaltung der Daten kümmert. Die sogenannten Clients können sich mit diesem Server verbinden und ihre Abfragen senden bzw. Daten empfangen. Die Verbindungen zwischen Server und Client werden über Netzwerkschnittstellen aufgebaut, sodass der Datenbankserver nicht auf demselben Rechner laufen muss wie seine Clients. Außerdem können auf diese Weise viele Clients, die auf verschiedenen Computern laufen, denselben Datenbestand benutzen, ohne sich darum kümmern zu müssen, wo die Daten letztendlich physikalisch abgelegt sind.

Neben der größeren Flexibilität ermöglicht der Client/Server-Ansatz eine Rechteverwaltung für die Benutzer der Datenbank. Jeder Client muss sich beim Verbindungsaufbau zu dem Server mit Zugangsdaten einloggen. Der Server kann dann anhand der Benutzerdaten bestimmen, welche Aktionen der Client durchführen darf, oder den Verbindungsaufbau ganz ablehnen, wenn der übergebene Benutzer nicht existiert oder das Passwort falsch war.

Für den Zugriff auf die Datenbank muss der Client neben den Login-Daten nur die Netzwerkadresse des Datenbankservers kennen.

Sie können eine aktuelle Version des `MySQLdb`-Moduls für Python unter der Adresse http://sourceforge.net/projects/mysql-python/

herunterladen.

Nach der Installation können Sie das Modul MySQLdb mittels import einbinden:

```
>>> import MySQLdb
```

Nun können Sie mit der connect-Funktion eine Verbindung zu einem Datenbankserver herstellen. Wir gehen bei den folgenden Beispielen davon aus, dass ein MySQL-Server unter der Adresse "192.168.0.128" erreichbar ist und einen Benutzer namens "root" mit dem Passwort "123456" hat:

```
>>> connection = MySQLdb.connect("192.168.0.128",
                                 "root", "123456", "test")
```

Der letzte Parameter mit dem Wert "test" gibt die Datenbank an, die wir auf dem Server verwenden möchten. MySQL ermöglicht es nämlich, mehrere Datenbanken auf demselben Server zu verwalten. Wir gehen davon aus, dass die Datenbank namens "test" bereits auf dem Server existiert, denn sie wird bei der Verbindung nicht automatisch erzeugt.

Nun können wir genau wie bei sqlite3 ein Cursor-Objekt für die Verbindung erzeugen und damit Anfragen an die Datenbank senden. Die Details zum Umgang mit Datenbankverbindungen und Cursor-Objekten können Sie im Abschnitt über SQLite nachlesen.

Zur Demonstration werden wir eine einfache Tabelle erzeugen, die die Marke, die Leistung und das Baujahr von Autos speichern kann:

```
import MySQLdb
connection = MySQLdb.connect("www.hostname.de",
                             "benutzername", "passwort",
"datenbank")
cursor = connection.cursor()
cursor.execute("""
   CREATE TABLE autos (marke TEXT, ps INTEGER, baujahr
INTEGER)
    """)
cursor.execute("INSERT INTO autos VALUES('Volvo', 130, 1995)")
daten = (("Audi", 350, 2005),
         ("Ford", 110, 2000),
         ("VW", 60, 2001))
cursor.executemany("INSERT INTO autos VALUES(%s, %s, %s)",
daten)

cursor.execute("SELECT marke, ps, baujahr FROM autos")
for row in cursor:
    print "Marke: %s, Leistung: %d PS, Baujahr: %d" % row
```

Die Ausgabe des Programms sieht folgendermaßen aus:

```
Marke: Volvo, Leistung: 130 PS, Baujahr: 1995
Marke: Audi, Leistung: 350 PS, Baujahr: 2005
Marke: Ford, Leistung: 110 PS, Baujahr: 2000
Marke: VW, Leistung: 60 PS, Baujahr: 2001
```

Im Unterschied zu sqlite3 verwendet MySQLdb bei Parameterübergaben nicht das Fragezeichen als Platzhalter, sondern die Zeichenfolge "%s". Wie im Beispiel gezeigt wird, können Sie "%s" auch für andere Datentypen als str verwenden. Das Modul MySQLdb kümmert sich dann um die entsprechende Umwandlung.

Anstelle von "%s" können Sie die Parameter auch mit Namen versehen und dann statt einer Sequenz ein Dictionary übergeben. Die folgenden beiden Aufrufe von execute erzeugen das gleiche Ergebnis:

```
>>> cursor.execute("""
    SELECT * FROM autos WHERE marke=%s AND ps=%s""",
    ("Volvo", 130)
    )
>>> cursor.execute("""
    SELECT * FROM autos
    WHERE marke=%(marke)s AND ps=%(leistung)s""",
    {"marke" : "Volvo", "Leistung" : 130}
    )
```

Allerdings bietet MySQLdb keine komfortable Schnittstelle zum Speichern eigener Datentypen. Sie müssen sich in Ihren Programmen selbst darum kümmern, dass Ihre Objekte beim Speichern in einen

String und beim Lesen wieder in den entsprechenden Datentyp konvertiert werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

19.4 Serialisierung von Instanzen – pickle

Das Modul `pickle` (dt. *pökeln*) bietet komfortable Funktionen für das *Serialisieren* von Objekten. Beim Serialisieren eines Objekts wird ein String erzeugt, der alle Informationen des Objekts speichert, sodass es später wieder durch das sogenannte *Deserialisieren* rekonstruiert werden kann.

Besonders für die dauerhafte Speicherung von Daten in Dateien ist `pickle` sehr gut geeignet. Folgende Datentypen können mithilfe von `pickle` serialisiert bzw. deserialisiert werden:

- `None`, `True`, `False`
- Numerische Datentypen (`int`, `long`, `float`, `complex`)
- `str`, `unicode`
- Sequenzielle Datentypen (`tuple`, `list`), Mengen (`set`, `frozenset`) und Dictionarys (`dict`), solange alle ihre Elemente auch von `pickle` serialisiert werden können.
- Globale Funktionen
- Built-in Functions
- Globale Klassen
- Klasseninstanzen, deren Attribute serialisiert werden können.

Bei Klassen und Funktionen muss beachtet werden, dass solche Objekte beim Serialisieren nur mit ihrem Klassennamen gespeichert werden. Der Code einer Funktion oder die Definition

der Klasse und ihre Attribute werden nicht gesichert. Wenn Sie also beispielsweise eine Instanz von einer selbst definierten Klasse deserialisieren möchten, muss die Klasse in dem aktuellen Kontext genauso wie bei der Serialisierung definiert sein. Ist das nicht der Fall, wird ein `UnpicklingError` erzeugt.

Es gibt drei verschiedene Formate, in denen `pickle` seine Daten speichern kann. Jedes dieser Formate hat eine Zahl, um es zu identifizieren:

Nummer	Beschreibung
0	Der resultierende String besteht nur aus ASCII-Zeichen und kann deshalb auch von Menschen beispielsweise zu Debug-Zwecken gelesen werden. Das Protokoll `0` ist der Standard für das `pickle`-Modul und auch zu früheren Versionen von Python (< 2.3) kompatibel.
1	Dieses Protokoll erzeugt einen Binärstring, der die Daten im Vergleich zur ASCII-Variante platzsparender speichert. Auch das Protokoll `1` ist abwärtskompatibel zu Python-Versionen vor 2.3.
2	Neues Binärformat, das besonders für Klasseninstanzen optimiert wurde. Objekte, die mit dem Protokoll `2` serialisiert wurden, können nur von Python-Versionen ab 2.3 gelesen werden.

Tabelle 19.8 Die pickle-Protokolle

Das Modul `pickle` bietet seine Funktionalität über zwei Schnittstellen an: eine imperative über die Funktionen `dump` und `load` und eine objektorientierte mit den Klassen `Pickler` und `Unpickler`.

Um `pickle` verwenden zu können, muss das Modul importiert werden:

```
>>> import pickle
```

Die imperative Schnittstelle

pickle.dump(obj, file[, protocol])

Schreibt die Serialisierung von *obj* in das Dateiobjekt *file*. Das übergebene Dateiobjekt muss dabei für den Schreibzugriff geöffnet worden sein.

Mit dem Parameter *protocol* kann das Protokoll für die Speicherung übergeben werden. Der Standardwert für *protocol* ist `0`. Wird ein Binärformat angegeben, so muss das für *file* übergebene Dateiobjekt im binären Schreibmodus geöffnet worden sein.

```
>>> f = open("pickle-test.txt", "w")
>>> pickle.dump([1, 2, 3], f)
```

Für *file* kann neben echten Dateiobjekten jedes Objekt übergeben werden, das eine `write`-Methode mit einem String-Parameter implementiert, zum Beispiel `StringIO`-Instanzen.

pickle.load(file)

Lädt das nächste serialisierte Objekt aus dem geöffneten Dateiobjekt, das für *file* übergeben wurde. Dabei erkennt *load*

selbstständig, in welchem Fomat die Daten gespeichert wurden.

Das folgende Beispiel setzt voraus, dass im aktuellen Arbeitsverzeichnis eine Datei mit dem Namen *pickle-test.txt* existiert, die eine serialisierte Liste enthält:

```
>>> f = open("pickle-test.txt")
>>> pickle.load(f)
[1, 2, 3]
```

pickle.dumps(obj[, protocol])

Gibt die serialisierte Repräsentation von *obj* als String zurück, wobei der Parameter *protocol* angibt, welches der drei Serialisierungsproktokolle verwendet werden soll. Standardmäßig wird das ASCII-Protokoll mit der Kennung `0` verwendet.

```
>>> pickle.dumps([1, 2, 3])
'(lp0\nI1\naI2\naI3\na.'
```

pickle.loads(string)

Stellt das in *string* serialisierte Objekt wieder her. Das verwendete Protokoll wird dabei automatisch erkannt, und überflüssige Zeichen am Ende des Strings werden ignoriert:

```
>>> s = pickle.dumps([1, 2, 3])
>>> pickle.loads(s)
[1, 2, 3]
```

Die objektorientierte Scbnittstelle

Gerade dann, wenn viele Objekte in dieselbe Datei serialisiert werden sollen, ist es lästig und schlecht für die Lesbarkeit, jedes Mal das Dateiobjekt und das zu verwendende Protokoll bei den Aufrufen von `dump` mit anzugeben.

Neben den schon vorgestellten Modulfunktionen gibt es deshalb noch die beiden Klassen `Pickler` und `Unpickler`.

`Pickler` und `Unpickler` haben außerdem den Vorteil, dass Klassen von ihnen erben und so die Serialisierung anpassen können.

pickle.Pickler(file[, protocol])

Die Parameter *file* und *protocol* haben die gleiche Bedeutung wie bei der `pickle.dump`-Funktion. Das resultierende Pickler-Objekt hat eine Methode namens `dump`, die als Parameter ein Objekt erwartet, das serialisiert werden soll.

Alle an die `load`-Methode gesendeten Objekte werden in das beim Erzeugen der `Pickler`-Instanz übergebene Dateiobjekt geschrieben.

```
>>> p = pickle.Pickler(file("eine_datei.txt", "w"), 2)
>>> p.dump({"vorname" : "Peter", "nachname" : "Kaiser"})
>>> p.dump([1, 2, 3, 4])
```

pickle.Unpickler(file)

Das Gegenstück zu `Pickler` ist `Unpickler`, um aus der übergebenen Datei die ursprünglichen Daten wiederherzustellen. `Unpickler`-Instanzen besitzen eine parameterlose Methode namens `load`, die jeweils das nächste Objekt aus der Datei liest.

Das folgende Beispiel setzt voraus, dass die im Beispiel zur `Pickler`-Klasse erzeugte Datei *eine_datei.txt* im aktuellen Arbeitsverzeichnis liegt:

```
>>> u = pickle.Unpickler(file("eine_datei.txt"))
>>> u.load()
{'nachname': 'Kaiser', 'vorname': 'Peter'}
>>> u.load()
[1, 2, 3, 4]
```

cPickle

Das Modul `pickle` ist komplett in Python implementiert und deshalb für sehr große Datenmengen langsam. Es existiert eine weitere Implementation des `pickle`-Moduls namens `cPickle`, die vollständig in der maschinennahen Sprache C geschrieben wurde.

Mit `cPickle` lassen sich Daten bis zu 1000-mal so schnell wie mit `pickle` serialisieren.

Als einzige Einschränkung muss dabei beachtet werden, dass die Bezeichner `Pickler` und `Unpickler` in `cPickle` keine wirklichen Klassen, sondern Funktionen sind. Deshalb können von ihnen keine eigenen Klassen abgeleitet werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

19.6 Temporäre Dateien – tempfile

Wenn Ihre Programme sehr umfangreiche Daten verarbeiten müssen, ist es oft nicht sinnvoll, alle Daten auf einmal im Arbeitsspeicher zu halten. Für diesen Zweck existieren temporäre Dateien, die es Ihnen erlauben, gerade nicht benötigte Daten vorübergehend auf die Festplatte auszulagern. Für die dauerhafte Speicherung der Daten eignen sich temporäre Dateien nicht.

Für den komfortablen Umgang mit temporären Dateien stellt Python das Modul `tempfile` bereit.

Die wichtigste Funktion dieses Moduls ist `TemporaryFile`, die ein geöffnetes Dateiobjekt zurückgibt, das mit einer neuen temporären Datei verknüpft ist. Die Datei wird für Lese- und Schreibzugriffe im Binärmodus (`"w+b"`) geöffnet. Wir als Benutzer der Funktion brauchen uns dabei um nichts weiter als das Lesen und Schreiben unserer Daten zu kümmern. Das Modul sorgt dafür, dass die temporäre Datei angelegt wird, und löscht sie auch wieder, wenn das Dateiobjekt von der Garbage Collection entsorgt wird.

Das Auslagern von Daten eines Programms auf die Festplatte kann ein Sicherheitsrisiko darstellen, weil andere Programme die Daten auslesen und damit unter Umständen Zugriff auf sicherheitsrelevante Informationen erhalten könnten. Deshalb versucht `TemporaryFile` die Datei sofort nach ihrer Erzeugung aus dem Dateisystem zu entfernen, um sie vor anderen Programmen zu verstecken, falls dies vom Betriebssystem unterstützt wird. Außerdem wird für den Dateiname ein zufälliger String benutzt, der aus sechs Zeichen besteht, wodurch es für andere Programme schwierig wird herauszufinden, zu welchem Programm eine temporäre Datei gehört.

Auch wenn Sie `TemporaryFile` in den meisten Fällen ohne Parameter aufrufen werden, wollen wir die vollständige Schnittstelle besprechen:

`TemporaryFile([mode[, bufsize[, suffix[, prefix[, dir]]]]])`

Die Parameter *mode* und *bufsize* entsprechen den gleichnamigen Argumenten der Built-in Function open (nachzulesen in Abschnitt 9.3, »Dateien«). Mit *suffix* und *prefix* können Sie bei Bedarf den Namen der neuen temporären Datei anpassen. Das, was für *prefix* übergeben wird, wird vor den automatisch erzeugten Dateinamen gesetzt, und der Wert für *suffix* wird hinten an den Dateinamen angehängt. Zusätzlich können Sie mit dem Parameter *dir* angeben, in welchem Ordner die Datei erzeugt werden soll. Standardmäßig kümmert sich `TemporaryFile` auch automatisch um einen Speicherort für die Datei.

Zur Veranschaulichung der Nutzung von `TemporaryFile` folgt ein kleines Beispiel, das erst einen String in einer temporären Datei ablegt und ihn anschließend wieder einliest:

```
>>> import tempfile
>>> tmp = tempfile.TemporaryFile()
>>> tmp.write("Hallo Zwischenspeicher")
>>> tmp.seek(0)
>>> string = tmp.read()
>>> string
'Hallo Zwischenspeicher'
```

Falls Sie nicht wünschen, dass die temporäre Datei verborgen wird, können Sie die Funktion `NamedTemporaryFile` benutzen, die die gleiche Schnittstelle wie `TemporaryFile` hat und sich auch ansonsten bis auf das Verstecken genauso verhält.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

20.2 Zugriff auf Ressourcen im Internet – urllib

Das Modul `urllib` bietet eine komfortable Schnittstelle, um auf Dateien im Internet zuzugreifen. Die zentrale Funktion dieser Bibliothek ist `urlopen`, die der Funktion open ähnelt, bis auf die Tatsache, dass statt eines Dateinamens eine URL übergeben wird. Außerdem können auf dem resultierenden Dateiobjekt aus naheliegendem Grunde keine Schreiboperationen durchgeführt werden.

top

20.2.1 Verwendung des Moduls

Im Folgenden sollen die wichtigsten im Modul `urllib` enthaltenen

Funktionen detailliert besprochen werden.

urllib.urlopen(url[, data[, proxies]])

Die Funktion `urlopen` greift auf die durch URL adressierte Netzwerkressource zu und gibt ein geöffnetes Dateiobjekt auf dieser Ressource zurück. Damit ermöglicht die Funktion es beispielsweise, den Quelltext einer Website herunterzuladen und wie eine lokale Datei einzulesen.

Wenn bei der URL kein Protokoll wie beispielsweise `http://` oder `ftp://` angegeben wurde, wird angenommen, dass die URL auf eine Ressource der lokalen Festplatte verweist. Für Zugriffe auf die lokale Festplatte kann das Protokoll `file://` angegeben werden.

Wenn kein Zugriff auf die Ressource erlangt werden kann, weil die Ressource beispielsweise nicht existiert oder der entsprechende Server nicht erreichbar ist, wird eine `IOError`-Exception geworfen.

Das von der Funktion `urlopen` zurückgegebene Dateiobjekt ist ein dateiähnliches Objekt (engl. *file-like object*), da es nur eine Untermenge der Funktionalität eines echten Dateiobjekts bereitstellt. Die folgende Tabelle zeigt die verfügbaren Methoden des zurückgegebenen dateiähnlichen Objekts mit einer kurzen Beschreibung.

Methode	Beschreibung
`read([size])`	Liest `size` Byte aus der Ressource aus. Wenn `size` nicht angegeben wurde, wird der komplette Inhalt ausgelesen. Die gelesenen Daten werden als String zurückgegeben.
`readline([size])`	Liest eine Zeile aus der Ressource aus. Wenn `size` angegeben wurde, werden maximal `size` Byte gelesen. Die gelesenen Daten werden als String zurückgegeben.
`readlines([sizehint])`	Liest alle Zeilen der Ressource aus und gibt diese in Form einer Liste von Strings zurück. Wenn `sizehint` angegeben wurde, wird nur so lange gelesen, bis ungefähr `sizehint` Bytes gelesen wurden.[1]
`fileno()`	Gibt den Dateideskriptor der geöffneten Ressource als ganze Zahl zurück.
`close()`	Schließt das geöffnete Objekt. Nach Aufruf dieser Methode sind keine weiteren Operationen mehr möglich.
`info()`	Gibt ein dictionary-ähnliches Objekt zurück, das Metainformationen der heruntergeladenen Seite enthält. Im Anschluss an diese Tabelle werden wir uns eingehend mit der von `info` zurückgegebenen Instanz beschäftigen.
`geturl()`	Gibt einen String mit der URL der Ressource zurück.

Tabelle 20.3 Methoden des datei-ähnlichen Objekts

Die Methode `info` des datei-ähnlichen Objekts stellt eine Instanz bereit, die verschiedene Informationen über die Netzwerkressource enthält. Auf diese Informationen kann wie bei einem Dictionary zugegriffen werden, weswegen die von `info`

zurückgegebene Instanz auch *dictionary-ähnliches Objekt* genannt wird. Dazu folgendes Beispiel: [In diesem Zusammenhang bedeutet »ungefähr«, dass die Anzahl der zu lesenden Bytes möglicherweise zu einer internen Puffergröße aufgerundet wird.]

```
>>> f = urllib.urlopen("http://www.galileo-press.de")
>>> d = f.info()
<httplib.HTTPMessage instance at 0x8106c8c>
>>> d.keys()
['content-length', 'expires', 'server', 'connection',
'cache-control', 'date', 'content-type']
```

Im Beispiel wurde auf die Internetressource http://www.galileo-press.de zugegriffen und durch Aufruf der Methode `info` das dictionary-ähnliche Objekt erzeugt, das Informationen zu der Website enthält. Durch die Methode keys eines Dictionarys lassen sich alle enthaltenen Schlüssel anzeigen. Welche Informationen enthalten sind, hängt vom verwendeten Protokoll ab. Beim HTTP-Protokoll enthält das dictionary-ähnliche Objekt alle vom Server gesendeten Informationen. So können beispielsweise über die Schlüssel `"content-length"` und `"server"` die Größe der heruntergeladenen Datei in Byte bzw. der Identifikationsstring der Serversoftware ausgelesen werden:

```
>>> d["content-length"]
'26180'
>>> d["server"]
'Zope/(Zope 2.7.6-final, python 2.3.5, linux2)
ZServer/1.1'
```

Beachten Sie, dass es sich bei dem in diesem Fall verwendeten Server *Zope* um einen Webserver für Python handelt.

Außerdem unterstützen Instanzen des datei-ähnlichen Objekts das Iteratorkonzept. Das heißt, dass sich `urlopen` folgendermaßen verwenden lässt:

```
f = urllib.urlopen("http://www.galileo-press.de")
for zeile in f:
    print zeile
f.close()
```

Dieser Beispielcode würde zeilenweise über den Quelltext von www.galileo-press.de iterieren und in jedem Iterationsschritt die aktuelle Zeile auf dem Bildschirm ausgeben.

Wenn das verwendete Protokoll `http` ist, kann der optionale Parameter *data* dazu verwendet werden, POST-Parameter an die Ressource zu übermitteln. Für den Parameter *data* müssen diese POST-Werte speziell aufbereitet werden. Dazu wird die Funktion `urlencode` verwendet:

```
>>> prm = urllib.urlencode({"prm1" : "wert1", "prm2" :
"wert2"})
>>> f = urllib.urlopen("http://www.beispiel.de", prm)
```

Beachten Sie, dass neben POST eine weitere Methode zur Parameterübergabe an eine Website namens GET existiert. Bei GET werden die Parameter direkt in die URL geschrieben:

```
>>> f = urllib.urlopen("http://www.beispiel.de?
prm1=wert1")
```

Wenn in Ihrem System für bestimmte Protokolle ein Proxy eingestellt ist, beispielsweise über eine Umgebungsvariable oder die Systemkonfiguration, wird diese Einstellung von `urlopen` respektiert und der Proxy verwendet. Es ist aber auch möglich, durch Angabe des dritten Parameters, *proxies*, die zu verwendenden Proxys direkt anzugeben. Hier muss ein Dictionary übergeben werden, das die Protokolle als Schlüssel und die Adresse des jeweiligen Proxys als Wert enthält. Ein Aufruf von `urlopen` unter Verwendung eines Proxys für HTTP- und FTP-Verbindungen sieht folgendermaßen aus:

```
>>> proxies = {
...     "http" : "http://www.proxy.de:8081",
...     "ftp" : "http://www.proxy.de:21"
...     }
>>> f = urllib.urlopen("http://www.google.de", None,
proxies)
```

urllib.urlretrieve(url[, filename[, reporthook[, data]]])

Macht den Inhalt der Ressource, auf die die URL *url* verweist, unter einem lokalen Dateinamen verfügbar. Dazu wird der Inhalt der Ressource heruntergeladen oder kopiert, sofern dies notwendig ist. Wenn sich die Ressource bereits auf der lokalen Festplatte befindet, wird sie nicht kopiert. Die Funktion `urlretrieve` gibt ein Tupel mit zwei Elementen zurück: dem Dateinamen der lokalen Datei und dem Rückgabewert der `info`-Methode des datei-ähnlichen Objekts:

```
>>> urllib.urlretrieve("http://www.galileo-press.de")
('/tmp/tmpHqjxaL', <httplib.HTTPMessage instance at
0xb78d36ec>)
```

Durch Angabe eines Dateinamens als zweiten Parameter kann festgelegt werden, wohin die heruntergeladene Ressource kopiert werden soll. Wenn dieser Parameter angegeben wurde, werden auch lokale Ressourcen kopiert. Normalerweise werden heruntergeladene Ressourcen als temporäre Dateien im entsprechenden Verzeichnis des Betriebssystems gespeichert.

Als dritter Parameter kann ein Funktionsobjekt übergeben werden. Diese Funktion wird aus `urlretrieve` heraus einmal aufgerufen, wenn die Verbindung zur Netzwerkressource hergestellt wurde, und dann öfter, wenn ein Block der Ressource heruntergeladen wurde. Der Callback-Funktion werden drei Parameter übergeben: die Anzahl der bisher übertragenen Blöcke, die Größe eines Blocks in Byte und die Gesamtgröße der Ressource in Byte. Mithilfe dieses dritten Parameters lässt sich also eine Statusanzeige des Downloads realisieren:

```
>>> def f(blocks, blocksize, size):
...     print "Status: %d%%" % (blocksize*blocks*100/size)
...
>>> url = "http://www.galileo-press.de"
>>> res = urllib.urlretrieve(url, "datei.html", f)
Status: 0%
Status: 29%
Status: 58%
Status: 87%
Status: 116%
>>> res
('datei.html', <httplib.HTTPMessage instance at
0xb78d39ac>)
```

Dass die Statusanzeige 116 % anzeigt, liegt daran, dass das letzte Paket nicht die volle Größe hat.

Der vierte Parameter, *data,* entspricht dem Parameter *data* der Funktion `urlopen` und wird auch so verwendet.

urllib.quote(string[, safe])

Ersetzt Sonderzeichen, die in einer URL nicht als solche vorkommen dürfen, durch Escape-Sequenzen der Form `%xx`, wie sie in URLs verwendet werden dürfen. Durch den optionalen Parameter *safe*, einen String, können Zeichen angegeben werden, die nicht in eine Escape-Sequenz umgewandelt werden sollen.

```
>>> urllib.quote("www.test.de/hallo welt.html")
'www.test.de/hallo%20welt.html'
```

urllib.unquote(string)

Die Funktion `unquote` ist das Gegenstück von `quote`. Escape-

Sequenzen der Form %xx im String *string* werden durch das Sonderzeichen ersetzt, und der resultierende String wird zurückgegeben.

```
>>> urllib.unquote("www.test.de/hallo%20welt.html")
'www.test.de/hallo welt.html'
```

urllib.urlencode(query[, doseq])

Erzeugt aus den Schlüssel/Wert-Paaren des Dictionarys *query* einen String des folgenden Formats:

```
>>> urllib.urlencode({"abc" : 1, "def" : "ghi"})
'abc=1&def=ghi'
```

Ein solcher String enthält Parameter, die per POST oder GET an ein serverseitiges Script übergeben werden können. Der Rückgabewert der Funktion `urlencode` kann als Parameter *data* der Funktionen `urlopen` und `urlretrieve` übergeben werden.

Wenn das übergebene Dictionary Sequenzen als Werte enthält und der optionale Parameter *doseq* `True` ist, werden diese Sequenzen zu eigenen Schlüssel/Wert-Paaren aufgebrochen. Dabei wird als Parametername der Schlüssel der jeweiligen Sequenz im Dictionary verwendet:

```
>>> urllib.urlencode({"abc" : [1,2,3], "def" : "ghi"},
True)
'abc=1&abc=2&abc=3&def=ghi'
>>> urllib.urlencode({"abc" : [1,2,3], "def" : "ghi"},
False)
'abc=%5B1%2C+2%2C+3%5D&def=ghi'
```

Wenn für *doseq* `False` übergeben wird, werden eventuell vorhandene Sequenzen als Text in den Parameterstring eingetragen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

20.3 Einlesen einer URL – urlparse

Das Modul `urlparse` ermöglicht es, verschiedene Teile einer URL zu extrahieren, ohne dabei einen unter Umständen außerordentlich komplexen regulären Ausdruck verwenden zu müssen. Im Folgenden werden die Funktionen des Moduls beschrieben.

Um die Beispiele ausführen zu können, muss zuvor das Modul `urlparse` eingebunden worden sein:

```
>>> import urlparse
```

urlparse.urlparse(urlstring[, default_scheme[,

allow_fragments]])

Die Funktion `urlparse` liest die URL *urlstring* ein und bricht sie in mehrere Teile auf. Dabei kann eine URL grundsätzlich aus sechs Teilen bestehen:

```
scheme://netloc/path;params?query#fragment
```

Der `netloc`-Bereich der URL wird außerdem in vier weitere Bereiche unterteilt:

```
username:password@host:port
```

Die meisten der angegebenen URL-Teile sind optional und können in URLs weggelassen werden.

Die sechs Bestandteile der URL werden in Form eines tupelähnlichen Objekts mit sechs Elementen zurückgegeben. Diese am meisten verwendeten Teile der URL können wie bei einem echten Tupel über die Indizes `0` bis `5` angesprochen werden. Zusätzlich – und das unterscheidet die zurückgegebene Instanz von einem Tupel – kann auf alle Teile der URL über Attribute der Instanz zugegriffen werden. Beachten Sie, dass Sie über Attribute auch auf die vier Unterbereiche des `netloc`-Teils zugreifen können, die nicht über einen Index erreichbar sind.

Die folgende Tabelle listet alle Attribute des Rückgabewertes der Funktion `urlparse` auf und erläutert sie jeweils mit einem kurzen Satz. Zusätzlich ist der entsprechende Index angegeben, sofern sich das entsprechende Attribut auch über einen Index ansprechen lässt. Die Attributnamen entsprechen den Namen der Bereiche, wie sie in den obigen URL-Beispielen verwendet wurden.

Attribut	Index	Beschreibung
`scheme`	`0`	Das Protokoll der URL, beispielsweise `http` oder `file`.
`netloc`	`1`	Die *Network Location* besteht üblicherweise aus einem Domainnamen mit Subdomain und TLD, beispielsweise `www.galileo-press.de`. Optional können auch Benutzername, Passwort und Portnummer in `netloc` enthalten sein.
`path`	`2`	Eine Pfadangabe, die einen Unterordner der Network Location kennzeichnet.
`params`	`3`	Parameter für das letzte Element des Pfades.
`query`	`4`	Über den *Query String* können zusätzliche Informationen an ein serverseitiges Script übertragen werden.
`fragment`	`5`	Das Fragment, auch *Anker* genannt. Ein geläufiges Beispiel für einen Anker ist eine Sprungmarke innerhalb einer HTML-Datei.
`username`		Der in der URL angegebene Benutzername, sofern vorhanden.
`password`		Das in der URL angegebene Passwort, sofern vorhanden.
`hostname`		Der Domainname der URL, beispielsweise `www.galileo-press.de`.
`port`		Die in der URL angegebene Portnummer, sofern vorhanden.

Tabelle 20.4 Teile einer URL

Über den optionalen Parameter *default_scheme* ist es möglich, ein Protokoll anzugeben, das in die resultierende Instanz eingetragen wird, wenn in der URL kein Protokoll angegeben wurde.

Der optionale Parameter *allow_fragments* legt fest, ob Fragmente, auch *Anker* genannt, in der URL vorkommen dürfen. Wenn hier `False` übergeben wird, referenziert das Attribut `fragment` der zurückgegebenen Instanz immer `None`, egal, ob ein Fragment übergeben wurde oder nicht. Der Parameter ist mit `True` vorbelegt.

Im folgenden Beispiel soll die URL

`http://www.beispiel.de/pfad/zur/datei.py?prm1=abc`

in ihre Bestandteile zerlegt werden:

```
>>> url = "http://www.beispiel.de/pfad/zur/datei.py?
prm1=abc"
>>> teile = urlparse.urlparse(url)
>>> teile.scheme
'http'
>>> teile.netloc
'www.beispiel.de'
>>> teile.path
'/pfad/zur/datei.py'
>>> teile.params
''
>>> teile.query
'prm1=abc'
>>> teile.fragment
''
>>> teile.hostname
'www.beispiel.de'
```

urlparse.urlunparse(parts)

Die Funktion `urlunparse` ist das Gegenstück zu `urlparse`. Sie erzeugt aus einem Tupel mit sechs Elementen einen URL-String. Statt eines reinen Tupels kann ein beliebiges iterierbares Objekt mit sechs Elementen, unter anderem beispielsweise auch der Rückgabewert von `urlparse`, übergeben werden.

```
>>> url = ("http", "beispiel.de", "/pfad/datei.py", "",
"", "")
>>> urlparse.urlunparse(url)
'http://beispiel.de/pfad/datei.py'
```

Beachten Sie, dass der Ausdruck

```
urlparse.urlunparse(urlparse.urlparse(url)) == url
```

nicht immer `True` ergibt, da überflüssige Angaben, wie beispielsweise ein leeres Fragment am Ende einer URL, beim Aufruf von `urlparse` verloren gehen.

urlparse.urlsplit(urlstring[, default_scheme[, allow_fragments]])

Die Funktion `urlsplit` funktioniert ähnlich wie `urlparse`, mit dem Unterschied, dass das Attribut `params` in der zurückgegebenen Instanz nicht vorhanden ist. Die Parameter werden dann dem Pfad zugeordnet und sind damit im Attribut `path` enthalten. Die Funktion `urlsplit` sollte dann verwendet werden, wenn die neuere URL-Syntax erlaubt sein soll, die es ermöglicht, Parameter an jedes Element des Pfades anzuhängen.

Ansonsten ist die Schnittstelle von `urlsplit` mit der von `urlparse` identisch.

urlparse.urlunsplit(parts)

Die Funktion `urlunsplit` ist das Gegenstück zu `urlsplit` und funktioniert damit genau so wie `urlunparse` in Bezug auf `urlparse`.

urlparse.urljoin(base, url[, allow_fragments])

Die Funktion `urljoin` kombiniert die Basis-URL *base* und die relative URL *url* zu einer absoluten Pfadangabe. Der optionale Parameter *allow_fragments* hat dieselbe Bedeutung wie bei

urlparse.

```
>>> base = "http://www.test.de"
>>> relativ = "pfad/zur/datei.py"
>>> urlparse.urljoin(base, relativ)
'http://www.test.de/pfad/zur/datei.py'
>>> base = "http://www.test.de/hallo/welt.py"
>>> relativ = "du.py"
>>> urlparse.urljoin(base, relativ)
'http://www.test.de/hallo/du.py'
```

Beachten Sie, dass urljoin die beiden übergebenen Pfade nicht einfach aneinanderhängt, sondern, wie im Beispiel zu sehen ist, Dateinamen am Ende der Basis-URL abschneidet.

urlparse.urldefrag(url)

Die Funktion urldefrag spaltet den Anker einer URL, sofern vorhanden, von der URL selbst ab. Die Funktion gibt ein Tupel zurück, dessen erstes Element die URL abzüglich des Ankers ist. Der Anker selbst ist als zweites Element im Tupel enthalten.

```
>>> urlparse.urldefrag("http://www.test.de#frag")
('http://www.test.de', 'frag')
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

20.4 FTP – ftplib

Das Modul `ftplib` ermöglicht es einer Anwendung, sich mit einem FTP-Server zu verbinden und Operationen auf diesem durchzuführen. *FTP* steht für *File Transfer Protocol* und bezeichnet ein Netzwerkprotokoll, das für Dateiübertragungen in TCP/IP-Netzwerken entwickelt wurde. Gerade im Internet ist FTP sehr verbreitet. So geschehen beispielsweise Dateiübertragungen auf einen Webserver üblicherweise via FTP.

Das Protokoll FTP ist sehr einfach aufgebaut und besteht aus einer umfangreichen Anzahl von Befehlen, die auch von Menschen gelesen werden können. Im Prinzip könnte man also auch direkt mit dem FTP-Server kommunizieren, ohne eine abstrahierende Bibliothek zwischenzuschalten. Die folgende Tabelle listet die wichtigsten FTP-Befehle auf und erläutert kurz ihre Bedeutung.

Sie werden sehen, dass sich das Modul `ftplib` sehr stark an diese Befehle anlehnt und man deshalb gut beraten ist, sich zumindest einen Überblick über die FTP-Befehle zu verschaffen.

Befehl	Beschreibung
`OPEN`	Baut eine Verbindung zu einem FTP-Server auf.
`USER`	Überträgt einen Benutzernamen zum Login an den FTP-Server.
`PASS`	Überträgt ein Passwort zum Login an den FTP-Server.
`CWD`	Ändert das aktuelle Arbeitsverzeichnis auf dem FTP-Server. (`CWD` steht für »*change working directory*«.)
`PWD`	Gibt das aktuelle Arbeitsverzeichnis auf dem FTP-Server zurück. (`PWD` steht für »*print working directory*«.)
`DELE`	Löscht eine Datei auf dem FTP-Server. (`DELE` steht für »*delete*«.)
`LIST` `LS`	Überträgt eine Liste aller im Arbeitsverzeichnis enthaltenen Dateien und Ordner. Die Liste wird über den Datenkanal übermittelt.
`MKD`	Erstellt ein Verzeichnis auf dem FTP-Server. (`MKD` steht für »*make directory*«.)
`RMD`	Löscht ein Verzeichnis auf dem FTP-Server. (`RMD` steht für »*remove directory*«.)
`RETR`	Überträgt eine Datei vom FTP-Server. (`RETR` steht für »*retrieve*«.)
`STOR`	Überträgt eine Datei vom Client an den FTP-Server. (`STOR` steht für »*store*«.)
`QUIT`	Beendet die Verbindung zwischen Server und Client.

Tabelle 20.5 FTP-Befehle

Die Kommunikation mit einem FTP-Server läuft auf zwei Kanälen ab: auf dem Steuerkanal zum Senden von Befehlen an den Server und auf dem Datenkanal zum Empfangen von Daten. Diese Trennung von Kommando- und Übertragungsebene ermöglicht es, dass auch während einer laufenden Datenübertragung Befehle, beispielsweise zum Abbruch der Übertragung, an den Server gesendet werden können. Grundsätzlich kann eine Datenübertragung in zwei Modi ablaufen. Im sogenannten *aktiven Modus* fordert der Client eine Datei an und öffnet gleichzeitig einen Port, über den dann die Übertragung der Datei ablaufen soll. Dem gegenüber steht der *passive Modus*, bei dem der Client den Server instruiert, einen Port zu öffnen, um die Datenübertragung durchzuführen. Das hat den Vorteil, dass auch Datenübertragungen mit Clients stattfinden können, die für den Server nicht direkt adressierbar sind, weil sie beispielsweise hinter einem Router oder einer Firewall stehen.

So viel zu den theoretischen Grundlagen. Ab jetzt werden wir behandeln, wie das Modul `ftplib` zur Kommunikation mit einem FTP-Server verwendet werden kann. Das Modul `ftplib` stellt die Klasse `FTP` zur Verfügung, die es einer Anwendung ermöglicht, sich mit einem FTP-Server zu verbinden und die dort unterstützten Operationen auszuführen. Mit diesem Modul kann man also einen vollwertigen FTP-Client implementieren.

Bereits beim Instanziieren der Klasse `FTP` kann eine Verbindung mit einem FTP-Server hergestellt werden. Dazu muss dem Konstruktor mindestens die Adresse des FTP-Servers als String übergeben werden. Der Konstruktor der Klasse `FTP` hat folgende Schnittstelle:

FTP([host[, user[, passwd[, acct]]]])

Der Konstruktor erzeugt eine Instanz der Klasse `FTP`, die mit dem FTP-Server *host* verbunden ist. Bei der Anmeldung an diesem Server werden der Benutzername *user*, das Passwort *passwd* und,

sofern notwendig, der Account *acct* verwendet.

Die Klasse FTP

Im Folgenden sollen die wichtigsten Methoden einer FTP-Instanz erläutert werden. Um die folgenden Beispiele ausführen zu können, müssen Sie sowohl das Modul `ftplib` importieren als auch eine FTP-Instanz `ftp` erzeugen, die mit einem FTP-Server Ihrer Wahl verbunden ist:

```
>>> import ftplib
>>> ftp = ftplib.FTP("ftp://ftp.test.de")
```

Im Folgenden sei `f` eine Instanz der Klasse `ftplib.FTP`.

f.connect(host[, port])

Verbindet zu dem FTP-Server *host* unter Verwendung des Ports *port*. Diese Methode sollte nicht aufgerufen werden, wenn bei der Instanziierung der Klasse FTP bereits die Adresse des FTP-Servers übergeben wurde.

f.getwelcome()

Gibt die Willkommensnachricht des verbundenen FTP-Servers als String zurück.

```
>>> ftp.getwelcome()
'220 Welcome to xyz FTP server. Please login...'
```

f.login([user[, passwd[, acct]]])

Loggt sich auf dem verbundenen FTP-Server ein. Beachten Sie, dass die Methode `connect` unbedingt aufgerufen werden muss, bevor ein Login durchgeführt werden kann.

Die Parameter haben dieselbe Bedeutung wie die des Konstruktors der Klasse FTP.

```
>>> ftp.login("Benutzername", "Passwort")
'230 User Benutzername logged in'
```

Wenn die Methode `login` aufgerufen wird, obwohl der Client bereits eingeloggt ist, wird eine `ftplib.error_perm`-Exception geworfen.

f.abort()

Unterbricht einen laufenden Dateitransfer. Beachten Sie, dass eine solche Unterbrechung je nach Server nicht zu jedem Zeitpunkt durchgeführt werden kann.

f.sendcmd(command)

Schickt den Kommandostring *command* an den verbundenen FTP-Server und gibt dessen Antwort ebenfalls als String zurück.

```
>>> ftp.sendcmd("PWD")
'257 "/" is the current directory'
```

f.retrbinary(command, callback[, maxblocksize])

Leitet einen Datentransfer im Binärmodus ein. Dazu muss als erster Parameter ein entsprechendes FTP-Kommando übergeben werden, aufgrund dessen der Server einen Datentransfer über den Datenkanal startet. Für einen simplen Dateitransfer dient das Kommando »RETR *dateiname*«.

An zweiter Stelle muss ein Funktionsobjekt übergeben werden. Die dahinter stehende Funktion muss exakt einen Parameter akzeptieren. Nach jedem erfolgreich übermittelten Block wird die Funktion *callback* aufgerufen. Die übertragenen Binärdaten werden dabei als Parameter in Form eines Strings übergeben.

Über den Parameter *maxblocksize* kann die maximale Größe der Blöcke angegeben werden, in die die Datei zum Herunterladen aufgeteilt wird.

```
bild = ""
def f(data):
    global bild
    bild += data
ftp.retrbinary("RETR bild.jpg", f)
```

Das Beispielprogramm lädt die Bilddatei *bild.jpg* aus dem aktuellen Arbeitsverzeichnis des FTP-Servers herunter und speichert die Binärdaten im String `bild`.

Alternativ könnte auch ein `LIST`-Kommando abgesetzt werden, aufgrund dessen der Verzeichnisinhalt ebenfalls über den Datenkanal geschickt wird:

```
>>> def f(data):
...     print data
...
>>> ftp.retrbinary("LIST", f)
drwxr-xr-x  11 user   group        360 Sep  5 02:45 .
drwxr-xr-x  11 user   group        360 Sep  5 02:45 ..
drwxr-xr-x   4 user   group         96 Jun 20  2006 ordner1
[…]
```

f.retrlines(command[, callback])

Leitet einen Dateitransfer im ASCII-Modus ein. Dazu muss als erster Parameter ein entsprechendes FTP-Kommando übergeben werden. Für einen simplen Dateitransfer wäre dies »`RETR` *dateiname*«. Möglich wäre aber beispielsweise auch, den Inhalt des Arbeitsverzeichnisses durch ein `LIST`-Kommando zu übertragen.

Eine Dateiübertragung im ASCII-Modus geschieht zeilenweise. Das heißt, die Callback-Funktion *callback* wird nach jeder vollständig übertragenen Zeile aufgerufen. Sie bekommt dabei die gelesene Zeile als Parameter übergeben. Beachten Sie, dass das abschließende Newline-Zeichen nicht mit übergeben wird.

Wenn keine Callback-Funktion angegeben wurde, werden die übertragenen Daten ausgegeben.

```
text = ""
def f(data):
    global text
    text = "".join((text, data, "\n"))
ftp.retrlines("RETR text.txt", f)
```

Dieses Beispielprogramm lädt die Textdatei *text.txt* zeilenweise herunter und fügt die heruntergeladenen Zeilen im String `text` wieder zu einem Gesamttext zusammen.

f.set_pasv(boolean)

Wenn für *boolean* `False` übergeben wird, wird die `FTP`-Instanz in den sogenannten aktiven Zustand versetzt. Ein Wert von `True` versetzt sie zurück in den passiven Zustand. Beachten Sie, dass der Client im aktiven Zustand für den Server erreichbar sein muss, sich also nicht hinter einer Firewall oder einem Router befinden darf.

f.storbinary(command, file[, blocksize])

Leitet einen Datei-Upload ein. Dabei muss als erster Parameter

ein entsprechender FTP-Befehl übergeben werden. Für einen simplen Datei-Upload lautet dieser Befehl »STOR *datei*«, wobei *datei* der Zielname der Datei auf dem FTP-Server ist. Als zweiter Parameter muss ein geöffnetes Dateiobjekt übergeben werden, dessen Inhalt hochgeladen werden soll.

Optional kann in Form des dritten Parameters, *blocksize*, die maximale Größe der Datenblöcke angegeben werden, in denen die Datei hochgeladen wird.

Das folgende Beispielprogramm führt einen binären Datei-Upload durch:

```
f = open("bla.txt", "r")
ftp.storbinary("STOR hallo.txt", f)
f.close()
```

Beachten Sie, dass die Datei im lokalen Arbeitsverzeichnis *bla.txt* heißt, auf den Server jedoch unter dem Namen *hallo.txt* hochgeladen wird.

f.storlines(command, file)

Verhält sich ähnlich wie `storbinary` mit dem Unterschied, dass die Datei im ASCII-Modus zeilenweise hochgeladen wird. Die Parameter *command* und *file* können wie bei `storbinary` verwendet werden.

f.nlst([dirname])

Gibt eine Liste mit dem Inhalt des aktuellen Arbeitsverzeichnisses auf dem FTP-Server zurück. Über den optionalen Parameter *dirname* kann ein Unterverzeichnis angegeben werden, dessen Inhalt aufgelistet werden soll:

```
>>> ftp.nlst()
['.', '..', 'ordner1', 'ordner2', 'hallo.txt']
>>> ftp.nlst("ordner1")
[' ordner1/.', ' ordner1/..', ' ordner1/test.py']
```

Neben den im Arbeitsverzeichnis existierenden Ordnern und Dateien sind die Verweise auf das aktuelle Verzeichnis (`.`) und das übergeordnete Verzeichnis (`..`) in der Liste enthalten.

f.dir([dirname[, callback]])

Gibt den Inhalt des aktuellen Arbeitsverzeichnisses auf dem FTP-Server in Form einer Aufzählung auf dem Bildschirm aus, wie sie vom FTP-Befehl `LIST` erzeugt würde. Optional kann über den Parameter *dirname* ein Unterverzeichnis angegeben werden, dessen Inhalt ausgegeben werden soll.

Außerdem kann eine Callback-Funktion übergeben werden, die anstelle einer Bildschirmausgabe aufgerufen wird. Die Callback-Funktion *callback* muss über die gleiche Schnittstelle verfügen wie die, die bei `retrlines` angegeben werden kann.

```
>>> ftp.dir()
drwxr-xr-x  11 user   group       360 Sep  5 02:45 .
drwxr-xr-x  11 user   group       360 Sep  5 02:45 ..
[…]
```

f.rename(fromname, toname)

Benennt die Datei *fromname* auf dem FTP-Server nach *toname* um.

```
>>> ftp.rename("ordner", "ordner2")
'250 Rename successful'
```

Es erübrigt sich zu sagen, dass der Ordner, der umbenannt werden soll, existieren muss. Ist dies nicht der Fall, wird eine `ftplib.error_perm`-Exception geworfen.

f.delete(filename)

Löscht die Datei *filename* auf dem FTP-Server.

```
>>> ftp.delete("hallo.txt")
'250 DELE command successful'
```

Auch hier wird eine `ftplib.error_perm`-Exception geworfen, wenn die zu löschende Datei nicht existiert.

f.cwd(pathname)

Ändert das aktuelle Arbeitsverzeichnis auf dem FTP-Server in *pathname*. Sollte das Verzeichnis *pathname* nicht existieren, wird eine `ftplib.error_perm`-Exception geworfen.

```
>>> ftp.cwd("ordner")
'250 CWD command successful'
```

f.mkd(pathname)

Erzeugt das Verzeichnis *pathname* auf dem FTP-Server. Die Methode `mkd` gibt den Pfad zu dem neu erstellten Verzeichnis zurück.

```
>>> ftp.mkd("ordner")
'/ordner'
```

Beachten Sie, dass kein Verzeichnis dieses Namens bereits existieren darf. In einem solchen Fall würde eine `ftplib.error_perm`-Exception geworfen.

f.pwd()

Gibt den Pfad des aktuellen Arbeitsverzeichnisses auf dem FTP-Server zurück.

```
>>> ftp.pwd()
'/'
```

f.rmd(dirname)

Löscht das Verzeichnis *dirname* auf dem FTP-Server. Beachten Sie, dass das Verzeichnis *dirname* vorhanden und leer sein muss, damit es erfolgreich gelöscht werden kann. Im Fehlerfall wird eine `ftplib.error_perm`-Exception geworfen.

```
>>> ftp.rmd("ordner")
'250 RMD command successful'
```

f.size(filename)

Ermittelt die Dateigröße der Datei *filename* auf dem FTP-Server. Wenn sie sich ermitteln ließ, wird die Dateigröße als ganze Zahl zurückgegeben, andernfalls ist der Rückgabewert `None`. Beachten Sie, dass das dieser Methode zugrunde liegende FTP-Kommando `SIZE` nicht standardisiert ist und somit nicht von allen FTP-Servern unterstützt wird.

f.quit()

Beendet die Verbindung zum FTP-Server, indem ihm ein `QUIT`-

Befehl gesendet wird. Dies ist die saubere Art, die Verbindung zu kappen, könnte aber eine Exception verursachen, wenn der Server mit einem Fehlercode antworten sollte.

Der Aufruf von `quit` erübrigt einen weiteren Aufruf von `close`.

f.close()

Beendet die Verbindung, ohne den FTP-Server darüber in Kenntnis zu setzen. Beachten Sie, dass dieselbe FTP-Instanz nach Aufruf dieser Funktion nicht wieder per `login` mit einem FTP-Server verbunden werden kann. Dazu sollte eine neue Instanz erzeugt werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

20.6 Telnet – telnetlib

Das Modul `telnetlib` ermöglicht die Verwendung des sogenannten *Telnet*-Netzwerkprotokolls (*Teletype Network*). Telnet wurde als möglichst einfaches bidirektionales Netzwerkprotokoll konzipiert. Häufig wird Telnet dazu verwendet, einen kommandozeilenbasierenden Zugriff auf einen entfernten Rechner zu ermöglichen. Da das Telnet-Protokoll aber keine Möglichkeit zur Verschlüsselung der übertragenen Daten bietet, wurde es nach und nach von anderen, in diesem Bereich stärkeren Protokollen wie beispielsweise SSH verdrängt.

Im Modul `telnetlib` ist im Wesentlichen die Klasse `Telnet` enthalten, über die die weitere Kommunikation mit dem entfernten Rechner abläuft. Der Konstruktor der Klasse `Telnet` hat die folgende Schnittstelle:

telnetlib.Telnet([host[, port]])

Erzeugt eine Instanz der Klasse `Telnet`. Optional können bereits hier der Hostname und der Port des Rechners übergeben werden, zu dem eine Verbindung hergestellt werden soll. Wenn keiner der Parameter angegeben wird, muss die erzeugte `Telnet`-Instanz explizit durch Aufruf der Methode open verbunden werden. Die Angabe einer Portnummer ist nur dann notwendig, wenn die Verbindung nicht über den Standardport 23 ablaufen soll.

Die Klasse Telnet

Nachdem sie erzeugt und mit dem Zielrechner verbunden wurde, kann eine `Telnet`-Instanz zur Kommunikation mit dem verbundenen Rechner verwendet werden. Dazu enthält sie eine Reihe Methoden, von denen die wichtigsten im Folgenden erläutert werden sollen. Dabei sei *t* eine Instanz der Klasse `telnetlib.Telnet`.

Generell gilt, dass, wenn während einer Lese- oder Schreiboperation die Verbindung geschlossen wird, eine `IOError`-Exception aus der gerade aktiven Methode heraus geworfen wird.

t.read_until(expected[, timeout])

Liest ankommende Daten, bis der String *expected* empfangen wurde. Alternativ kann ein Timeout in Sekunden als zweiter Parameter angegeben werden, nach dessen Ablaufen der Lesevorgang abgebrochen wird. Die gelesenen Daten werden als String zurückgegeben.

t.read_all()

Liest alle ankommenden Daten, bis die Verbindung geschlossen wird. Beachten Sie, dass diese Methode das Programm auch so lange blockiert. Die gelesenen Daten werden zurückgegeben.

t.open(host[, port])

Verbindet die `Telnet`-Instanz zum entfernten Rechner *host* unter Verwendung des Ports *port*. Diese Funktion sollte nur aufgerufen werden, wenn die Verbindungsdaten nicht bereits dem Konstruktor der Klasse `Telnet` übergeben wurden.

t.close()

Schließt die Telnet-Verbindung zum entfernten Rechner.

t.write(buffer)

Sendet den String *buffer* zum Verbindungspartner. Diese Funktion kann das Programm blockieren, wenn die Daten nicht sofort geschrieben werden können.

Beispiel

Im folgenden Beispielprogramm soll das Modul `telnetlib` dazu verwendet werden, zu einem POP3-Server zu verbinden. Dabei möchten wir auf die abstrahierte Schnittstelle des Moduls `poplib` verzichten und dem Server direkt POP3-Kommandos senden. Da das POP3-Protokoll jedoch relativ simpel ist und auf lesbaren Kommandos basiert, stellt dies kein großes Problem dar.

Das Ziel des Programms ist es, die Ausgabe des POP3-Kommandos `LIST` zu erhalten, das die Indizes aller im Posteingang liegenden Mails auflistet.

Im Programm soll die Telnet-Kommunikation möglichst komfortabel über eine auf POP3 zugeschnittene Klasse ablaufen:

```
import telnetlib

class POP3Telnet(object):

    def __init__(self, host, port):
        self.tel = telnetlib.Telnet(host, port)
        self.lese_daten()

    def close(self):
        self.tel.close()

    def lese_daten(self):
        return self.tel.read_until(".\r\n", 20.0)

    def kommando(self, kom):
        self.tel.write("%s\r\n" % kom)
        return self.lese_daten()
```

Dem Konstruktor der Klasse POP3Telnet werden Hostname und Port des POP3-Servers übergeben. Intern wird dann eine Instanz der Klasse Telnet erzeugt und mit diesem Server verbunden. Durch Aufruf der Methode lese_daten wird die Begrüßungsnachricht des Servers ausgelesen und verworfen, da sie nicht weiter von Interesse ist, aber bei späteren Lesevorgängen stören würde.

Wichtig sind die Methoden lese_daten und kommando. Die Methode lese_daten liest genau einen Antwortstring des POP3-Servers ein. Eine solche Antwort wird stets durch den String ".\r\n" beendet. Der gelesene String wird zurückgegeben. Damit dieser Lesevorgang das Programm bei einem unerreichbaren Server nicht auf unbestimmte Zeit blockiert, wurde ein Timeout von 20 Sekunden festgelegt.

Die zweite wichtige Methode ist kommando. Sie erlaubt es, einen POP3-Befehl an den Server zu senden. Dieser Befehl wird inklusive eines abschließenden "\r\n" in die Telnet-Instanz geschrieben und von dieser an den verbundenen Rechner weitergeleitet. Schlussendlich wird die Antwort des Servers eingelesen und zurückgegeben.

Doch die Klasse ist nur der erste Teil des Beispielprogramms. Im nun folgenden zweiten Teil wird die Klasse POP3Telnet zur Kommunikation mit einem POP3-Server eingesetzt. Dazu werden zunächst die Zugangsdaten für den POP3-Server festgelegt:

```
host = "pop.test.de"
port = 110
user = "benutzername"
passwd = "passwort"
```

Jetzt wird eine Instanz der Klasse POP3Telnet erzeugt, die mit dem angegebenen POP3-Server verbunden ist. Dann wird die Anmeldeprozedur durch Senden der Kommandos USER und PASS durchgeführt.

```
pop = POP3Telnet(host, port)
pop.kommando("USER %s" % user)
pop.kommando("PASS %s" % passwd)
```

An dieser Stelle sind wir, wenn bei der Anmeldung alles gut gelaufen ist, dazu in der Lage, mit beliebigen POP3-Kommandos auf den Posteingang zuzugreifen. Dann schicken wir das eingangs erwähnte LIST-Kommando. Das LIST-Kommando des POP3-Protokolls liefert eine Liste aller im Posteingang enthaltenen E–Mails. Jeder Eintrag besteht dabei aus dem ganzzahligen Index der jeweiligen E-Mail und ihrer Größe in Byte.

Beachten Sie, dass der Server auf LIST zwei Antwortstrings sendet, von denen uns nur der zweite interessiert, da dieser die Daten über vorhandene E-Mails enthält. Aus diesem Grund muss nach dem Aufruf der Methode kommando noch einmal der zweite Antwortstring eingelesen werden. Der zurückgegebene String wird ausgegeben. Im Code sieht das folgendermaßen aus:

```
pop.kommando("LIST")
print pop.lese_daten()
pop.kommando("QUIT")
pop.close()
```

Zum Schluss schicken wir das Kommando QUIT an den Server und schließen die Telnet-Verbindung. Die Ausgabe des Beispielprogramms könnte folgendermaßen aussehen:

```
1 623
2 614
3 1387
.
```

In diesem Fall befinden sich drei E-Mails mit den Größen 623, 614 und 1387 Byte im Posteingang.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

20.7 XML-RPC

Der Standard *XML-RPC* (*Extensible Markup Language Remote Procedure Call*) ermöglicht den entfernten Funktions- und Methodenaufruf über eine Netzwerkschnittstelle. Dabei können entfernte Funktionen aus Sicht des Programmierers aufgerufen werden, als gehörten sie zum lokalen Programm. Das Übertragen der Funktionsaufrufe und insbesondere der Parameter und des Rückgabewertes wird vollständig von der XML-RPC-Bibliothek übernommen, sodass der Programmierer die Funktionen tatsächlich nur aufzurufen braucht.

Neben XML-RPC existieren weitere mehr oder weniger standardisierte Verfahren zum entfernten Funktionsaufruf. Da aber XML-RPC auf zwei bereits bestehenden Standards, nämlich XML und HTTP, basiert und keine völlig neuen binären Protokolle

einführt, ist es vergleichsweise einfach umzusetzen und daher in vielen Programmiersprachen verfügbar.

Da XML-RPC unabhängig von einer bestimmten Programmiersprache entwickelt wurde, ist es durchaus möglich, Client und Server in zwei verschiedenen Sprachen zu schreiben. Aus diesem Grund musste sich bei der XML-RPC-Spezifikation auf einen kleinsten gemeinsamen Nenner geeinigt werden, was die Eigenheiten bestimmter Programmiersprachen und besonders die verfügbaren Datentypen anbelangt. Sie werden feststellen, dass Sie bei einer Funktion mit einer XML-RPC-fähigen Schnittstelle bestimmte Einschränkungen zu beachten haben.

Im Folgenden werden wir uns zunächst damit beschäftigen, wie durch einen XML-RPC-Server bestimmte Funktionen nach außen hin aufrufbar werden. Danach widmen wir uns der Client-Seite und klären, wie solche Funktionen dann aufgerufen werden.

top

20.7.1 Der Server ▼▲

Zum Aufsetzen eines XML-RPC-Servers wird das Modul `SimpleXMLRPCServer` benötigt. Dieses Modul enthält im Wesentlichen die Klasse `SimpleXMLRPCServer`, die einen entsprechenden Server aufsetzt und Methoden zur Verwaltung desselben bereitstellt. Der Konstruktor der Klasse hat folgende Schnittstelle:

SimleXMLRPCServer(addr[, requestHandler[, logRequests[, allow_none[, encoding]]]])

Der einzige zwingend erforderliche Parameter ist *addr* und spezifiziert die IP-Adresse und den Port, an die der Server gebunden wird. Die Angaben müssen in einem Tupel der Form `(ip, port)` übergeben werden, wobei die IP-Adresse ein String und die Portnummer eine ganze Zahl zwischen `0` und `65535` ist. Technisch wird der Parameter an die zugrunde liegende `Socket`-Instanz weitergereicht. Der Server kann sich nur an Adressen binden, die ihm auch zugeteilt sind. Wenn für `ip` im Tupel ein leerer String angegeben wird, wird der Server an alle dem PC zugeteilten Adressen gebunden, beispielsweise auch an `127.0.0.1` oder `localhost`.

Über den optionalen Parameter *requestHandler* kann eine Art Backend festgelegt werden. In den meisten Fällen reicht die Voreinstellung des Standard-Handlers `SimpleXMLRPCRequestHandler`. Die Aufgabe dieser Klasse ist es, eingehende Daten in einen Funktionsaufruf zurückzuverwandeln.

Über den Parameter *logRequest* kann festgelegt werden, ob einkommende Funktionsaufrufe protokolliert werden sollen oder nicht. Der Parameter ist mit `True` vorbelegt.

Der vierte Parameter, *allow_none*, ermöglicht es, sofern hier `True` übergeben wird, `None` in XML-RPC-Funktionen zu verwenden. Normalerweise verursacht die Verwendung von `None` eine Exception, da kein solcher Datentyp im XML-RPC-Standard vorgesehen ist. Da dies aber eine übliche Erweiterung des Standards darstellt, wird *allow_none* von vielen XML-RPC-Implementationen unterstützt.

Als letzter optionaler Parameter kann ein Encoding zur Datenübertragung festgelegt werden. Standardmäßig wird hier UTF-8 verwendet.

Für gewöhnlich reicht zur Instanziierung des XML-RPC-Servers folgender Aufruf des Konstruktors:

```
>>> from SimpleXMLRPCServer import SimpleXMLRPCServer as
Server
>>> srv = Server(("", 1337))
```

Die Klasse SimpleXMLRPCServer

Nachdem eine Instanz der Klasse SimpleXMLRPCServer erzeugt wurde, verfügt diese über Methoden, um beispielsweise Funktionen zum entfernten Aufruf zu registrieren. Die wichtigsten Methoden einer SimpleXMLRPCServer-Instanz sollen im Folgenden erläutert werden.

Dazu ist noch zu sagen, dass die Klasse SimpleXMLRPCServer von der Klasse SocketServer des gleichnamigen Moduls erbt. Das bedeutet insbesondere, dass ein XML-RPC-Server ebenfalls über die Methode serve_forever dazu instruiert wird, eine unbestimmte Anzahl von Anfragen zu beantworten. Näheres zum Modul SocketServer erfahren Sie in Abschnitt 20.1.8.

Im Folgenden sei *s* eine Instanz der Klasse SimpleXMLRPCServer.

s.register_function(function[, name])

Registriert das Funktionsobjekt *function* für einen RPC-Aufruf. Das bedeutet, dass ein zu diesem Server verbundener XML-RPC-Client die Funktion *function* über das Netzwerk aufrufen kann.

Optional kann der Funktion ein anderer Name gegeben werden, über den sie für den Client zu erreichen ist. Wenn ein solcher Name angegeben wird, kann dieser aus beliebigen Unicode-Zeichen bestehen, auch solchen, die in einem Python-Bezeichner eigentlich nicht erlaubt sind, beispielsweise ein Umlaut oder ein Punkt.

s.register_instance(instance[, allow_dotted_names])

Registriert die Instanz *instance* für den entfernten Zugriff. Wenn der verbundene Client eine Methode dieser Instanz aufruft, wird der Aufruf durch die spezielle Methode _dispatch geleitet. Die Methode muss folgendermaßen definiert sein:

```
def _dispatch(self, method, params):
    pass
```

Bei jedem entfernten Aufruf einer Methode dieser Instanz wird _dispatch aufgerufen. Der Parameter method enthält den Namen der aufgerufenen Methode, und params enthält die dabei angegebenen Parameter. Wenn die Parameter entpackt und an eine Methode weitergereicht werden sollen, kann folgender Methodenaufruf verwendet werden:

```
self.meth(*params)
```

Eine konkrete Implementierung der Methode _dispatch, die die tatsächliche Methode der registrierten Instanz mit dem Namen method aufruft und die Parameter übergibt, sähe folgendermaßen aus:

```
def _dispatch(self, method, params):
    try:
        return getattr(self, method)(*params)
    except (AttributeError, TypeError):
        return None
```

Diese Funktion gibt sowohl dann None zurück, wenn keine Methode mit dem Namen method vorhanden ist, als auch dann, wenn die Methode mit der falschen Zahl oder einem unpassenden Parameter aufgerufen wird.

Wenn für den optionalen Parameter *allow_dotted_names* True übergeben wird, sind Punkte im entfernten Methodenaufruf möglich. Dadurch können auch Methoden von Attributen über das Netzwerk aufgerufen werden. Beachten Sie unbedingt, dass es

damit einem Angreifer möglich gemacht wird, auf die globalen Variablen des Programms zuzugreifen und möglicherweise schädlichen Code auszuführen. Sie sollten *allow_dotted_names* nur innerhalb eines lokalen, vertrauenswürdigen Netzes auf `True` setzen, da sonst eine massive Sicherheitslücke geöffnet würde.

s.register_introspection_functions()

Registriert die Funktionen `system.listMethods`, `system.methodHelp` und `system.methodSignature` für den entfernten Zugriff. Diese Funktionen ermöglichen es einem verbundenen Client, eine Liste aller verfügbaren Funktionen und Informationen zu einzelnen dieser Funktionen zu bekommen.

Näheres zur Verwendung der Funktionen `system.listMethods`, `system.method Help` und `system.methodSignature` erfahren Sie in Abschnitt 20.7.2, »Der Client«.

s.register_multicall_functions()

Registriert die Funktion `system.multicall` für den entfernten Zugriff. Durch Aufruf der Funktion `system.multicall` kann der Client mehrere Methodenaufrufe bündeln, um so Traffic zu sparen. Auch die Rückgabewerte der Methodenaufrufe werden gebündelt zurückgegeben.

Näheres zur Verwendung der Funktion `system.multicall` erfahren Sie in Abschnitt 20.7.2, »Der Client«.

Beispiel

Nachdem die wichtigsten Funktionen der Klasse `SimpleXMLRPCServer` erläutert wurden, soll an dieser Stelle ein kleines Beispielprogramm entwickelt werden. Bei dem Programm handelt es sich um einen XML-RPC-Server, der zwei mathematische Funktionen (genauer gesagt die Berechnungsfunktionen für die Fakultät und das Quadrat einer ganzen Zahl) bereitstellt, die ein verbundener Client aufrufen kann.

```
from SimpleXMLRPCServer import SimpleXMLRPCServer as Server

def fak(n):
    """ Berechnet die Fakultaet der ganzen Zahl n. """
    erg = 1
    for i in xrange(2, n+1):
        erg *= i
    return erg

def quad(n):
    """ Berechnet das Quadrat der Zahl n. """
    return n*n

srv = Server(("", 1337))
srv.register_function(fak)
srv.register_function(quad)
srv.serve_forever()
```

Zunächst werden die beiden Berechnungsfunktionen `fak` und `quad` für die Fakultät bzw. das Quadrat einer Zahl erstellt. Danach wird ein auf Port 1337 horchender XML-RPC-Server erzeugt und werden die soeben erstellten Funktionen registriert. Schlussendlich wird der Server durch Aufruf der Methode `serve_forever` gestartet und ist nun bereit, eingehende Verbindungsanfragen und Methodenaufrufe entgegenzunehmen und zu bearbeiten.

Der hier vorgestellte Server ist natürlich nur eine Hälfte des Beispielprogramms. Im nächsten Abschnitt werden wir besprechen, wie ein XML-RPC-Client auszusehen hat, und schließlich werden wir am Ende des folgenden Abschnitts einen Client entwickeln, der mit diesem Server kommunizieren kann.

top

20.7.2 Der Client ▼▲

Um einen XML-RPC-Client zu schreiben, wird das Modul `xmlrpclib` der Standardbibliothek verwendet. In diesem Modul ist vor allem die Klasse `ServerProxy` enthalten, über die die Kommunikation mit einem XML-RPC-Server abläuft. Hier sehen Sie zunächst die Schnittstelle des Konstruktors der Klasse `ServerProxy`:

ServerProxy(uri[, transport[, encoding[, verbose[, allow_none [, use_datetime]]]]])

Erzeugt eine Instanz der Klasse `ServerProxy`, die mit dem XML-RPC-Server verbunden ist, den die URI *uri* beschreibt.

An zweiter Stelle kann wie bei der Klasse `SimpleXMLRPCServer` ein Art Backend festgelegt werden. Die voreingestellten Klassen `Transport` für das HTTP- Protokoll und `SafeTransport` für das HTTPS-Protokoll dürften in den meisten Anwendungsfällen genügen.

Wenn für den vierten Parameter, *verbose*, `True` übergeben wird, gibt die `Server Proxy`-Instanz alle ausgehenden und ankommenden XML-Pakete auf dem Bildschirm aus. Dies kann besonders zur Fehlersuche hilfreich sein.

Wenn für den letzten Parameter *use_datetime* `True` übergeben wird, wird zur Repräsentation von Datums- und Zeitangaben statt der `xmlrpclib`-internen Klasse `DateTime` die Klasse `datetime` des gleichnamigen Moduls verwendet, die einen wesentlich größeren Funktionsumfang besitzt.

Auf die Parameter *encoding* und *allow_none* muss hier nicht weiter eingegangen werden, da sie dieselbe Bedeutung haben wie die gleichnamigen Parameter des Konstruktors der Klasse `SimpleXMLRPCServer`, der zu Beginn des letzten Kapitels besprochen wurde.

Die Klasse ServerProxy

Nach der Instanziierung der Klasse `ServerProxy` ist diese mit einem XML-RPC-Server verbunden. Das bedeutet insbesondere, dass alle bei diesem Server registrierten Funktionen wie Methoden der `ServerProxy`-Instanz aufgerufen und verwendet werden können. Es ist also keine weitere Sonderbehandlung nötig.

Zusätzlich beinhaltet eine `ServerProxy`-Instanz drei Methoden, die weitere Informationen über die verfügbaren entfernten Funktionen bereitstellen. Beachten Sie jedoch, dass der Server diese Methoden explizit zulassen muss. Dies geschieht durch Aufruf der Methoden `register_introspection_functions` der `SimpleXMLRPCServer`-Instanz.

Im Folgenden sei *s* eine Instanz der Klasse `ServerProxy`.

s.system.listMethods()

Gibt die Namen aller beim XML-RPC-Server registrierten entfernten Funktionen in Form einer Liste von Strings zurück. Die Liste enthält keine dieser Systemmethoden.

s.system.methodSignature(name)

Diese Methode gibt Auskunft über die Schnittstelle der registrierten Funktion mit dem Funktionsnamen *name*. Die Schnittstellenbeschreibung ist ein String im Format:

```
"string, int, int, int"
```

wobei die erste Angabe dem Datentyp des Rückgabewertes und alle weiteren den Datentypen der Funktionsparameter entsprechen. Der XML-RPC-Standard sieht vor, dass zwei verschiedene Funktionen den gleichen Namen haben dürfen,

sofern sie anhand ihrer Schnittstelle unterscheidbar sind. Aus diesem Grund wird von der Methode `system.methodSignature` nicht ein einzelner String, sondern eine Liste von Strings zurückgegeben.

Beachten Sie, dass der Methode `system.methodSignature` nur eine tiefere Bedeutung zukommt, wenn der XML-RPC-Server in einer Sprache geschrieben wurde, bei der Funktionsparameter jeweils an einen Datentyp gebunden werden. Solche Sprachen sind beispielsweise C, C++, C# oder Java. Sollten Sie `system.method Signature` bei einem XML-RPC-Server aufrufen, der in Python geschrieben wurde, so wird schlicht `"signatures not supported"` zurückgegeben.

s.system.methodHelp(name)

Gibt den Docstring der entfernten Funktion *name* zurück, wenn ein solcher existiert. Wenn kein Docstring gefunden werden konnte, wird ein leerer String zurückgegeben.

Beispiel

Damit wäre die Verwendung einer `ServerProxy`-Instanz beschrieben und eigentlich denkbar einfach. Das folgende Beispiel implementiert einen zu dem XML-RPC-Server des letzten Kapitels passenden Client.

```
import xmlrpclib

cli = xmlrpclib.ServerProxy("http://ip:1337")
print cli.fak(5)
print cli.quad(5)
```

Sie sehen, dass das Verbinden zu einem XML-RPC-Server und das Ausführen einer entfernten Funktion nur wenige Codezeilen benötigt und damit fast so einfach ist, als befände sich die Funktion im Client-Programm selbst.

top

20.7.3 Multicall ▼▲

Im Modul `xmlrpc` ist eine Klasse namens `MultiCall` enthalten. Diese Klasse ermöglicht es, mehrere Funktionsaufrufe gebündelt an den Server zu schicken, und instruiert diesen, die Rückgabewerte ebenfalls gebündelt zurückzusenden. Auf diese Weise lässt sich bei häufigen Funktionsaufrufen die Netzlast minimieren.

Die Verwendung der `MultiCall`-Klasse ist denkbar einfach und soll an folgendem Beispiel verdeutlicht werden. Das Beispiel benötigt einen laufenden Server, der die Funktionen `fak` und `quad` für den entfernten Zugriff bereitstellt, also genau so einen, wie wir ihn in Abschnitt 20.7.1 vorgestellt haben. Zusätzlich muss der Server den Einsatz von Multicall durch Aufruf der Methode `register_multicall_functions` erlauben.

```
import xmlrpclib

cli = xmlrpclib.ServerProxy("http://ip:1337")

mc = xmlrpclib.MultiCall(cli)
for i in xrange(10):
    mc.fak(i)
    mc.quad(i)

for ergebnis in mc():
    print ergebnis
```

Zunächst wird wie gehabt eine Verbindung zu dem XML-RPC-Server hergestellt. Danach wird eine Instanz der Klasse `MultiCall` erzeugt und dem Konstruktor die zuvor erzeugte `ServerProxy`-Instanz übergeben.

Ab jetzt läuft die gebündelte Kommunikation mit dem Server über die `MultiCall`-Instanz. Dazu können die entfernten Funktionen `fak` und `quad` aufgerufen werden, als wären es lokale Methoden der `MultiCall`-Instanz. Beachten Sie aber, dass diese Methodenaufrufe keinen sofortigen entfernten Funktionsaufruf zur Folge haben und somit auch zu dieser Zeit keinen Wert zurückgeben.

Im Beispiel werden `fak` und `quad` jeweils zehnmal mit einer fortlaufenden ganzen Zahl aufgerufen.

Durch Aufruf der `MultiCall`-Instanz (`mc()`) werden alle gepufferten entfernten Funktionsaufrufe zusammen an den Server geschickt. Als Ergebnis wird ein Iterator zurückgegeben, der über alle Rückgabewerte in der Reihenfolge des jeweiligen Funktionsaufrufes iteriert. Im Beispielprogramm nutzen wir den Iterator dazu, die Ergebnisse mittels `print` auszugeben.

Gerade bei wenigen Rückgabewerten ist es sinnvoll, diese direkt zu referenzieren.

```
wert1, wert2, wert3 = mc()
```

Hier wird davon ausgegangen, dass zuvor drei entfernte Funktionsaufrufe durchgeführt wurden und dementsprechend auch drei Rückgabewerte vorliegen.

top

20.7.4 Einschränkungen ▲

Der XML-RPC-Standard ist nicht auf Python allein zugeschnitten, sondern es wurde bei der Ausarbeitung des Standards versucht, einen kleinsten gemeinsamen Nenner vieler Programmiersprachen zu finden, sodass beispielsweise Server und Client auch dann problemlos miteinander kommunizieren können, wenn sie in verschiedenen Programmiersprachen geschrieben wurden.

Aus diesem Grund bringt das Verwenden von XML-RPC einige Einschränkungen mit sich, was die komplexeren bzw. exotischeren Datentypen von Python betrifft. So gibt es im XML-RPC-Standard beispielsweise keine Repräsentation der Datentypen `complex`, `set` und `frozenset`. Auch `None` darf nur verwendet werden, wenn dies bei der Instanziierung der Server- bzw. Clientklasse explizit angegeben wurde. Das bedeutet natürlich nur, dass Instanzen dieser Datentypen nicht über die XML-RPC-Schnittstelle versendet werden dürfen. Programmintern können sie weiterhin verwendet werden. Sollten Sie versuchen, beispielsweise eine Instanz des Datentyps `complex` als Rückgabewert einer Funktion über die XML-RPC-Schnittstelle zu versenden, so wird eine `xmlrpclib.Fault`-Exception geworfen. Beachten Sie, dass es natürlich dennoch möglich ist, eine komplexe Zahl über eine XML-RPC-Schnittstelle zu schicken, indem Real- und Imaginärteil getrennt als jeweils ganze Zahl übermittelt werden.

Die folgende Tabelle listet alle im XML-RPC-Standard vorgesehenen Datentypen auf und beschreibt, wie sich diese in Python verwenden lassen.

XML-RPC	Python	Anmerkungen
Boolesche Werte	`bool`	Keine Anmerkungen
Ganze Zahlen	`int`	Keine Anmerkungen
Gleitkommazahlen	`float`	Keine Anmerkungen
Strings	`str`	Keine Anmerkungen
Arrays	`list`	In der Liste dürfen als Elemente nur XML-RPC-konforme Instanzen verwendet werden.
		Alle Schlüssel müssen Strings sein.

Strukturen	dict	Als Werte dürfen nur XML-RPC-konforme Instanzen verwendet werden.
Datum/Zeit	DateTime	Der spezielle Datentyp xmlrpclib.DateTime wird verwendet.
Binärdaten	Binary	Der spezielle Datentyp xmlrpclib.Binary wird verwendet.

Tabelle 20.10 Erlaubte Datentypen bei XML-RPC

Es ist möglich, Instanzen von selbst erstellten Klassen zu verwenden. In einem solchen Fall wird die Instanz in ein Dictionary, also eine Struktur, umgewandelt, in der die Namen der enthaltenen Attribute als Schlüssel eingetragen werden und die jeweils referenzierten Instanzen als Werte. Dies geschieht automatisch. Beachten Sie jedoch, dass das auf der Gegenseite ankommende Dictionary nicht automatisch wieder in eine Instanz der ursprünglichen Klasse umgewandelt wird.

Die letzten beiden Datentypen, die in der Tabelle aufgelistet sind, sind neu. Es handelt sich dabei um Datentypen, die im Modul xmlrpclib enthalten und speziell auf die Verwendung im Zusammenhang mit XML-RPC zugeschnitten sind. Die beiden erwähnten Datentypen DateTime und Binary sollen im Folgenden kurz erläutert werden.

Der Datentyp DateTime

Der Datentyp DateTime des Moduls xmlrpclib kann verwendet werden, um Datums- und Zeitangaben über eine XML-RPC-Schnittstelle zu versenden. Statt einer DateTime-Instanz kann, sofern der entsprechende Parameter bei der Instanziierung der ServerProxy-Instanz übergeben wurde, auch direkt eine Instanz der bekannten Datentypen datetime.date, datetime.time oder datetime.datetime verwendet werden.

Bei der Erzeugung einer Instanz des Datentyps DateTime kann entweder einer der Datentypen des Moduls datetime übergeben werden oder ein UNIX-Timestamp als ganze Zahl:

```
>>> import xmlrpclib
>>> import datetime
>>> xmlrpclib.DateTime(987654321)
<DateTime '20010419T06:25:21' at 80fda4c>
>>> xmlrpclib.DateTime(datetime.date(2007, 9, 7))
<DateTime '20070907T00:00:00' at 8108f8c>
```

Die erste DateTime-Instanz wurde aus einem UNIX-Timestamp erzeugt, während dem DateTime-Konstruktor bei der zweiten Instanziierung eine datetime.date-Instanz übergeben wurde.

Instanzen des Datentyps DateTime können Sie bedenkenlos in Form eines Rückgabewertes oder eines Parameters über eine XML-RPC-Schnittstelle senden.

Der Datentyp Binary

Der Datentyp Binary des Moduls xmlrpclib wird zum Versenden von Binärdaten über eine XML-RPC-Schnittstelle verwendet. Im Gegensatz zu Python macht der XML-RPC-Standard einen Unterschied zwischen ASCII-Strings und Binärdaten, was zur Folge hat, dass es eines zusätzlichen Datentyps für Binärdaten bedarf.

Bei der Instanziierung des Datentyps Binary wird ein String übergeben, der die binären Daten enthält. Diese können auf der Gegenseite über das Attribut data wieder ausgelesen werden:

```
>>> import xmlrpclib
>>> b = xmlrpclib.Binary("\x00\x01\x02\x03")
>>> b.data
'\x00\x01\x02\x03'
```

Instanzen des Datentyps `Binary` können Sie bedenkenlos in Form eines Rückgabewertes oder eines Parameters über eine XML-RPC-Schnittstelle senden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

21.2 Inspizieren von Instanzen – inspect

Das Modul `inspect` stellt Funktionen bereit, über die der Programmierer detaillierte Informationen über eine Instanzgi erlangen kann. So könnte beispielsweise der Inhalt einer Klasse oder die Parameterliste einer Funktion ermittelt werden. Damit eignet sich `inspect` besonders zum Erstellen von detaillierten Debug-Ausgaben.

Grundsätzlich lässt sich die Funktionalität von `inspect` in vier Teilbereiche gliedern, die im Folgenden erklärt werden sollen:

- Funktionen, die sich auf Datentypen, Attribute und Methoden einer Instanz beziehen
- Funktionen, die sich auf ein Stück des Quellcodes beziehen, das im Zusammenhang mit einer Instanz steht
- Funktionen, die sich auf Klassen- und Funktionsobjekte beziehen

Bevor Sie die Beispiele dieser Kapitel ausführen können, müssen Sie das Modul `inspect` einbinden:

```
>>> import inspect
```

top

21.2.1 Datentypen, Attribute und Methoden

In diesem Kapitel sollen die Funktionen des Moduls inspect besprochen werden, mit deren Hilfe der Programmierer Informationen über den Datentyp, Attribute oder Methoden einer Instanz abfragen kann. Im Folgenden werden die wichtigsten dieser Funktionen besprochen.

inspect.getmembers(object[, predicate])

Gibt alle Attribute und Methoden, auch *Member* genannt, der Instanz *object* in Form einer Liste von Tupeln zurück. Jedes Tupel enthält dabei den Namen des jeweiligen Members als erstes Element und den Wert des Members als zweites Element. Im Falle einer Methode entspricht das Funktionsobjekt dem Wert des Members. Die zurückgegebene Liste ist nach Member-Namen sortiert.

```
>>> class klasse(object):

...     def __init__(self):
...         self.a = 1
...         self.b = 2
...         self.c = 3

...     def hallo(self):
...         return "welt"
...
>>> inspect.getmembers(klasse())
 [('__class__', <class '__main__.klasse'>), […]
('a', 1), ('b', 2), ('c', 3), ('hallo', <bound method
klasse.hallo
of <__main__.klasse object at 0xb7a9d08c>>)]
```

Für den optionalen Parameter *predicate* kann eine Filterfunktion übergeben werden. Diese Funktion wird für jeden Member der Instanz *object* aufgerufen und bekommt diesen als einzigen Parameter übergeben. Es werden alle Member in die Ergebnisliste aufgenommen, für die die Filterfunktion True zurückgegeben hat.

```
>>> inspect.getmembers(klasse(), lambda x:
inspect.ismethod(x))
[('__init__', <bound method klasse.__init__ of
<__main__.klasse
object at 0xb7a9df2c>>), ('hallo', <bound method
klasse.hallo of
<__main__.klasse object at 0xb7a9df2c>>)]
```

In diesem Fall wurde eine Lambda-Form übergeben, die True zurückgibt, wenn es sich bei dem Member x um eine Methode handelt. Dementsprechend klein ist die Ergebnisliste. Auf die verwendete Funktion ismethod wird später noch eingegangen.

Zusätzlich zu getmembers sind im Modul inspect eine Menge Funktionen enthalten, deren Namen allesamt mit is beginnen. Eine dieser Funktionen wurde im vorangegangenen Beispiel bereits erfolgreich eingesetzt. Die folgende Tabelle listet die wichtigsten dieser Funktionen auf.

Funktion	Rückgabewert
isclass(object)	True, wenn object eine Klasse ist, andernfalls False.
ismodule(object)	True, wenn object ein Modul ist, andernfalls False.
ismethod(object)	True, wenn object eine Methode ist, andernfalls False.
isfunction(object)	True, wenn object eine Funktion ist, andernfalls False.
	True, wenn object ein Traceback-Objekt

istraceback(object)	ist, andernfalls False.
iscode(object)	True, wenn object ein Code-Objekt ist, andernfalls False. Näheres zum Code-Objekt erfahren Sie im nächsten Kapitel.
isbuiltin(object)	True, wenn object eine Built-in Function ist, andernfalls False.
isroutine(object)	True, wenn object eine Funktion oder Methode ist, andernfalls False.

Tabelle 21.1 Auf Member bezogene Funktionen des Moduls inspect

Grundsätzlich werden die Methoden folgendermaßen verwendet:

```
>>> def a():
...     print "Hallo Welt"
...
>>> inspect.isbuiltin(a)
False
>>> inspect.isfunction(a)
True
>>> inspect.isroutine(a)
True
```

Damit sind die wichtigsten Funktionen des Moduls inspect, die sich auf Datentypen, Attribute oder Methoden beziehen, beschrieben.

top

21.2.2 Quellcode ▼▲

In diesem Abschnitt sollen die wichtigsten Funktionen des Moduls inspect behandelt werden, die sich direkt auf den Quellcode beispielsweise einer Funktion beziehen.

inspect.getfile(object)

Gibt den Namen der Datei zurück, in der das Objekt *object* definiert wurde. Dabei kann es sich sowohl um eine Quellcode-Datei als auch um eine Bytecode-Datei handeln.

Diese Funktion wirft eine TypeError-Exception, wenn es sich bei *object* um ein eingebautes Objekt, beispielsweise eine Built-in Function handelt. Das liegt daran, dass Built-in Functions intern in C implementiert sind und somit keiner Quelldatei zugeordnet werden können.

```
>>> inspect.getfile(inspect.getfile)
'/usr/lib/python2.5/inspect.py'
```

In diesem Beispiel wurde die Funktion getfile verwendet, um herauszufinden, in welcher Quelldatei sie selbst definiert ist.

inspect.getmodule(object)

Gibt die Modulinstanz des Moduls zurück, in dem das Objekt *object* definiert wurde.

```
>>> inspect.getmodule(inspect.getmodule)
<module 'inspect' from
'C:\Programme\Python25\lib\inspect.pyc'>
```

In diesem Beispiel wurde getmodule dazu verwendet, den Pfad des Moduls herauszufinden, in dem sie selbst definiert ist.

inspect.getsourcefile(object)

Gibt den Namen der Quellcode-Datei zurück, in der das Objekt *object* definiert wurde. Diese Funktion wirft eine TypeError-Exception, wenn es sich bei *object* um ein eingebautes Objekt,

beispielsweise eine Built-in Function handelt. Das liegt daran, dass diese Objekte entweder intern in C implementiert sind oder die Quelldatei nur als Kompilat vorliegt. Dem Objekt kann also kein tatsächlicher Quellcode zugeordnet werden. Diese Einschränkung betrifft auch viele Funktionen der Standardbibliothek.

```
>>> inspect.getsourcefile(inspect.getsourcefile)
'/usr/lib/python2.5/inspect.py'
```

In diesem Fall wurde `getsourcefile` dazu verwendet, die Quellcodedatei herauszufinden, in der die Funktion selbst definiert ist.

inspect.getsourcelines(object)

Gibt ein Tupel mit zwei Elementen zurück. Das erste ist eine Liste von Strings, die alle dem Objekt *object* zugeordneten Quellcodezeilen enthält. Das zweite Element des zurückgegebenen Tupels ist die Zeilennummer der ersten dem Objekt *object* zugeordneten Quellcodezeile.

Für *object* kann ein Modul, eine Methode, eine Funktion, ein Traceback-Objekt oder ein Frame-Objekt übergeben werden. Näheres zum Frame-Objekt erfahren Sie im nächsten Kapitel.

Die Funktion wirft eine `IOError`-Exception, wenn der Quellcode zum Objekt *object* nicht geladen werden konnte.

```
>>> inspect.getsourcelines(inspect.getsourcelines)
(['def getsourcelines(object):\n', […], 610)
```

In diesem Fall wurde die Funktion `getsourcelines` dazu verwendet, die Quellcodezeilen ihrer eigenen Definition zurückzugeben. Beachten Sie, dass das Auslassungszeichen keine rekursive Liste andeutet, sondern weitere Quellcodezeilen andeuten soll.

inspect.getsource(object)

Gibt die dem Objekt *object* zugeordneten Quellcodezeilen in einem einzigen String zurück. Die Funktion unterscheidet sich demzufolge nur im Rückgabewert von `getsourcelines`.

Für *object* kann ein Modul, eine Methode, eine Funktion, ein Traceback-Objekt oder ein Frame-Objekt übergeben werden.

Die Funktion wirft eine `IOError`-Exception, wenn der Quellcode zum Objekt *object* nicht geladen werden konnte.

top

21.2.3 Klassen und Funktionen ▲

In diesem Abschnitt sollen die wichtigsten Funktionen des Moduls `inspect` behandelt werden, die sich auf Klassen und Funktionen beziehen.

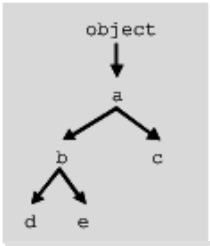
inspect.getclasstree(classes[, unique])

Gibt die Vererbungshierarchie der übergebenen Klassen in Form eines Baums [Bei einem Baum handelt es sich grundsätzlich um nichts anderes als um eine verschachtelte Liste. Gerade im Zusammenhang mit `getclasstree` ist die zurückgegebene Liste ausdrücklich als Baum zu sehen.] zurück. Im ersten Teil des Beispielprogramms bauen wir daher zunächst eine Klassenhierarchie auf:

```
>>> class a(object):
...     pass
...
```

```
>>> class b(a):
...     pass
...
>>> class c(a):
...     pass
...
>>> class d(b):
...     pass
...
>>> class e(b):
...     pass
...
```

Abbildung 21.2 veranschaulicht die Hierarchie.



Hier klicken, um das Bild zu vergrößern

Abbildung 21.2 Die Klassenhierarchie

Die Funktion `getclasstree` bereitet Teile dieser Hierarchie zu einem Baum auf. Dazu müssen beim Funktionsaufruf die Klassen übergeben werden, die im resultierenden Baum enthalten sein sollen. Im folgenden Beispiel soll die Hierarchie um die Klasse b erzeugt werden:

```
>>> inspect.getclasstree([b])
[(<class '__main__.a'>, (<type 'object'>,)),
    [(<class '__main__.b'>, (<class '__main__.a'>,))
    ]
]
```

Der resultierende Baum, hier aus Gründen der Übersichtlichkeit formatiert, besteht aus einer Liste von Tupeln, die jeweils eine Klasse repräsentieren. Ein solches Tupel ist folgendermaßen aufgebaut:

```
(Klasse, (Basisklassen))
```

Nach jedem solchen Tupel kann eine eingebettete Liste folgen, die alle Klassen enthält, die von der Klasse abgeleitet sind, die das Tupel beschreibt. Auch diese eingebetteten Listen bestehen aus Tupeln des obigen Formats.

Wie Sie sehen, wurde im Beispiel ein Baum erzeugt, der nur die Klasse b selbst und ihre Basisklasse a enthält.

Für den ersten Parameter *classes* von *getclasstree* muss eine Liste übergeben werden, die auch, anders als im vorangegangenen Beispiel, mehrere Klassen enthalten darf, die dann alle im resultierenden Baum vorkommen. So soll im folgenden Beispiel die Hierarchie um die Klassen b und a erstellt werden:

```
>>> inspect.getclasstree([b, a])
[(<type 'object'>, ()),
    [(<class '__main__.a'>, (<type 'object'>,)),
        [(<class '__main__.b'>, (<class '__main__.a'>,))
        ]
    ]
]
```

Der resultierende Baum enthält die aufgelisteten Klassen a und b sowie die Basisklasse `object` von a.

Wenn für den optionalen Parameter *unique* `True` übergeben wird, taucht jede Klasse auch im Falle von Mehrfachvererbung nur ein einziges Mal im Klassenbaum auf.

inspect.getmro(cls)

Gibt ein Tupel zurück, das alle Basisklassen der Klasse *cls* inklusive der Klasse *cls* selbst enthält. Die Klassen sind dabei so angeordnet, dass die allgemeinste Basisklasse als letzte im Tupel enthalten ist. Das bedeutet umgekehrt, dass die Klasse *cls* selbst als erstes Element des resultierenden Tupels geführt wird.

```
>>> inspect.getmro(b)
(
<class '__main__.b'>,
<class '__main__.a'>,
<type 'object'>
)
```

Dieses Beispiel bezieht sich auf die bei `getclasstree` angelegte Klassenhierarchie.

inspect.getargspec(func)

Gibt die Funktionsschnittstelle der Funktion *func*, also die Namen und vordefinierten Werte der Funktionsparameter zurück. Das Ergebnis ist ein Tupel mit vier Elementen. Das erste Element ist eine Liste mit den Namen aller Parameter. Das zweite und das dritte Element des zurückgegebenen Tupels enthalten den Parameternamen, über den beliebige Positions- oder Schlüsselwortparameter funktionsintern angesprochen werden. Das letzte Element des Tupels ist ein Tupel mit vordefinierten Werten der Parameter.

```
>>> def funktion(a, b, c=1, d=2, *args):
...     pass
...
>>> inspect.getargspec(funktion)
(['a', 'b', 'c', 'd'], 'args', None, (1, 2))
```

In diesem Fall besteht die Funktionsschnittstelle also aus den Parametern `a`, `b`, `c` und `d` sowie aus einer Möglichkeit, beliebige weitere Positionsparameter zu übergeben. Auf diese zusätzlichen Parameter wird funktionsintern über den Namen `args` zugegriffen. Eine Möglichkeit, beliebige Schlüsselwortparameter zu übergeben, gibt es nicht, das entsprechende Element des Tupels ist `None`. Schlussendlich können dem Tupel noch die vordefinierten Werte für die letzten beiden Parameter `c` und `d` entnommen werden.

inspect.getargvalues(frame)

Gibt Informationen über die einer Funktion tatsächlich übergebenen Werte zurück. Der Rückgabewert ist ein Tupel mit vier Elementen. Die ersten drei Elemente des Tupels entsprechen denen des Tupels, das von der Funktion `getargspec` zurückgegeben wird. Das vierte Element des Tupels enthält ein Dictionary mit den Namen und Werten aller lokalen Variablen des übergebenen Frame-Objekts *frame*.

Das sogenannte *Frame-Objekt* repräsentiert einen Codeblock und kann mithilfe der Funktion `inspect.currentframe` erzeugt werden.

```
>>> def funktion(a, b, c=1, d=2):
...     print inspect.getargvalues(inspect.currentframe())
...
>>> funktion("Hallo", "Welt")
(['a', 'b', 'c', 'd'], None, None,
{'a': 'Hallo', 'c': 1, 'b': 'Welt', 'd': 2})
```

Beachten Sie, dass die beiden `None`-Elemente des Tupels nur Bedeutung haben, wenn die Funktionsschnittstelle das Übergeben beliebiger Positions- oder Schlüsselwortparameter ermöglicht, was

hier nicht der Fall ist.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

21.3 Formatierte Ausgabe von Instanzen – pprint

Das Modul `pprint` (für *pretty print*) gibt eine formatierte Repräsentation eines Python-Datentyps auf dem Bildschirm aus. Damit macht das Modul insbesondere die Ausgabe großer Listen oder Dictionarys besser lesbar und bietet sich somit förmlich an, in einer interaktiven Debug-Sitzung zur Ausgabe verschiedener Werte verwendet zu werden.

Im Modul `pprint` sind die folgenden Funktionen enthalten. Um die Beispiele ausführen zu können, muss zuvor das Modul `pprint` eingebunden werden:

```
>>> import pprint
```

pprint.pprint(object[, stream[, indent[, width[, depth]]]])

Gibt die Instanz *object* formatiert auf dem Stream *stream* aus. Wenn der Parameter *stream* nicht übergeben wird, wird nach `sys.stdout` geschrieben. Über die Parameter *indent* und *depth* lässt sich die Formatierung der Ausgabe steuern. Dabei kann für

indent die Anzahl der Leerzeichen übergeben werden, die für eine Einrückung verwendet werden soll. Der Parameter *indent* ist mit 1 vorbelegt.

Über den optionalen Parameter *width* kann die maximale Anzahl an Zeichen angegeben werden, die die Ausgabe breit sein darf. Dieser Parameter ist mit 80 Zeichen vorbelegt.

Der Parameter *depth* ist ebenfalls eine ganze Zahl und gibt an, bis zu welcher Tiefe Unterinstanzen, beispielsweise also verschachtelte Listen, ausgegeben werden sollen. Im folgenden Beispiel soll die Funktion pprint dazu verwendet werden, die Liste sys.path formatiert und damit lesbar auszugeben:

```
>>> import sys
>>> pprint.pprint(sys.path, indent=4)
[   'C:\\Python25\\Lib\\idlelib',
    'C:\\WINDOWS\\system32\\python25.zip',
    'C:\\Python25\\DLLs',
    'C:\\Python25\\lib',
    'C:\\Python25\\lib\\plat-win',
    'C:\\Python25\\lib\\lib-tk',
    'C:\\Python25',
    'C:\\Python25\\lib\\site-packages']
```

Zum Vergleich geben wir sys.path noch einmal unformatiert mittels print aus:

```
>>> print sys.path
['C:\\Python25\\Lib\\idlelib',
'C:\\WINDOWS\\system32\\python25.zip',
 'C:\\Python25\\DLLs', 'C:\\Python25\\lib',
'C:\\Python25\\lib\\plat-win', 'C:\\Python25',
 'C:\\Python25\\lib\\lib-tk', 'C:\\Python25',
'C:\\Python25\\lib\\site-packages']
```

Die Ausgabe eines Dictionarys sieht folgendermaßen aus:

```
>>> d = {}
>>> d["path"] = sys.path
>>> d["hallo"] = "welt"
>>> d["aaa"] = "Sortierte Ausgabe"
>>> pprint.pprint(d)

{'aaa': 'Sortierte Ausgabe',
 'hallo': 'welt',
 'path': ['C:\\Python25\\Lib\\idlelib',
          'C:\\WINDOWS\\system32\\python25.zip',
          'C:\\Python25\\DLLs',
          'C:\\Python25\\lib',
          'C:\\Python25\\lib\\plat-win',
          'C:\\Python25\\lib\\lib-tk',
          'C:\\Python25',
          'C:\\Python25\\lib\\site-packages']}
```

Es wurde ein Dictionary angelegt, in dem unter anderem die Liste *sys.path* als Wert enthalten ist. Damit ist dies nicht nur ein Beispiel für die formatierte Ausgabe eines Dictionarys, sondern zugleich dafür, wie verschachtelte Instanzen formatiert werden. Beachten Sie, dass die Schlüssel/Wert-Paare des Dictionarys bei der Ausgabe nach den Schlüsseln sortiert werden.

pprint.pformat(object[, indent[, width[, depth]]])

Die Funktion pformat hat eine ähnliche Bedeutung wie pprint, mit dem Unterschied aber, dass pformat die formatierte Ausgabe in Form eines Strings zurückgibt, anstatt sie in einen Stream zu schreiben. Die Schnittstelle von pformat ist bis auf den hier überflüssigen *stream*-Parameter mit der von pprint identisch.

pprint.isreadable(object)

Gibt True zurück, wenn die formatierte Ausgabe der Instanz *object* lesbar ist. Lesbar bedeutet in diesem Zusammenhang, dass die Ausgabe als Python-Code gelesen werden kann und mithilfe der Built-in Function eval aus der Ausgabe wieder die zugrunde liegende Python-Instanz rekonstruiert werden könnte.

Wenn *object* eine Instanz eines Basisdatentyps ist, gibt die Funktion in den meisten Fällen True zurück:

```
>>> pprint.isreadable(sys.path)
True
>>> a = []
>>> a.append(a)
>>> pprint.isreadable(a)
False
```

Im zweiten Teil des Beispiels wurde `False` zurückgegeben, weil es sich um eine rekursive Liste handelt, die sich selbst als Element enthält. Eine solche Liste kann aus naheliegenden Gründen nicht vollständig ausgegeben werden. Auch Instanzen selbst erstellter Klassen können nicht lesbar repräsentiert werden. Letzteres liegt daran, dass die Ausgabe dann den Code zum Erstellen der Klasse sowie der Instanz enthalten müsste, was keinen Sinn macht.

pprint.isrecursive(object)

Gibt `True` zurück, wenn die Instanz *object* rekursiv ist, also sich selbst als Element enthält.

```
>>> a = []
>>> a.append(a)
>>> pprint.isrecursive(a)
True
```

Damit sind die grundlegendsten Möglichkeiten, die das Modul `inspect` bietet, erläutert, und wir werden uns im Folgenden mit dem Modul `logging` beschäftigen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

21.4 Logdateien – logging

Das Modul `logging` stellt ein flexibles Interface zum Protokollieren eines Programmlaufs bereit. Protokolliert wird der Programmablauf, indem an unterschiedlichen Stellen im Programm Meldungen an das `logging`-Modul abgesetzt werden. Diese Meldungen können unterschiedliche Dringlichkeitsstufen haben. So gibt es beispielsweise Fehlermeldungen, Warnungen oder Informationen. Das Modul `logging` kann diese Meldungen auf vielfältige Weise verarbeiten. Üblich ist es, die Meldung mit einem Timestamp zu versehen und entweder auf dem Bildschirm auszugeben oder in eine Datei zu schreiben.

In diesem Kapitel soll die Verwendung des Moduls `logging` anhand mehrerer Beispiele im interaktiven Modus gezeigt werden. Um die Beispielprogramme korrekt ausführen zu können, muss zuvor das Modul `logging` eingebunden sein:

```
>>> import logging
```

Bevor Meldungen an den sogenannten *Logger* geschickt werden können, muss dieser durch Aufruf der Funktion `basicConfig` initialisiert werden. Die Funktion `basicConfig` bekommt

verschiedene Schlüsselwortparameter übergeben. Im folgenden Beispiel wird ein Logger eingerichtet, der alle eingehenden Meldungen in die Logdatei *programm.log* schreibt:

```
>>> logging.basicConfig(
...     filename = "programm.log",
...     filemode = "a")
```

Über den Schlüsselwortparameter `filemode` kann der Modus angegeben werden, in dem die Logdatei geöffnet werden soll. Eine Liste aller verfügbaren Modi und ihrer Bedeutung finden Sie in Abschnitt 9.3.4. Der Parameter ist mit "a" für »append« vorbelegt und könnte hier auch weggelassen werden.

Jetzt können mithilfe der im Modul enthaltenen Funktion `log` Meldungen an den Logger übergeben werden. Die Funktion *log* bekommt dabei die Dringlichkeitsstufe der Meldung als ersten und die Meldung selbst in Form eines Strings als zweiten Parameter übergeben:

```
>>> logging.log(logging.ERROR, "Ein Fehler ist
aufgetreten")
>>> logging.log(logging.INFO, "Dies ist eine Information")
```

Durch das Aufrufen der Funktion `shutdown` wird der Logger korrekt deinitialisiert, und eventuell noch anstehende Schreiboperationen werden durchgeführt:

```
>>> logging.shutdown()
```

Natürlich sind nicht nur die Dringlichkeitsstufen `ERROR` und `INFO` verfügbar. Die folgende Tabelle listet alle vordefinierten Stufen auf, aus denen Sie wählen können. Die Tabelle ist dabei nach Dringlichkeit geordnet, wobei die dringendste Stufe zuletzt aufgeführt wird.

Level	Beschreibung
`loggin.DEBUG`	Eine Meldung, die nur für den Programmierer zur Fehlersuche interessant ist
`loggin.INFO`	Eine Informationsmeldung über den Programmstatus
`loggin.WARNING`	Eine Warnmeldung, die auf einen möglichen Fehler hinweist
`loggin.ERROR`	Eine Fehlermeldung, nach der das Programm weiterarbeiten kann
`loggin.CRITICAL`	Eine Meldung über einen kritischen Fehler, der das sofortige Beenden des Programms oder der aktuell durchgeführten Operation zur Folge hat

Tabelle 21.2 Vordefinierte Dringlichkeitsstufen

Beachten Sie, dass aus Gründen des Komforts zu jeder Dringlichkeitsstufe eine eigene Funktion existiert. So sind die beiden Funktionsaufrufe von `logging.log` aus dem letzten Beispiel äquivalent zu:

```
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
```

Wenn Sie sich die Logdatei nach dem Aufruf dieser beiden Funktionen ansehen, werden Sie feststellen, dass es lediglich einen einzigen Eintrag gibt:

```
ERROR:root:Ein Fehler ist aufgetreten
```

Das liegt daran, dass der Logger in seiner Basiskonfiguration nur Meldungen loggt, deren Dringlichkeit größer oder gleich der einer Warnung sind. Um auch Debug- und Info-Meldungen mitzuloggen, muss beim Aufruf der Funktion basicConfig im Schlüsselwortparameter level ein geeigneter Wert übergeben werden:

```
logging.basicConfig(
    filename="programm.log",
    level = logging.DEBUG)
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
```

In diesem Beispiel wurde die Mindestdringlichkeit auf DEBUG gesetzt. Das bedeutet, dass alle Meldungen, die mindestens eine Dringlichkeit von DEBUG haben, geloggt werden. Folglich erscheinen auch beide Meldungen in der Logdatei:

```
ERROR:root:Ein Fehler ist aufgetreten
INFO:root:Dies ist eine Information
```

top

21.4.1 Das Meldungsformat anpassen

Zu Beginn von Abschnitt 21.4 wurde erwähnt, dass man Einträge in einer Logdatei gerne mit der Uhrzeit versieht, zu der die Meldung eingegangen ist. Eine solche Uhrzeit wird aber, wie in den vorangegangenen Beispielen zu sehen war, standardmäßig nicht angezeigt. Es gibt allerdings eine Möglichkeit, das Format der geloggten Meldung anzupassen. Dazu muss beim Funktionsaufruf von basicConfig der Schlüsselwortparameter format übergeben werden:

```
logging.basicConfig(
    filename="programm.log",
    level = logging.DEBUG,
    format = "%(asctime)s [%(levelname)-8s] %(message)s")
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
logging.error("Und schon wieder ein Fehler")
```

Sie sehen, dass ein Format-String übergeben wurde, der die Vorlage für eine Meldung enthält, wie sie später in der Logdatei stehen soll. Beachten Sie, dass es sich dabei um die bekannte String-Formatierung handelt, bei der die Bezeichner asctime für den Timestamp, levelname für die Dringlichkeitsstufe und message für die Meldung verwendet wurden. Die von diesem Beispiel generierten Meldungen sehen folgendermaßen aus:

```
2007-09-27 03:28:55,811 [ERROR   ] Ein Fehler ist
aufgetreten
2007-09-27 03:29:00,690 [INFO    ] Dies ist eine
Information
2007-09-27 03:29:12,686 [ERROR   ] Und schon wieder ein
Fehler
```

Die folgende Tabelle listet die wichtigsten Bezeichner auf, die innerhalb des format-Formatstrings verwendet werden dürfen. Beachten Sie, dass einige der Bezeichner je nach Kontext, in dem die Meldung erzeugt wird, keine Bedeutung haben.

Level	Beschreibung
%(levelname)s	Die Dringlichkeitsstufe der Meldung
%(pathname)s	Der Pfad zur Programmdatei, in der die Meldung abgesetzt wurde
%(filename)s	Der Dateiname der Programmdatei, in der die Meldung abgesetzt wurde
%(module)s	Der Name des Moduls, in dem die Meldung abgesetzt wurde. Der Modulname entspricht dem Dateinamen ohne Dateiendung.

%(funcName)s	Der Name der Funktion, in der die Meldung abgesetzt wurde
%(lineno)s	Die Quellcodezeile, in der die Meldung abgesetzt wurde
%(asctime)s	Zeitpunkt der Meldung. Beachten Sie, dass das Datums- und Zeitformat beim Funktionsaufruf von basicConfig über den Parameter datefmt angegeben werden kann. Näheres dazu folgt im Anschluss an diese Tabelle.
%(thread)s	Die ID des Threads, in dem die Meldung abgesetzt wurde
%(process)s	Die ID des Prozesses, in dem die Meldung abgesetzt wurde
%(message)s	Der Text der Meldung

Tabelle 21.3 Bezeichner im Formatstring

Was uns an diesen Meldungen vielleicht noch stört, ist das Format des Timestamps. Zum einen wird das amerikanische Datumsformat verwendet, und zum anderen ist eine Auflösung bis auf die Millisekunde genau für unsere Zwecke ein bisschen zu fein. Das Format des Timestamps kann beim Aufruf von basicConfig über den Schlüsselwortparameter datefmt angegeben werden:

```
logging.basicConfig(
    filename="programm.log",
    level = logging.DEBUG,
    format = "%(asctime)s %(levelname)s: %(message)s",
    datefmt = "%d.%m.%Y %H:%M:%S")
logging.error("Ein Fehler ist aufgetreten")
```

Die in der Vorlage für das Datumsformat verwendeten Platzhalter wurden in Abschnitt 16.1 eingeführt. Die von diesem Beispiel erzeugte Meldung sieht folgendermaßen aus:

```
27.09.2007 03:38:49 ERROR: Ein Fehler ist aufgetreten
```

top

21.4.2 Logging Handler

Bisher wurde ausschließlich besprochen, wie das Modul logging dazu verwendet werden kann, alle eingehenden Meldungen in eine Datei zu schreiben. Tatsächlich ist das Modul in dieser Beziehung sehr flexibel und erlaubt es, nicht nur in Dateien, sondern beispielsweise auch in Streams zu schreiben oder die Meldungen über eine Netzwerkverbindung zu schicken. Dafür werden sogenannte Logging Handler verwendet. Um genau zu sein, haben wir in den vorherigen Abschnitten bereits einen impliziten Handler verwendet, ohne uns darüber im Klaren zu sein.

Um einen speziellen Handler einzurichten, muss eine Instanz der Handler-Klasse erzeugt werden. Diese kann dann vom Logger verwendet werden. Im folgenden Beispiel sollen alle Meldungen auf einen Stream, nämlich sys.stdout, geschrieben werden. Dazu wird die Handler-Klasse logging.StreamHandler verwendet.

```
import logging
import sys

handler = logging.StreamHandler(sys.stdout)
frm = logging.Formatter("%(asctime)s %(levelname)s:
%(message)s",
                        "%d.%m.%Y %H:%M:%S")
handler.setFormatter(frm)

logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.critical("Ein wirklich kritischer Fehler")
logger.warning("Und eine Warnung hinterher")
logger.info("Dies hingegen ist nur eine Info")
```

Zunächst wird der Handler, in diesem Fall ein `StreamHandler`, instanziiert. Im nächsten Schritt wird eine Instanz der Klasse `Formatter` erzeugt. Diese Klasse kapselt die Formatierungsanweisungen, die wir in den vorherigen Beispielen beim Aufruf der Funktion `basicConfig` übergeben haben. Mithilfe der Methode `set Formatter` werden dem Handler die Formatierungsanweisungen bekannt gegeben.

Um den Handler beim Logger zu registrieren, benötigen wir Zugriff auf die bisher implizit verwendete Logger-Instanz. Diesen Zugriff erlangen wir über die Funktion `logging.getLogger`. Danach wird über `addHandler` der Handler hinzugefügt und über `setLevel` die gewünschte Dringlichkeitsstufe eingestellt.

Beachten Sie, dass die Meldungen im Folgenden nicht über Funktionen des Moduls `logging`, sondern über die Methoden `critical`, `warning` und `info` der Logger-Instanz `logger` abgesetzt werden. Das Beispielprogramm gibt folgenden Text auf dem Bildschirm aus:

```
27.09.2007 17:21:46 CRITICAL: Ein wirklich kritischer
Fehler
27.09.2007 17:21:46 WARNING: Und eine Warnung hinterher
27.09.2007 17:21:46 INFO: Dies hingegen ist nur eine Info
```

Im Folgenden sollen die wichtigsten zusätzlichen Handler-Klassen beschrieben werden, die im Modul `logging` enthalten sind.

logging.FileHandler(filename[, mode)

Dieser Handler schreibt die Logeinträge in die Datei *filename*. Dabei wird die Datei im Modus `mode` geöffnet. Beachten Sie, dass der Handler `FileHandler` auch implizit durch Angabe der Schlüsselwortparameter `filename` und `filemode` beim Aufruf der Funktion `basicConfig` verwendet werden kann.

logging.StreamHandler([stream])

Dieser Handler schreibt die Logeinträge in den Stream *stream*. Beachten Sie, dass der Handler `StreamHandler` auch implizit durch Angabe des Schlüsselwortparameters `stream` beim Aufruf der Funktion `basicConfig` verwendet werden kann.

logging.handler.SocketHandler(host, port)

Dieser Handler sendet die Logeinträge über eine TCP-Netzwerkschnittstelle an den Rechner mit dem Hostnamen *host* unter Verwendung des Ports *port*. Beachten Sie, dass die Meldungen komprimiert werden und der Rechner, der die Logeinträge entgegennimmt, diese speziell entpacken muss. Aus Platzgründen werden wir darauf hier nicht eingehen, Sie finden aber ein ausführliches Beispielprogramm in der Dokumentation des Moduls `logging` unter dem Stichwort »*Sending and receiving events across a network*«.

logging.handler.DatagramHandler(host, port)

Dieser Handler sendet die Logeinträge über eine UDP-Netzwerkschnittstelle und wird ähnlich verwendet wie der Handler `SocketHandler`.

logging.handler.SMTPHandler(mailhost, fromaddr, toaddr, subject)

Dieser Handler sendet die Logeinträge als E-Mail an die Adresse *toaddr*. Dabei wird *subject* als Betreff und *fromaddr* als Absenderadresse eingetragen. Über den Parameter *mailhost* wird der zu verwendende SMTP-Server angegeben.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

21.6 Traceback-Objekte – traceback

Das Modul `traceback` existiert zum Umgang mit dem sogenannten *Traceback-Objekt*. Ein Traceback-Objekt hält den Kontext fest, aus dem eine Exception geworfen wurde, und liefert damit die Informationen, die bei einem Traceback auf dem Bildschirm angezeigt werden. Zu diesen Informationen gehört vor allem die Funktionshierarchie, der sogenannte *Callstack*. Ein Traceback-Objekt wird beim Werfen einer Exception automatisch erzeugt. Generell können Sie auf das Traceback-Objekt einer gerade abgefangenen Exception über die Funktion `sys.exc_info` des Moduls `sys` zugreifen. Im Folgenden sollen die im Modul `traceback` enthaltenen Funktionen besprochen werden. Dabei werden alle Beispiele im folgenden Kontext ausgeführt:

```
>>> import traceback
>>> import sys
>>> def f1():
...     raise TypeError
...
>>> def f2():
...     f1()
...
>>> try:
...     f2()
... except TypeError:
...     tb = sys.exc_info()[2]
```

```
...
>>> tb
<traceback object at 0xb7c49414>
```

Es existiert also ein Traceback-Objekt `tb`, das den Callstack einer `TypeError`-Exception beschreibt, die zuvor aus einer verschachtelten Funktionshierarchie heraus geworfen wurde.

traceback.print_tb(traceback[, limit[, file]])

Gibt den Stacktrace des Traceback-Objekts formatiert auf dem Bildschirm aus. Über den optionalen Parameter `limit` kann angegeben werden, wie viele Einträge des Stacktraces maximal ausgegeben werden sollen. Für den dritten, optionalen Parameter kann ein geöffnetes Dateiobjekt übergeben werden, in das der Stacktrace geschrieben wird. Standardmäßig wird in den Stream `sys.stderr` geschrieben.

```
>>> traceback.print_tb(tb)
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in f2
  File "<stdin>", line 2, in f1
```

traceback.print_exception(type, value, traceback[, limit[, file]])

Diese Funktion gibt einen vollständigen Traceback auf dem Bildschirm aus. Die Ausgabe ist genau so formatiert wie die einer normalen, nicht abgefangenen Exception. Für die beiden zusätzlichen Parameter *type* und *value* müssen Exception-Typ und Exception-Wert übergeben werden. Die restlichen Parameter haben dieselbe Bedeutung wie bei `print_tb`. So könnte beispielsweise der Funktionsaufruf von print_exception im interaktiven Modus folgendermaßen aussehen:

```
>>> traceback.print_exception(TypeError, "Hallo Welt", tb)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in f2
  File "<stdin>", line 2, in f1
TypeError: Hallo Welt
```

traceback.print_exc([, limit[, file]])

Wie `print_exception`, jedoch immer für die aktuell abgefangene Exception. Die Parameter *limit* und *file* haben dieselbe Bedeutung wie bei `print_tb`. Beachten Sie, dass diese Funktion nur innerhalb eines except-Zweiges aufgerufen werden kann.

```
>>> try:
...     raise TypeError
... except TypeError:
...     traceback.print_exc()
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError
```

traceback.format_exc([limit])

Diese Funktion arbeitet wie `print_exc`, mit dem Unterschied, dass der formatierte Traceback nicht auf dem Bildschirm bzw. in eine Datei ausgegeben, sondern als String zurückgegeben wird.

traceback.print_last([, limit[, file]])

Diese Funktion arbeitet wie `print_exception`, jedoch immer für die zuletzt abgefangene Exception. Diese Funktion kann auch außerhalb eines except-Zweiges aufgerufen werden.

```
>>> traceback.print_last()
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

```
  File "<stdin>", line 2, in f2
  File "<stdin>", line 2, in f1
TypeError
```

traceback.extract_tb(traceback[, limit])

Gibt eine Liste von aufbereiteten Stacktrace-Einträgen des Traceback-Objekts *traceback* zurück. Ein aufbereiteter Stacktrace-Eintrag ist ein Tupel der folgenden Form:

```
(Dateiname, Zeilennummer, Funktionsname, Text)
```

Der optionale Parameter *limit* hat die gleiche Bedeutung wie bei `print_tb`.

```
>>> traceback.extract_tb(tb)
[('<stdin>', 2, '<module>', None),
('<stdin>', 2, 'f2', None),
('<stdin>', 2, 'f1', None)]
```

traceback.format_list(lst)

Die Funktion `format_list` bekommt eine Liste von Tupeln übergeben, wie sie beispielsweise von der Funktion `extract_tb` zurückgegeben wird. Aus diesen Informationen erzeugt `format_list` eine Liste von aufbereiteten Strings der folgenden Form:

```
>>> traceback.format_list(traceback.extract_tb(tb))
['  File "<stdin>", line 2, in <module>\n',
'  File "<stdin>", line 2, in f2\n',
'  File "<stdin>", line 2, in f1\n']
```

traceback.format_exception_only(type, value)

Formatiert eine Exception mit dem Typ *type* und dem Wert *value* zu einer Liste von Strings. Jeder der in dieser Liste enthaltenen Strings repräsentiert eine Zeile der Ausgabe.

```
>>> traceback.format_exception_only(TypeError, "Hallo
Welt")
['TypeError: Hallo Welt\n']
```

traceback.format_exception(type, value, tb[, limit])

Formatiert eine Exception mit dem Typ *type*, dem Wert *value* und dem Stacktrace *tb* zu einer Liste von Strings. Jeder String dieser Liste repräsentiert eine Zeile der Ausgabe. Der Parameter *limit* hat die gleiche Bedeutung wie bei der Funktion `print_tb`.

```
>>> traceback.format_exception(TypeError, "Hallo Welt",
tb)
['Traceback (most recent call last):\n',
'  File "<stdin>", line 2, in <module>\n',
'  File "<stdin>", line 2, in f2\n',
'  File "<stdin>", line 2, in f1\n',
'TypeError: Hallo Welt\n']
```

traceback.format_tb(tb[, limit])

Der Aufruf dieser Funktion ist äquivalent zu `format_list(extract_tb(tb, limit))`.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

21.7 Analyse des Laufzeitverhaltens ▼

Die Optimierung eines Programms ist ein wichtiger Teilbereich der Programmierung und kann viel Zeit in Anspruch nehmen. In der Regel wird zunächst ein lauffähiges Programm erstellt, das alle gewünschten Anforderungen erfüllt, bei dem jedoch noch nicht unbedingt Wert auf die Optimierung der Algorithmik gelegt wird. Das liegt vor allem daran, dass man oftmals erst beim fertigen Programm die tatsächlichen Engpässe erkennt und im frühen Stadium somit eventuell viel Zeit in die Optimierung völlig unkritischer Bereiche investieren würde.

Um das Laufzeitverhalten eines Python-Programms möglichst genau zu erfassen, existieren die drei Module `timeit`, `profile` und `cProfile` in der Standardbibliothek von Python. Diese Module sollen das Thema der nächsten Abschnitt sein.

top

21.7.1 Laufzeitmessung – timeit ▼▲

Das Modul `timeit` der Standardbibliothek ermöglicht es, genau zu messen, wie lange ein Python-Programm zur Ausführung braucht. Üblicherweise wird `timeit` dazu verwendet, die Laufzeit zweier

verschiedener Algorithmen für dasselbe Problem zu vergleichen.

Sie erinnern sich sicherlich noch, dass im Kapitel über Funktionen zwei Algorithmen zur Berechnung der Fakultät einer ganzen Zahl besprochen wurden: ein iterativer und ein rekursiver. Es wurde gesagt, dass ein laufzeitoptimierter iterativer Algorithmus im Vergleich zu seinem rekursiven Pendant stets effizienter ist. Das wollen wir in diesem Kapitel anhand des `timeit`-Moduls überprüfen, und zusätzlich wollen wir testen, um wie viel Prozent schneller die iterative Variante tatsächlich ausgeführt werden kann.

Um die Laufzeit eines Python-Codes zu testen, muss die im Modul `timeit` enthaltene Klasse `Timer` instanziiert werden. Der Konstruktor der Klasse `Timer` hat folgende Schnittstelle:

timeit.Timer([stmt[, setup[, timer]]])

Erzeugt eine Instanz der Klasse `Timer`. Der zu analysierende Python-Code kann dem Konstruktor in Form des Parameters *stmt* als String übergeben werden. Für den zweiten Parameter *setup* kann ebenfalls ein String übergeben werden, der den Python-Code enthält, der zur Initialisierung von *stmt* benötigt wird. Demzufolge wird *setup* auch vor *stmt* ausgeführt. Beide Parameter sind optional und mit dem String "pass" vorbelegt.

Als dritter optionaler Parameter *timer* kann eine Zeitgeberfunktion übergeben werden. Dies sollte eine der Funktionen `time.time` oder `time.clock` des Moduls `time` sein. Standardmäßig wird diejenige dieser beiden Funktionen verwendet, die auf dem aktuellen System die höchste Auflösung bietet. Das ist `time.time` unter Windows und `time.clock` unter Unix-artigen Betriebssystemen. Es ist normalerweise nicht notwendig, diesen Parameter anzugeben.

Die Klasse Timer

Nachdem eine Instanz der Klasse `Timer` erzeugt worden ist, besitzt sie drei Methoden, die im Folgenden besprochen werden sollen. Dabei sei *t* eine Instanz der Klasse `Timer`.

t.timeit([number])

Diese Methode führt zunächst den *setup*-Code einmalig aus und wiederholt danach den beim Konstruktor für *stmt* übergebenen Code *number*-mal. Wenn der optionale Parameter *number* nicht angegeben wurde, wird der zu messende Code 1.000.000-mal ausgeführt.

Die Funktion gibt die Zeit zurück, die das Ausführen des gesamten Codes (also inklusive aller Wiederholungen, jedoch exklusive des Setup-Codes) in Anspruch genommen hat. Der Wert wird in Sekunden als Gleitkommazahl zurückgegeben.

Hinweis

Um das Ergebnis von äußeren Faktoren so unabhängig wie möglich zu machen, wird für die Dauer der Messung die Garbage Collection des Python-Interpreters deaktiviert. Sollte die Garbage Collection ein wichtiger mitzumessender Teil Ihres Codes sein, so lässt sie sich mit einem Setup-Code von "`gc.enable()`" wieder aktivieren.

t.repeat([repeat[, number]])

Ruft die Methode `timeit` *repeat*-mal auf und gibt die Ergebnisse in Form einer Liste von Gleitkommazahlen zurück. Der Parameter *number* wird dabei der Methode `timeit` bei jedem Aufruf übergeben.

Hinweis

Es ist normalerweise keine gute Idee, den Mittelwert aller von repeat zurückgegebenen Werte zu bilden und diesen als durchschnittliche Laufzeit auszugeben. Andere Prozesse, die auf Ihrem System laufen, verfälschen die Ergebnisse aller Messungen. Vielmehr sollten Sie den kleinsten Wert der zurückgegebenen Liste als minimale Laufzeit annehmen, da dies die Messung mit der geringsten Systemaktivität war.

t.print_exc([file])

Sollte im zu analysierenden Code eine Exception geworfen werden, wird die Analyse sofort abgebrochen und ein Traceback ausgegeben. Der Stacktrace dieses Tracebacks ist jedoch nicht immer optimal, da er sich nicht auf den tatsächlich ausgeführten Quellcode bezieht.

Um einen aussagekräftigeren Stacktrace auszugeben, kann eine geworfene Exception abgefangen und die Methode `print_exc` aufgerufen werden. Diese Methode gibt einen Traceback auf dem Bildschirm aus, der sich direkt auf den zu analysierenden Code bezieht und damit die Fehlersuche erleichtert.

Durch Angabe des optionalen Parameters *file* lässt sich die Ausgabe in eine beliebige Datei umleiten.

Beispiel

Eingangs wurde erwähnt, dass wir das Modul `timeit` dazu verwenden werden, zu prüfen, um wie viel Prozent die iterative Fakultätsberechnung schneller ist als die rekursive.

Dazu binden wir zunächst das Modul `timeit` ein und implementieren die beiden Berechnungsfunktionen.

```
import timeit

def fak1(n):
    res = 1
    for i in xrange(2, n+1):
        res *= i
    return res

def fak2(n):
    if n > 0:
        return fak2(n-1)*n
    else:
        return 1
```

Danach wird für beide Funktionen jeweils eine Instanz der Klasse `Timer` erzeugt:

```
t1 = timeit.Timer("fak1(50)", "from __main__ import fak1")
t2 = timeit.Timer("fak2(50)", "from __main__ import fak2")
```

Beachten Sie, dass wir im Setup-Code zunächst die gewünschte Berechnungsfunktion aus dem Namensraum des Hauptprogramms `__main__` in den Namensraum des zu testenden Programms importieren müssen. Im eigentlich zu analysierenden Code wird nur noch die Berechnung der Fakultät von 50 unter Verwendung der jeweiligen Berechnungsfunktion angestoßen.

Schlussendlich wird die Laufzeitmessung mit 1.000.000 Wiederholungen gestartet und das jeweilige Ergebnis ausgegeben:

```
print "Iterativ: ", t1.timeit()
print "Rekursiv: ", t2.timeit()
```

Die Ausgabe des Programms lautet:

```
Iterativ:  16.1009089947
Rekursiv:  28.7318170071
```

Das würde bedeuten, dass der iterative Algorithmus um ca. 40 % schneller ist als der rekursive. Doch diese Daten sind noch nicht wirklich repräsentativ, denn es könnte sein, dass der Test der rekursiven Funktion durch einen im System laufenden Prozess ausgebremst wurde. Aus diesem Grund starten wir einen erneuten Test:

```
print "Iterativ: ", min(t1.repeat(100, 10000))
print "Rekursiv: ", min(t2.repeat(100, 10000))
```

Dieses Mal führen wir eine Testreihe durch, die einen Test mit 10.000 Einzelwiederholungen 100-mal wiederholt und das kleinste der Ergebnisse ausgibt. Die Ergebnisse sind, relativ gesehen, annäherungsweise deckungsgleich mit denen der vorherigen Tests:

```
Iterativ:  0.162111997604
Rekursiv:  0.272562026978
```

Beachten Sie, dass die absoluten Zahlenwerte sehr stark vom verwendeten System abhängen. Auf einem schnelleren Computer sind sie dementsprechend kleiner.

top

21.7.2 Profiling – cProfile

Um eine Laufzeitanalyse eines vollständigen Python-Programms anzufertigen, wird ein sogenannter *Profiler* verwendet. Ein Profiler überwacht einen kompletten Programmdurchlauf und listet nach Beenden des Programms detailliert auf, wie viel Prozent der Laufzeit beispielsweise in welcher Funktion verbraucht wurden. Auf diese Weise kann der Programmierer die laufzeittechnischen Engpässe des Programms erkennen und an sinnvollen Stellen mit der Optimierung des Programms beginnen.

Grundsätzlich gilt: Je mehr Prozent der Laufzeit in einer bestimmten Funktion verbracht werden, desto mehr Zeit sollte man investieren, um diese Funktion zu optimieren. Dagegen wäre es Zeitverschwendung, stundenlang eine Funktion zu optimieren, die vielleicht nur einmal zur Initialisierung des Programms aufgerufen wird.

Seit Python Version 2.5 ist in der Standardbibliothek ein neuer Profiler namens `cProfile` enthalten. Dieser bildet die Schnittstelle des alten Profilers `profile` ab, ist jedoch im Gegensatz zu diesem in C statt in Python geschrieben. Aus diesem Grund ist der Overhead von `cProfile` kleiner, und die Zeitmessungen sind somit besser. Wir werden hier den Profiler `cProfile` besprechen. Da dieser jedoch über die gleiche Schnittstelle wie `profile` verfügt, gilt die Beschreibung genauso für den alten Profiler.

Beachten Sie, dass der Profiler `cProfile` möglicherweise nicht für alle Python-Interpreter verfügbar ist. Das reine Python-Pendant `profile` hingegen kann überall verwendet werden.

Verwendung des Moduls

Im Modul `cProfile` sind im Wesentlichen zwei wichtige Funktionen enthalten, die im Folgenden besprochen werden sollen.

cProfile.run(command[, filename])

Diese Funktion führt den als *command* übergebenen String mithilfe einer exec-Anweisung aus und führt während der Ausführung eine detaillierte Laufzeitanalyse durch. Üblicherweise

wird für *command* ein Funktionsaufruf der Hauptfunktion eines größeren Programms übergeben.

Über den zweiten, optionalen Parameter *filename* kann eine Datei angegeben werden, in die das Ergebnis der Laufzeitanalyse geschrieben wird. Wenn dieser Parameter nicht angegeben wurde, wird das Ergebnis auf dem Bildschirm ausgegeben. Bei diesem Ergebnis der Analyse handelt es sich um eine tabellarische Auflistung aller Funktionsaufrufe. Wie diese Tabelle aussieht und wie sie zu lesen ist, wird im Beispiel-Teil dieses Kapitels geklärt.

cProfile.runctx(command, globals, locals[, filename])

Diese Funktion verhält sich wie `run`, mit dem Unterschied, dass über die Parameter *globals* und *locals* der globale und lokale Kontext festgelegt werden kann, in dem *command* ausgeführt wird. Für die Parameter *globals* und *locals* kann ein Dictionary übergeben werden, wie es von den Built-in Functions `globals` und `locals` zurückgegeben wird.

Beispiel

Im Folgenden soll eine Laufzeitanalyse für ein kleines Beispielprogramm erstellt werden. Dazu betrachten wir zunächst den Quelltext des Programms:

```
import math

def calc1(n):
    return n**2

def calc2(n):
    return math.sqrt(n)

def calc3(n):
    return math.log(n+1)

def programm():
    for i in xrange(100):
        calc1(i)
        for j in xrange(100):
            calc2(j)
            for k in xrange(100):
                calc3(k)

programm()
```

Im Programm existieren drei kleine Funktionen namens `calc1`, `calc2` und `calc3`, die jeweils eine ganze Zahl als Parameter übergeben bekommen, dann eine mathematische Operation auf dieser Zahl anwenden und das Ergebnis zurückgeben. In der Hauptfunktion `programm` befinden sich drei ineinander verschachtelte Schleifen, die jeweils über alle ganzen Zahlen von `0` bis `99` iterieren und eine der drei Berechnungsfunktionen aufrufen. Die Frage, die wir mithilfe des Profilers lösen möchten, lautet, an welcher Stelle sich eine Optimierung des Programms besonders lohnen würde und wo sie überflüssig wäre.

Der Profiler wird folgendermaßen in das Programm eingebunden:

```
import cProfile
[…]
cProfile.run("programm()")
```

wobei die Auslassungszeichen für den Code des Beispielprogramms stehen. Beachten Sie, dass die Codezeile `programm()` des Beispielprogramms jetzt überflüssig ist. Das Ausführen der Laufzeitanalyse gibt folgendes Ergebnis aus:

```
         2020103 function calls in 6.115 CPU seconds
Ordered by: standard name
ncalls  tottime percall cumtime percall
filename:lineno(function)
      1 0.000   0.000   6.115   6.115
<string>:1(<module>)
1000000 2.916   0.000   4.467   0.000
programm.py:10(calc3)
```

```
      1 1.603   1.603   6.115   6.115
programm.py:13(programm)
    100 0.000   0.000   0.000   0.000
programm.py:4(calc1)
  10000 0.030   0.000   0.045   0.000
programm.py:7(calc2)
1000000 1.551   0.000   1.551   0.000   {math.log}
  10000 0.015   0.000   0.015   0.000   {math.sqrt}
      1 0.000   0.000   0.000   0.000   {method 'disable'
of
                                        '_lsprof.Profiler'
objects}
```

Jede Zeile dieser Tabelle bezieht sich auf eine Funktion des Beispielprogramms. Die Spaltenbeschriftungen der Tabelle sind vielleicht nicht ganz klar, weswegen sie kurz erläutert werden sollen:

- `ncalls` steht für die Anzahl von Funktionsaufrufen der Funktion.
- `tottime` steht für die Gesamtzeit in Sekunden, die in der Funktion verbracht wurde. Dabei werden Aufrufe von Unterfunktionen nicht einbezogen.
- `percall` steht für den Quotienten von `tottime` und `ncalls`.
- `cumtime` steht für die Gesamtzeit in Sekunden, die in der Funktion verbracht wurde. Dabei werden Aufrufe von Unterfunktionen mit einbezogen.
- `percall` steht für den Quotienten von `cumtime` und `ncalls`.
- `filename:lineno(function)` steht für den Funktionsnamen, inklusive Angabe der Programmdatei und der Zeile, an der die Funktion im Quellcode steht.

Die vom Profiler angezeigte Tabelle gibt einen guten Überblick darüber, wo die zeitkritischen Funktionen des Programms liegen. In diesem Fall sticht die Funktion `calc3` hervor, die insgesamt 1.000.000-mal aufgerufen wird und in der sich satte 73 % der Laufzeit abspielen. Die 10.000-mal aufgerufene Funktion `calc2` macht hingegen nur 0,7 % der Gesamtlaufzeit aus. Die restliche Laufzeit wird, abgesehen von einem verschwindend geringen Prozentsatz in `calc1`, in der Hauptfunktion `programm` verbracht.

Zugegebenermaßen hätte man dieses Ergebnis auch anhand des Programms abschätzen können. Jede Schleife iteriert über 100 Zahlen und ruft in jedem Iterationsschritt »ihre« Funktion auf. Damit wird die innerste Funktion 1003 = 1.000.000-mal aufgerufen. Auch die prozentuale Laufzeit der Funktionen `calc3` und `calc2` liegt in etwa um Faktor 100 auseinander. Etwaige Schwankungen rühren daher, dass unterschiedliche Berechnungen durchgeführt werden (Logarithmusfunktion gegen Wurzelfunktion).

Auch wenn dieses Beispiel etwas künstlich wirkt, kann man die Vorgehensweise auf ein größeres, zeitkritisches Projekt übertragen. Im Falle unseres Beispiels wäre man gut damit beraten, alle Ressourcen in die Optimierung der Funktion `calc3` zu stecken, da diese mit 1.000.000 Aufrufen und 73 % Laufzeitanteil doch stark dominiert.

top

21.7.3 Tracing – trace

Im letzten Abschnitt haben wir besprochen, welche Möglichkeiten Python bietet, ein Programm mithilfe eines Profilers zu untersuchen. Dies funktioniert im besprochenen Beispiel sehr gut, hat aber auch einen großen Nachteil: Der Profiler arbeitet auf der Funktionsebene. Das bedeutet, dass immer nur die Laufzeit ganzer Funktionen gemessen wird. Häufig ist es aber so, dass es auch innerhalb einer größeren Funktion Teile gibt, die laufzeittechnisch gesehen bedeutungslos sind, und welche, die sehr laufzeitintensiv sind. In einem solchen Fall greift man zu einem anderen Hilfsmittel, dem sogenannten *Tracer*.

Ein Tracer, in Python über das Modul `trace` verfügbar, überwacht

einen Programmlauf und registriert dabei, wie oft jede einzelne Codezeile des Programms ausgeführt wurde. Eine solche *Überdeckungsanalyse* wird im Wesentlichen aus zwei Gründen durchgeführt:

- Mithilfe einer Überdeckungsanalyse lassen sich Codezeilen ausfindig machen, die besonders häufig aufgerufen werden und daher möglicherweise besonders laufzeitintensiv sind. Diese Zeilen könnten dann gezielt optimiert werden. Beachten Sie aber, dass ein Tracer nicht die tatsächliche Laufzeit einer Codezeile misst, sondern nur, wie oft diese Zeile im Programmfluss ausgeführt wurde.

- Häufig muss bei sicherheitsrelevanten Programmen eine Überdeckungsanalyse vorgelegt werden, um zu beweisen, dass bei einem Test jede Codezeile mindestens einmal ausgeführt wurde. Auf diese Weise versucht man zu vermeiden, dass beispielsweise der Autopilot eines Flugzeugs ausfällt, weil eine Codezeile ausgeführt wurde, an die man beim Testen der Software nicht gedacht hat.

In diesem Kapitel möchten wir die Überdeckungsanalyse durchführen, um laufzeitkritische Stellen in einem Programm zu identifizieren. Dazu erstellen wir eine leicht modifizierte Version des Beispielprogramms aus dem vorangegangenen Kapitel. »Modifiziert« bedeutet, dass der Code ohne Unterfunktionen geschrieben wurde.

```
import math

def programm():
    for i in xrange(100):
        i**2
        for j in xrange(100):
            math.sqrt(j)
            for k in xrange(100):
                math.log(k+1)
```

Die Überdeckungsanalyse wird mithilfe des Moduls `trace` durchgeführt. Dazu ist folgender zusätzlicher Code nötig:

```
import trace
import sys

tracer = trace.Trace(
    ignoredirs = [sys.prefix, sys.exec_prefix],
    trace = 0)
tracer.run("programm()")

r = tracer.results()
r.write_results(show_missing=True, coverdir="ergebnis")
```

Zunächst wird eine Instanz der Klasse `Tracer` erzeugt. Diese bekommt zwei Schlüsselwortparameter übergeben. Über den Parameter `ignoredirs` wird eine Liste von Verzeichnissen übergeben, deren enthaltene Module nicht in die Überdeckungsanalyse mit einbezogen werden sollen. In diesem Fall möchten wir keine Module der Standardbibliothek übergeben und fügen deshalb die entsprechenden Verzeichnisse `sys.prefix` und `sys.exec_prefix` an. Den zweiten Parameter, `trace`, setzen wir auf `0`, da sonst jede während des Programmlaufs ausgeführte Zeile auf dem Bildschirm ausgegeben wird.

Danach führen wir, analog zum Profiler, die Methode `run` der `Trace`-Instanz aus und übergeben dabei den auszuführenden Python-Code. Nachdem der Tracer durchgelaufen ist, können die Ergebnisse über die Methode `results` der `Trace`-Instanz abgeholt werden. Wir möchten die Ergebnisse in diesem Fall nicht weiterverarbeiten und speichern sie deshalb mithilfe der Methode `write_results` auf der Festplatte. Dabei wird über den Parameter `coverdir` das Unterverzeichnis angegeben, in dem die Ergebnisse gespeichert werden sollen. Wenn für den Parameter `show_missing` `True` übergeben wird, werden Codezeilen, die während des Programmlaufs niemals ausgeführt wurden, mit einem Pfeil gekennzeichnet.

Das Ergebnis wird im Unterordner *ergebnis* als Textdatei mit dem Dateinamen *modulname.cover* abgespeichert, wobei *modulname* durch den Namen Ihres getesteten Moduls ersetzt wird. In unserem Beispiel sieht das Ergebnis folgendermaßen aus:

```
       import trace
>>>>>> import sys

>>>>>> import math
>>>>>> def programm():
  101:     for i in xrange(100):
  100:         i**2
10100:         for j in xrange(100):
10000:             math.sqrt(j)
1010000:             for k in xrange(100):
1000000:                 math.log(k+1)

>>>>>> tracer = trace.Trace(
>>>>>>     ignoredirs = [sys.prefix, sys.exec_prefix],
>>>>>>     trace = 0)
>>>>>> tracer.run("programm()")

>>>>>> r = tracer.results()
>>>>>> r.write_results(show_missing=True,
coverdir="ergebnis")
```

Sie sehen, dass die Ergebnisse zu einer gut lesbaren Datei aufbereitet werden. Im Prinzip ist die Datei in zwei Spalten aufgeteilt: Rechts steht der Quellcode des Programms und links die Anzahl der Aufrufe jeder Codezeile. Die Pfeile in der linken Spalte weisen auf Codezeilen hin, die während des überwachten Programmlaufs niemals ausgeführt wurden. Beachten Sie, dass diese Zeilen natürlich nur nicht ausgeführt wurden, solange die Überwachung des Programms aktiv war.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

22.2 Erstellen von EXE-Dateien – py2exe

Das Thema dieses Kapitels ist eine Erweiterung des Pakets `distutils` namens py2exe. Durch das Modul py2exe (sprich: *py to exe*) wird es möglich, ein Python-Programm zusammen mit dem Python-Interpreter zu einer einzelnen ausführbaren Datei (Windows-EXE) zu schnüren, sodass das fertige Programm auch in Umgebungen ausgeführt werden kann, in denen keine Python-Installation vorhanden ist. Beachten Sie, dass py2exe kein Installer ist, sondern dass das Programm über die resultierende EXE direkt ausführbar ist. Beachten Sie außerdem, dass py2exe eine EXE erzeugt, die sich ausschließlich unter Windows ausführen lässt und dass das Erstellen einer ausführbaren Datei mit py2exe ebenfalls unter Windows geschehen muss.

Das Modul py2exe ist nicht im Lieferumfang von Python enthalten. Sie müssen es von der Website des Open-Source-Projekts unter http://www.py2exe.org herunterladen.

In diesem Kapitel soll ein kleines Beispielprogramm geschrieben werden, das dann mittels py2exe in eine für sich stehende ausführbare Datei verwandelt werden soll. Hier sehen Sie zunächst den Quelltext des Beispielprogramms:

```
import random
import sys

def verwirble_text(text):
    liste = []
    for wort in text.split():
        w = list(wort[1:-1])
        random.shuffle(w)
        w = "".join(w)
        liste.append("".join([wort[0], w, wort[-1]]))
    return " ".join(liste)
```

```
if __name__ == "__main__":
    text = u""" Hallo
    Python erm\u00F6glicht einen glatten \u00DCbergang von
    kleinen, einfachen Scripten zu sehr komplexen
Anwendungen.
    Dieses Buch vermittelt umfassende Python-Kenntnisse
    """
    print verwirble_text(text)
```

Es handelt sich dabei um die Funktion `verwirble_text` des vorherigen Beispielprogramms, die dahingehend erweitert wurde, dass sie automatisch einen »verwirbelten« Beispieltext ausgibt, wenn das Programm ausgeführt wird.

Da py2exe eine Erweiterung des Pakets `distutils` ist, müssen wir auch in diesem Fall die Programmdatei *setup.py* implementieren. Bezogen auf unser obiges Beispielprogramm sieht sie folgendermaßen aus:

```
from py2exe.build_exe import py2exe
from distutils.core import setup

setup(
      […]
      console=["programm.py"]
      )
```

Statt der Auslassungszeichen stehen die üblichen Parameter der Funktion `setup` im Quellcode, über die beispielsweise der Name oder die E-Mail-Adresse des Autors angegeben werden kann. Um das Installationsscript für py2exe nutzbar zu machen, muss durch den Schlüsselwortparameter `console` eine Liste mit mindestens einem Eintrag übergeben werden. Jeder Eintrag steht dabei für den Namen einer Programmdatei, die später zu einem eigenständigen ausführbaren Programm werden soll. In unserem Fall soll dies nur für die Programmdatei *programm.py* durchgeführt werden.

Beachten Sie, dass alle Programmdateien, die für den Schlüsselwortparameter `console` übergeben werden, später als reine Konsolenanwendung enden. Alternativ kann der Parameter `windows` für eine Anwendung mit grafischer Benutzeroberfläche verwendet werden.

Nachdem das Installationsscript fertig ist, kann die ausführbare Datei erzeugt werden. Dazu muss das Installationsscript *setup.py* mit dem Argument py2exe aufgerufen werden:

```
setup.py py2exe
```

Der Rest geschieht automatisch. Nachdem sich das Installationsscript beendet hat, finden Sie im Programmverzeichnis zwei Unterordner namens *build* und *dist*. Der erste Ordner enthält temporäre Dateien von py2exe und kann problemlos gelöscht werden. Im Ordner *dist* ist die fertige Distribution Ihres Python-Programms enthalten. Das beinhaltet nicht nur die ausführbare Datei, in unserem Fall *programm.exe*, sondern noch weitere ausführbare und nicht-ausführbare Dateien, die für das endgültige Programm wichtig sind. So sind beispielsweise der Python-Interpreter in der DLL *Python25.dll* und die benötigten Teile der Standardbibliothek im Archiv *library.zip* ausgelagert.

Hinweis

Im Verzeichnis *dist* finden Sie neben den angesprochenen Dateien vor allem auch die DLL *MSVCR71.dll*. Diese Datei stammt von Microsoft und sollte Ihrem Programm daher nur beiliegen, wenn Sie die Erlaubnis dazu haben.

Neben *MSVCR71.dll* werden noch weitere DLLs benötigt, damit Ihr Programm ausgeführt werden kann. Welche das sind, teilt Ihnen das Installationsscript mit, kurz bevor es sich beendet:

```
*** binary dependencies ***
Your executable(s) also depend on these dlls which are not
included,
you may or may not need to distribute them.
Make sure you have the license if you distribute any of
them, and
make sure you don't distribute files belonging to the
operating
system.
   ADVAPI32.dll - C:\WINDOWS\system32\ADVAPI32.dll
   USER32.dll - C:\WINDOWS\system32\USER32.dll
   SHELL32.dll - C:\WINDOWS\system32\SHELL32.dll
   KERNEL32.dll - C:\WINDOWS\system32\KERNEL32.dll
```

Beachten Sie, dass die hier aufgelisteten DLLs im Gegensatz zu *MSVCR71.dll* auf jeder Windows-Installation verfügbar sein sollten.

Hinweis

Das Modul py2exe arbeitet problemlos mit einem Großteil der Standardbibliothek von Python zusammen. Auch viele Bibliotheken von Drittanbietern funktionieren einwandfrei. Trotzdem sind in einigen Fällen einige Tricks vonnöten, um ein bestimmtes Modul oder Paket mit py2exe lauffähig zu machen. Für solche Fälle listet die Website http://www.py2exe.org unter dem Stichwort »Working with various packages and modules« Tipps für bestimmte Pakete und Module auf.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

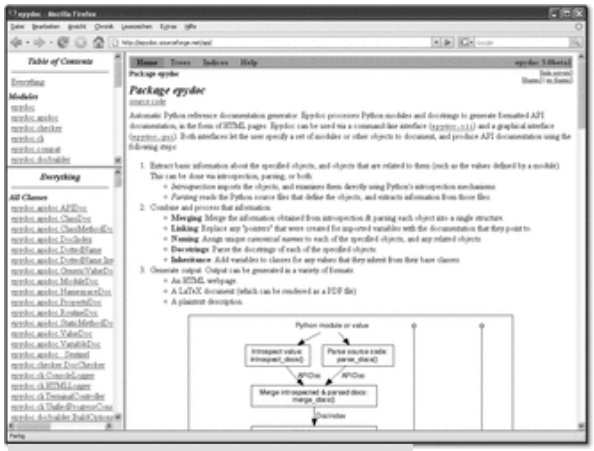
top

22.3 Automatisches Erstellen einer Dokumentation – epydoc

Wenn Sie Ihre Programme und Module an Benutzer und andere Entwickler weitergeben, ist eine gute Dokumentation sehr wichtig. Ohne vernünftige Funktionsbeschreibung ist selbst das beste Programm wertlos, da es zu aufwendig ist, die Funktionsweise allein aus dem Quellcode herauszulesen oder durch Ausprobieren zu erahnen.

Von Programmierern wird das Schreiben von Dokumentationen oft als lästig empfunden, weil es verhältnismäßig viel Zeit in Anspruch nimmt, ohne die Programme selbst zu verbessern. Aus diesem Grund gibt es auch in der Python-Welt viele Module und Programme, die nicht ausreichend dokumentiert sind.

Es gibt allerdings Werkzeuge und Methoden, um das Schreiben von Dokumentationen so einfach wie möglich zu machen. Wir werden uns in diesem Abschnitt mit dem Programm *epydoc* beschäftigen, das Python-Programme anhand ihres Quellcodes dokumentieren kann. Das Werkzeug epydoc analysiert den Programmtext und sammelt insbesondere die Informationen aus den Docstrings. Die gesammelten Informationen werden aufbereitet und als HTML- oder PDF-Datei exportiert.



Hier klicken, um das Bild zu vergrößern

Abbildung 22.1 Beispiel für eine Dokumentation mit epydoc

Das Programm epydoc selbst ist in Python geschrieben und deshalb auf allen Python-Plattformen lauffähig. Sie können die aktuellste Version von der Projekt-Homepage unter http://epydoc.sourceforge.net herunterladen. [Falls Sie Linux einsetzen, lohnt sich außerdem ein Blick in die Paketliste Ihrer Distribution. Es ist recht wahrscheinlich, dass dort bereits ein Paket mit epydoc bereitsteht.]

Benutzung von epydoc

Nach der Installation können Sie epydoc per Konsole aufrufen, wobei der Befehl zum Aufruf unter Unix »epydoc« und unter Windows »epydoc.py« lautet. Wir werden im weiteren Verlauf des Kapitels die Unix-Variante verwenden. Windows-Benutzer müssen beim Nachvollziehen der Aufrufe ein ».py« an »epydoc« anhängen.

Zu epydoc existiert außerdem eine grafische Benutzeroberfläche, die aber aufgrund ihrer eingeschränkten Möglichkeiten nicht besprochen wird.

Im einfachsten Fall ruft man epydoc mit den Modulen als Parameter auf, die man dokumentieren möchte. Voraussetzung für eine erfolgreiche Dokumentationsgenerierung ist, dass die übergebenen Module von Python importiert werden können. Die Module müssen sich also unter dem Pfad befinden, zu dem epydoc ausgeführt wird, oder Standardmodule sein. Als Beispiel erzeugen wir die Dokumentation des Standardmoduls `time`:

```
$ epydoc time
```

Nach dem Aufruf gibt es einen neuen Ordner namens *html* im aktuellen Arbeitsverzeichnis, der die Dokumentation des Moduls `time` im HTML-Format enthält.

Um die Erzeugung der Dokumentation anzupassen, kann man epydoc eine Reihe weiterer Parameter übergeben, von denen die folgende Tabelle die drei wichtigsten zeigt:

Parameter	Beschreibung
`--html`	Erzeuge die Dokumentation im HTML-Format (Standardeinstellung)
`--pdf`	Generiere eine PDF-Datei, die die Dokumentation enthält.
`--output` *DIR*	Schreibe die Dokumentation in das Verzeichnis `DIR`. Standardmäßig werden je nach Format die Ordnernamen *html* oder *pdf* verwendet.

Tabelle 22.3 Die drei wichtigsten Parameter von epydoc

Um beispielsweise die Dokumentation des Moduls `time` als PDF im Verzeichnis *time_dokumentation* zu erzeugen, dient folgender Aufruf:

```
$ epydoc --pdf --output time_documentation time
```

22.3.1 Docstrings und ihre Formatierung für epydoc

Die wichtigste Informationsquelle für epydoc stellen die Docstrings dar. [Wir haben Docstrings in Abschnitt 13.3 behandelt.] Allerdings gibt es in Python keine Docstrings für Variablen und Klassenmember, die aber oft auch dokumentiert werden müssen. Deshalb führt epydoc zwei Notationen ein, um auch solche Elemente mit Dokumentation versehen zu können:

1. Steht in der Zeile hinter einer Wertezuweisung ein String-Literal, so wird dieses als Docstring für die Variable interpretiert, der der Wert zugewiesen wurde.
2. Stehen unmittelbar vor einer Wertezuweisung Kommentare, die durch die Zeichenfolge `#:` eingeleitet werden, interpretiert epydoc die Kommentare als Docstrings für die Variable, der der Wert zugewiesen wurde.

Damit kann man die Attribute `X` und `Y` der Klasse `A` wie folgt dokumentieren:

```
class A(object):
    X = 10
    """ A.X speichert eine Zahl und wird mit 10
initialisiert """

    #: Der Klassenmember A.Y speichert eine Zahl und wird
    #: mit 1337 initialisiert
    Y = 1337

    pass
```

In diesem Beispiel würde epydoc das Klassenattribut `A.X` mit dem Docstring `"A.X speichert eine Zahl und wird mit 10 initialisiert"` verknüpfen und `A.Y` den Docstring `"Der Klassenmember A.Y speichert eine Zahl und wird\nmit 1337 initialisiert"` zuordnen. [Der Zeilenvorschub im Docstring von `A.Y` bleibt erhalten, wie das »\n« andeutet.]

Die Sprache Epytext

Damit epydoc die Docstrings richtig interpretiert und im Ergebnis auch so angezeigt wie von Ihnen gewünscht, gibt es einige Regeln, die Sie beim Schreiben von Docstrings beachten sollten, wenn Sie epydoc für die Dokumentationsgenerierung verwenden möchten.

Diese Regeln werden von der epydoc-spezifischen, sehr einfachen Beschreibungssprache *Epytext* festgelegt. Epytext soll möglichst intuitiv verwendbar sein und ist deshalb relativ elementar gehalten.

Jeder Docstring besteht gemäß Epytext aus mehreren Blöcken, die durch Leerzeilen voneinander getrennt sind. Bei der Interpretation durch epydoc werden innerhalb eines Blocks Leerzeilen, Zeilenvorschübe und sonstige Whitespaces zu einzelnen Leerzeichen zusammengefasst.

Die folgende Tabelle zeigt eine Funktion namens `test`, die in ihrem Docstring drei Textblöcke enthält:

Python-Code	Generierte Dokumentation

Python-Code	Generierte Dokumentation

`def test():`
`  """`
`  Dies ist der erste Block,`
`  der in der Ausgabe vom nachstehenden abgegrenzt wird.`
 	
`  Ein weiterer Block.`
`  """`
`  pass`
	Dies ist der erste Block, der in der Ausgabe vom nachstehenden abgegrenzt wird.
	
	Ein weiterer Block.

Tabelle 22.4 Docstring mit zwei Blöcken

Wichtig ist, dass Epytext genau wie Python selbst die Einrückung benutzt, um zu entscheiden, welche Zeilen zu einem Block gehören. Deshalb müssen im letzten Beispiel die vier Zeilen des ersten Blocks alle auf einer Einrücktiefe stehen.

Man kann Blöcke ineinander verschachteln, indem man die Einrücktiefe vergrößert:

Python-Code	Generierte Dokumentation

`def test2():`
`  """`
`  Dieser Block hat einen untergeordneten Block.`
 	
`  Ich bin untergeordnet.`
 	
`  Ich nicht.`
`  """`
`  pass`
	Dieser Block hat einen untergeordneten Block.
	
	Ich bin untergeordnet.
	
	Ich nicht.

Tabelle 22.5 Verschachtelung bzw. Unterordnung von Blöcken

Neben einfachen Textblöcken gibt es andere Blöcke, die durch spezielle Zeichen am Anfang eingeleitet werden. Die folgende Tabelle gibt eine kurze Übersicht über eine Auswahl der unterstützten Blocktypen:

Zeichen	Blocktyp
-	Listenelement ohne Nummerierung
1.	Nummeriertes Listenelement. Anstelle der »1« können beliebige Zahlen stehen. Auch Punkte können zur Gliederung der Liste verwendet werden, z. B. »1.3.2.«
>>>	Doctest-Block. Damit werden Tests des Doctest-Moduls eingeleitet (siehe hierzu Abschnitt 21.5.1).
@	Felder, die bestimmte Informationen zu Parametern enthalten

Tabelle 22.6 Übersicht über einige der Blocktypen von Epytext

Listen

Listenelemente können mehrere Blöcke und auch weitere Listen enthalten. Alle in einer Liste enthaltenen Blöcke müssen genauso tief wie oder tiefer als das einleitende Zeichen eingerückt werden:

Python-Code	Generierte Dokumentation

`def test3():`
`  """`
`  1. Erstes Listenelement einer`	

```
geordneten
Liste.

    2. Noch
ein Element.
Man
    beachte
die
Einruecktiefe.

    – Eine
Liste ohne

Nummerierung.
    –
Listenelemente
muessen
        nicht
zwingend durch

Leerzeilen
voneinander

getrennt
werden.
    """
    pass
```

1. Erstes Listenelement einer geordneten Liste.

2. Noch ein Element. Man beachte die Einruecktiefe.

3. Das letzte Element der Liste.

– Eine Liste ohne Nummerierung.

– Listenelemente muessen nicht zwingend durch Leerzeilen voneinander getrennt werden.

Tabelle 22.7 Ein Beispiel mit Listen

Listen können genau wie Textblöcke ineinander verschachtelt werden. Dabei müssen sie tiefer als der übergeordnete Block eingerückt sein.

Doctest-Abschnitte

Doctest-Blöcke werden durch drei Größerzeichen, >>>, eingeleitet und müssen von den umliegenden Blöcken durch Leerzeilen getrennt werden. Außerdem dürfen sie selbst keine Leerzeilen enthalten. Weitere Informationen zu Doctest finden Sie in Abschnitt 21.5.1.

Felder

Sogenannte *Felder* dienen dazu, spezielle Informationen wie Parameterbeschreibungen anzugeben. Sie werden von einem @ eingeleitet, das von einem Schlüsselwort gefolgt wird. Dieses Schlüsselwort gibt an, welche Information der folgende Block enthält. Die Zeichenfolge `@param` leitet beispielsweise eine Parameterbeschreibung ein.

Wir beschränken uns hier aufgrund der Fülle von Schlüsselwörtern auf die wichtigsten: `param`, `type` und `return`. Die Verwendung dieser Felder zeigt das folgende Beispiel:

Python-Code	Generierte Dokumentation

`def quadrat(x):`
`  """`
`  @param x: Gibt den Zahlenwert an, dessen Quadrat bestimmt werden soll.`
`  @type x: int oder float`
`  @return: Gibt das Quadrat von x zurueck.`
`  """`
`  pass`
	Parameters:
	
	x – Gibt den Zahlenwert an, dessen Quadrat bestimmt werden soll.
	
	(type=int oder float)
	
	Returns:
	
	Gibt das Quadrat von x zurueck.

Tabelle 22.8 Ein Beispiel, das Felder nutzt.

Sonstige Formatierungsanweisungen

Neben den bisher besprochenen eher strukturellen Formatierungen kann auch das Aussehen der Ausgabe direkt verändert werden. Dazu zählt beispielsweise das Verändern des Schriftbilds zu Fett oder Kursiv. Da diese Art der Formatierung innerhalb von beliebigen Blöcken vorkommen kann, wird sie *Inline Markup* (dt. *Eingebettete Auszeichnung*) genannt.

Inline Markup hat immer die Form `x{...}`, wobei `x` ein einzelner Großbuchstabe ist, der die Art der Auszeichnung angibt, und `...` den zu formatierenden String symbolisiert.

Beispielsweise steht der Buchstabe `I` für kursiven Text (von engl. *italic*). Die Inline-Markup-Anweisung `I{Hallo Welt}` gibt also den String `Hallo Welt` kursiv aus.

In der nachstehenden Tabelle sind die wichtigsten Buchstaben für das Inline Markup aufgeführt:

Buchstabe	Bedeutung
I	Kursiver Text
B	Fetter Text
C	Quellcodegerechte Formatierung. Schrift mit gleicher Breite für alle Zeichen.
M	Mathematische Ausdrücke
U	Interpretiert den Inhalt der Klammern als URL und erzeugt in der Ausgabe einen Hyperlink zu dieser Adresse
E	Escaping – Es verhindert, dass das Zeichen zwischen den geschweiften Klammern als Epytext-Anweisung interpretiert wird. Steht mehr als ein Zeichen zwischen den Klammern, wird dies als kodiertes Sonderzeichen aufgefasst. Mit »E{rb}« und »E{lb}« lassen sich Beispielsweise die geschweiften Klammern } und { kodieren.

Tabelle 22.9 Auswahl von Inline-Markup-Arten

Insbesondere der Buchstabe `E` ist sehr wichtig, um zu verhindern, dass Teile von Docstrings, die zufällig gültige Epytext-Anweisungen ergeben, nicht fälschlicherweise als solche interpretiert werden. Das folgende Beispiel zeigt einen Docstring, der ohne Escaping ein unbeabsichtigtes Listenelement hätte:

```
def funktion():
    """
    Eine komplett sinnlose Funktion, die
    E{-} wenn sie richtig angewendet wird -
    vielleicht doch einmal von Nutzen sein
    kann...
    """
    return 10
```

Stünde anstelle von `E{-}` ein einfacher Bindestrich (`-`), würde die zweite Zeile des Docstrings als Element einer Liste ohne Nummerierung interpretiert.

Ausblick

Das Programm epydoc ist sehr mächtig und umfangreich. Dieses Kapitel sollte Ihnen einen kleinen Einblick in seine Funktionalität gewähren, wobei jedoch nicht alle Aspekte berücksichtigt werden konnten.

Wenn Sie sich ausführlich mit epydoc beschäftigen möchten, empfehlen wir ihnen die sehr gute Dokumentation auf der Homepage. Diese finden Sie unter der folgenden Adresse:

http//epydoc.sourceforge.net

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.2 Strings

Bei der Arbeit mit Strings sollten Sie immer im Hinterkopf behalten, dass Strings *immutable* sind. Das bedeutet, dass bei jeder Veränderung eines Strings eine neue Instanz des Datentyps `str` erzeugt werden muss. Dieses Verhalten lässt sich nicht ändern, was sich aber erreichen lässt, ist, dass möglichst wenig neue Instanzen erzeugt werden müssen. Betrachten Sie dazu folgendes Beispiel, in dem wir alle in einer Liste enthaltenen Strings durch eine Funktion `f` schicken möchten. Die Rückgabewerte all dieser Funktionsaufrufe sollen zu einem einzigen langen String zusammengefasst werden. Ganz blauäugig würden wir das Problem erst einmal folgendermaßen lösen:

```
def f(s):
    return s.upper()

alle_strings = ["Hallo Welt"]*200000
string = ""
for s in alle_strings:
    string += f(s)
```

Zunächst wird die Funktion `f` definiert, die den übergebenen String `s` in Großbuchstaben konvertiert und zurückgibt. Beachten Sie, dass diese Funktion zunächst nicht zur Debatte stehen soll, sondern dass wir uns auf den nun folgenden Code konzentrieren möchten. In diesem wird über eine Liste von 200.000 Strings iteriert. In jedem Iterationsschritt wird die Funktion `f` für den aktuellen String aufgerufen und das Ergebnis an den String `string` gehängt. Bei jedem Anhängen eines neuen Strings wird eine neue Instanz des Datentyps `str` erzeugt, was laufzeittechnisch gesehen glatter Wahnsinn ist.

Das ständige Erzeugen neuer `str`-Instanzen kann unterbunden werden, indem die von `f` zurückgegebenen Strings zunächst in einem veränderlichen Datentyp, beispielsweise einer Liste, zwischengespeichert und erst nach Durchlauf der Schleife in einer einzelnen Operation zusammengefügt werden. Dazu kann die String-Methode `join` verwendet werden:

```
alle_strings = ["Hallo Welt"]*200000
lst = [None]*len(alle_strings)
for i in xrange(len(alle_strings)):
    lst[i] = f(alle_strings[i])
string = "".join(lst)
```

In diesem Beispiel wird zunächst eine Liste erzeugt, die ebenso viele Einträge umfasst wie die Liste `alle_strings`. In jedem Element der neuen Liste `lst` wird `None` referenziert. Das ist besonders günstig, da von `None` immer nur eine Instanz existiert und somit alle Elemente der Liste auf dieselbe Instanz verweisen. Damit wird keine Laufzeit zur Erzeugung überflüssiger Instanzen verschwendet. Im zweiten Schritt werden in einer `for`-Schleife alle ganzzahligen Indizes zwischen `0` und `199.999` durchlaufen und das jeweilige Element der Liste `alle_strings` durch die Funktion `f` geschickt und in die neue Liste `lst` eingetragen. Das Eintragen in die neue Liste ist besonders günstig, da nur Elemente ersetzt werden müssen, sich die Länge der Liste also nicht verändert. In der letzten Zeile des Beispiels werden schlussendlich alle Elemente der neu erzeugten Liste `lst` durch Aufruf der String-Methode `join` zu einem großen String zusammengefasst. Wenn man die Laufzeit der beiden Beispiele vergleicht, stellt man fest, dass die Laufzeit des unteren Beispiels um 20 % geringer ist als die des oberen.

Beachten Sie, dass sich dieser Optimierungsansatz nur bei wirklich großen Listen auszahlt und bei kleinen sogar eher kontraproduktiv ist. Dennoch gilt die Maxime, mehrere Strings nicht mit dem Operator + zu verbinden. So könnte der Code

```
string = a + b + c + d
```

mit den Strings a, b, c und d viel effizienter in dieser Form geschrieben werden:

```
string = "%s%s%s%s", (a, b, c, d)
```

da in diesem Fall nur eine statt drei neuer Instanzen für die jeweiligen Zwischenergebnisse der Addition erzeugt werden muss.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.3 Funktionsaufrufe

Damit ist das obige Beispiel aber noch keineswegs vollständig optimiert, denn auch Funktionsaufrufe sind vergleichsweise laufzeitintensiv. Aus diesem Grund sollten Sie Funktionsaufrufe in häufig durchlaufenen Schleifen stets gründlich auf ihre Notwendigkeit prüfen. In diesem Fall wäre es am besten, die Funktion zu »*inlinen*«, das bedeutet, den Funktionsinhalt direkt in die Schleife zu schreiben:

```
alle_strings = ["Hallo Welt"]*200000
lst = [None]*len(alle_strings)
for i in xrange(len(alle_strings)):
    lst[i] = alle_strings[i].upper()
string = "".join(lst)
```

In diesem Fall wurde auf den Funktionsaufruf verzichtet und der Inhalt der Funktion direkt in die Schleife geschrieben. Die Laufzeit dieses Beispiels ist noch einmal um ca. 25 % geringer als die des optimierten Beispiels aus dem vorherigen Abschnitt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.4 Schleifen

Rekapitulieren wir noch einmal, was im Beispielprogramm abläuft: Im Prinzip wird über die Liste `alle_strings` iteriert, und aus den in `alle_strings` enthaltenen Elementen wird eine neue Liste `lst` erzeugt. Das klingt doch nach einem Paradebeispiel für List Comprehensions. In der Regel können List Comprehensions schneller ausgeführt werden als eine vergleichbare `for`-Schleife. Die letzte Version des Beispielprogramms lässt sich mithilfe einer List Comprehension folgendermaßen formulieren:

```
alle_strings = ["Hallo Welt"]*200000
string = "".join([s.upper() for s in alle_strings])
```

Diese Herangehensweise an das Problem ist nicht nur sehr elegant, sondern wird auch um ca. 15 % schneller ausgeführt als die vorangegangene Version des Beispiels.

Insgesamt läuft die in diesem Kapitel vorgestellte Version des Programms also um ca. 50 % schneller als der erste Versuch, das Problem anzugehen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.6 Lookup

Wenn über einen Modulnamen auf eine Funktion zugegriffen wird, die in diesem Modul enthalten ist, muss bei jedem Funktionsaufruf ein sogenannter *Lookup* durchgeführt werden. Dieser Lookup muss nicht durchgeführt werden, wenn eine direkte Referenz auf das Funktionsobjekt besteht. Stellen Sie sich einmal vor, Sie wollten die Quadratwurzeln aller natürlichen Zahlen zwischen 0 und 100 bestimmen. Dazu kommt einem zunächst folgender Ansatz in den Sinn:

```
import math
for i in xrange(100):
    math.sqrt(i)
```

Wesentlich effizienter ist es jedoch, die Funktion `sqrt` des Moduls `math` direkt zu referenzieren und über diese Referenz anzusprechen:

```
import math
s = math.sqrt
for i in xrange(100):
    s(i)
```

Die Schleife der zweiten Variante kann um ca. 25 % schneller ausgeführt werden als die Schleife der ersten.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.7 Lokale Referenzen

Der nächste Optimierungsansatz ist in gewisser Weise ähnlich zu dem des vorherigen Kapitels: Das Verwenden von lokalen Referenzen ist grundsätzlich schneller als das Verwenden von globalen Referenzen. Das liegt daran, dass jeder verwendete Bezeichner zunächst im lokalen Namensraum vermutet wird. Erst danach wird die Suche auf den globalen Namensraum ausgeweitet.

Dieser Optimierungsansatz soll anhand der folgenden Beispielfunktion veranschaulicht werden, die, analog zum Beispiel des vorherigen Kapitels, das Quadrat aller natürlichen Zahlen von 0 bis 100 berechnet. Beachten Sie, dass die dafür verwendete Built-in Function pow im globalen Namensraum existiert.

```
def f1():
    for i in xrange(100):
        pow(i, 2)
```

Die Funktion f1 dient ausschließlich zu Demonstrationszwecken, weswegen das berechnete Quadrat direkt nach der Berechnung verworfen wird. Dennoch wird bei jedem Aufruf der Built-in Function pow zunächst der lokale Namensraum nach dem Namen pow durchsucht, bevor die Funktion schließlich im globalen Namensraum gefunden wird. Effizienter wäre es also, die Funktion pow durch eine Referenz im lokalen Namensraum direkt zu referenzieren, wie es in der Funktion f2 gemacht wird:

```
def f2():
    p = pow
    for i in xrange(100):
        p(i, 2)
```

Die Funktion f2 kann um ca. 10 % schneller ausgeführt werden als die Funktion f1.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.8 Exceptions

Stellen Sie sich einmal vor, Sie müssten in einer sehr frequentierten Schleife mit einem sich ständig ändernden Index `i` auf eine Liste `liste` zugreifen, können aber nicht sicher sein, ob ein Element `liste[i]` tatsächlich existiert. Wenn ein Element mit dem Index `i` existiert, soll dieses zurückgegeben werden, andernfalls `0`. In einem solchen Fall ist es in der Regel ineffizient, vor dem Zugriff zu prüfen, ob ein `i`-tes Element existiert:

```
def f(liste, i):
    if i in liste:
        return liste[i]
    else:
        return 0
```

In der Regel ist es wesentlich effizienter, einfach auf das `i`-te Element zuzugreifen und im Falle einer geworfenen `IndexError`-Exception den Wert `0` zurückzugeben:

```
def f(liste, i):
    try:
        return liste[i]
    except IndexError:
        return 0
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.9 Keyword arguments

Das Übergeben von Positionsparametern beim Funktions- oder Methodenaufruf ist im Vergleich zu Schlüsselwortparametern grundsätzlich effizienter. Dazu soll folgende Funktion betrachtet werden, die vier Parameter erwartet:

```
def f(a, b, c, d):
    return "%s %s %s %s", (a,b,c,d)
```

Der Funktionsaufruf

```
f("Hallo", "du", "schoene", "Welt")
```

kann fast 50 % schneller ausgeführt werden als der Funktionsaufruf

```
f(a="Hallo", b="du", c="schoene", d="Welt")
```

Beachten Sie, dass dies kein Grund sein sollte, allgemein auf Schlüsselwortparameter zu verzichten, denn die absolute Laufzeitersparnis liegt auch noch bei relativ vielen Funktionsaufrufen in einem nicht wahrnehmbaren Bereich.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

24.2 Einführung in PyQt

Sie haben aus der vorangegangenen Zusammenfassung bereits einen groben Überblick darüber, was das Framework *Qt* (sprich *cute*, dt. *pfiffig*) und damit auch die Python-Bindings *PyQt* ausmacht. Dieser grobe Überblick soll hier nun verfeinert werden, und wir werden die Konzepte und Stärken von Qt vor allem von der technischen Seite her beleuchten.

Dazu beschäftigen wir uns im nächsten Abschnitt zunächst mit der Frage, wo und wie Qt und PyQt bezogen und installiert werden können. Danach geben wir eine Übersicht über die grundlegenden Konzepte des Qt-Frameworks.

top

24.2.1 Installation

Um Qt-Anwendungen mit Python schreiben zu können, müssen zwei Komponenten installiert werden: das Qt-Framework und die zugehörigen Python-Bindings für dieses. Sollten Sie beides noch nicht auf Ihrem PC installiert haben und ein Windows-Betriebssystem einsetzen, können Sie das Komplettpaket von der Website des PyQt-Entwicklers unter http://www.riverbankcomputing.com herunterladen. Eine Qt-Installation ist dann nicht mehr vonnöten. Neben Qt und PyQt ist in diesem Komplettpaket auch die Entwicklungsumgebung *Eric* enthalten, mit der sich sehr komfortabel PyQt-Anwendungen schreiben lassen. Da wir Sie nicht auf eine bestimmte IDE festnageln möchten, werden wir Eric [Weitere Informationen über *Eric* und andere Entwicklungsumgebungen finden Sie im Anhang.] für die folgenden Beispielprojekte nicht verwenden. Alternativ können Sie PyQt von der CD installieren, die diesem Buch beiliegt.

Wenn aus gewissen Gründen das Komplettpaket für Sie nicht in Frage kommt, beispielsweise weil Sie Linux einsetzen, können Qt und PyQt getrennt voneinander installiert werden. Qt kann auf der Website http://www.trolltech.com von Trolltech unter Developer Zone • Downloads • Qt kostenlos heruntergeladen werden. Alternativ können Sie Qt von der CD installieren, die diesem Buch beiliegt.

Beachten Sie noch mal, dass Qt und PyQt unter einem dualen Lizenzsystem stehen. Projekte, die Sie mit den angebotenen freien Versionen von Qt und PyQt entwickeln, dürfen Sie nur unter einer ebenfalls freien Lizenz, beispielsweise der GPL, veröffentlichen. Um ein kommerzielles Programm veröffentlichen zu dürfen, muss eine Lizenzgebühr entrichtet werden.

Hinweis

Sollten Sie während der Installation von Qt eine Warnung bekommen, dass keine MinGW-Installation auf Ihrem Rechner gefunden wurde, können Sie diese ignorieren, sofern Sie ausschließlich mit PyQt arbeiten möchten. MinGW ist eine freie Compilerdistribution für C und C++.

top

24.2.2 Grundlegende Konzepte von Qt

Als Einführung in die Programmierung mit Qt bzw. PyQt soll in

diesem Abschnitt eine Übersicht über die wichtigsten Konzepte und Stärken des Qt-Frameworks gegeben werden.

Umfang

Eine der größten Stärken von Qt ist der Funktionsumfang des Frameworks. Ihnen wird bereits aufgefallen sein, dass im Zusammenhang mit Qt nicht von einem Toolkit, sondern von einem Framework gesprochen wurde. Das hängt damit zusammen, dass der überwiegende Anteil an Klassen, die das Qt-Framework enthält, nichts mit der Programmierung grafischer Benutzeroberflächen zu tun hat, sondern anderweitige nützliche Funktionalität bereitstellt. So enthält das Qt-Framework beispielsweise Klassen zum Arbeiten mit XML-Daten oder zur Netzwerkkommunikation. Viele dieser Klassen sind zwar in Kombination mit Python aufgrund der Standardbibliothek faktisch überflüssig, bieten aber in Programmiersprachen wie C++, das über keine so umfangreiche Standardbibliothek verfügt, einen erheblichen Mehrwert.

Signale und Slots

Einer der größten Unterschiede zu anderen GUI-Toolkits ist das Signal-und-Slot-Prinzip, das Qt zur Kommunikation einzelner Objekte untereinander einsetzt. Bei jedem Ereignis, das in einem Qt-Objekt auftritt, beispielsweise beim Anklicken eines Buttons, wird ein sogenanntes *Signal* gesendet, das dann von verbundenen Slots empfangen werden kann. Diese Form der Kommunikation wird bei anderen Toolkits häufig durch Callback-Funktionen implementiert, was zwar effizienter ist, jedoch auch einige Nachteile mit sich bringt.

Näheres zum Signal-und-Slot-Prinzip erfahren Sie in Abschnitt 24.4.

Layouts

Das Qt-Framework unterstützt sogenannte Layouts in der grafischen Oberfläche eines Programms. Mithilfe eines Layouts lassen sich Steuerelemente relativ zueinander automatisch positionieren. Diese Gruppe von Steuerelementen kann dann gemeinsam verschoben oder in der Größe verändert werden, ohne ihre relativen Positionen zueinander zu verlieren.

Näheres zu Layouts erfahren Sie in Abschnitt 24.3.1.

Zeichenfunktionen

Qt stellt umfangreiche Funktionalität zum Zeichnen in der grafischen Benutzeroberfläche bereit. So erlaubt Qt das Zeichnen verschiedenster Formen mit verschiedensten Arten der Füllung oder des Linienstils. Darüber hinaus bietet Qt Möglichkeiten zur Transformation von Zeichnungen mithilfe einer Transformationsmatrix, was erstaunliche Effekte ermöglicht und wodurch sich Qt von vielen anderen GUI-Toolkits abhebt. Außerdem ist die Integration einer 3D-OpenGL-Szene in eine grafische Benutzeroberfläche mittels Qt sehr einfach. Dadurch könnte beispielsweise ein Anzeigeprogramm für 3D-Modelle sehr komfortabel in Qt geschrieben werden.

Darüber hinaus ermöglicht Qt das Lesen und Schreiben vieler Grafikformate, darunter vor allem auch des Vektorformats SVG.

Näheres zum Zeichnen mittels Qt erfahren Sie in Abschnitt 24.6.

Das Model-View-Konzept

Qt implementiert das sogenannte Model-View-Konzept, das eine Trennung von Form und Inhalt ermöglicht. So ist es mittels Qt möglich, die Daten, die ein Programm verarbeitet, in einer eigenen Klassenstruktur zu speichern – getrennt von der Funktionalität, die diese Daten anzeigt. Auf diese Weise wird der Programmaufbau insgesamt übersichtlicher, und die unabhängige

Weiterentwicklung und Wiederverwertung einzelner Komponenten wird erleichtert.

Näheres zum Model-View-Konzept erfahren Sie in Abschnitt 24.7.

Behindertengerechte Oberflächen

Qt bietet als eines der wenigen Frameworks umfassende Unterstützung zum Schreiben barrierefreier Benutzeroberflächen. So arbeitet Qt mit den Technologien der drei größten Desktop-Betriebssysteme Windows, Linux und Mac OS X zusammen, um Programme mit grafischer Benutzeroberfläche für Menschen bedienbar zu machen, die aufgrund einer Behinderung sonst nicht dazu in der Lage wären.

Dieses Thema ist sehr speziell, weswegen wir im Folgenden nicht näher darauf eingehen werden. Sollten Sie daran interessiert sein, behindertengerechte Qt-Anwendungen zu schreiben, finden Sie Anleitungen dazu auf der Trolltech-Website unter dem Stichwort *Accessibility*.

Tools

Der nächste herausragende Bereich von Qt sind die mitgelieferten Tools. Bei einer Qt-Installation werden für gewöhnlich die Programme *Qt Designer*, *Qt Assistant* und *Qt Linguist* mit installiert. Ersteres ist ein Programm zum komfortablen Editieren einer grafischen Benutzeroberfläche. Wir werden im Laufe dieses Kapitels noch auf den Qt Designer zurückkommen.

Die ebenfalls mit gelieferten Tools Qt Assistant und Qt Linguist werden in diesem Buch nicht besprochen. Es sei nur erwähnt, dass es sich bei Qt Assistant um ein Tool zum Lesen der Qt-Hilfe und bei Qt Linguist um ein Hilfsmittel zur Lokalisierung von Qt-Anwendungen handelt.

Dokumentation

Als letzter Punkt ist die Dokumentation des Qt-Frameworks zu nennen, die entweder Ihrer Qt-Installation beiliegt oder im Internet auf der Website des Qt-Herstellers Trolltech zu finden ist. Die Qt-Dokumentation ist sehr umfangreich und mit vielen Beispielen versehen. Zudem finden Sie zahlreiche Beispielprogramme, die unterschiedliche Anwendungsgebiete von Qt abdecken.

Mit der PyQt-Installation erhalten Sie zudem eine auf Python abgestimmte Version der Qt-Dokumentation, in der alle Schnittstellen und ein Teil der Beispiele für Python beschrieben werden. Auch diese Dokumentation ist mittlerweile umfangreich genug, um mit ihr arbeiten zu können. Beachten Sie, dass in der PyQt-Dokumentation viele Beispiele noch nicht von C++ nach Python übersetzt worden sind.

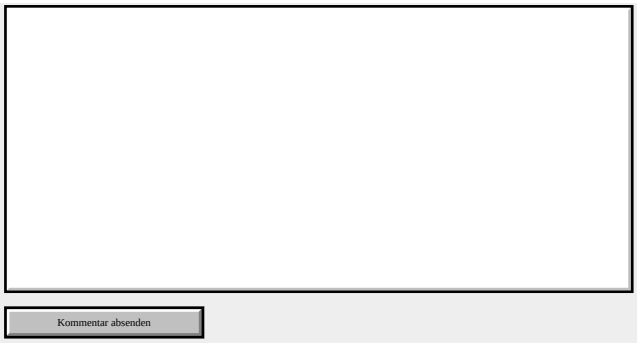
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

24.3 Entwicklungsprozess

In diesem Abschnitt soll der vollständige Entwicklungsprozess einer einfachen PyQt-Anwendung dargestellt werden. Auf dem hier erarbeiteten Wissen werden wir später aufbauen, wenn es an eine komplexere Anwendung geht.

Bei dem Beispielprogramm, das in diesem Kapitel entwickelt wird, handelt es sich um ein Formular, das einige persönliche Daten vom Benutzer einliest. Sie kennen ein solches Formular von anmeldepflichtigen Internetportalen.

top

24.3.1 Erstellen des Dialogs

Bevor wir damit beginnen, das Programm zu schreiben, müssen wir uns Gedanken darüber machen, wie die grafische Oberfläche aussehen soll. Dazu gehört zunächst einmal die Frage, welche Informationen vom Benutzer eingelesen werden sollen.

In diesem Beispiel sollen Vorname, Nachname, Geburtsdatum und Adresse des Benutzers eingelesen und gespeichert werden. Zusätzlich soll der Benutzer die allgemeinen Geschäftsbedingungen unseres Pseudo-Portals akzeptieren müssen und optional einen Warenkatalog bestellen können.

Auf Basis dieser Voraussetzungen können wir im nächsten Schritt einen *Dialog* erstellen. Unter einem Dialog versteht man ein einzelnes Fenster der grafischen Benutzeroberfläche. Häufig besitzt eine Anwendung einen *Hauptdialog*, in dem der Großteil der Kommunikation mit dem Benutzer abläuft.

Der Hauptdialog unserer Beispielanwendung soll so aussehen wie in Abbildung 24.1.

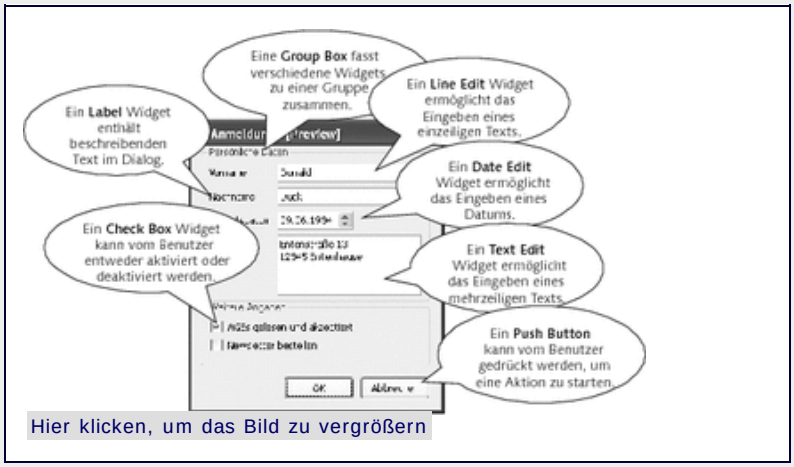
Hier klicken, um das Bild zu vergrößern

Abbildung 24.1 Hauptdialog der Anwendung

Der Hauptdialog enthält mehrere sogenannte *Widgets*. Unter einem Widget (dt. *Dingsbums*) versteht man ein einzelnes Steuer- oder Bedienelement der grafischen Oberfläche. Ein Widget kann beliebig viele untergeordnete Widgets enthalten, sogenannte *Children* (dt. *Kinder*). Im Dialog aus Abbildung 24.1 sind beispielsweise die Eingabefelder, die ihrerseits ebenfalls Widgets sind, dem Gruppierungswidget »Persönliche Daten«

untergeordnet.

Abbildung 24.2 gibt Aufschluss über die Namen und Bedeutungen der einzelnen Widgets, aus denen der Hauptdialog besteht.



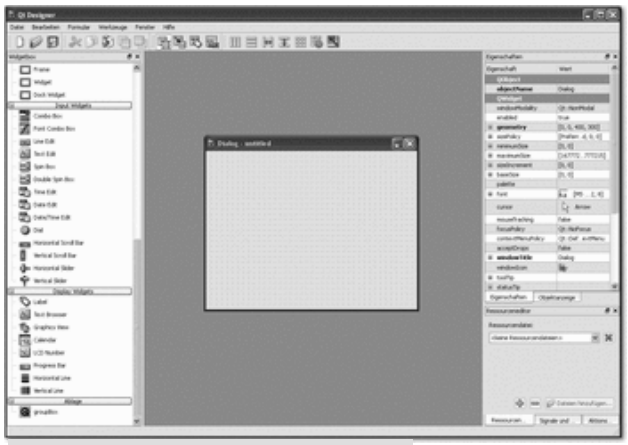
Hier klicken, um das Bild zu vergrößern

Abbildung 24.2 Verschiedene Widgets

Jetzt können wir damit beginnen, den Dialog zu erstellen. Qt stellt dafür ein komfortables Entwicklungswerkzeug, den *Qt Designer*, bereit. Mithilfe des Qt Designers lassen sich auch komplexe Dialoge problemlos editieren.

Unter Windows finden Sie das Programm Qt Designer im Startmenü unter dem Eintrag **Qt** bzw. **PyQt**. Sollten Sie Linux oder Mac OS X einsetzen, müssen Sie den Qt Designer gegebenenfalls über den Befehl `designer` aus einer Shell heraus starten.

Nach dem Starten des Designers wird die Möglichkeit angeboten, ein sogenanntes *Template* zu laden. Wir entscheiden uns in diesem Fall für das Template »Dialog without buttons«, das einen vollständig leeren Dialog bereitstellt. Danach präsentiert sich der Qt Designer so wie in Abbildung 24.3.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.3 Der Qt-Designer

Die Arbeitsfläche des Qt Designers lässt sich grob in drei Bereiche unterteilen:

- In der Mitte finden Sie Ihre Dialogvorlage, die zu diesem Zeitpunkt noch leer ist.
- Auf der linken Seite befindet sich eine Liste aller verfügbaren Steuerelemente. Um eines dieser Steuerelemente zum Dialog hinzuzufügen, wählen Sie es aus und ziehen es auf den Dialog. Nachdem ein Steuerelement zum Dialog hinzugefügt worden ist, kann es beliebig positioniert oder in seiner Größe verändert werden.

- Auf der rechten Seite können die Eigenschaften des aktuell ausgewählten Steuerelements bearbeitet werden. Beachten Sie, dass diese Eigenschaften teilweise sehr speziell sind und in diesem Buch nicht zur Gänze beschrieben werden können.

Wir beginnen damit, die beiden Buttons **OK** und **Abbrechen** im Dialog zu platzieren. Dazu ziehen Sie zwei Buttons aus der Liste am linken Rand und platzieren sie im Dialog. Die Aufschrift der Buttons lässt sich durch einen Doppelklick auf den bisherigen Text verändern. Um die Schaltfläche **OK** zur Standardschaltfläche des Dialogs zu machen, ändern Sie die Eigenschaft »default« des Buttons in der Liste auf der rechten Seite auf »true«.

Danach sollen die Group Boxes zur Gruppierung der weiteren Steuerelemente im Dialog platziert werden. Dazu ziehen Sie zwei Group Boxes auf den Dialog und passen Größe und Position Ihren Wünschen an. Der Titel einer Group Box lässt sich durch einen Doppelklick auf den bisherigen Text anpassen (siehe Abbildung 24.4).

Hier klicken, um das Bild zu vergrößern

Abbildung 24.4 Dialog mit Group Boxes

Jetzt könnten im Prinzip weitere Widgets zum Dialog hinzugefügt werden, indem sie in eine der beiden Gruppen platziert werden. Das macht sie automatisch zu untergeordneten Widgets der jeweiligen Gruppe.

Wenn wir die Eingabefelder jedoch mitsamt erklärendem Text absolut im Dialog positionieren würden, müssten wir jedes Mal alle Steuerelemente anpassen, wenn wir die Größe des Dialogs verändern. Um dieses Dilemma zu vermeiden, unterstützt Qt sogenannte *Layouts*.

Ein Layout ist eine Art Container, in dem mehrere Widgets automatisch angeordnet werden. Um ein Layout zu erstellen, markieren Sie alle Widgets, die dazugehören sollen, und wählen eine dieser drei hervorgehobenen Schaltflächen aus der Toolbar (siehe Abbildung 24.5).

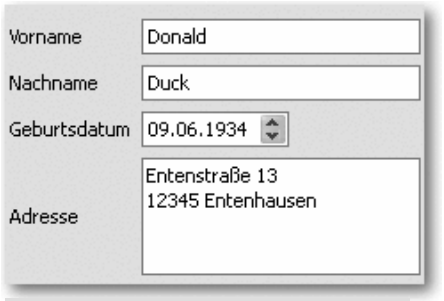
Hier klicken, um das Bild zu vergrößern

Abbildung 24.5 Layout-Toolbar

Ein Klick auf das erste Icon ordnet die Steuerelemente horizontal an, ein Klick auf das zweite Icon vertikal. Die beiden folgenden Icons erstellen einen sogenannten *Splitter* (dt. *Trenner*) zwischen den Steuerelementen, der später vom Benutzer verschoben werden kann. Dies ist für unseren Dialog nicht weiter wichtig und wird deshalb von uns nicht benötigt. Interessant ist allerdings noch die dritte Schaltfläche, die die markierten Steuerelemente

tabellarisch anordnet. Ein Klick auf das letzte Icon löst ein bestehendes Layout auf.

Wenn Sie sich den Inhalt der Group Box »Persönliche Daten« im geplanten Dialog noch einmal ansehen, werden Sie feststellen, dass es sich um einen zweispaltigen tabellarischen Aufbau handelt, bei dem stets eine Beschreibung der anzugebenden Daten links und das entsprechende Eingabefeld rechts steht.



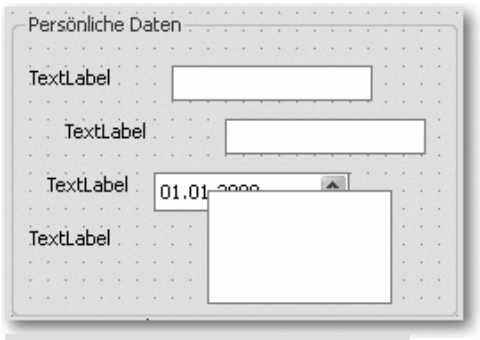
Hier klicken, um das Bild zu vergrößern

Abbildung 24.6 Tabellarischer Aufbau

Um diese Anordnung der Steuerelemente durch ein Layout zu erzielen, fügen wir die Steuerelemente zunächst an ihrem ungefähren Platz in den Dialog ein. Dabei finden Sie die Beschriftungen unter dem Namen *Label* und die Eingabefelder unter den Namen *Edit Box*, *Date Box* und *Text Box* im Steuerelemente-Menü des Qt Designers.

Nachdem die Steuerelemente ungefähr an ihrem Platz sind, markieren wir sie und klicken auf das oben beschriebene Icon in der Toolbar zum Aufbau eines tabellarischen Layouts. Beachten Sie, dass es egal ist, in welcher Reihenfolge die Steuerelemente zuvor markiert wurden, da Qt die Position im Layout anhand ihrer vorherigen Position im Dialog abschätzt. Kleinere Ungenauigkeiten in der Positionierung werden dabei automatisch korrigiert. Das fertige Layout wird durch einen roten Rahmen angezeigt und kann jetzt als Ganzes neu positioniert oder in der Größe verändert werden. Sie werden feststellen, dass alle enthaltenen Steuerelemente bei einer Größenänderung automatisch mit angepasst werden.

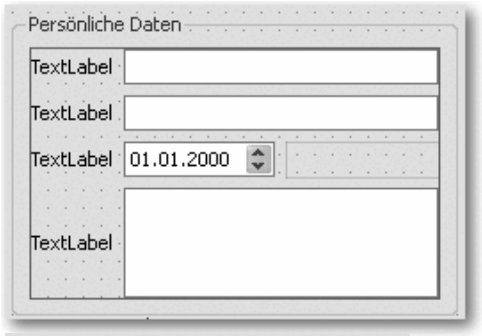
Abbildung 24.7 soll noch einmal veranschaulichen, was das Hinzufügen eines Layouts für das Arrangement der Steuerelemente bedeutet.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.7 Vor dem Festlegen eines Layouts

Wenn jetzt alle Widgets markiert werden und ein tabellarisches Layout ausgewählt wird, ordnet der Qt Designer die Steuerelemente so an, wie es in Abbildung 24.8 zu sehen ist.



Hier klicken, um das Bild zu vergrößern

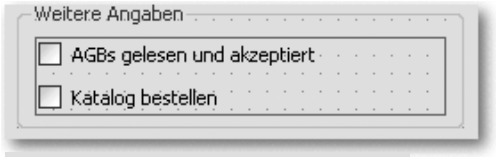
Abbildung 24.8 Nach dem Festlegen eines Layouts

Das Auswählen und Positionieren des Layouts kann etwas »fummelig« sein, da Sie auf den dünnen Rahmen klicken müssen.

Beachten Sie, dass dem Date Edit Widget automatisch nur so viel Platz eingeräumt wurde, wie zum Anzeigen des Datums benötigt wird. Der dahinter eingefügte Leerraum wird durch ein dünnes rotes Rechteck gekennzeichnet. Wenn Sie selbst solche Aussparungen in ein Layout einbauen möchten, können Sie dies mithilfe eines sogenannten *Spacers* tun, den Sie im Steuerelemente-Menü finden. Ein Spacer verhält sich wie ein Widget, wird aber im späteren Dialog nicht angezeigt.

Im nächsten Schritt müssen wir den hinzugefügten Label Widgets einen beschreibenden Text verpassen. Dazu klicken Sie, wie bei einer Group Box, doppelt auf den bisherigen Text.

Ähnlich wie eben besprochen können Sie das Hinzufügen der beiden Check Boxes angehen. Allerdings sollten Sie in diesem Fall ein vertikales Layout wählen (siehe Abbildung 24.9).



Hier klicken, um das Bild zu vergrößern

Abbildung 24.9 Check Boxes im vertikalen Layout

Damit wäre der Dialog äußerlich gestaltet, doch eine wichtige Sache fehlt noch. Jedes Steuerelement benötigt einen Namen, unter dem es nachher im Programm angesprochen werden kann. Dieser Name kann im Qt Designer festgelegt werden, und es ist auch empfehlenswert, das zu tun. Um einem Widget einen neuen Namen zu verpassen, markieren Sie es, öffnen mit einem Rechtsklick das entsprechende Kontextmenü und wählen den Menüpunkt **Objektnamen ändern**.

Die folgende Tabelle listet die im Beispiel vergebenen Namen für alle wichtigen Steuerelemente auf. Steuerelemente, die nur aus Layoutgründen existieren, beispielsweise die Group Boxes und Labels, müssen nicht benannt werden, da später keine Operationen mit ihnen durchgeführt werden sollen.

Steuerelement	Name
Der Dialog selbst	`Hauptdialog`
Das Line Edit Widget »Vorname«	`vorname`
Das Line Edit Widget »Nachname«	`nachname`
Das Date Edit Widget »Geburtsdatum«	`geburtsdatum`
Das Text Edit Widget »Adresse«	`adresse`
Das Check Box Widget »AGB«	`agb`
Das Check Box Widget »Newsletter«	`newsletter`

Das Button Widget »OK«	buttonOK
Das Button Widget »Abbrechen«	buttonAbbrechen

Tabelle 24.1 Die Namen der wichtigen Steuerelemente

Nachdem alle Namen vergeben wurden, ist das Layout des Hauptdialogs fertig und kann im Projektverzeichnis gespeichert werden. Dazu wählen Sie den Menüpunkt DATEI • FORMULAR SPEICHERN. Ein mit dem Qt Designer erstellter Dialog wird als UI-Datei (**.ui*) gespeichert.

Im Falle dieses Beispielprogramms soll der Dialog unter dem Dateinamen *hauptdialog.ui* gespeichert werden.

Die Tabulatorreihenfolge

Ein wichtiges, aber nicht essenzielles Thema ist die sogenannte *Tabulatorreihenfolge*. Bei grafischen Benutzeroberflächen gilt die Maxime, dass ihre Bedienung zwar für die Maus entworfen ist, jederzeit aber auch vollständig über die Tastatur erfolgen kann. Ein wichtiges Instrument dafür ist die Tabulatortaste, mit der der Eingabefokus zwischen den einzelnen Widgets eines Dialogs gewechselt werden kann. Damit dies in einer für den Benutzer nachvollziehbaren Reihenfolge funktioniert, sollten Sie im Qt Designer für jeden Dialog Ihres Projekts die Tabulatorreihenfolge festlegen.

Um die Tabulatorreihenfolge zu editieren, müssen Sie den entsprechenden Modus über das letzte Icon der Werkzeuge-Toolbar aktivieren. Der normale Editiermodus kann durch einen Klick auf das linke Icon wiederhergestellt werden.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.10 Festlegen der Tabulatorreihenfolge

Nachdem Sie in den Modus zur Festlegung der Tabulatorreihenfolge gewechselt sind, klicken Sie in der Reihenfolge auf die Widgets Ihres Dialogs, in der sie nachher beim Drücken der Tabulator-Taste den Eingabefokus bekommen sollen. Die aktuelle Position eines Widgets in der Tabulatorreihenfolge wird durch eine nicht zu übersehende Zahl über dem Widget verdeutlicht. Die Tabulatorreihenfolge unseres Hauptdialogs sieht so aus wie in Abbildung 24.11.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.11 Tabulatorreihenfolge im Beispieldialog

Nachdem auch die Tabulatorreihenfolge angepasst wurde, können wir uns endlich dem Schreiben der Anwendung widmen.

top

24.3.2 Schreiben des Programms

Bevor wir uns an das Schreiben der eigentlichen Anwendung heranwagen können, muss noch ein Schritt erledigt werden. Der gespeicherte Hauptdialog ist in Form einer UI-Datei für unser Beispielprogramm nicht zu gebrauchen. Bevor es also losgeht, muss die UI-Datei in eine Python-Klasse umgewandelt werden. Diese Klasse kann dann im Beispielprogramm eingebunden und instanziiert werden. Das Konvertieren der UI-Datei in Python-Code geschieht mit dem Kommandozeilenprogramm `pyuic4`, das mit PyQt installiert wurde und sich im Hauptverzeichnis von Python befindet.

Das Programm `pyuic4` bekommt den Pfad zu einer UI-Datei als Kommandozeilenparameter übergeben und gibt die daraus erstellte Python-Klasse standardmäßig auf dem Bildschirm aus. Um die Ausgabe in eine Programmdatei umzulenken, kann die Kommandozeilenoption `-o` verwendet werden:

```
C:\Python25\pyuic4 -o hauptdialog.py hauptdialog.ui
```

Der obige Aufruf ist natürlich nur möglich, wenn das Projektverzeichnis als aktuelles Arbeitsverzeichnis ausgewählt ist und Python im Verzeichnis *C:\Python25* installiert wurde. In anderen Fällen müssen Sie die Pfade entsprechend anpassen. Beachten Sie, dass das Programm `pyuic4` bei vielen Linux-Distributionen nach der Installation über das Kommando `pyuic4` direkt gestartet werden kann, also kein Pfad angegeben werden muss.

Nachdem das Programm durchgelaufen ist, befindet sich die Programmdatei *hauptdialog.py* im Projektverzeichnis. Wenn Sie diese Datei öffnen, werden Sie feststellen, dass eine Python-Klasse namens `Ui_Hauptdialog` erzeugt wurde, die in zwei Methoden den Python-Code enthält, der zum Erstellen unseres Hauptdialogs vonnöten ist.

Jetzt sind endlich alle Vorbereitungen erledigt, sodass wir die Programmdatei *programm.py* erstellen und uns ganz dem Programmieren der Beispielanwendung widmen können. Das einfachste Programm, das unseren Hauptdialog einfach nur anzeigt und sonst keine weiteren Operationen durchführt, sieht folgendermaßen aus:

```
import sys
from PyQt4 import QtGui
from hauptdialog import Ui_Hauptdialog as Dlg

class MeinDialog(QtGui.QDialog, Dlg):
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.setupUi(self)

app = QtGui.QApplication(sys.argv)
dialog = MeinDialog()
dialog.show()
sys.exit(app.exec_())
```

Zunächst werden alle benötigten Module eingebunden. Das sind insbesondere die generierte Programmdatei *hauptdialog.py* sowie das Modul `QtGui`, in dem alle Klassen des Qt-Frameworks gekapselt sind, die thematisch mit grafischen Benutzeroberflächen zusammenhängen. Eine Übersicht darüber, welche Namensräume es innerhalb des Pakets `PyQt4` gibt, finden Sie weiter unten.

Danach wird die Klasse `MeinDialog` erstellt, die zum einen von der Basisklasse aller Qt-Dialoge, `QDialog`, und zum anderen von der in *hauptdialog.py* implementierten Klasse `Ui_Hauptdialog` abgeleitet wird. Beachten Sie, dass der Klasse `Ui_Hauptdialog` beim Einbinden das Alias `Dlg` verliehen wurde.

Im Konstruktor der Klasse `MeinDialog` muss der Konstruktor der Klasse `QDialog` aufgerufen werden. Schließlich werden durch

Aufruf der von `Dlg` geerbten Methode `setupUi` alle Steuerelemente des Hauptdialogs initialisiert.

Im nun folgenden Codeblock wird eine Instanz der Klasse `QApplication` erstellt, die den Rahmen einer Anwendung mit grafischer Benutzeroberfläche bereitstellt. Dazu gehört beispielsweise der sogenannte *Main Event Loop*, die Hauptschleife der Anwendung. Dem Konstruktor der Klasse `QApplication` werden die Kommandozeilenparameter `sys.argv` übergeben. Beachten Sie, dass jede Qt-Anwendung immer nur eine einzige `QApplication`-Instanz haben darf, unabhängig davon, wie viele Dialoge später angezeigt werden sollen.

Nachdem der Rahmen für die Qt-Anwendung erstellt worden ist, kann der Hauptdialog erzeugt werden, indem die Klasse `MeinDialog` instanziiert wird. Durch die von der Basisklasse `QDialog` vererbte Methode `show` wird der Dialog sichtbar. Zu guter Letzt muss der eben angesprochene Main Event Loop durch die Methode `exec_` der `QApplication`-Instanz gestartet werden. Da wir im Moment keine weiteren Operationen durchführen möchten, nachdem der Dialog vom Benutzer geschlossen wurde, geben wir den Rückgabewert von `app.exec_` direkt an `sys.exit` weiter und beenden damit das Beispielprogramm. Beachten Sie, dass das Programm nach Aufruf von `app.exec_` in einer Endlosschleife läuft, bis der Benutzer den Hauptdialog schließt.

Dieses zugegebenermaßen ziemlich simple Programm soll im Folgenden sinnvoll erweitert werden. So sollen die vom Benutzer eingegebenen Werte ausgelesen und in das neben der Benutzeroberfläche existierende Konsolenfenster geschrieben werden. Anhand der dazu notwendigen Erweiterungen wird im Folgenden das *Signal-und-Slot*-Konzept von Qt erklärt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

24.4 Signale und Slots

Beim Schreiben eines Programms mit grafischer Benutzeroberfläche wird das Prinzip der ereignisgesteuerten Programmierung angewandt. Dieses Prinzip sieht nicht vor, dass ein Programm sequenziell von oben nach unten abgearbeitet wird, sondern führt beim Auftreten bestimmter Ereignisse einen Codeabschnitt aus, der vom Programmierer für dieses Ereignis vorgesehen wurde. Die Anwendung der ereignisgesteuerten Programmierung ist in diesem Fall notwendig, da im Falle einer grafischen Benutzeroberfläche der Benutzer das Programm steuert und nicht das Programm den Benutzer, wie es bei einer Konsolenanwendung der Fall war. Der Benutzer steuert das Programm durch seine Eingaben, die im Programm in Form von Ereignissen ankommen. Wann und in welcher Reihenfolge der Benutzer seine Eingaben macht, ist durch das Programm nicht vorgegeben.

In Qt finden sich zwei Techniken der ereignisgesteuerten Programmierung: zum einen sogenannte *Events* und zum anderen Signale und Slots. Beide Techniken sollen im Folgenden besprochen werden.

Jedes Widget in der grafischen Benutzeroberfläche wird programmintern durch eine Instanz einer entsprechenden Qt-Klasse repräsentiert. Jede dieser Klassen bietet sogenannte *Eventhandler* an. Das sind Methoden, die der Programmierer in einer abgeleiteten Klasse überschreiben kann, um beim Eintreten eines speziellen Ereignisses (Event) eigenen Code ausführen zu können. Events werden nur für wenige Ereignisse verwendet, die aber häufig eintreten. Ein Beispiel für ein solches Ereignis ist das `paintEvent`, das immer dann eintritt, wenn der Inhalt eines Widgets neu gezeichnet werden muss. Das Widget reagiert auf das Event durch Ausführung seines Eventhandlers. Dies kann unter Umständen sehr häufig passieren. Ein Beispiel für die Implementation eines Eventhandlers finden Sie in Abschnitt 24.6 im Zusammenhang mit der Zeichenfunktionalität von Qt.

Neben den Events bietet das Qt-Framework das Konzept von *Signalen und Slots* für die Behandlung von Ereignissen an. Dieses zentrale Konzept, dessen Ziel es ganz allgemein ist, die Kommunikation von Qt-Objekten, beispielsweise Widgets, untereinander zu gewährleisten, ist womöglich das größte Unterscheidungsmerkmal zwischen Qt und anderen GUI-Toolkits.

Ein *Signal* wird von einem Widget gesendet, wenn bestimmte Ereignisse, beispielsweise eine Benutzereingabe, eingetreten sind. Es gibt für jedes Widget in Qt vordefinierte Signale für die meisten Anwendungsfälle. Es ist jedoch auch möglich, eigene Signale zu selbst bestimmten Ereignissen zu senden. Dazu wird die `emit`-Methode eines Qt-Objekts, beispielsweise also eines Widgets, verwendet.

Um ein Signal zu erhalten, muss ein sogenannter *Slot* (dt. *Steckplatz*) eingerichtet werden. Ein Slot ist eine Funktion oder Methode, die immer dann aufgerufen wird, wenn ein bestimmtes Signal gesendet wird. Dazu muss ein Slot mit einem Signal verbunden werden. Es ist durchaus möglich, einen Slot mit mehreren Signalen zu verbinden.

Im Folgenden wird das Beispiel des letzten Kapitels zu einer sinnvollen Anwendung erweitert. Diese Anwendung soll die Daten, die der Benutzer in den Dialog eingibt, in das parallel geöffnete Konsolenfenster ausgeben, sofern der Benutzer die Eingaben durch Klicken des OK-Buttons bestätigt. Beim Drücken des

Abbrechen-Buttons sollen keine Daten ausgegeben werden.

```
import sys
from PyQt4 import QtGui, QtCore
from hauptdialog import Ui_Hauptdialog as Dlg

class MeinDialog(QtGui.QDialog, Dlg):
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.setupUi(self)

        # Slots einrichten
        self.connect(self.buttonOK,
                QtCore.SIGNAL("clicked()"), self.onOK)
        self.connect(self.buttonAbbrechen,
                QtCore.SIGNAL("clicked()"),
self.onAbbrechen)

    def onOK(self):
        # Daten auslesen
        d = {}
        print "Vorname: %s" % self.vorname.text()
        print "Nachname: %s" % self.nachname.text()
        print "Adresse: %s" % self.adresse.toPlainText()
        datum =
self.geburtsdatum.date().toString("dd.MM.yyyy")
        print "Geburtsdatum: %s" % datum

        if self.agb.checkState():
            print "AGBs akzeptiert"
        if self.newsletter.checkState():
            print "Katalog bestellt"
        self.close()

    def onAbbrechen(self):
        print "Schade"
        self.close()
```

Im Konstruktor der Dialogklasse `MeinDialog` werden mithilfe der von `QDialog` vererbten Methode `connect` zwei Slots angelegt und mit jeweils einem Signal verbunden. Die Methode `connect` bekommt zunächst die Instanz übergeben, von der ein Signal empfangen werden soll. In diesem Fall sind das die Button Widgets `self.buttonOK` und `self.buttonAbbrechen`. Der zweite Parameter spezifiziert, welches Signal empfangen werden soll. Dazu wird die Funktion `QtCore.SIGNAL` mit einer Beschreibung des Signals in Textform aufgerufen. Diese Signalbeschreibung besteht zum einen aus dem Namen des Signals, in diesem Fall `clicked` und zum anderen aus der Parametersignatur des Signals, die in Klammern hinter den Signalnamen geschrieben wird. Da es sich bei Qt eigentlich um eine C++-Bibliothek handelt, müssen die Parametersignaturen in C++-Syntax angegeben werden. Das soll uns aber im Moment noch nicht stören, da die meisten grundlegenden Qt-Signale parameterlos sind. Der dritte und letzte Parameter der Methode `connect` ist das Funktionsobjekt der Methode, die bei Empfangen des entsprechenden Signals aufgerufen werden soll. Die Parametersignatur der Methode muss mit der des Signals übereinstimmen.

Die Methode `onOK` ist ein Slot für das Signal `clicked` des OK-Button-Widgets. Das bedeutet, dass die Methode immer dann aufgerufen wird, wenn der Benutzer auf den OK-Button klickt. Analog dazu wird die Methode `onAbbrechen` aufgerufen, wenn der Benutzer auf den Abbrechen-Button klickt.

In der Methode `onOK` sollen die Eingaben des Benutzers aus den verschiedenen Widgets des Hauptdialogs ausgelesen werden. Jedes dieser Widgets wird durch eine Instanz einer entsprechenden Qt-Klasse repräsentiert. Diese Instanzen werden durch Attribute der automatisch generierten Basisklasse `Dlg` referenziert, deren Namen wir im Qt Designer festgelegt haben. Welches Widget dabei welchen Namen bekam, können Sie in der Tabelle am Ende von Abschnitt 24.3.1 nachlesen. Die Attributnamen sollten allerdings selbsterklärend sein.

Über die angesprochenen Attribute können wir den Inhalt der Steuerelemente auslesen. Wie dies geschieht, ist jedoch von Widget zu Widget verschieden, da sich ja auch die enthaltenen Daten stark unterscheiden. So kann beispielsweise auf den Inhalt eines Line Edit Widgets über die Methode `text` zugegriffen werden. Näheres zu den wichtigsten Widget-Klassen des Qt-Frameworks finden Sie in Abschnitt 24.8. Erwähnenswert ist noch,

dass die Methode `date` der Date-Edit-Instanz `geburtsdatum` das gespeicherte Datum nicht direkt in Form eines Strings, sondern in Form einer `QDate`-Instanz zurückgibt. Diese muss erst durch Aufruf der Methode `toString` in einen String konvertiert werden. Die ausgelesenen Daten werden mithilfe des Schlüsselworts `print` in die Konsole ausgegeben. Zum Schluss, nachdem alle Daten ausgelesen wurden, wird der Dialog durch Aufruf der Methode `close` geschlossen.

Im zweiten Slot, onAbbrechen, sind abgesehen vom Schließen des Dialogs keine weiteren Operationen vonnöten.

```
app = QtGui.QApplication(sys.argv)
dialog = MeinDialog()
dialog.show()
sys.exit(app.exec_())
```

Bei dem Code, der die Applikations- und Dialogklasse instanziiert und die Main Event Loop, startet handelt es sich um denselben, der schon im letzten Beispielprogramm seinen Dienst getan hat.

Hinweis

Wie das Beispiel demonstriert, öffnet auch ein Python-Programm mit grafischer Benutzeroberfläche unter Windows immer noch ein Konsolenfenster, in das mittels `print` geschrieben werden kann.

Das mag in einigen Fällen zwar wünschenswert sein, ist jedoch häufig störend, wenn die Kommunikation mit dem Benutzer vollständig über die grafische Oberfläche ablaufen soll. Wenn Sie nicht wünschen, dass ein Konsolenfenster geöffnet wird, können Sie die Dateiendung der Python-Programmdatei von *.py* nach *.pyw* ändern. Dann werden alle Ausgaben in die Konsole unterdrückt und wird kein Konsolenfenster geöffnet.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

24.6 Zeichenfunktionalität ▼

Nachdem die praxisorientierte Einführung in die Programmierung grafischer Benutzeroberflächen mithilfe des Qt-Frameworks hinter uns liegt, möchten wir uns dem ersten Spezialgebiet von Qt zuwenden: der Zeichenfunktionalität. Wenn Sie ein eigenes Widget erstellen, also eine Klasse definieren, die von einem Steuerelement oder direkt von `QWidget` erbt, haben Sie die Möglichkeit, selbst beliebige Inhalte in das Widget zu zeichnen. Das ist besonders dann interessant, wenn eine Anwendung Inhalte in einem Widget anzeigen möchte, für die es im Qt-Framework keine vorgefertigte Klasse gibt. Das könnte zum Beispiel ein Diagramm oder eine spezifische Grafik sein.

Im Folgenden werden wir uns zunächst mit den Grundlagen des Zeichnens in Qt beschäftigen und danach einige einfache Formen, beispielsweise einen Kreis oder ein Rechteck, auf den Bildschirm bringen.

Die in den folgenden Unterkapiteln präsentieren Beispielklassen verstehen sich im folgenden Kontext:

```
from PyQt4 import QtGui
import sys

class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

app = QtGui.QApplication(sys.argv)
widget = MeinWidget()
widget.resize(150, 150)
widget.show()
sys.exit(app.exec_())
```

Dabei werden wir in den Beispielen jeweils nur die Klasse `MeinWidget` neu implementieren. Um aus diesen Beispielen ein tatsächlich lauffähiges PyQt-Programm zu erstellen, muss die vorgestellte Klasse in den obigen Kontext eingefügt werden. Beachten Sie, dass je nach Beispielprogramm auch noch der Namensraum `QtCore` eingebunden werden muss.

top

24.6.1 Werkzeuge ▼▲

Um innerhalb eines Widgets zeichnen zu können, muss der Eventhandler `paintEvent` implementiert werden. Dabei handelt es sich um eine Methode, die vom Qt-Framework immer dann aufgerufen wird, wenn das Widget teilweise oder vollständig neu gezeichnet werden muss. Das passiert beispielsweise dann, wenn das Anwendungsfenster teilweise verdeckt oder minimiert war und vom Benutzer in den Vordergrund geholt wurde. Die Methode `paintEvent` bekommt eine Instanz der Klasse `QPaintEvent` übergeben, die unter anderem den Bereich des Widgets enthält, der neu gezeichnet werden soll. [In unseren einfachen Beispielen werden wir beim Aufrufen der `paintEvent`-Methode stets die komplette Zeichnung neu zeichnen. Je komplexer und aufwendiger eine Zeichnung jedoch zu zeichnen ist, desto eher sollte man nur den durch die `QPaintEvent`-Instanz spezifizierten Bereich tatsächlich neu zeichnen.]

Innerhalb der `paintEvent`-Methode muss der sogenannte *Painter* (dt. *Maler*) erzeugt werden, mit dessen Hilfe die Zeichenoperationen später durchgeführt werden können. Bei einem Painter handelt es sich um eine Instanz der Klasse `QtGui.QPainter`. Ein grundlegendes Widget, das das Paint-Event

implementiert und einen Painter erzeugt, sieht folgendermaßen aus:

```
class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

    def paintEvent(event):
        painter = QtGui.QPainter(self)
```

Nach dem Erzeugen der `QPainter`-Instanz können mithilfe des Painters beliebige Zeichenoperationen durchgeführt werden.

Zum Zeichnen gibt es in Qt neben dem Painter zwei grundsätzliche Werkzeuge: einen Pen und einen Brush.

Als *Pen* (dt. *Stift*) wird eine Instanz der Klasse `QtGui.QPen` bezeichnet. Um einen Pen zu verwenden, muss dieser dem Painter – also einer `QPainter`-Instanz – mithilfe der Methode `setPen` bekannt gegeben werden. Grundsätzlich wird ein Pen zum Zeichnen von Linien, beispielsweise für Umrandungen bestimmter Figuren, verwendet. Dazu enthält ein Pen im Wesentlichen drei Informationen: die Linienfarbe, die Liniendicke und den Linienstil. Ein Pen wird folgendermaßen erzeugt:

```
pen = QtGui.QPen(QtGui.QColor(255,0,0))
```

Dem Konstruktor des Pens wird eine Instanz der Klasse `QColor` übergeben, um die Farbe des Pens, in diesem Fall Rot [Eine Farbangabe besteht aus drei einzelnen Werten zwischen `0` und `255`. Der erste übergebene Wert spezifiziert den Rot-, der zweite den Grün- und der dritte den Blauanteil der Farbe. Nähere Informationen dazu finden Sie im Internet unter dem Stichwort »RGB«.] , zu spezifizieren. Nachdem ein Pen erzeugt worden ist, kann seine Liniendicke bzw. der Linienstil mithilfe der Methoden `setWidth` und `setStyle` festgelegt werden:

```
pen.setWidth(7)
pen.setStyle(QtCore.Qt.DashLine)
```

Die der Methode `setWidth` übergebene ganze Zahl entspricht der Liniendicke in Pixeln, die eine mit diesem Pen gezeichnete Linie später auf dem Bildschirm haben wird. Der Methode `setStyle` können verschiedene Konstanten übergeben werden, die jeweils einen bestimmten Linienstil vorschreiben. Eine Auswahl dieser Konstanten finden Sie in der folgenden Tabelle:

Konstante	Beschreibung
`QtCore.Qt.SolidLine`	Eine durchgezogene Linie. Dies ist die Standardeinstellung und braucht nicht explizit gesetzt zu werden.
`QtCore.Qt.DashLine`	Eine gestrichelte Linie
`QtCore.Qt.DotLine`	Eine gepunktete Linie
`QtCore.Qt.DashDotLine`	Eine Linie, die abwechselnd gestrichelt und gepunktet ist

Tabelle 24.3 Linienstile eines Pens

Das zweite wichtige Werkzeug zum Zeichnen ist der sogenannte *Brush* (dt. *Pinsel*), mit dessen Hilfe Flächen gefüllt werden. Ein Brush spezifiziert, ähnlich wie ein Pen, zunächst einmal die Farbe, in der eine Fläche gefüllt werden soll. Analog zum Pen wird ein Brush folgendermaßen erzeugt:

```
brush = QtGui.QBrush(QtGui.QColor(0,0,255))
```

Auch dem Konstruktor des Brushs wird der Farbwert, in diesem

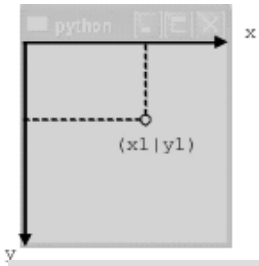
Fall Blau, in Form einer QColor-Instanz übergeben. Nachdem der Brush erzeugt worden ist, kann auch hier mit der Methode setStyle ein Stil festgelegt werden. Mithilfe eines solches Stils ist es beispielsweise möglich, Flächen verschieden stark bzw. in unterschiedliche Richtungen schraffiert zu füllen. Näheres dazu erfahren Sie in der Qt-Dokumentation.

Allgemein gilt, dass Pens und Brushes selektiert werden müssen, bevor sie benutzt werden können. Dazu werden die Methoden setPen bzw. setBrush eines Painters aufgerufen und wird die jeweilige QPen- bzw. QBrush-Instanz als Parameter übergeben. Eine darauf folgende Zeichenoperation wird dann mit den ausgewählten Werkzeugen durchgeführt. Beachten Sie, dass immer nur ein Brush und ein Pen gleichzeitig selektiert sein können.

top

24.6.2 Koordinatensystem ▼▲

Bevor es ans Zeichnen einfacher geometrischer Formen geht, müssen wir uns Gedanken über das in Qt verwendete Koordinatensystem machen. Dieses lehnt sich an andere GUI-Toolkits an und soll durch Abbildung 24.12 veranschaulicht werden.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.12 Das Koordinatensystem

Jeder Pixel innerhalb des Widgets kann mithilfe des Koordinatensystems beschrieben werden. Beachten Sie, dass der Ursprung des Koordinatensystems in der oberen linken Ecke des Widgets liegt und dass die Y-Achse, im Gegensatz zum in der Mathematik verwendeten kartesischen Koordinatensystem, nach unten zeigt. Die Einheit des Koordinatensystems ist Pixel.

Jedes Widget verfügt über ein eigenes lokales Koordinatensystem, dessen Ursprung stets relativ zur Position des Widgets in dessen oberer linken Ecke liegt. Das hat den Vorteil, dass eine Zeichnung nicht angepasst werden muss, wenn das Widget in seiner Position auf dem Bildschirm oder innerhalb eines anderen Widgets verändert wird.

top

24.6.3 Einfache Formen ▼▲

Das Qt-Framework bietet eine ganze Reihe von abstrakten Zeichenoperationen, die das Zeichnen einfacher geometrischer Formen, wie beispielsweise eines Rechtecks oder einer Ellipse, ermöglichen. Grundsätzlich sind Zeichenoperationen als Methoden der Klasse QPainter, also eines Painters, implementiert.

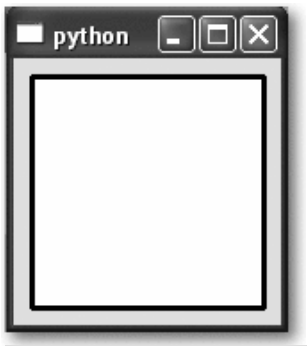
Wir beginnen damit, ein Rechteck zu zeichnen. Dazu wird die Methode drawRect eines Painters verwendet. Bevor ein Rechteck gezeichnet werden kann, sollten ein Pen für den Rand des Rechtecks und ein Brush für die Füllung erzeugt und ausgewählt werden:

```
class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.pen = QtGui.QPen(QtGui.QColor(0,0,0))
```

```
        self.pen.setWidth(3)
        self.brush =
QtGui.QBrush(QtGui.QColor(255,255,255))

    def paintEvent(self, event):
        painter = QtGui.QPainter(self)
        painter.setPen(self.pen)
        painter.setBrush(self.brush)
        painter.drawRect(10, 10, 130, 130)
```

Im Konstruktor der Widget-Klasse `MeinWidget` werden Pen und Brush angelegt, die zum Zeichnen des Rechtecks verwendet werden sollen. In diesem Fall handelt es sich um einen schwarzen Pen mit einer Stiftdicke von drei Pixeln sowie um einen weißen Brush. In der Methode `paintEvent` wird zunächst ein Painter erzeugt und werden die für die Zeichnung zu verwendenden Werkzeuge, also ein Brush und ein Pen, selektiert. Danach wird mittels `drawRect` ein Rechteck auf dem Bildschirm gemalt. Die übergebenen Parameter kennzeichnen der Reihe nach die X-Koordinate der oberen linken Ecke, die Y-Koordinate der oberen linken Ecke, die Breite des Rechtecks sowie die Höhe des Rechtecks. Alle Werte werden in Pixel angegeben. Beachten Sie, dass das Koordinatensystem, in dem das Rechteck gezeichnet wird, relativ zur Position des Widgets liegt.

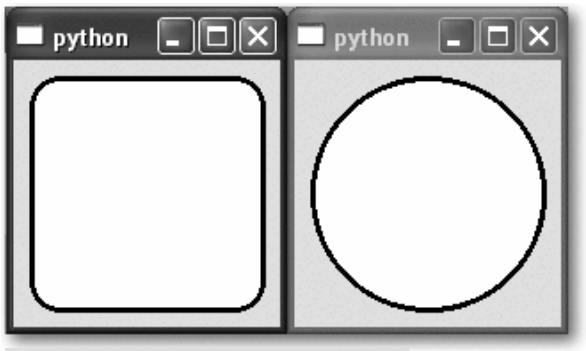
Auf dem Bildschirm erscheint das Rechteck, genauer betrachtet ein Quadrat (siehe Abbildung 24.13).



Hier klicken, um das Bild zu vergrößern

Abbildung 24.13 Ein mit `drawRect` gezeichnetes Quadrat

Auf ganz ähnliche Weise können noch weitere Figuren gezeichnet werden, deren Form durch Angabe eines umschließenden Rechtecks beschrieben ist. So braucht beispielsweise nur der Methodenname `drawRect` ausgetauscht zu werden, um ein Rechteck mit runden Ecken (`drawRoundRect`) oder eine Ellipse (`drawEllipse`) zu zeichnen.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.14 Ein Rechteck mit runden Ecken und eine Ellipse

Um eine dieser Figuren in ihrer Größe an das Widget anzupassen, kann die parameterlose Methode `rect` einer Widgetklasse verwendet werden, die die Dimensionen des Widgets als `QRect`-Instanz [Die Klasse `QRect` beschreibt ein Rechteck und verfügt unter anderem über die Methoden `x`, `y`, `width` und `height`, mit denen auf die Koordinaten der oberen linken Ecke und die

Dimensionen des Rechtecks zugegriffen werden kann. Näheres zur Klasse QRect finden Sie in der Qt-Dokumentation.] zurückgibt. Auf diese Weise ist es beispielsweise möglich, das gesamte Widget mit einer Form zu füllen.

Neben diesen drei grundlegenden Formen existiert eine Reihe weiterer Methoden zum Zeichnen spezieller Formen. Die folgende Tabelle gibt eine Übersicht über die wichtigsten dieser Methoden, die in diesem Kapitel nicht weiter besprochen werden. [Ein Polygon ist eine Fläche, die durch einen Linienzug begrenzt ist. Eine Fläche heißt konvex, wenn die Verbindungslinie zwischen je zwei Punkten in der Fläche vollständig innerhalb der Fläche verläuft. Das Gegenteil eines konvexen Polygons ist ein konkaves Polygon, das wesentlich aufwendiger darzustellen ist.]

Methode	Beschreibung
drawArc	Zeichnet einen geöffneten Bogen mit dem selektierten Pen. »Geöffnet« bedeutet in diesem Fall, dass die beiden Enden des Bogens nicht durch eine Linie miteinander verbunden sind.
drawChord	Zeichnet einen geschlossenen Bogen mit dem selektierten Pen, der mit dem selektierten Brush gefüllt wird. »Geschlossen« bedeutet, dass die beiden Enden des Bogens durch eine Linie miteinander verbunden sind.
drawConvexPolygon	Zeichnet ein konvexes Polygon[5] mit dem selektierten Pen, das mit dem selektierten Brush gefüllt wird.
drawLine	Zeichnet eine Linie mit dem selektierten Pen.
drawLines	Zeichnet einen Linienzug mit dem selektierten Pen.
drawPie	Zeichnet mit dem selektierten Pen einen Ausschnitt einer Ellipse, der umgangssprachlich als »Tortenstück« bezeichnet wird.
drawPolygon	Zeichnet mit dem selektierten Pen ein beliebiges Polygon, das mit dem selektierten Brush gefüllt wird. Diese Methode ist allgemeiner und hat eine komplexere Schnittstelle als drawConvexPolygon.
fillRect	Zeichnet ein Rechteck ohne Rand. Dieses Rechteck wird mit dem selektierten Brush gefüllt.

Tabelle 24.4 Methoden eines Painters

top

24.6.4 Grafiken ▼▲

Neben dem Zeichnen der grundlegenden geometrischen Formen ermöglicht es das Qt-Framework komfortabel, Grafiken der verschiedensten Formate von der Festplatte zu laden und mithilfe eines Painters anzuzeigen.

Das Laden einer Grafik von der Festplatte hat noch nichts mit der Anzeige und damit auch nichts mit der Klasse QPainter zu tun. Dafür existiert die Klasse QImage, die eine Grafik repräsentiert und Methoden anbietet, um Grafiken diverser Formate zu laden, zu manipulieren und zu speichern. [Eine Liste aller Grafikformate, die Qt »versteht«, finden Sie in der Dokumentation der Klasse QImage.] Das folgende Beispielprogramm lädt die Grafik *buch.png* und zeigt sie mithilfe der Methode drawImage des Painters an:

```
class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.grafik = QtGui.QImage("buch.png")
        self.ziel = QtCore.QRect(10, 10, 130, 130)
        self.quelle = QtCore.QRect(0, 0,
                                   self.grafik.width(),
                                   self.grafik.height())

    def paintEvent(self, event):
        painter = QtGui.QPainter(self)
        painter.drawImage(self.ziel, self.grafik,
self.quelle)
```

Im Konstruktor der Widgetklasse `MeinWidget` wird zunächst eine Instanz der Klasse `QImage` erzeugt. Dem Konstruktor der `QImage`-Klasse wird der Dateipfad der zu ladenden Grafik übergeben.

Nachdem die Grafik geladen wurde, werden zwei Rechtecke namens `self.quelle` und `self.ziel` erzeugt. Das Rechteck `self.ziel` spezifiziert das Rechteck im Widget, in das die Grafik gezeichnet werden soll. Das Rechteck `self.quelle` spezifiziert den Ausschnitt der Grafik, der dabei gezeichnet werden soll. In diesem Fall umschließt das Quellrechteck das gesamte Bild.

Das mit diesem Code erstellte Widget sieht so aus wie in Abbildung 24.15.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.15 Eine Grafik in einem Widget

Beachten Sie, dass in diesem Fall ein nicht quadratisches Bild auf eine quadratische Fläche gezeichnet und somit beim Zeichnen leicht gestreckt wird. Um dies zu vermeiden, müsste das Seitenverhältnis des Zielrechtecks an das des Quellrechtecks angepasst werden.

Abgesehen vom bloßen Laden eines Bildes bietet die Klasse `QImage` eine Fülle von Methoden, mit denen das geladene Bild manipuliert werden kann. Es ist nicht sinnvoll, in dieser Einführung einen vollständigen Überblick über diese Möglichkeiten zu geben. Stattdessen verweisen wir auf die ausführliche Qt-Dokumentation, in der Sie weiterführende Informationen zu `QImage` finden.

top

24.6.5 Text ▼▲

Nachdem wir sowohl geometrische Formen als auch Grafiken in ein Widget zeichnen können, fehlt noch eine mehr oder weniger große Disziplin: das Zeichnen von Text. Für viele Zeichnungen wird die Ausgabe von Text benötigt, sei es für die Beschriftungen eines Diagramms oder das Ziffernblatt einer Uhr.

Zum Zeichnen von Text in einem Widget wird die Methode `drawText` eines Painters verwendet. Im folgenden Beispiel wird der Text »Hallo Welt« im Widget zentriert ausgegeben:

```
class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.font = QtGui.QFont("Helvetica", 16)
        self.pen = QtGui.QPen(QtGui.QColor(0,0,255))
```

```
    def paintEvent(self, event):
        painter = QtGui.QPainter(self)
        painter.setPen(self.pen)
        painter.setFont(self.font)
        painter.drawText(self.rect(),
QtCore.Qt.AlignCenter,
                         "Hallo Welt")
```

Im Konstruktor der Klasse `MeinWidget` wird zunächst eine Instanz der Klasse `QFont` erzeugt. Diese Klasse repräsentiert einen Font, also einen Schrifttyp in einer bestimmten Größe und mit bestimmten weiteren Eigenschaften. Zudem wird ein Pen erzeugt, der die Schriftfarbe vorgibt, in der unser Text geschrieben werden soll.

In der Methode `paintEvent` wird zunächst, wie gehabt, ein Painter erzeugt, und dann werden mittels `setFont` und `setPen` Font und Pen selektiert. Durch einen Aufruf der Methode `drawText` wird der Text gezeichnet. Die Methode bekommt ein Rechteck als ersten und eine Positionsanweisung innerhalb dieses Rechtecks als zweiten Parameter übergeben. Als dritter Parameter wird der zu schreibende Text übergeben. Zur Positionierung des Texts innerhalb des angegebenen Rechtecks können mehrere Konstanten mithilfe des binären ODERs verknüpft werden. Die wichtigsten dieser Konstanten sind in folgender Tabelle aufgelistet und kurz erläutert:

Konstante	Beschreibung
`QtCore.Qt.AlignLeft`	Richtet den Text am linken Rand des Rechtecks aus.
`QtCore.Qt.AlignRight`	Richtet den Text am rechten Rand des Rechtecks aus.
`QtCore.Qt.AlignHCenter`	Richtet den Text horizontal zentriert im Rechteck aus.
`QtCore.Qt.AlignTop`	Richtet den Text am oberen Rand des Rechtecks aus.
`QtCore.Qt.AlignBottom`	Richtet den Text am unteren Rand des Rechtecks aus.
`QtCore.Qt.AlignVCenter`	Richtet den Text vertikal zentriert im Rechteck aus.
`QtCore.Qt.AlignCenter`	Richtet den Text horizontal und vertikal zentriert im Rechteck aus.

Tabelle 24.5 Konstanten zur Positionierung des Texts

Das mit diesem Code erstellte Widget sieht so aus wie in Abbildung 24.16.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.16 Mittels `drawText` kann Text gezeichnet werden.

Es gibt noch eine zweite, vereinfachte Variante, `drawText` zu verwenden. Dabei werden ebenfalls drei Parameter übergeben: die X-Koordinate, an die der Text geschrieben werden soll, die Y-Koordinate, an die der Text geschrieben werden soll, und ein

String, der den Text enthält. Wenn im obigen Beispielprogramm der Aufruf von drawText durch folgende Codezeile ersetzt wird,

```
painter.drawText(0, 16, "Hallo Welt")
```

dann sieht das erstellte Widget so aus wie in Abbildung 24.17.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.17 Eine Variante von drawText

Beachten Sie, dass sich die Koordinaten, die bei der zweiten Variante von drawText übergeben werden, auf die untere linke Ecke des Texts beziehen, sodass der Text nicht an die Position 0/0 geschrieben werden kann. Der als Y-Koordinate übergebene Wert von 16 Pixeln entspricht genau der gewählten Schriftgröße, weswegen der Text direkt unter dem oberen Rand des Widgets erscheint.

top

24.6.6 Eye-Candy ▲

Eingangs wurde erwähnt, dass das Qt-Framework unter anderem in Bezug auf seine Zeichenfunktionalität aus der Masse der GUI-Toolkits heraussticht. Zugegebenermaßen sind die bislang besprochenen Grundlagen zum Zeichnen in einem Qt-Widget zwar wichtig, aber auch nicht besonders beeindruckend. Funktionalität zum Zeichnen von grundlegenden geometrischen Formen, Grafiken und Text finden Sie so oder so ähnlich auch in vielen anderen Toolkits. Aus diesem Grund möchten wir in diesem Kapitel einige Aspekte der Zeichenfunktionalität von Qt in den Vordergrund holen und als »Eye-Candy« (dt. *Blickfang*, *Augenweide*) präsentieren. Die hier besprochenen Aspekte des Zeichnens in Qt dienen als Demonstration der Zeichenfunktionalität und sollen Stichwörter liefern, unter denen in der Qt-Dokumentation näher nachgeforscht werden kann. Den Quelltext der hier vorgeführten Beispielanwendungen finden Sie auf der CD, die diesem Buch beiliegt.

Farbverläufe

Abgesehen von einem flächigen Farbanstrich kann ein Brush einen beliebigen Bereich auch mit komplexeren Strukturen füllen. So kann ein Brush beispielsweise verwendet werden, um das Innere eines Rechtecks mit einem Farbverlauf zu füllen, wie in dem Widget aus Abbildung 24.18 zu sehen ist.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.18 Ein Farbverlauf mit `QLinearGradient`

Um einen Brush zu erstellen, der Flächen mit einem Farbverlauf füllt, muss zunächst eine Instanz einer Gradient-Klasse erzeugt werden. Eine solche Gradient-Klasse (dt. *Gefälle*) enthält alle Informationen, die benötigt werden, um einen Farbverlauf zu zeichnen. Es existieren drei verschiedene Gradient-Klassen, die jeweils einen eigenen Typus von Farbverlauf beschreiben. Die folgende Tabelle benennt und erläutert kurz jede dieser Klassen:

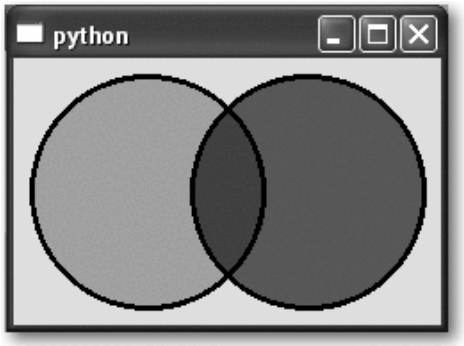
Klasse	Beschreibung
`QtGui.QConicalGradient`	Beschreibt einen konischen Farbverlauf. Das Ergebnis ähnelt der Draufsicht eines Kegels.
`QtGui.QLinearGradient`	Beschreibt einen linearen Farbverlauf. Ein solcher wurde im Beispielwidget aus Abbildung 24.18 verwendet.
`QtGui.QRadialGradient`	Beschreibt einen radialen (kreisförmigen) Farbverlauf.

Tabelle 24.6 Gradient-Klassen

Nachdem eine Instanz einer Gradient-Klasse mit den erforderlichen Informationen über den Farbverlauf erzeugt wurde, kann diese dem Konstruktor eines Brushs als Parameter übergeben werden. Ein auf diese Weise erzeugter Brush kann dann verwendet werden, um eine Fläche mit einem Farbverlauf zu füllen.

Transparenz

Das Qt-Framework unterstützt sowohl bei einem Brush als auch bei einem Pen das sogenannte *Alpha-Blending*. Darunter versteht man einen Transparenzwert, den jeder Farbwert besitzt und der bei der Erzeugung einer `QColor`-Instanz als vierter Parameter übergeben werden kann. Auf diese Weise ist es möglich, teilweise transparente, Formen zu zeichnen, die sich überlappen. Das soll das Beispiel-Widget aus Abbildung 24.19 demonstrieren.

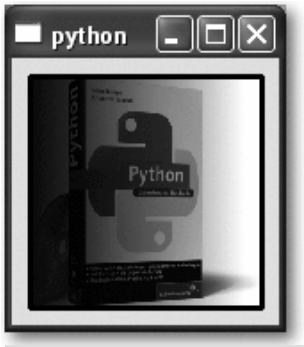


Hier klicken, um das Bild zu vergrößern

Abbildung 24.19 Alpha-Blending

Zum Verwenden von Alpha-Blending reicht es tatsächlich aus, bei der Erzeugung eines Brushs bzw. eines Pens eine `QColor`-Instanz mit einem Transparenzwert zu übergeben.

Diese Möglichkeiten zur Darstellung von Transparenzen lassen sich beispielsweise im Zusammenhang mit den bereits besprochenen Farbverläufen für interessante Effekte nutzen (siehe Abbildung 24.20).



Hier klicken, um das Bild zu vergrößern

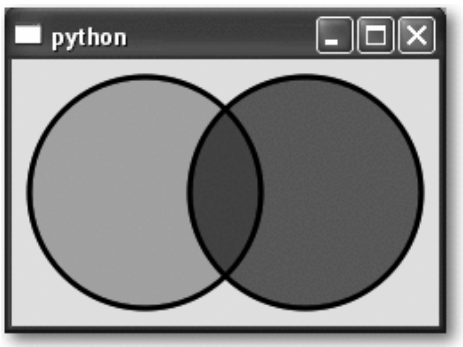
Abbildung 24.20 Transparenzeffekt

In diesem Beispielwidget wird eine Grafik angezeigt, die von einem Rechteck überlagert wird. Das Innere dieses Rechtecks ist mit einem Farbverlauf-Brush gezeichnet. Die Zielfarbe dieses Farbverlaufs ist vollständig transparent. Dieses Beispiel soll demonstrieren, dass das Qt-Framework tatsächlich eine grundlegende Unterstützung für Transparenzen in allen Bereichen des Zeichnens bietet.

Anti-Aliasing

Wenn Sie sich das Beispielwidget mit den beiden überlappenden, teilweise transparenten Kreisen noch einmal ansehen, werden Sie feststellen, dass man die einzelnen Pixel, aus denen der Umriss der Kreise besteht, erkennen kann. Die Kreise sehen deswegen nicht besonders ansprechend aus. In vielen Fällen soll eine solche oder ähnliche Zeichnung »sauber« aussehen. Genau zu diesem Zweck existiert eine Technik namens *Anti-Aliasing*, von der Sie vielleicht schon im Zusammenhang mit Computerspielen gehört haben. Beim Anti-Aliasing werden die Randbereiche einer Zeichnung geglättet, sodass einzelne Pixel nicht mehr auszumachen sind. Das Qt-Framework bietet grundlegende Unterstützung zum Zeichnen mit Anti-Aliasing.

Das Transparenz-Beispiel mit aktiviertem Anti-Aliasing sieht so aus wie in Abbildung 24.21.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.21 Anti-Aliasing

Um Anti-Aliasing bei einem Painter zu aktivieren, wird die Codezeile

```
painter.setRenderHints(QtGui.QPainter.Antialiasing)
```

verwendet, wobei `painter` eine `QPainter`-Instanz ist.

Transformationen

Eine weitere interessante Möglichkeit, die Qt bietet, sind *Transformationen*, die mithilfe einer *Transformationsmatrix* auf eine beliebige zu zeichnende Form angewandt werden können. Eine Transformationsmatrix wird durch die Klasse `QMatrix` repräsentiert.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.22 Matrixtransformationen

Im Beispiel aus Abbildung 24.22 wurde zunächst eine Figur erstellt, die nachher transformiert werden sollte. Da es sich dabei nicht um eine Form handelt, die in Qt bereits vorgesehen ist, wie beispielsweise ein Rechteck oder eine Ellipse, muss die Figur mithilfe eines sogenannten *Painter Path* zu einer Einheit zusammengefügt werden. Ein Painter Path ist eine Instanz der Klasse `QPainterPath`. Die Form dieses Beispiels besteht aus zwei Linien und einer *Beziérkurve* [Eine Beziérkurve ist eine Kurve, die durch eine mathematische Funktion mit, im Falle einer kubischen Beziérkurve, vier Parametern beschrieben wird. Beziérkurven können auch in vielen Grafikprogrammen erstellt werden.] .

Nachdem sowohl die Matrix als auch der Painter Path erstellt worden sind, kann die Matrix auf den Painter Path angewandt und der resultierende, transformierte Painter Path schließlich gezeichnet werden. Im Beispiel wurde die Matrix in fünf Iterationsschritten immer wieder verändert und die entstandende Figur jeweils mit einem unterschiedlichen Pen gezeichnet.

Beide Klassen, `QPainterPath` und `QMatrix`, enthalten viele Methoden, die das Arbeiten mit ihnen erheblich erleichtern. In der Qt-Dokumentation finden Sie ausführliche und grafisch sehr ansprechende Beispielprogramme zu dieser Thematik.

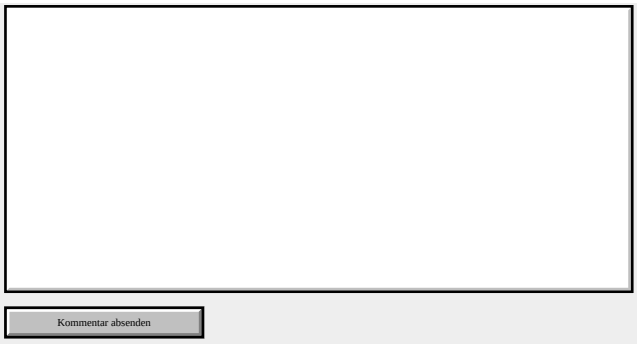
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

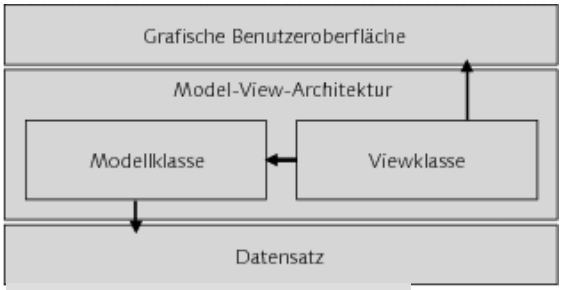
top

24.7 Model-View-Architektur

Mit Qt4 wurde die sogenannte *Model-View-Architektur* in das Framework eingeführt. Die grundsätzliche Idee dieser Art der Programmierung ist es, Form und Inhalt voneinander zu trennen. Bezogen auf Qt bedeutet das, dass Klassen, die bestimmte Daten enthalten, von Klassen getrennt werden sollen, die diese Daten an der grafischen Benutzeroberfläche anzeigen. So soll es eine *Modellklasse* geben, die ein bekanntes Interface für die gespeicherten Daten bereitstellt, und eine *Viewklasse*, die über die Modellklasse auf die Daten zugreift und auf der grafischen Oberfläche anzeigt. Beachten Sie dabei, dass nicht vorausgesetzt wird, dass die Daten tatsächlich in der Modellklasse enthalten sind, sondern nur, dass die Modellklasse Methoden bereitstellt, um auf die Daten zuzugreifen. Die Daten selbst können durchaus in einer Datenbank oder Datei gespeichert sein.

Das Aufteilen der Programmlogik in Modell- und Viewklassen hat den Vorteil, dass das Programm insgesamt einfacher und besser strukturiert wird. Außerdem führen Änderungen beispielsweise in der Art, wie die Daten gespeichert sind, nicht dazu, dass die Anzeigeklasse angepasst werden muss. Umgekehrt ist es der Modellklasse egal, in welcher Form die von ihr bereitgestellten Daten am Bildschirm angezeigt werden.

Das Verhältnis zwischen Modell- und Viewklasse lässt sich durch Abbildung 24.23 anschaulich beschreiben.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.23 Die Model-View-Architektur

Das Qt-Framework bietet einige Klassen, die dem Programmierer beim Erstellen einer Model-View-Architektur helfen. Darunter finden sich Basisklassen sowohl für die Modell- als auch für die Viewklassen.

Im Folgenden soll die Verwendung einiger dieser Klassen anhand einer einfachen Anwendung mit Model-View-Architektur demonstriert werden. Bei dieser Anwendung soll es sich um ein rudimentäres Adressbuch im Stil von Microsoft Outlook handeln.

top

24.7.1 Beispielprojekt: Ein Adressbuch

In diesem Abschnitt bieten wir einen praxisorientierten Einstieg in die Programmierung einer Model-View-Architektur anhand eines einfachen Beispielprogramms. Dazu dient ein grafisches Adressbuch das beim Starten mehrere Adresssätze aus einer Textdatei einliest und dann grafisch auf dem Bildschirm anzeigt. Intern sollen dabei die Datensätze durch eine Modellklasse eingelesen und aufbereitet werden. Eine Viewklasse soll sich dann

um die Anzeige der Daten kümmern.

Wir werden uns zunächst auf das bloße Einlesen und Anzeigen konzentrieren. Danach werden wir das Programm um bestimmte sinnvolle Extras erweitern, anhand derer weitere Aspekte von Model-View-Architekturen in Qt vorgestellt werden. Die vorläufige Anwendung, die in diesem Kapitel entwickelt wird, soll so aussehen wie in Abbildung 24.24.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.24 Ein Adressbuch

Die Adressdaten sollen aus einer Datei des folgenden Formats ausgelesen werden:

```
Donald Duck
don@ld.de
Pechvogelstraße 13
12345 Entenhausen
01234/313

Dagobert Duck
d@gobert.de
Geldspeicherweg 42
12345 Entenhausen
0190/123456
[…]
```

Die Adressdaten sind also zeilenweise in einer Datei gespeichert. Zwei Einträge im Adressbuch werden durch eine Leerzeile in der Quelldatei voneinander getrennt. Abgesehen davon, dass der Name der Person, zu der der Eintrag gehört, in der ersten Zeile des Eintrags stehen sollte, gibt es keine weiteren Anforderungen an die Formatierung der Daten. [Tatsächlich ist das Dateiformat für den vorgestellten Verwendungszweck eher ungeeignet, da es beispielsweise für das Programm, das die Datei einliest, keine effiziente Möglichkeit gibt, die einzelnen Teilinformationen des Eintrags zuzuordnen, beispielsweise also die E-Mail-Adresse herauszufiltern. Das Dateiformat wird hier jedoch aufgrund seiner Einfachheit verwendet, schließlich geht es nicht darum, eine perfekte Applikation zu schreiben.]

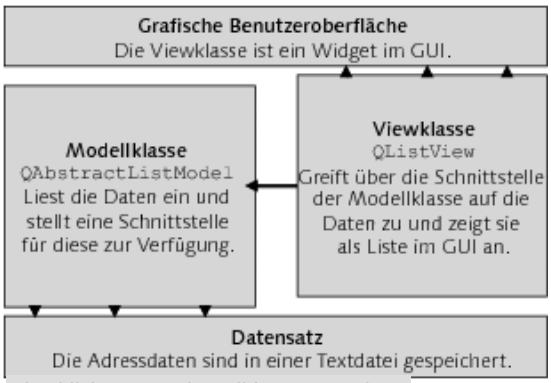
Das Adressbuch soll eine Beispielimplementation für eine Model-View-Architektur darstellen. Es ist auch relativ klar, welche Aufgaben dabei der Modell- und welche der Viewklasse zukommen.

Die Modellklasse hat die Aufgabe, die Quelldatei mit den Adressdaten einzulesen und eine Schnittstelle bereitzustellen, über die auf diese Daten zugegriffen werden kann.

Die Viewklasse soll auf die in der Modellklasse gespeicherten Daten zugreifen und diese dann in geeigneter Form auf dem Bildschirm präsentieren. Da es sich bei dem Adressbuch im Prinzip um eine Liste von Adresseinträgen handelt, können wir hier auf die Basisklasse `QListView` des Qt-Frameworks zurückgreifen, die die grundlegende Funktionalität zum Anzeigen von Modelldaten mit Listenstruktur bereitstellt. Hätten die Daten eine andere Struktur, könnten wir die Basisklassen `QTreeView` oder

QTableView verwenden, die eine baumartige bzw. tabellarische Struktur der Daten visualisieren.

Abbildung 24.25 stellt die Programmstruktur grafisch dar.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.25 Die Model-View-Architektur unseres Beispielprogramms

Der Quellcode der Modellklasse befindet sich in der Programmdatei *modell.py* und sieht folgendermaßen aus:

```
from PyQt4 import QtCore

class Modell(QtCore.QAbstractListModel):
    def __init__(self, dateiname):
        QtCore.QAbstractListModel.__init__(self)
        self.datensatz = []

        # Lade Datensatz
        f = open(dateiname)
        try:
            lst = []
            for zeile in f:
                if not zeile.strip():
self.datensatz.append(QtCore.QVariant(lst))
                    lst = []
                else:
                    lst.append(zeile.strip())
            if lst:
self.datensatz.append(QtCore.QVariant(lst))
        finally:
            f.close()

    def rowCount(self, parent=QtCore.QModelIndex()):
        return len(self.datensatz)

    def data(self, index, role=QtCore.Qt.DisplayRole):
        return
QtCore.QVariant(self.datensatz[index.row()])
```

Es wird die Modellklasse `Modell` definiert, die von der Basisklasse `QtCore. QAbstractListModel` abgeleitet ist. Diese Basisklasse implementiert grundlegende Funktionalität einer Modellklasse für einen Datensatz, der als eindimensionale Folge von Werten, also als Liste, angesehen werden soll.

Im Konstruktor der Klasse `Modell` sollen die Adressdaten aus einer Textdatei des oben beschriebenen Formats geladen werden. Dazu bekommt der Konstruktor den Dateinamen dieser Datei übergeben. Da das Dateiformat, in dem die Daten vorliegen, sehr einfach ist, ist auch der Einlesevorgang vergleichsweise simpel und braucht nicht näher erläutert zu werden. Wichtig ist aber, dass die einzelnen Einträge des Adressbuchs klassenintern in einer Liste gespeichert werden, die durch das Attribut `self.datensatz` referenziert wird. Jeder Eintrag dieser Liste ist wiederum eine Liste von Strings, die jeweils eine Zeile des Eintrags repräsentieren.

Beachten Sie zudem, dass die einzelnen Einträge als Instanzen der Klasse `QVariant` gespeichert werden. An der Schnittstelle zwischen Modell- und Viewklasse werden grundsätzlich `QVariant`-Instanzen übertragen. Die Klasse `QVariant` ist ein Konzept der C++-Welt, aus der Qt kommt, und ermöglicht es dort, Werte

beliebiger Datentypen über dieselbe Schnittstelle zu schicken. Dieses Konzept erlaubt sehr flexible Schnittstellen in C++, ist aber im Zusammenhang mit Python eher kontraproduktiv, da Schnittstellen in Python von Haus aus flexibel bezüglich der verwendeten Datentypen sind. Trotzdem müssen wir `QVariant` verwenden, da PyQt auf größtmögliche Kompatibilität zu den C++-Schnittstellen setzt.

Am Ende der Klassendefinition werden noch zwei Methoden definiert, die jede Modellklasse implementieren muss. Diese Methoden bilden die Schnittstelle, über die die Viewklasse später auf die in der Modellklasse gespeicherten Daten zugreifen kann.

Die Methode `rowCount` muss die Anzahl der Elemente als ganze Zahl zurückgeben, die der Datensatz enthält. Der dabei übergebene Parameter `parent` soll an dieser Stelle keine Rolle spielen.

Die Methode `data` wird von der Viewklasse aufgerufen, um auf ein bestimmtes Element des Datensatzes zuzugreifen. Welches das ist, wird beim Aufruf der Methode `data` über den Parameter `index` mitgeteilt. Bei `index` handelt es sich aber nicht um eine ganze Zahl, sondern um eine `QModelIndex`-Instanz. Auf den tatsächlichen Index kann über die Methode `row` dieser Instanz zugegriffen werden. Die Methode `data` muss eine `QVariant`-Instanz zurückgeben.

So viel zur Modellklasse. Die dazu passende Viewklasse sieht folgendermaßen aus und ist in der Programmdatei *view.py* enthalten:

```
class View(QtGui.QListView):
   def __init__(self, modell, parent=None):
       QtGui.QListView.__init__(self, parent)
       self.delegate = ViewDelegate()
       self.setItemDelegate(self.delegate)
       self.setModel(modell)

self.setVerticalScrollMode(QtGui.QListView.ScrollPerPixel)
```

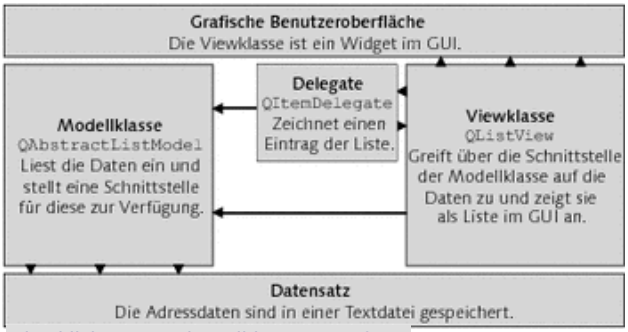
Die Viewklasse `View` wird von der Basisklasse `QtGui.QListView` abgeleitet. Diese Basisklasse stellt die Funktionalität bereit, die benötigt wird, um einen listenartigen Datensatz grafisch darzustellen. Alternativ hätten auch die Klassen `QTreeView` und `QTableView` als Basisklassen dienen können, wenn zur Darstellung der Daten eine baumartige oder tabellarische Struktur verwendet werden soll.

Dem Konstruktor der Klasse `View` wird die Instanz der soeben definierten Modellklasse `Modell` übergeben, die mithilfe der Viewklasse grafisch dargestellt werden soll. Um die Daten jedoch tatsächlich anzuzeigen, wird eine weitere Klasse benötigt, der sogenannte *Delegate* (dt. *Abgesandter*). Eine Instanz der Delegate-Klasse, die im Anschluss an die Viewklasse besprochen werden soll, wird der Viewklasse über die Methode `setItemDelegate` zugewiesen. Die Delegate-Klasse enthält die Zeichenroutinen für ein Element des Datensatzes.

Zum Schluss wird noch das Modell mittels `setModel` eingebunden und, was eher eine kosmetische Angelegenheit ist, der Scrollmodus auf »pixelweise« gesetzt. Im Normalzustand würde das `QListView`-Widget beim Verschieben der Scrollbar immer um ganze Einträge weiter scrollen, was bei wenigen großen Einträgen nicht schön aussieht.

Neben der Viewklasse wurde eine sogenannte Delegate-Klasse angesprochen, von der innerhalb der Viewklasse eine Instanz erstellt wurde. Die Aufgabe einer solchen Delegate-Klasse, die mehr oder weniger als Teil der Viewklasse zu betrachten ist, besteht darin, das Zeichnen eines einzelnen Eintrags in der Liste zu erledigen. Dazu kann die Delegate-Klasse über die Schnittstelle der Modellklasse auf den eingelesenen Datensatz zugreifen. Die Grafik, die eingangs die Model-View-Architektur des Beispielprogramms veranschaulichte, enthielt aus Gründen der Einfachheit keine Informationen über die Delegate-Klasse. Das

möchten wir an dieser Stelle nachholen und zeigen in Abbildung 24.26, wie sich die Delegate-Klasse in die Model-View-Architektur integiert.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.26 Die Model-View-Architektur des Beispielprogramms

Die Delegate-Klasse positioniert sich zwischen der View- und der Modellklasse und ist für das Zeichnen eines einzelnen Eintrags im ListView Widget zuständig. Bei der Delegate-Klasse handelt es sich nicht um ein Widget, sondern nur um eine Hilfsklasse der Viewklasse. Die Viewklasse ruft die Methode paint der Delegate-Klasse für jeden Eintrag im Datensatz auf und stellt aus den Einzelzeichnungen das ListView Widget zusammen, das an der grafischen Benutzeroberfläche angezeigt wird. Wie bei der View- und Modellklasse auch, existiert im Qt-Framework eine Basisklasse, von der eine selbst definierte Delegate-Klasse abgeleitet werden muss. Um das Zeichnen eines Eintrags dann an die jeweiligen Bedürfnisse anzupassen, müssen in der abgeleiteten Klasse diverse Methoden implementiert werden. Näheres dazu erfahren Sie anhand des Beispielprogramms im Laufe dieses Kapitels.

Um einen Eintrag adäquat zeichnen zu können, kann die Delegate-Klasse über die von der Modellklasse bereitgestellte Schnittstelle auf den Datensatz zugreifen. Selbstverständlich kann auch die Viewklasse selbst auf diesem Wege Daten des Datensatzes lesen.

Im Folgenden soll die Delegate-Klasse für die vorher besprochene Viewklasse erläutert werden. Die Delegate-Klasse ist in der gleichen Programmdatei definiert wie die Viewklasse. Da die Delegate-Klasse vergleichsweise umfangreich ist, werden wir sie Methode für Methode besprechen:

```
from PyQt4 import QtGui, QtCore

class ViewDelegate(QtGui.QItemDelegate):
    def __init__(self):
        QtGui.QItemDelegate.__init__(self)

        self.rahmenStift =
QtGui.QPen(QtGui.QColor(0,0,0))
        self.titelTextStift = QtGui.QPen(

QtGui.QColor(255,255,255))
        self.titelFarbe =
QtGui.QBrush(QtGui.QColor(120,120,120))
        self.textStift = QtGui.QPen(QtGui.QColor(0,0,0))
        self.titelSchriftart = QtGui.QFont("Helvetica",
10,

QtGui.QFont.Bold)
        self.textSchriftart = QtGui.QFont("Helvetica", 10)

        self.zeilenHoehe = 15
        self.titelHoehe = 20
        self.abstand = 4
        self.abstandInnen = 2
        self.abstandText = 4
```

Im Konstruktor der Klasse ViewDelegate werden einige Attribute initialisiert, die zum Zeichnen eines Adresseintrags von Bedeutung sind. Dazu zählen zum einen die Zeichenwerkzeuge wie beispielsweise Brushs und Pens, mit denen der Adresseintrag gezeichnet werden soll, und zum anderen einige Konstanten, die Abstände und Richtgrößen zum Zeichnen eines Eintrags festlegen.

Um zu besprechen, welches Attribut wofür gedacht ist, vergegenwärtigen wir uns anhand von Abbildung 24.27 noch einmal, wie ein Eintrag im späteren Programm gezeichnet werden soll.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.27 Ein Eintrag im Adressbuch

Die folgende Tabelle listet alle Attribute, darunter vor allem die angelegten Zeichenwerkzeuge, der Klasse `ViewDelegate` mit einer kurzen Beschreibung der jeweiligen Bedeutung auf.

Attribut	Beschreibung
`rahmenStift`	Der Pen, mit dem der dünne schwarze Rahmen um den Eintrag gezeichnet werden soll
`titelTextStift`	Der Pen, mit dem die Überschrift geschrieben werden soll
`titelFarbe`	Der Brush, mit dem das graue Rechteck unter der Überschrift gezeichnet wird
`titelSchriftart`	Die Schriftart, in der die Überschrift geschrieben werden soll
`textStift`	Der Pen, mit dem die Adressdaten geschrieben werden sollen
`textSchriftart`	Die Schriftart, in der die Adressdaten geschrieben werden sollen
`zeilenHoehe`	Die Höhe einer Zeile der Adressdaten in Pixel
`titelHoehe`	Die Höhe der Überschrift in Pixel
`abstand`	Der Abstand des Eintrags vom Dialograd und anderen Einträgen in Pixel
`abstandInnen`	Der Abstand zwischen dem grauen Rechteck unter der Überschrift und der Umrandung des Eintrags in Pixel
`abstandText`	Der Abstand des Texts von der Umrandung des Eintrags auf der linken Seite in Pixel

Tabelle 24.7 Attribute der Klasse »ViewDelegate«

Damit wäre der Konstruktor vollständig beschrieben. Es folgt die Methode `sizeHint`, die jede Delegate-Klasse implementieren muss. Diese Methode wird vom `QListView`-Widget aufgerufen, um die Dimensionen herauszufinden, die ein bestimmter Eintrag des Datensatzes in der Anzeige benötigt.

```
    def sizeHint(self, option, index):
        anz = len(index.data().toList())
        return QtCore.QSize(170,
                            self.zeilenHoehe*anz +
self.titelHoehe)
```

Die Methode wird aufgerufen, um die Höhe und die Breite eines einzelnen Eintrags in Erfahrung zu bringen. Dabei bekommt sie zwei Parameter übergeben: `option` und `index`.

Für den Parameter `option` wird eine Instanz der Klasse `QStyleOptionViewItem` übergeben, die verschiedene Anweisungen enthalten kann, in welcher Form der Eintrag gezeichnet werden soll. Da diese Formatanweisungen möglicherweise auch Einfluss

auf die Maße eines Eintrags haben, werden sie auch der Funktion sizeHint übergeben. In unserem Beispielprogramm ist der Parameter option nicht von Belang und wird nicht weiter erläutert.

Mit dem zweiten Parameter, index, wird das Element spezifiziert, dessen Dimensionen zurückgegeben werden sollen. Für index wird eine Instanz der Klasse QModelIndex übergeben. Wichtig ist vor allem die Methode data der QModel Index-Instanz, über die auf die Daten des Eintrags zugegriffen werden kann. Bei den Daten handelt es sich um die QVariant-Instanz, die in der Methode data der Modellklasse zurückgegeben wird.

In der Methode sizeHint wird jetzt über die Methode data der übergebenen QModelIndex-Instanz auf die Daten des Adresseintrags zugegriffen. Da es sich dabei um eine QVariant-Instanz handelt, muss diese erst durch Aufruf der Methode toList in eine Liste konvertiert werden. Danach wird die Größe berechnet, die der Eintrag beim späteren Zeichnen haben wird, und in Form einer QSize-Instanz zurückgegeben. Beachten Sie, dass die Breite der Einträge in diesem Beispiel bei konstanten 170 Pixeln liegt. [Dabei handelt es sich um eine Vereinfachung des Beispielprogramms. In einem wirklichen Programm müsste die Breite des Eintrags anhand der längsten Zeile berechnet werden. Dazu kann die Methode width einer QFontMetrics-Instanz verwendet werden. Näheres dazu finden Sie in der Qt- bzw. PyQt-Dokumentation.]

Die folgende Methode paint muss von einer Delegate-Klasse implementiert werden und wird immer dann aufgerufen, wenn ein einzelner Eintrag neu gezeichnet werden muss. Beachten Sie, dass paint pro Aufruf immer nur einen Eintrag zeichnet.

```
    def paint(self, painter, option, index):
        rahmen = option.rect.adjusted(self.abstand,
self.abstand,
                                      -self.abstand,
-self.abstand)
        rahmenTitel = rahmen.adjusted(self.abstandInnen,
                                      self.abstandInnen,
-self.abstandInnen+1, 0)
        rahmenTitel.setHeight(self.titelHoehe)
        rahmenTitelText =
rahmenTitel.adjusted(self.abstandText,
                                           0,
self.abstandText, 0)
        datensatz = index.data().toList()
        painter.save()
        painter.setPen(self.rahmenStift)
        painter.drawRect(rahmen)
        painter.fillRect(rahmenTitel, self.titelFarbe)

        # Titel schreiben
        painter.setPen(self.titelTextStift)
        painter.setFont(self.titelSchriftart)
        painter.drawText(rahmenTitelText,
                         QtCore.Qt.AlignLeft |
QtCore.Qt.AlignVCenter,
                         datensatz[0].toString())

        # Adresse schreiben
        painter.setPen(self.textStift)
        painter.setFont(self.textSchriftart)
        for i, eintrag in enumerate(datensatz[1:]):
            painter.drawText(rahmenTitel.x() +
self.abstandText,
                             rahmenTitel.bottom() +
(i+1)*self.zeilenHoehe,
                             "%s" % eintrag.toString())
        painter.restore()
```

Die Methode paint bekommt die drei Parameter painter, option und index übergeben. Für den Parameter painter wird eine QPainter-Instanz übergeben, die dazu verwendet werden soll, den Eintrag zu zeichnen. Die beiden Parameter option und index haben die gleiche Bedeutung wie bei der Methode sizeHint der Delegate-Klasse.

In der Methode paint werden zunächst einige Rechtecke berechnet, die nachher zum Zeichnen des Eintrags verwendet werden. Beachten Sie, dass option.rect eine QRect-Instanz referenziert, die das Rechteck beschreibt, in das der Eintrag gezeichnet werden soll. Alle Zeichenoperationen sollten sich also an diesem Rechteck ausrichten. Die angelegten lokalen Referenzen haben folgende Bedeutung:

Attribut	Beschreibung
rahmen	Das Rechteck, um das der dünne schwarze Rahmen gezogen werden soll
rahmenTitel	Das Rechteck der grau hinterlegten Titelzeile
rahmenTitelText	Das Rechteck, in das der Text in der Titelzeile geschrieben wird. Dazu wird ein Rechteck benötigt, da der Text vertikal zentriert werden soll.

Tabelle 24.8 Lokale Referenzen in der Methode »paint«

Nachdem die lokalen Referenzen angelegt wurden, wird der Status des Painters mittels save gespeichert, um ihn am Ende der Methode mittels restore wiederherstellen zu können. Beachten Sie, dass ein auf einem solchen Wege übergebener Painter immer in den Ausgangszustand zurückversetzt werden sollte, nachdem die Zeichenoperationen durchgeführt wurden, da sonst ein ungewollter Seiteneffekt in der übergeordneten Funktion, in diesem Fall also im Qt-Framework, auftritt.

Danach werden mithilfe der Methoden drawRect und fillRect des Painters der Rahmen um den Eintrag und die grau hinterlegte Titelzeile gezeichnet. Jetzt fehlen nur noch die Beschriftungen des Eintrags. Dazu werden zunächst die passende Schriftart und das gewünschte Stiftwerkzeug mittels setFont und setPen ausgewählt. Die Titelzeile des Eintrags wird mit der fetten Schriftart titel Schriftart und einem weißen Pen geschrieben. Außerdem wird sie im Rechteck rahmenTitelText vertikal zentriert und horizontal linksbündig positioniert.

Beachten Sie, dass die Methode drawText des Painters in mehreren Varianten aufgerufen werden kann. So ist es beispielsweise möglich (wie bei der Titelzeile) ein Rechteck und eine Positionsanweisung innerhalb dieses Rechtecks zu übergeben oder (wie bei den Adresszeilen des Eintrags) direkt die Koordinaten anzugeben, an die der Text geschrieben werden soll.

Zu guter Letzt riskieren wir noch einen Blick auf das Hauptprogramm, das in der Programmdatei *programm.py* stehen soll:

```
from PyQt4 import QtGui
import sys
import modell
import view

m = modell.Modell("adressbuch.txt")

app = QtGui.QApplication(sys.argv)
liste = view.View(m)
liste.resize(200, 500)
liste.show()
sys.exit(app.exec_())
```

Nachdem vor allem die lokalen Module modell und view eingebunden wurden, wird eine Instanz der Klasse Modell erzeugt, die den Datensatz aus der Datei *adressbuch.txt* repräsentieren soll. Nachdem die Modellklasse instanziiert wurde, wird nach dem bekannten Schema eine PyQt-Applikation erstellt.

Beachten Sie dabei, dass die Viewklasse View als einziges Widget der Applikation gleichzeitig als Fensterklasse dient. Bevor das Widget mittels show angezeigt wird, wird seine Größe durch Aufruf der Methode resize auf einen sinnvollen Wert (200 Pixel breit und 500 Pixel hoch) gesetzt.

Wenn das Hauptprogramm ausgeführt wird, können Sie sehen, dass sich die Basisklasse QListView der Viewklasse tatsächlich um Feinheiten wie das Scrollen von Einträgen oder das Anpassen der Einträge bei einer Größenänderung kümmert (siehe Abbildung 24.28).

Hier klicken, um das Bild zu vergrößern

Abbildung 24.28 Scrollen im Adressbuch

top

24.7.2 Auswählen von Einträgen

Nachdem wir die Adressdaten erfolgreich als eine Art Liste in einem Widget angezeigt haben, drängt sich die Frage auf, ob abgesehen vom bloßen Anzeigen der Daten noch weitere Aktionen durchgeführt werden können. So soll das Programm in diesem Abschnitt dahingehend weiterentwickelt werden, dass der Benutzer einen Eintrag des Adressbuchs auswählen kann.

An der Grundstruktur des Beispielprogramms und insbesondere der Viewklasse muss dafür nicht viel verändert werden, denn genau genommen ist das Auswählen im vorherigen Beispielprogramm schon möglich gewesen, allerdings haben wir bis dato alle Einträge der Liste gleich gezeichnet. Was noch fehlt, ist also die grafische Hervorhebung des ausgewählten Eintrags, damit der Benutzer erkennen kann, welcher Eintrag momentan selektiert ist.

Ein ausgewählter Eintrag im Adressbuch soll später folgendermaßen aussehen:

Hier klicken, um das Bild zu vergrößern

Abbildung 24.29 Ein ausgewählter Eintrag im Adressbuch

Der ausgewählte Eintrag soll sich in der Farbe der Titelleiste von den anderen unterscheiden. Statt in einem dunklen Grau soll sie in einem Blauton gezeichnet werden. Dazu legen wir im Konstruktor der Delegate-Klasse zunächst einen neuen Brush mit diesem Blauton als Farbe an:

```
def __init__(self):
    QtGui.QItemDelegate.__init__(self)
    [...]
    self.titelFarbeAktiv =
QtGui.QBrush(QtGui.QColor(0,0,120))
```

```
    self.hintergrundFarbeAktiv = QtGui.QBrush(
QtGui.QColor(230,230,255))
[…]
```

Jetzt muss nur noch beim Zeichnen eines Eintrags, also in der Methode `paint`, unterschieden werden, ob es sich bei dem zu zeichnenden Eintrag um den momentan ausgewählten handelt oder nicht. Dies lässt sich anhand des Attributs `state` der `QStyleOptionViewItem`-Instanz feststellen, die beim Aufruf der Methode `paint` für den Parameter `option` übergeben wird.

Wir ändern also das Zeichnen des grauen Titelrechtecks in folgenden Code:

```
if option.state & QtGui.QStyle.State_Selected:
    painter.fillRect(rahmen, self.hintergrundFarbeAktiv)
    painter.fillRect(rahmenTitel, self.titelFarbeAktiv)
else:
    painter.fillRect(rahmenTitel, self.titelFarbe)
```

Beachten Sie, dass dieser Code vor dem Zeichnen des dünnen schwarzen Rahmens stehen muss:

```
painter.setPen(self.rahmenStift)
painter.drawRect(rahmen)
```

Das waren tatsächlich schon alle notwendigen Schritte, um es dem Benutzer zu erlauben, einen Eintrag des Adressbuchs auszuwählen. Beachten Sie, dass mit dem binären UND-Operator `&` überprüft wird, ob das Statusflag `Status_Selected` gesetzt ist oder nicht.

Neben `Status_Selected` existieren noch weitere vordefinierte Zustände, von denen mitunter auch mehrere gleichzeitig gesetzt sein können. Diese Zustände sind teilweise sehr speziell und sollen hier nicht näher erläutert werden.

top

24.7.3 Editieren von Einträgen ▲

Nachdem wir uns damit beschäftigt haben, wie die Adressdaten in einem `QList View`-Widget angezeigt werden können, und das Beispielprogramm dahingehend erweitert haben, dass ein Eintrag vom Benutzer ausgewählt werden kann, liegt die Frage nahe, ob wir dem Benutzer auch das Editieren eines Datensatzes erlauben können. Es ist zwar nicht ganz so banal wie das Selektieren eines Eintrags im vorherigen Kapitel, doch auch für das Editieren eines Eintrags bietet die Model-View-Architektur von Qt eine komfortable Schnittstelle an.

Im späteren Programm soll das Editieren eines Eintrags so aussehen, wie es in Abbildung 24.30 gezeigt ist.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.30 Editieren eines Adresseintrags

Um das Editieren von Einträgen zu ermöglichen, müssen die einzelnen Einträge des Datensatzes von der Modellklasse zunächst explizit als editierbar gekennzeichnet werden. Dazu muss die Modellklasse die Methode `flags` implementieren:

```
def flags(self, index):
    return QtCore.Qt.ItemIsSelectable |
QtCore.Qt.ItemIsEditable\
                                    |
QtCore.Qt.ItemIsEnabled
```

Diese Methode wird immer dann aufgerufen, wenn das `QListView`-Widget nähere Informationen über den Eintrag erhalten will, der durch die `QModelIndex`-Instanz `index` spezifiziert wird. In unserem Fall werden unabhängig vom Index des Eintrags pauschal die Flags `ItemIsSelectable`, `ItemIsEditable` und `Item IsEnabled` zurückgegeben, die für einen selektierbaren, editierbaren und aktivierten Eintrag stehen. Standardmäßig – also wenn die Methode `flags` nicht implementiert wird – erhält jeder Eintrag die Flags `ItemIsSelectable` und `Item IsEnabled`.

Zusätzlich zur Methode `flags` sollte die Modellklasse die Methode `setData` implementieren, die die Aufgabe hat, die vom Benutzer veränderten Einträge in den Datensatz zu übernehmen.

```
def setData(self, index, value, role=QtCore.Qt.EditRole):
    self.datensatz[index.row()] = value
    self.emit(QtCore.SIGNAL("layoutChanged()"))
    return True
```

Die Methode bekommt den Index des veränderten Eintrags und den veränderten Inhalt dieses Eintrags übergeben. Der zusätzliche Parameter `role` soll an dieser Stelle nicht weiter interessieren. Im Körper der Methode wird der alte Eintrag in dem in der Modellklasse gespeicherten Datensatz `self.datensatz` durch den veränderten ersetzt. Danach wird das Signal `layoutChanged` gesendet, das die Viewklasse dazu anhält, die Anzeige vollständig neu aufzubauen. Das ist sinnvoll, da sich durch die Änderungen des Benutzers die Zeilenzahl und damit die Höhe des jeweiligen Eintrags verändert haben könnte.

Das sind alle Änderungen, die an der Modellklasse vorgenommen werden müssen, um das Editieren eines Eintrags zu erlauben. Doch auch die Delegate-Klasse muss einige zusätzliche Methoden implementieren. Dabei handelt es sich um die Methoden `createEditor`, `setEditorData`, `updateEditorGeometry`, `setModel Data` und `eventFilter`, die im Folgenden besprochen werden.

Die Methode `createEditor` wird aufgerufen, wenn der Benutzer doppelt auf einen Eintrag klickt, um diesen zu editieren. Die Methode `createEditor` muss ein Widget zurückgeben, das dann statt des entsprechenden Eintrags zum Editieren angezeigt wird.

```
def createEditor(self, parent, option, index):
    return QtGui.QTextEdit(parent)
```

Die Methode bekommt die bereits bekannten Parameter `option` und `index` übergeben, die den zu editierenden Eintrag spezifizieren. Zusätzlich wird für `parent` das Widget übergeben, das als Elternwidget des Editorwidgets eingetragen werden soll. In diesem Fall erstellen wir ein `QTextEdit`-Widget, in dem der Benutzer den Eintrag editieren soll.

Die Methode `setEditorData` wird vom `QListView`-Widget aufgerufen, um das von `createEditor` erzeugte Widget mit Inhalt zu füllen.

```
def setEditorData(self, editor, index):
    l = [unicode(s.toString()) for s in
index.data().toList()]
    text = "\n".join(l)
    editor.setPlainText(text)
```

Dazu bekommt die Methode das Editorwidget in Form des Parameters `editor` und den bekannten Parameter `index` übergeben, der den zu editierenden Eintrag spezifiziert. Im Methodenkörper werden die Daten des zu editierenden Eintrags ausgelesen und mittels `join` zu einem einzigen String zusammengefügt. Dieser String wird dann durch Aufruf der Methode `setPlainText` in das `QTextEdit`-Widget geschrieben.

Die Methode `updateEditorGeometry` wird vom `QListView`-Widget aufgerufen, um die Größe des Editorwidgets festlegen zu lassen.

```
def updateEditorGeometry(self, editor, option, index):
    rahmen = option.rect.adjusted(self.abstand,
self.abstand,
                                  -self.abstand,
-self.abstand)
    editor.setGeometry(rahmen)
```

Die Methode bekommt die bekannten Parameter `option` und `index` und zusätzlich das Editorwidget `editor` übergeben. In diesem Fall verpassen wir dem Editorwidget mittels `setGeometry` die gleiche Größe, die der entsprechende Eintrag gehabt hätte, wenn er normal gezeichnet werden würde.

Die Methode `setModelData` wird aufgerufen, wenn das Editieren durch den Benutzer erfolgt ist, um die veränderten Daten aus dem Editorwidget auszulesen und an die Modellklasse weiterzureichen.

```
def setModelData(self, editor, model, index):
    txt = editor.toPlainText()
    l = [QtCore.QVariant(s) for s in txt.split("\n")]
    model.setData(index, QtCore.QVariant(l))
```

Die Methode bekommt sowohl das Editorwidget als auch die Modellklasse in Form der Parameter `editor` und `model` übergeben. Zusätzlich wird eine `QModel Index`-Instanz übergeben, die den editierten Eintrag spezifiziert. In der Methode wird der Text des `QTextEdit`-Widgets ausgelesen und in einzelne Zeilen unterteilt. Danach wird die vorhin angelegte Methode `setData` der Modellklasse aufgerufen.

Damit ist die grundlegende Funktionalität zum Editieren eines Eintrags implementiert. Allerdings werden Sie beim Ausführen des Programms feststellen, dass die Enter-Taste beim Editieren eines Eintrags sowohl eine neue Zeile beginnt als auch das Editieren des Eintrags beendet. Das ist kein besonders glücklicher Umstand und sollte behoben werden. Dazu implementieren wir die Methode `eventFilter`, die immer dann aufgerufen wird, wenn ein sogenanntes *Event* eintritt. Ein Event ist beispielsweise das Drücken einer Taste während des Editierens eines Eintrags.

```
def eventFilter(self, editor, event):
    if event.type() == QtCore.QEvent.KeyPress \
       and event.key() in (QtCore.Qt.Key_Return,
                           QtCore.Qt.Key_Enter):
        return False
    return QtGui.QItemDelegate.eventFilter(self, editor,
event)
```

Die Methode bekommt das Editorwidget `editor` und eine `QEvent`-Instanz übergeben, die das eingetretene Event spezifiziert. Im Körper der Methode wird überprüft, ob es sich bei dem Event um einen Tastendruck handelt und wenn ja, ob es sich bei der gedrückten Taste um die Enter- oder die Return-Taste handelt. Beachten Sie, dass es einen Unterschied zwischen diesen beiden Tasten gibt. Enter finden Sie auf der Tastatur unten rechts vom Nummernblock, während die Return-Taste diejenige ist, die Sie

verwenden, um in einem Text eine neue Zeile zu beginnen.

Nur wenn es sich bei dem Event nicht um eine gedrückte Enter- oder Return-Taste handelt, wird die Standardimplementation der Methode aufgerufen, beispielsweise soll also bei gedrückter Escape-Taste weiterhin das Editieren des Eintrags abgebrochen werden. Im Falle der Enter- oder der Return-Taste wird nichts dergleichen unternommen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

24.8 Wichtige Widgets

Nachdem wir bereits zwei Beispielprojekte zur Programmierung grafischer Benutzeroberflächen in PyQt entwickelt haben, finden Sie in diesem Abschnitt eine Übersicht über die wichtigsten Qt-Widgets und ihre Verwendung. Wie Sie wissen, werden Widgets im Qt-Framework durch Klassen repräsentiert, die sogenannten *Widgetklassen*. Die Aufzählung von Widgetklassen, die Sie in diesem Kapitel vorfinden werden, ist keineswegs vollständig, und auch die Verwendung der Widgetklassen kann hier nicht erschöpfend beschrieben werden. Wenn Sie eine umfassende Beschreibung dieser Klassen benötigen, ist die Qt- bzw. PyQt-Dokumentation Ihr Freund und Helfer. Eine umfassende Übersicht über alle Klassen des Qt-Frameworks finden Sie dort unter dem Stichwort »class reference«.

Die folgende Übersicht ist in alphabetischer Reihenfolge geordnet.

top

24.8.1 QCheckBox

Die Klasse `QCheckBox` repräsentiert eine *Check Box* in der grafischen Benutzeroberfläche. Eine Check Box ist ein Steuerelement, das vom Benutzer entweder aktiviert oder deaktiviert werden kann und dabei in seiner Bedeutung unabhängig von anderen Check Boxes ist. Eine Alternative zur Check Box ist der Radio Button, der in Abschnitt 24.8.13 erklärt wird.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.31 Zwei Check Boxes

Die Verwendung der Klasse `QCheckBox` ist aufgrund der einfachen Struktur des Steuerelements schnell erklärt. Sie können mit den Methoden `setCheckState` und `checkState` den sogenannten Check State setzen oder lesen. Der Check State beschreibt, ob die Check Box momentan mit einem Haken versehen ist oder nicht. Dabei werden folgende Konstanten verwendet, wobei `setCheckState` eine solche Konstante als Parameter erwartet und `checkState` eine dieser Konstanten zurückgibt.

Attribut	Beschreibung
`QtCore.Qt.Checked`	Die Check Box zeigt einen Haken.
`QtCore.Qt.Unchecked`	Die Check Box zeigt keinen Haken.
`QtCore.Qt.PartiallyChecked`	Die Check Box befindet sich im optionalen dritten Modus: »teilweise gecheckt«.

Tabelle 24.9 Lokale Referenzen in der Methode `paint`

Eine Check Box kann mithilfe der Methode `setTristate` mit dem in der Tabelle genannten optionalen dritten Status versehen werden.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.32 Eine Check Box im optionalen dritten Status

Diesen dritten Status kennen Sie beispielsweise von der »Schreibgeschützt«-Check Box im Ordnereigenschaften-Dialog von Windows. Sie zeigt an, dass die Option für einen Teil der untergeordneten Elemente aktiv und für den anderen Teil inaktiv ist. Den Tristate-Status können Sie mit der Methode `isTristate` abfragen.

Wenn der Benutzer den Status einer Check Box ändert, wird das `stateChanged`-Signal gesendet, das die Signatur `stateChanged(int)` besitzt.

top

24.8.2 QComboBox

Die Klasse `QComboBox` repräsentiert eine Combo Box, besser bekannt als Dropdown-Menü, in der grafischen Benutzeroberfläche.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.33 Zwei Combo Boxes

Zunächst einmal können einer `QComboBox`-Instanz mithilfe der Methode `addItem` bzw. `addItems` Einträge hinzugefügt werden. Diese Methoden bekommen einen String bzw. eine Liste von Strings übergeben, die jeweils den Namen des Eintrags enthalten, der in der Combo Box erscheinen soll. Der Methode `addItem` kann zudem optional eine `QIcon`-Instanz an erster Stelle übergeben werden, die das Icon repräsentiert, das zusammen mit dem Eintrag in der Combo Box angezeigt werden soll.

Wenn der Benutzer einen Eintrag einer Combo Box auswählt, werden zwei Signale gesendet: `currentIndexChanged(int)` und `activated(int)`. Der Unterschied zwischen diesen beiden Signalen besteht darin, dass `currentIndex Changed` auch gesendet wird, wenn vonseiten des Programms, also ohne Eingriff des Benutzers, ein Eintrag ausgewählt wurde, während `activated` ausschließlich dann gesendet wird, wenn der Benutzer einen Eintrag der Combo Box auswählt.

Eine Combo Box kann klassisch, beispielsweise mit der oben beschriebenen Methode `addItem`, mit Werten gefüllt werden, unterstützt aber auch eine Model-View-Architektur. Dazu kann mithilfe der Methoden `setModel` und `setView` der `QComboBox`-Klasse die Modell- bzw. die Viewklasse festgelegt werden.

top

24.8.3 QDateEdit

Die Klasse `QDateEdit` repräsentiert ein Widget in der grafischen Benutzeroberfläche, das eine Datumsangabe vom Benutzer einliest. Dabei ist es mit dem DateEdit-Widget möglich, dem Benutzer einen übersichtlichen Kalender anzuzeigen, in dem er das gewünschte Datum auswählen kann.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.34 Zwei DateEdit-Widgets

Bei der Instanziierung der Klasse `QDateEdit` kann als erster Parameter eine `QDate`-Instanz übergeben werden, die dem Datum entspricht, das im Widget voreingestellt sein soll. Anstelle von `QDate` kann auch eine Instanz der Klasse `date` `time.date` aus Pythons Standardbibliothek übergeben werden.

Das Anzeigen eines Kalenders kann durch Aufruf der Methode `setCalendar` Popup aktiviert werden, indem beim Methodenaufruf `True` übergeben wird.

Über die Methoden `setMinimumDate` und `setMaximumDate` kann dem Benutzer ein bestimmter Zeitraum vorgegeben werden, aus dem er ein Datum auswählen soll. Diese Werte können dementsprechend über die Methoden `minimumDate` und `maximumDate` ausgelesen werden. Beachten Sie, dass die beiden letzten Methoden das jeweilige Datum als `QDate`-Instanz zurückgeben. Das Datum, das momentan vom Widget angezeigt wird, kann mithilfe der Methode `date` gelesen und mit `setDate` geschrieben werden.

Wenn das Datum vom Benutzer geändert wurde, wird vom DateEdit-Widget das Signal `dateChanged(const QDate &)` gesendet.

Beachten Sie, dass neben dem DateEdit-Widget noch das DateTimeEdit- und das TimeEdit-Widget existieren, bei denen der Benutzer noch zusätzlich bzw. ausschließlich eine Uhrzeit angeben kann.

top

24.8.4 QDateTimeEdit ▼▲

Die Klasse `QDateTimeEdit` repräsentiert ein Widget, das sich ganz ähnlich verhält wie das DateEdit-Widget, mit dem Unterschied aber, dass zusätzlich zum Datum eine Uhrzeit angegeben werden kann. Ähnlich wie beim DateEdit-Widget kann optional ein Kalender zum Auswählen des Datums angezeigt werden. Zum Auswählen der Uhrzeit existiert keine solche Komfortfunktion, sie muss durch den Benutzer eingetippt werden.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.35 Zwei DateTimeEdit-Widgets

Bei der Klasse `QDateTimeEdit` handelt es sich genauer betrachtet um die Basisklasse von `QDateEdit`. Die Verwendung von `QDateTimeEdit` ähnelt also der Verwendung von `QDateEdit` sehr. Um die Zeitangabe verwerten zu können, existieren zusätzlich zu den bei `QDateEdit` besprochenen Methoden die Methoden `setMinimumTime`, `setMaximumTime`, `setTime`, `minimumTime`, `maximumTime` und `time` mit den Bedeutungen für die Zeitangabe, die die jeweiligen Datumsäquivalente für die Datumsangabe haben. Beachten Sie analog zu den Datumsmethoden, dass Qt die Klasse `QTime` für Zeitangeben verwendet.

Neben dem Signal `dateChanged(const QDate &)` [Das Signal wurde mit seiner C++-Parametersignatur angegeben. Diese muss zum einen beim Verbinden eines Slots zu diesem Signal angegeben werden und gibt zum anderen Aufschluss über die Anzahl und Typen der Parameter, die einem verbundenen Slot übergeben werden.] , das bei einer Änderung des Datums gesendet wird, existieren die Signale `timeChanged(const QTime &)` sowie `dateTimeChanged(const QDateTime &)`, die bei einer Änderung der Zeit bzw. in beiden Fällen gesendet werden.

top

24.8.5 QDial ▼▲

Die Klasse `QDial` repräsentiert ein sogenanntes Dial-Widget in der grafischen Benutzeroberfläche. Ein Dial-Widget ist eine Art Drehregler ähnlich dem Lautstärkeregler einer Stereoanlage.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.36 Zwei Dial-Widgets

Das Dial-Widget wird intern ganz ähnlich angesprochen wie ein sogenanntes Slider-Widget, das einen Schieberegler repräsentiert. Beide Widgets erben von derselben Basisklasse, `QAbstractSlider`.

Der einzige Unterschied zwischen einem Slider und dem Dial-Widget ist das sogenannte Wrapping des Dial-Widgets. Dieser Modus ist standardmäßig deaktiviert und bedeutet, dass übergangslos zwischen dem ersten und dem letzten möglichen Wert gewechselt werden kann. Wenn das Wrapping deaktiviert ist, besteht zwischen dem ersten und dem letzten Wert eine Lücke. In Abbildung 24.36 ist das an den Strichen deutlich zu sehen. Eine Einsatzmöglichkeit für den Wrapping-Modus wäre ein Dial-Widget, bei dem ein Winkel in Grad angegeben werden soll. Dabei sollte übergangslos zwischen 0° und 359° gewechselt werden können.

Der Wrapping-Modus kann mithilfe der Methode `setWrapping` aktiviert bzw. deaktiviert und mithilfe der Methode `wrapping` ausgelesen werden.

Näheres zur Verwendung des Dial-Widgets erfahren Sie in Abschnitt 24.8.19, in dem das Slider-Widget besprochen wird.

top

24.8.6 QDialog ▼▲

Die Klasse `QDialog` repräsentiert einen Dialog in der grafischen Benutzeroberfläche. Ein Dialog unterscheidet sich von normalen Widgets dahingehend, dass ein Dialog immer ein sogenanntes Toplevel-Widget ist, sich also nicht in ein anderes Widget einbetten lässt. Abgesehen von diesem Unterschied, kann `QDialog` wie eine Widgetklasse verwendet werden, da sie, wie alle Widgets, von der Basisklasse `QWidget` abgeleitet ist. Eine `QDialog`-Instanz kann verwendet werden, um im wahrsten Sinne des Wortes einen Dialog mit dem Benutzer zu führen, beispielsweise um gewisse Daten von diesem einzulesen.

Grundsätzlich unterscheidet man zwei Arten von Dialogen – modale und nicht modale Dialoge:

- Unter einem *modalen Dialog* versteht man einen Dialog, der sich im Vordergrund der Anwendung platziert und sich den Eingabefokus klaut. Es können keine anderen Dialoge vom Benutzer bedient werden, während ein modaler Dialog geöffnet ist. Ein modaler Dialog bietet sich also für eine wichtige Teilkommunikation mit dem Benutzer an, die für den weiteren Programmlauf essenziell ist.
- Dem gegenüber steht der *nicht modale Dialog*. Wird ein Dialog nicht modal geöffnet, so kann er parallel zum restlichen Teil der Anwendung bedient werden. Ein bekanntes Beispiel für einen nicht modalen Dialog ist der »Suchen und Ersetzen«-Dialog von vielen Textverarbeitungsprogrammen, bei dem es dem Benutzer möglich sein muss, während des geöffneten Dialogs Änderungen im Hauptfenster vorzunehmen.

Um einen Dialog modal anzuzeigen, wird die Methode `exec` aufgerufen. Diese blockiert den Programmfluss so lange, bis der Benutzer den Dialog beendet hat. Der Rückgabewert von `exec` gibt an, in welcher Form der Dialog beendet wurde. Es wird eine der beiden Konstanten `QtCore.Qt.Accepted` und `QtCore.Qt.Rejec ted` zurückgegeben, wobei die erste ein Beenden des Dialogs über den **OK**- und die zweite ein Beenden über den **Abbrechen**-Button repräsentiert. Innerhalb der Dialogklasse können die Methoden `accept` und `reject` aufgerufen werden, um den Dialog mit dem entsprechenden Rückgabewert zu beenden.

Ein nicht modaler Dialog wird mithilfe der Methode `show` angezeigt. Diese Methode kehrt sofort zurück, ohne auf das Beenden des Dialogs zu warten, und ermöglicht somit das parallele Verarbeiten von Dialog und Hauptanwendung.

top

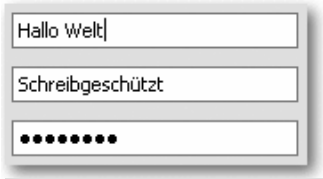
24.8.7 QGLWidget ▼▲

Die Klasse `QGLWidget` repräsentiert ein Widget in der grafischen Benutzeroberfläche, das eine 3D OpenGL-Szene anzeigt. Auf diese Weise kann Qt beispielsweise als Benutzeroberfläche für ein Computerspiel oder für einen Editor für dreidimensionale Daten verwendet werden. Die Klasse `QGLWidget` ist zwar sehr interessant und bietet eine Fülle von Möglichkeiten, liegt aber auch nicht im Fokus dieses Buches und soll deshalb hier nur der Vollständigkeit halber erwähnt werden. Näheres zum GL-Widget finden Sie in der Qt- bzw. PyQt-Dokumentation.

top

24.8.8 QLineEdit

Die Klasse `QLineEdit` repräsentiert ein LineEdit-Widget in der grafischen Benutzeroberfläche. Ein LineEdit-Wdiget ermöglicht es, einen einzeiligen Text vom Benutzer einzulesen. Ein LineEdit-Widget kann optional sowohl lese- als auch schreibgeschützt sein, sodass beispielsweise auch eine Passworteingabe mit der Klasse `QLineEdit` realisiert werden kann.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.37 Drei LineEdit-Widgets

Die Verwendung des Eingabefeldes ist recht einfach und basiert im Wesentlichen auf den Methoden `setText` und `text`, mit denen der enthaltene Text geschrieben bzw. ausgelesen werden kann.

Zusätzlich kann ein LineEdit-Eingabefeld (wie in Abbildung 24.37 zu sehen ist) in verschiedene Modi versetzt werden:

- Im normalen Modus kann der enthaltene Text sowohl vom Benutzer als auch vom Programm verändert und gelesen werden.
- Im schreibgeschützten Modus kann der enthaltene Text vom Benutzer zwar gelesen, aber nicht verändert werden. Für das Programm selbst ist das Eingabefeld dann selbstverständlich nicht schreibgeschützt.
- Im lesegeschützten Modus kann der enthaltene Text vom Benutzer zwar geschrieben, aber nicht gelesen werden. Das ist insbesondere für Passworteingaben interessant. Das Programm selbst kann selbstverständlich auch ein lesegeschütztes Feld auslesen.

Um ein LineEdit-Widget schreibgeschützt zu schalten, muss die Methode `setReadOnly` aufgerufen und `True` übergeben werden. Um den Wert auszulesen, kann die Methode `readOnly` aufgerufen werden. Um ein LineEdit-Widget lesegeschützt zu schalten, muss die Methode `setEchoMode` aufgerufen und eine der folgenden Konstanten übergeben werden:

Attribut	Beschreibung
`QtGui.QLineEdit.Normal`	Aktiviert den normalen Eingabemodus.
`QtGui.QLineEdit.NoEcho`	Aktiviert einen lesegeschützten Eingabemodus, bei dem überhaupt nichts angezeigt wird.
`QtGui.QLineEdit.Password`	Aktiviert einen lesegeschützten Eingabemodus, bei dem jeder eingegebene Buchstabe durch einen dicken Punkt ersetzt wird.

Tabelle 24.10 Eingabemodi eines LineEdit-Widgets

top

24.8.9 QListView

Die Klasse `QListView` repräsentiert ein ListView-Widget in der grafischen Benutzeroberfläche. Ein ListView-Widget zeigt die Daten einer entsprechenden Modellklasse in Form einer Liste an. Üblicherweise wird ein eigenes Widget erstellt, das von der Klasse `QListView` abgeleitet wird, um die Art der Anzeige anzupassen.

Eine Einführung in das Konzept der Modell-View-Programmierung anhand des ListView-Widgets finden Sie in Abschnitt 24.7.

top

24.8.10 QListWidget

Die Klasse `QListWidget` repräsentiert ein List-Widget in der grafischen Benutzeroberfläche. Ein List-Widget dient einem ähnlichen Zweck wie das ListView-Widget, nämlich dazu, aufbereitete Daten in Form einer Liste anzuzeigen. Der Unterschied besteht darin, dass für `QListView` keine Modell-View-Architektur erforderlich ist.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.38 Ein List-Widget

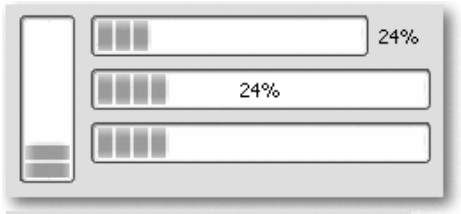
Ein List-Widget wird immer dann eingesetzt, wenn das Erstellen einer Model-View-Architektur für die anzuzeigenden Daten unverhältnismäßig aufwendig wäre und die Flexibilität, die eine Model-View-Architektur bietet, nicht benötigt wird.

Ein einzelner Eintrag in der Liste wird durch eine Instanz der Klasse `QListWidget Item` repräsentiert, die mithilfe der Methode `addItem` zur Liste hinzugefügt werden kann. Die Methode `currentItem` gibt den momentan ausgewählten Eintrag zurück.

top

24.8.11 QProgressBar

Die Klasse `QProgressBar` repräsentiert einen Fortschrittsbalken (engl. *progress bar*) in einer grafischen Benutzeroberfläche. Ein Fortschrittsbalken zeigt den Fortschritt einer langwierigen Operation an. Der Benutzer kann dabei aus naheliegenden Gründen keine Eigenschaften des Fortschrittsbalkens verändern; das ProgressBar-Widget ist für den Benutzer ausschließlich lesbar.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.39 Vier ProgressBar-Widgets

Einem ProgressBar-Widget wird keine Prozentzahl übergeben, die es anzuzeigen hat, sondern es wird auf der Basis von Einzelschritten gearbeitet. Mithilfe der Methoden `setMinimum` und `setMaximum` wird ein ganzzahliger Bereich festgelegt, in dem sich der momentane Status bewegt. Mit `setValue` wird dem ProgressBar-Widget der aktuelle, ebenfalls ganzzahlige Status mitgeteilt, der zwischen dem angegebenen Minimum und dem Maximum liegen muss. Das Widget errechnet daraus eine Prozentzahl und zeigt den Fortschrittsbalken dementsprechend an.

Wenn sowohl das Minimum als auch das Maximum auf `0` gesetzt

wurden, zeigt die ProgressBar den sogenannten *Busy Indicator* an. Das ist ein kurzes Stück des Fortschrittsbalkens, das im Widget hin- und herläuft (siehe Abbildung 24.40).

Hier klicken, um das Bild zu vergrößern

Abbildung 24.40 Der Busy Indicator

Optional kann der Fortschritt zusätzlich zum Balken in Form einer Prozentanzeige visualisiert werden. Dazu muss die Methode `setTextVisible` aufgerufen und `True` übergeben werden.

Ein ProgressBar-Widget kann, wie in Abbildung 24.39 zu sehen ist, entweder horizontal oder vertikal verlaufen. Um diese Eigenschaft an die jeweiligen Umstände anzupassen, wird die Methode `setOrientation` mit einer der beiden Konstanten `QtCore.Qt.Horizontal` bzw. `QtCore.Qt.Vertical` als Parameter aufgerufen.

top

24.8.12 QPushButton

Die Klasse `QPushButten` repräsentiert einen Button, auch Schaltfläche genannt, in der grafischen Benutzeroberfläche. Ein Button ist ein beschriftetes Widget, auf das der Benutzer klicken kann, um eine mit dem Button assoziierte Aktion auszulösen.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.41 Drei PushButton-Widgets

Die häufigste Art des Buttons ist die, die in Abbildung 24.41 mit »Klick mich« beschriftet ist. Auf diese Weise präsentiert sich ein PushButton-Widget im Normalfall. Die Beschriftung des Buttons kann mit der Methode `setText` festgelegt werden.

Das Einzige, was zur Verwendung des Buttons noch fehlt, ist das Signal, das beim Klicken auf selbigen gesendet wird. Dabei handelt es sich um das `clicked()`-Signal. Das Programm sollte einen Slot mit diesem Signal verbinden, um eine Aktion beim Klicken des Buttons zu starten.

Eine Variante des Push Buttons ist der Toggle Button. Das ist ein Button, der sich ähnlich wie eine Check Box aktivieren und deaktivieren lässt. Ein solcher Button ist in Abbildung 24.41 zu sehen und mit »Check mich« beschriftet. Der Toggle-Modus wird aktiviert, indem die Methode `setCheckable` der Klasse `QPushButton` aufgerufen und `True` übergeben wird. Dann kann der Status des Buttons, ähnlich wie der einer Check Box, mit der Methode `isChecked` herausgefunden und mittels `setCheck` gesetzt werden.

Zu guter Letzt kann ein Button durch Aufruf der Methoden `setDafault` noch zu einem sogenannten *Default Button* gemacht werden. Ein solcher Button wird immer dann aktiviert, wenn der Benutzer auf die Return- oder Leertaste drückt, egal, welches Steuerelement momentan den Fokus hat. Der Default Button wird hervorgehoben. Es kann in einem Dialog immer nur einen solchen Button geben.

top

24.8.13 QRadioButton

Die Klasse `QRadioButton` repräsentiert einen Radio Button in der grafischen Benutzeroberfläche. Ein Radio Button wird verwendet, um den Benutzer eine Auswahl aus mehreren vorgegebenen Möglichkeiten treffen zu lassen.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.42 Drei Radio Buttons

Alle Radio Buttons eines Dialogs, die über dasselbe Elternwidget verfügen, gehören zu einer Gruppe. Innerhalb einer Gruppe kann immer nur ein Radio Button aktiviert sein. Wenn der Benutzer einen anderen aktiviert, wird der vorher aktivierte inaktiv. Um mehrere Gruppen von Radio Buttons in demselben Dialog unterzubringen, müssen diese über verschiedene Elternwidgets verfügen. Dazu bietet sich das sogenannte *Frame-Widget*, repräsentiert durch die Klasse `QFrame`, an, das einem unsichtbaren Rahmen im Widget entspricht und als Elternwidget für die Radio Buttons dienen kann.

Ein Radio Button sendet das Signal `toggled()`, wenn der Benutzer ihn aktiviert oder deaktiviert. Ansonsten kann der Status eines Radio Buttons, ähnlich wie bei einer Check Box, über die Methode `isChecked` erfragt werden.

top

24.8.14 QScrollArea

Die Klasse `QScrollArea` repräsentiert ein ScrollArea-Widget in der grafischen Benutzeroberfläche. Ein ScrollArea-Widget besitzt ein untergeordnetes Widget, dessen Größe möglicherweise die Größe des ScrollArea-Widgets überschreitet. Wenn dies der Fall ist, sorgt das ScrollArea-Widget dafür, dass Scrollbars für die entsprechende Seite entstehen, mit deren Hilfe das übergroße untergeordnete Widget gescrollt werden kann.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.43 Ein ScrollArea-Widget

Das untergeordnete Widget einer `QScrollArea`-Instanz wird mithilfe der Methode `setWidget` festgelegt. In Abbildung 24.43 wurde dem ScrollArea-Widget ein zugegebenermaßen ungewöhnlich großes Dial-Widget zum Scrollen übergeben.

top

24.8.15 QSlider

Die Klasse `QSlider` repräsentiert einen Slider in der grafischen Benutzeroberfläche. Bei einem Slider handelt es sich um einen Schieberegler, wie Sie ihn beispielsweise von einem Mischpult her

kennen. Grundsätzlich wird ein Slider dazu verwendet, den Benutzer einen ganzzahligen Wert auswählen zu lassen, der innerhalb eines bestimmten Bereichs liegen muss. Dabei sollte der tatsächliche Wert für den Benutzer eher weniger interessant sein, da dieser von einem Slider nicht angezeigt wird.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.44 Drei Slider

Der Slider ist von der Basisklasse `QAbstractSlider` abgeleitet, von der auch das Dial-Widget erbt. Die Verwendung dieser beiden Widgets ist weitestgehend identisch.

Ein Slider wird durch Aufruf der Methoden `setMinimum` und `setMaximum` initialisiert. Diese Methoden legen den ganzzahligen Mindest- bzw. Maximalwert fest, zwischen denen der Benutzer einen beliebigen Wert wählen darf. Mithilfe der Methode `setValue` kann der Anfasser des Sliders auf eine bestimmte Position gesetzt werden. Analog dazu erlaubt es die Methode `value`, die Position des Anfassers auszulesen. Der Slider sendet das Signal `valueChanged()` immer dann, wenn der Benutzer den Anfasser auf eine neue Position bewegt und losgelassen hat.

Optional kann ein Slider mit einer Skala ausgestattet werden. Dazu muss die Methode `setTickPosition` mit einem geeigneten Parameter aufgerufen werden. Als Parameter kann beispielsweise `QtCore.Qt.NoTicks` übergeben werden, um eine Skala abzuschalten, oder `QtCore.Qt.TicksAbove` bzw. `QtCore.Qt.Ticks Below`, um die Skala über bzw. unter dem Slider anzuzeigen.

Zu guter Letzt kann ein Slider, wie in Abbildung 24.44 zu sehen ist, sowohl horizontal als auch vertikal ausgerichtet sein. Um die Ausrichtung eines Sliders zu verändern, wird die Methode `setOrientation` aufgerufen und eine der beiden Konstanten `QtCore.Qt.Horizontal` bzw. `QtCore.Qt.Vertical` übergeben.

top

24.8.16 QTableView

Die Klasse `QTableView` repräsentiert ein TableView-Widget in der grafischen Benutzeroberfläche. Ein TableView Widget zeigt tabellarische Datenstrukturen, beispielsweise die Bundesliga-Tabelle, unter Verwendung einer Model-View-Architektur an.

Ein Beispiel für eine Model-View-Architektur, allerdings bezogen auf ein ListView-Widget, finden Sie in Abschnitt 24.7. Sie können `QTableView` als View- und beispielsweise `QAbstractTableModel` als Modellklasse verwenden. Genauere Informationen und Beispiele zu diesem Thema sind in der Qt-Dokumentation zu finden.

top

24.8.17 QTableWidget

Die Klasse `QTableWidget` repräsentiert ein Table-Widget in der grafischen Benutzeroberfläche. Ein Table Widget hat einen ähnlichen Zweck wie ein TableView-Widget, mit dem Unterschied aber, dass das Table Widget keine Model-View-Architektur voraussetzt.

Hier klicken, um das Bild zu vergrößern

Abbildung 24.45 Ein Table-Widget

Das Table-Widget wird besonders dann eingesetzt, wenn das Schreiben einer eigenen Model-View-Architektur unverhältnismäßig viel Arbeit bedeutet und die Flexibilität, die eine solche Architektur bietet, nicht benötigt wird.

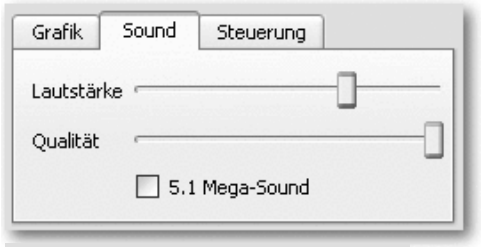
Nachdem eine Instanz der Klasse `QTableWidget` erzeugt wurde, können mithilfe der Methoden `setRowCount` und `setColumnCount` die Anzahl der Zeilen und Spalten festgelegt werden. Danach können einzelne Einträge in jeweils eine Zelle der Tabelle geschrieben werden. Um einen Eintrag hinzuzufügen, wird die Methode `setItem` aufgerufen, die den Zeilen- und Spaltenindex der Zelle und eine `QTableWidgetItem`-Instanz übergeben bekommt. Diese Instanz beschreibt das einzufügende Element.

Die Zeilen und Spalten einer Tabelle können mithilfe der Methoden `setHorizontalHeaderLabels` und `setVerticalHeaderLabels` beschriftet werden. Diese Methoden bekommen jeweils eine Liste von Strings als Parameter übergeben.

top

24.8.18 QTabWidget ▼▲

Die Klasse `QTabWidget` repräsentiert ein Tab-Widget in der grafischen Benutzeroberfläche. Ein Tab-Widget zeigt mehrere Karteikartenreiter an, zwischen denen der Benutzer wechseln kann. Je nachdem, welcher Reiter aktiv ist, werden unterschiedliche Steuerelemente im Tab-Widget angezeigt. Das Tab-Widget findet häufig in Konfigurationsdialogen Anwendung, wenn viele Steuerelemente auf relativ wenig Raum einfach zu erreichen sein müssen.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.46 Tab-Widget im Konfigurationsdialog eines fiktiven Spiels

Nachdem eine Instanz der Klasse `QTabWidget` erzeugt worden ist, können die einzelnen Tabs und die dazugehörenden sogenannten Seiten erstellt werden. Eine Seite ist nichts anderes als ein Widget, dem die in der Seite gewünschten Steuerelemente untergeordnet sind. Beachten Sie, dass das Seitenwidget kein Elternwidget benötigt. Mithilfe der Funktion addTab wird ein Tab mitsamt Seite zum Tab-Widget hinzugefügt.

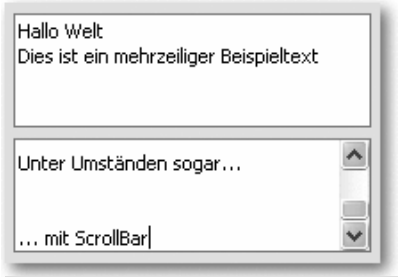
Auf die aktuell angezeigte Seite kann mithilfe der Methoden `currentIndex` oder `currentWidget` zugegriffen werden. Analog dazu kann die aktuelle Seite mit `setCurrentIndex` bzw. `setCurrentWidget` verändert werden.

Wenn der Benutzer eine neue Seite auswählt, wird das Signal `currentChanged(int)` gesendet.

top

24.8.19 QTextEdit

Die Klasse `QTextEdit` repräsentiert ein mehrzeiliges Eingabefeld in der grafischen Benutzeroberfläche. In ein solches Eingabefeld kann der Benutzer beliebigen Text schreiben.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.47 Zwei TextEdit-Widgets

Ein TextEdit-Widget liegt nicht nur in der Vorstellung sehr nahe beim bereits besprochenen LineEdit-Widget, sondern wird programmintern auch ganz ähnlich angesprochen. Im Gegensatz zum einzeiligen Eingabefeld ist das TextEdit-Widget dazu in der Lage, Rich Text und HTML darzustellen.

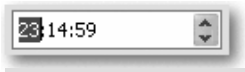
Im Falle von reinem Text, wenn es sich also weder um HTML noch um Rich Text handelt, kann der Inhalt eines TextEdit-Widgets mit der Methode `toPlainText` ausgelesen und mittels `setPlainText` geschrieben werden. Außerdem kann das TextEdit-Widget ähnlich wie bei LineEdit durch Aufruf der Methode `setReadOnly` in einen schreibgeschützten Modus versetzt werden, in dem der Benutzer den Inhalt des Widgets nicht verändern kann.

Das TextEdit-Widget bietet insbesondere in puncto HTML und Rich Text eine Fülle von Möglichkeiten, die in der Qt-Dokumentation im Detail besprochen werden.

top

24.8.20 QTimeEdit

Die Klasse `QTimeEdit` repräsentiert ein TimeEdit-Widget in der grafischen Benutzeroberfläche. Ein solches Widget erlaubt es, eine Zeitangabe vom Benutzer einzulesen. Die Klasse hängt eng mit den bereits besprochenen Klassen `QDateEdit` und `QDateTimeEdit` zusammen.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.48 Ein TimeEdit-Widget

Die Klasse `QTimeEdit` unterstützt die Funktionalität der Klasse `QDateTimeEdit`, die sich auf Zeitangaben bezieht.

top

24.8.21 QTreeView

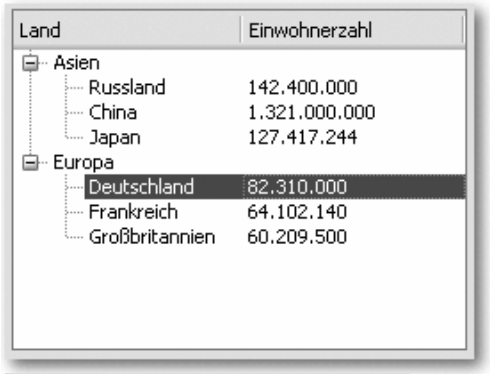
Die Klasse `QTreeView` repräsentiert ein TreeView-Widget in der grafischen Benutzeroberfläche. Ein TreeView-Widget zeigt baumförmige Datenstrukturen, beispielsweise den Inhalt eines Ordners mitsamt Unterordnern, unter Verwendung einer Model-View-Architektur an.

Ein Beispiel für eine Model-View-Architektur, allerdings bezogen auf ein ListView-Widget, finden Sie in Abschnitt 24.7. Dabei können Sie `QTreeView` als View- und beispielsweise `QAbstractItemModel` als Modellklasse verwenden. Genauere Informationen und Beispiele zu diesem Thema sind in der Qt-Dokumentation zu finden.

top

24.8.22 QTreeWidget ▼▲

Die Klasse `QTreeWidget` repräsentiert ein Tree-Widget in der grafischen Benutzeroberfläche. Das Tree-Widget zeigt, ähnlich wie das TreeView-Widget, hierarchisch aufgebaute Datenstrukturen an. Im Gegensatz zum TreeView-Widget setzt das Tree-Widget jedoch keine Model-View-Architektur voraus.



Hier klicken, um das Bild zu vergrößern

Abbildung 24.49 Ein Tree-Widget

Die Klasse `QTreeWidget` implementiert die klassische Schnittstelle eines Tree-Widgets, die so ähnlich auch in vielen anderen GUI-Toolkits vorhanden ist. Allgemein sollte das Tree-Widget nur dann verwendet werden, wenn das Erstellen einer eigenen Model-View-Architektur einen unverhältnismäßig hohen Aufwand darstellt und auf die Flexibilität des TreeView-Widgets verzichtet werden kann.

Nach dem Instanziieren der Klasse `QTreeWidget` kann eine optionale Kopfzeile eingerichtet werden, die die Spaltenbeschreibungen enthält. In Abbildung 24.49 kennzeichnet die Kopfzeile die Spalten »Land« und »Einwohnerzahl«. Zum Anlegen einer Kopfzeile und zum Beschriften der Spalten wird die Methode `setHea derLabels` aufgerufen. Diese bekommt eine Liste von Strings übergeben und legt für jeden der enthaltenen Strings eine eigene Spalte im Tree-Widget an.

Nachdem das Widget initialisiert wurde, kann mithilfe der Methode `addTop LevelItem` ein Eintrag auf höchster Ebene hinzugefügt werden. Bei einem Eintrag handelt es sich um eine Instanz der Klasse `QTreeWidgetItem`, die ihrerseits über untergeordnete Einträge verfügen kann. Beim Instanziieren eines Eintrags wird dem Konstruktor der gewünschte Elterneintrag übergeben und die Baumstruktur auf diese Weise erzeugt. Wenn die Einträge erstellt wurden, kann der oberste Eintrag mittels `addTopLevelItem` zum Tree-Widget hinzugefügt werden, was alle untergeordneten Einträge einschließt.

top

24.8.23 QWidget ▲

Die Klasse `QWidget` ist die Basisklasse aller Steuerelemente und

implementiert einen Großteil der steuerelementübergreifenden Funktionalität. Allgemein können Widgets in zwei Sorten unterteilt werden:

- Der häufigste Fall ist ein *untergeordnetes Widget*, das dadurch charakterisiert ist, dass es ein übergeordnetes Widget (*Parent*) besitzt. Ein solches untergeordnetes Widget wird relativ zu seinem Parent positioniert und kann nicht über die Ränder des Parent-Widgets hinausgehen.
- Ein Widget ohne Parent-Widget ist ein sogenanntes *Fenster* (engl. *Window*). Theoretisch kann jede Instanz einer von `QWidget` abgeleiteten Klasse als Fenster dienen, beispielsweise also auch ein Button oder eine Check Box. Praktisch macht dies jedoch wenig Sinn, und es werden nur geeignetere Klassen wie beispielsweise `QDialog` als Fenster verwendet.

In diesem Kapitel soll ein knapper Überblick über die Funktionalität gegeben werden, die die Klasse `QWidget` bereitstellt und die demzufolge an jedes Steuerelement weitervererbt wird. Beachten Sie dabei, dass es sich bei `QWidget` um eine ausgesprochen umfangreiche Klasse handelt, die hier keinesfalls vollständig beschrieben werden kann.

Die folgende Tabelle listet die wichtigsten Eventhandler auf, die beim Auftreten verschiedener Events aufgerufen werden. Ein Eventhandler kann eingerichtet werden, indem eine selbst erstellte Klasse, die von `QWidget` direkt oder von einer Steuerelementklasse erbt, die genannte Methode implementiert.

Event	Beschreibung
`paintEvent`	Wird immer dann aufgerufen, wenn das Widget neu gezeichnet werden muss. Die Methode bekommt eine Instanz der Klasse `QPaintEvent` übergeben, die beispielsweise den neu zu zeichnenden Bereich enthält.
`resizeEvent`	Wird immer dann aufgerufen, wenn das Widget in seiner Größe verändert wurde. Der Methode wird eine Instanz der Klasse `QResizeEvent` übergeben, die unter anderem die alte und neue Größe des Widgets enthält.
`mousePressEvent`	Wird immer dann aufgerufen, wenn der Benutzer mit der Maus auf das Widget klickt. Der Methode wird eine Instanz der Klasse `QMouseEvent` übergeben, die beispielsweise die Koordinaten des Mausklicks enthält.
`mouseDoubleClickEvent`	Wird immer dann aufgerufen, wenn der Benutzer doppelt auf das Widget klickt. Beachten Sie, dass vor diesem Event immer ein `mousePressEvent`, ein `mouseReleaseEvent` und möglicherweise ein `mouseMoveEvent` auftreten. Es ist aus naheliegenden Gründen nicht möglich, bereits beim ersten Klick zwischen einem einfachen und einem doppelten Klick zu unterscheiden. Es wird eine `QMouseEvent`-Instanz übergeben.
`mouseMoveEvent`	Wird immer dann aufgerufen, wenn der Benutzer die Maus bewegt, während er eine Maustaste gedrückt hält. Wenn zuvor die Methode `setMouseTracking` aufgerufen und `True` übergeben wurde, wird auch dann ein `mouseMoveEvent`

	erzeugt, wenn der Benutzer die Maus bewegt, ohne eine Taste gedrückt zu halten.
`keyPressEvent`	Wird immer dann aufgerufen, wenn das Widget den Eingabefokus hat und der Benutzer eine Taste drückt. Wenn die Taste ausreichend lange gedrückt wird, wird die Methode `keyPressEvent` immer wieder aufgerufen. Diese Funktionalität wird `auto-repeat` genannt. Es wird eine Instanz der Klasse `QKeyEvent` übergeben, über die beispielsweise die gedrückte Taste ausgelesen werden kann.

Tabelle 24.11 Events eines Widgets

Und damit beschließen wir das Kapitel über die Programmierung grafischer Benutzeroberflächen und wenden uns einem weiteren interessanten Themenkomplex zu: der Webentwicklung mit Python.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

25.2 Konzepte und Besonderheiten im Überblick

Django setzt auf das *Model-View-Konzept*, das Sie schon im Kapitel über Qt kennengelernt haben. Eine typische Django-Anwendung definiert ein *Datenmodell*, das automatisch von Django in einer Datenbank verwaltet wird. Die Ausgabe für Endbenutzer übernehmen sogenannte *Views* (dt. *Anzeigen*), die auf das Datenmodell zurückgreifen können.

Zurzeit unterstützt Django die Datenbanken MySQL, SQLite3 und PostgreSQL. [SQLite3 ist als Modul in Pythons Standardbibliothek enthalten und kann ohne besondere Konfiguration sofort benutzt werden. Für MySQL und PostgreSQL benötigen Sie einen separaten Datenbankserver.] Außerdem befinden sich Anbindungen für Microsofts SQL-Server MSSQL und für den Oracle-Datenbankserver in der Entwicklung.

Wir als Webentwickler werden bei dem Umgang mit den Datenmodellen vor technischen Details wie der Abfragesprache SQL vollkommen verschont. Unsere Aufgabe besteht nur darin, Django mitzuteilen, welche Datenbank wir gerne verwenden möchten. Unsere Modelle werden über Python-Code definiert, und Django übernimmt die komplette Kommunikation mit der Datenbank für uns.

Als besonderes Bonbon für den Webentwickler erstellt Django anhand der Modelldefinition automatisch eine Administrationsoberfläche, um die Daten des Modells zu verwalten.

Django unterscheidet zwischen *Projekten* und *Applikationen*. Als Projekt wird eine Website als Ganzes bezeichnet, die beispielsweise eine News-Seite, ein Forum und ein Gästebuch umfassen kann. Applikationen sind dafür zuständig, die

Funktionen eines Projekts zu realisieren. Unser Beispielprojekt hätte also drei Applikationen: eine News-Applikation, eine Forum-Applikation und eine Applikation für das Gästebuch.

Django vertritt das Prinzip des *Loose Coupling* (dt. *schwache Kopplung*) für alle Teile der Webanwendung. Das bedeutet, dass möglichst viel unabhängig voneinander entwickelt werden kann, sodass die einzelnen Teile auf anderen Seiten verwendet werden können. Insbesondere ist es problemlos möglich, Applikationen in verschiedenen Projekten zu verwenden, ohne den Code anpassen zu müssen.

Wir werden im nächsten Abschnitt mit der Entwicklung des Beispielprojekts beginnen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

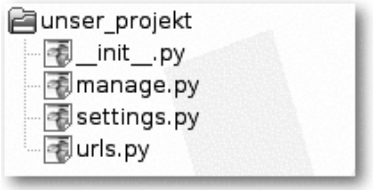
25.3 Erstellen eines neuen Django-Projekts

Bevor wir mit Django arbeiten können, müssen wir einen Projektordner erstellen, der die Konfiguration unseres Projekts enthält.

Zu diesem Zweck liefert Django ein Tool namens *django-admin.py*, mit dessen Hilfe die Erzeugung eines neuen Projekts zum Kinderspiel wird. Sie müssen einfach in der Kommandozeile Folgendes eingeben:

```
$ django-admin.py startproject unser_projekt
```

Die Zeichenfolge `unser_projekt` ist dabei der gewünschte Name des neuen Projekts. Nach der erfolgreichen Ausführung von *django-admin.py* existiert eine neue Ordnerstruktur mit den Projektdateien auf der Festplatte (siehe Abbildung 25.2).



Hier klicken, um das Bild zu vergrößern

Abbildung 25.2 Ordnerstruktur eines frischen Django-Projekts

In der nachstehenden Tabelle sind die Zwecke der einzelnen Dateien erklärt:

Dateiname	Zweck

__init__.py	Die Datei ist standardmäßig leer und nur deshalb notwendig, damit das Projekt als Python-Modul importierbar ist. Jedes Django-Projekt muss ein gültiges Python-Modul sein.
manage.py	Dieses Programm dient zur Verwaltung unseres Projekts. Mit ihm können wir beispielsweise die Datenbank aktualisieren oder das Projekt für die Fehlersuche ausführen.
settings.py	In dieser Datei sind alle Einstellungen für unser Projekt gespeichert. Dazu zählt insbesondere die Konfiguration der Datenbank.
urls.py	Legt fest, über welche Adressen die Seiten unseres Projekts erreichbar sein sollen.

Tabelle 25.1 Bedeutung der Dateien in einem Django-Projekt

Nun können wir mit dem Projekt arbeiten.

Der Entwicklungswebserver

Das frische Projektskelett kann in diesem Zustand schon gestestet werden. Um die Entwicklung von Django-Anwendungen zu erleichtern, bietet das Framework einen einfachen Webserver an. Dieser Server überwacht ständig die Änderungen, die Sie am Code Ihrer Anwendung vornehmen, und lädt automatisch die veränderten Programmteile nach, ohne dass er dazu neu gestartet werden muss.

Den Entwicklungswebserver können Sie über das Verwaltungsprogramm *manage.py* starten, indem Sie den Parameter runserver übergeben. Bevor Sie den Befehl ausführen, müssen Sie in das Projektverzeichnis *unser_projekt* wechseln. Unter Windows entfällt das Kommando python zu Beginn des folgenden Beispiels.

```
$ python manage.py runserver
Validating models...
0 errors found.

Django version 0.96, using settings
'unser_projekt.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Wie Sie der Ausgabe entnehmen können, ist der Entwicklungsserver unter der Adresse http://127.0.0.1:8000/ erreichbar. Die Seite sieht im Browser folgendermaßen aus (siehe Abbildung 25.3).

Wichtig

Djangos Einwicklungswebserver ist ausschließlich für die *Entwicklung* von Webanwendungen gedacht und sollte niemals für eine *öffentliche* Seite benutzt werden. Verwenden Sie für den produktiven Einsatz Ihrer Webanwendungen »echte« Webserver wie beispielsweise den Apache-Webserver. Eine Anleitung zur Konfiguration des Apache mit Django finden Sie in der Django-Dokumentation.

Hier klicken, um das Bild zu vergrößern

Abbildung 25.3 Ausgabe des noch leeren Projekts im Browser

Konfiguration des Projekts

Bevor wir an unserem Projekt arbeiten können, muss es konfiguriert werden. Dazu öffnen wir die Datei *settings.py* in einem Texteditor. Es handelt sich bei der Datei um ein einfaches Python-Programm, das Django über globale Variablen konfiguriert. Allerdings beinhaltet es auch sehr spezielle Einstellungen, die uns an dieser Stelle nicht interessieren.

Wir beschränken uns im ersten Schritt auf die Einstellungen, die die Datenbank von Django betreffen. Relativ weit am Anfang der Datei finden Sie einen Block, der folgendermaßen aussieht:

```
DATABASE_ENGINE = ''           # 'postgresql_psycopg2',
'postgresql',
                               # 'mysql', 'sqlite3' or
'ado_mssql'.
DATABASE_NAME = ''             # Or path to database file if
using
                               # sqlite3.
DATABASE_USER = ''             # Not used with sqlite3.
DATABASE_PASSWORD = ''         # Not used with sqlite3.
DATABASE_HOST = ''             # Set to empty string for
localhost. Not
                               # used with sqlite3.
DATABASE_PORT = ''             # Set to empty string for
default. Not
                               # used with sqlite3.
```

Zusammen mit den Kommentaren sollten diese Einstellungen selbsterklärend sein. Da SQLite standardmäßig mit Python ausgeliefert wird und daher bei der Verwendung von SQLite keine zusätzlichen Module installiert werden müssen, wird unser Beispielprojekt eine SQLite-Datenbank nutzen. Dazu setzen wir die beiden für SQLite relevanten Einstellungen `DATABASE_ENGINE` und `DATABASE_NAME` wie folgt:

```
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = r'C:\unser_projekt\datenbank.dat'
```

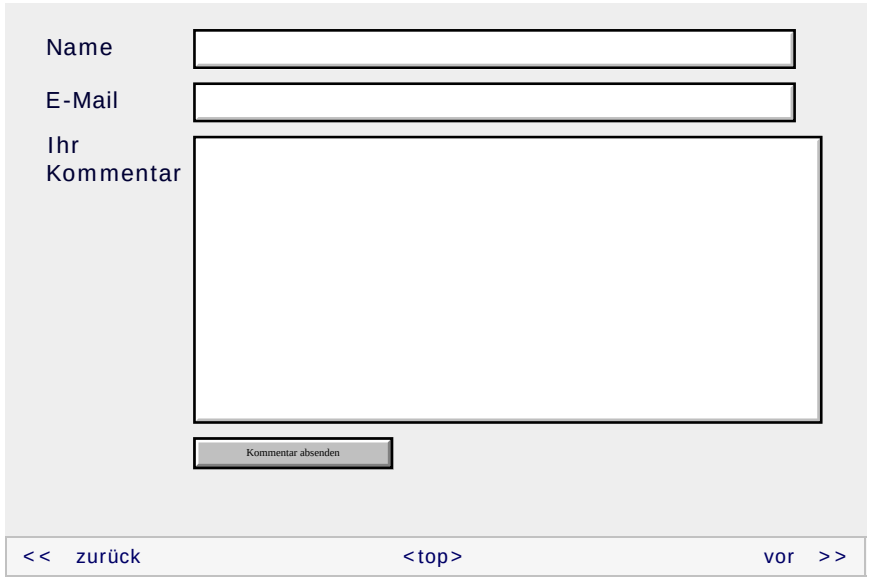
Außerdem sollten Sie die Sprache von Django auf die Gegebenheiten von Deutschland umstellen. Dies geschieht mit der Anpassung der `TIME_ZONE`- und `LANGUAGE_CODE` -Variable:

```
TIME_ZONE = 'Europe/Berlin'
LANGUAGE_CODE = 'de-DE'
```

Damit sind die grundlegenden Einstellungen gemacht, und wir können uns an unsere erste Django-Applikation wagen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen
Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

25.4 Erstellung der Applikation

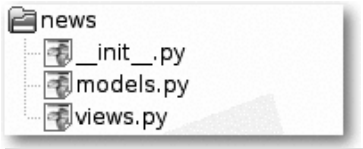
Ein Projekt ist nur der Rahmen für eine Webseite. Die eigentliche Funktionalität wird durch sogenannte *Applikationen* implementiert, die in das Projekt eingebunden werden können. Genau wie für Projekte bietet Django auch für Applikationen ein Werkzeug an, mit dem das Grundgerüst einer Applikation erzeugt werden kann.

Um unsere News-Applikation zu erzeugen, wechseln wir mit einer System-Shell in das Projektverzeichnis und führen den nachstehenden Befehl aus (das python am Anfang kann unter Windows entfallen):

```
$ python manage.py startapp news
```

Dabei ist news der gewünschte Name der neuen Applikation.

Das Programm erzeugt in unserem Projektverzeichnis einen neuen Ordner namens news, der drei Dateien enthält (siehe Abbildung 25.4).

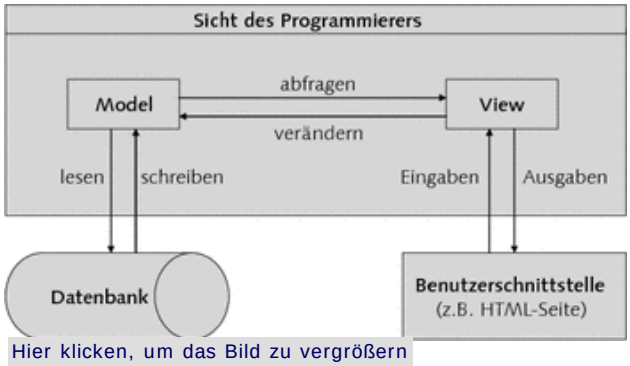


Hier klicken, um das Bild zu vergrößern

Abbildung 25.4 Ordner einer neuen Django-Applikation

Die Datei *__init__.py* ist wie bei Projekten deshalb notwendig, damit die Applikation ein importierbares Python-Modul wird. Sie ist standardmäßig leer.

Interessanter sind die beiden anderen Dateien, *models.py* und *views.py*, die dazu dienen, das Django zugrunde liegende *Model-View-Konzept* umzusetzen (siehe Abbildung 25.5).



Hier klicken, um das Bild zu vergrößern

Abbildung 25.5 Das Model-View-Konzept

Das sogenannte *Model* definiert die Struktur unserer Daten und wie damit umgegangen werden muss. Außerdem bieten Models eine komfortable Schnittstelle für den Zugriff auf die gespeicherten Daten. Wie die konkrete Kommunikation mit der Datenbank vonstatten geht, bleibt dem Programmierer verborgen. Er kann beim Umgang mit den Daten auf die Schnittstellen des Models zurückgreifen, ohne sich um technische Details wie SQL-Statements kümmern zu müssen. [Natürlich ist es möglich, eigene SQL-Befehle an die Datenbank zu senden, falls die mächtige API von Django für einen speziellen Fall nicht ausreichen sollte. In der Praxis werden Sie allerdings höchstwahrscheinlich niemals darauf zurückgreifen müssen.] Die Datenbank selbst bleibt ihm »verborgen«, weil er sie nur indirekt durch das Model »sieht«.

Die sogenannte *View* (dt. *Sicht*) kümmert sich um die Aufbereitung der Daten für den Benutzer. Sie kann dabei auf die Models zurückgreifen und deren Daten auslesen und verändern. Wie dabei die Benutzerschnittstelle konkret aussieht, ist der View egal. Die Aufgabe der Views ist nur, die vom Benutzer abgefragten Daten zu ermitteln, diese zu verarbeiten und dann an ein sogenanntes *Template* zu übergeben, das die eigentliche Anzeige übernimmt. Mit Templates werden wir uns später beschäftigen.

Ein Model definieren

Der erste Schritt nach der Erstellung einer neuen Applikation ist die Definition eines Datenmodells. Diese Definition sieht so aus, dass für alle Arten von Datensätzen, die die Applikation braucht, eine Klasse erstellt wird. Diese Klassen müssen von der Basisklasse `models.Model` im Paket `django.db` abgeleitet werden und legen die Eigenschaften der Datensätze und ihre Verknüpfungen untereinander fest.

Unser Beispielmodell für die News-Applikation definiert eine Modellklasse für die Meldungen und eine für die Besucherkommentare. Wir schreiben die Definition in die Datei *models.py*, die dann Folgendes enthält:

```
from django.db import models

class Meldung(models.Model):
    titel = models.CharField(max_length=100)
    zeitstempel = models.DateTimeField()
    text = models.TextField("Meldungstext")

class Kommentar(models.Model):
    news = models.ForeignKey(Meldung)
    autor = models.CharField(max_length=70)
    text = models.TextField("Text")
```

Die Attribute der Datensätze werden über Klassenmember

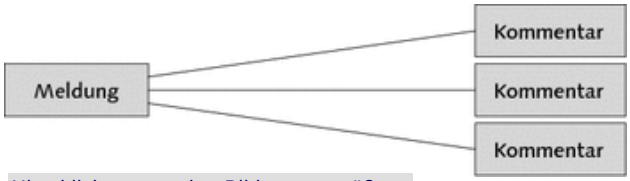
festgelegt, wobei jedes Attribut eine Instanz eines speziellen Feldtyps von Django sein muss. Über die Parameter der Feldtyp-Konstruktoren werden dabei die Eigenschaften der Attribute angegeben.

Die Klasse `Meldung` besitzt ein `CharField` namens `titel`, das eine maximale Länge von `100` Zeichen haben soll. Der Feldtyp `CharField` dient zum Speichern von Texten begrenzter Länge. Das Attribut `zeitstempel` soll den Veröffentlichungszeitpunkt jeder Meldung angeben und benutzt den für Zeitangaben gedachten Feldtyp `DateTimeField`. Im letzten Attribut namens `text` wird der eigentliche Meldungstext gespeichert. Der verwendete Feldtyp `TextField` kann beliebig lange Texte speichern.

Die beiden Attribute `autor` und `text` der Klasse `Kommentar` speichern den Namen desjenigen Besuchers, der den Kommentar geschrieben hat, und den Kommentartext selbst. Interessanter ist das Attribut `meldung`, das eine Beziehung zwischen den Meldungen und Kommentaren herstellt.

Beziehungen zwischen Modellen

Es ist so, dass es zu einer Meldung mehrere Kommentare geben kann und dass sich umgekehrt jeder Kommentar genau auf eine Meldung bezieht. Mit dem Feldtyp `ForeignKey` (dt. *Fremdschlüssel*) wird eine *One-To-Many Relation* (dt. *Eins-zu-viele-Relation*) festgelegt, die besagt, dass es zu einem Kommentar genau eine Meldung gibt.



Hier klicken, um das Bild zu vergrößern

Abbildung 25.6 One-To-Many Relation bei Meldung und Kommentar

Neben den One-To-Many Relations unterstützt Django auch *Many-To-Many* (dt. *Viele-zu-vielen*) und *One-To-One* Relations (dt. *Eins-zu-eins-Relation*), die wir aber nicht thematisieren werden.

Wir sind nun mit der Definition des Models fertig und können Django anweisen, eine entsprechende Datenbank anzulegen. Vorher müssen wir allerdings unsere Applikation noch in das Projekt einbinden. Dies wird dadurch erreicht, dass wir den Modulnamen der Applikation in das Tupel `INSTALLED_APPS` am Ende der *settings.py* unseres Projekts einfügen:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'unser_projekt.news'
)
```

Das Tupel `INSTALLED_APPS` enthält dabei die Importnamen aller Applikationen, die das Projekt verwendet. Sie werden sich jetzt wundern, warum unser eigentlich leeres Projekt schon vier Applikation enthält. Django bindet diese Applikationen standardmäßig ein, weil sie in eigentlich jedem Projekt gebraucht werden.

Nachdem wir nun unsere Applikation erzeugt, ihr Datenmodell definiert und sie in das Projekt eingefügt haben, können wir die Datenbank erstellen. Wir rufen hierzu das Programm *manage.py* mit dem Parameter syncdb auf:

```
$ python manage.py syncdb
Creating table auth_message
Creating table auth_group
```

```
Creating table auth_user
Creating table auth_permission
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table news_meldung
Creating table news_kommentar
```

Django erzeugt anhand der Model-Definition automatisch die passenden SQL-Statements für die verwendete Datenbank. Für jede Modellklasse wird eine eigene Tabelle angelegt, die Spalten für alle Attribute der Klasse enthält. Die Namen der Spalten entsprechen dabei den Attributnamen. Zusätzlich gibt es in jeder Tabelle eine Spalte namens `id`, mit der jeder Datensatz eindeutig über eine Ganzzahl identifiziert werden kann. [Sie können auch eigene `id`-Spalten definieren. Nähres finden Sie in der Django-Dokumentation.]

Nachdem die notwendigen Tabellen in der Datenbank erzeugt worden sind, fragt uns das Script, ob wir einen Benutzeraccount für die Administrationsoberfläche anlegen möchten. Wir beantworten die Frage mit »yes« und geben entsprechende Daten ein. Natürlich sollten Sie an dieser Stelle andere Zugangsdaten wählen als im Beispiel.

```
You just installed Django's auth system, which means you
don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'revelation'): Johannes
E-mail address: je@revelation-soft.de
Password:
Password (again):
Superuser created successfully.
```

Die Model-API

Bevor wir uns mit der automatisch generierten Administrationsoberfläche auseinandersetzen, wollen wir das Model-API kennenlernen, mit dem wir auf die Daten unseres Models zugreifen können. Das Programm *manage.py* kann mit dem Parameter `shell` in einem Shell-Modus gestartet werden, in dem wir über eine interaktive Python-Shell unsere Models verwenden können.

Anlegen von Datensätzen

Zuerst wollen wir die Shell nutzen, um eine News-Meldung in die Datenbank zu schreiben:

```
$ python manage.py shell
>>> from unser_projekt.news.models import Meldung,
Kommentar
>>> from datetime import datetime
>>> m = Meldung(titel="Unsere erste Meldung",
                zeitstempel=datetime.now(),
                text="Klassischerweise steht hier 'Hallo
Welt'.")
>>> m.save()
```

Mit diesem einfachen Code wurde eine neue Meldung erzeugt und auch schon in der Datenbank abgelegt. Da sowohl das Projekt aus auch die Applikation einfache Python-Module sind, können wir sie über ein `import`-Statement einbinden.

Um neue Datensätze zu erzeugen, müssen wir nur die dazugehörige Model-Klasse instanziieren. Der Konstruktor der Model-Klasse erwartet dabei Schlüsselwortparameter für alle Attribute des Datensatzes. Wichtig ist außerdem, dass Django für jeden Spaltenwert einen Wert mit einem Datentyp erwartet, der zu der Spaltendefinition passt. Deshalb muss für den Umgang mit Datumsangaben der Typ `datetime.datetime` importiert werden. Für die Textspalten sind Strings passend.

Sie können auf die Attribute einer Model-Instanz auf gewohnte Weise zugreifen und sie auch über Zuweisungen verändern:

```
>>> m.titel
'Unsere erste Meldung'
```

```
>>> m.titel = "'Hallo Welt'-Meldung"
>>> m.save()
>>> m.id
1
```

Mit der letzten Abfrage `m.id` greifen wir auf die automatisch von Django eingefügte `id`-Spalte des Models zu. Da es sich bei `m` um den ersten Eintrag handelt, hat `m.id` den Wert `1`.

Kommentare können wir direkt über die jeweilige Meldung einfügen. Durch die Bindung der Kommentar-Klasse an die Meldung bekommt jede `Meldung`-Instanz ein Attribut `kommentar_set`, das Zugriff auf die Kommentare der Meldung bietet:

```
>>> m2 = Meldung(titel="Umfrage zu Django",
                 zeitstempel=datetime.now(),
                 text="Wie findet ihr das Framework?")
>>> m2.save()
>>> k1 = m2.kommentar_set.create(autor="Heinz",
text="Super!")
>>> k2 = m2.kommentar_set.create(autor="Jens",
text="Klasse!")
>>> m2.kommentar_set.count()
2
>>> m2.save()
```

Nun gibt es eine zweite Meldung in unserer News-Tabelle, die bereits mit zwei Kommentaren versehen ist. Das erste `m2.save()` ist deshalb erforderlich, da erst beim Speichern ein `id`-Spaltenwert von der Datenbank erzeugt wird, um Kommentare mit dem Datensatz zu verknüpfen.

Es gibt noch eine unschöne Stelle in unserem Modul, die wir beheben sollten. Schauen Sie sich einmal an, was Python ausgibt, wenn wir eine `Meldung`-Instanz ausgeben lassen:

```
>>> m2
<Meldung: Meldung object>
```

Diese Form der Darstellung ist nicht sehr nützlich, da sie uns keine Informationen über den Inhalt des Objekts liefert. Sie können in der *models.py* jeder Klasse eine Methode namens `__unicode__` angeben, die eine aussagekräftige Repräsentation des Objektinhalts zurückgeben sollte. Wir ändern unsere *models.py* so ab, dass die `__unicode__`-Methoden der Klassen `Meldung` und `Kommentar` jeweils das kennzeichnende Attribut `text` zurückgeben:

```
class Meldung(models.Model):
    ...
    def __unicode__(self):
        return self.text

class Kommentar(models.Model):
    ...
    def __unicode__(self):
        return self.text
```

Damit die Änderungen auch für unsere Shell wirksam werden, müssen wir sie neu starten. Sie beenden dazu einfach den Python-Interpreter mit der Tastenkombination Strg + D oder Strg + Z und starten ihn dann erneut. Dabei sollten Sie nicht vergessen, nach dem Neustart auch wieder die Model-Klassen und `datetime` zu importieren.

Wenn wir nun in der neuen Shell eine Meldung erzeugen, können wir sie auch in sinnvoller Weise ausgeben lassen:

```
>>> m = Meldung(titel="Nun auch mit guten Ausgaben",
                zeitstempel=datetime.now(),
                text="Jetzt sehen die Ausgaben auch gut
aus.")
>>> m
<Meldung: Jetzt sehen die Ausgaben auch gut aus.>
>>> m.save()
```

Sie sollten nach Möglichkeit alle Ihre Model-Klassen mit einer __unicode__-Methode ausstatten, da Django oft darauf zurückgreift, um Informationen zu den Objekten auszugeben.

Abfrage von Datensätzen

Mittlerweile wissen Sie, wie man neue Datensätze in die Datenbank eines Django-Projekts einfügt. Genauso wichtig wie das Anlegen neuer Datensätze ist aber auch das Abfragen von Datensätzen aus der Datenbank. Für den Zugriff auf die bereits in der Datenbank vorhandene Datensätze bietet jede Model-Klasse ein Attribut namens objects an, dessen Methoden ein komfortables Auslesen von Daten ermöglichen:

```
>>> Meldung.objects.all()
[<Meldung: Klassischerweise steht hier 'Hallo Welt'.>,
 <Meldung: Wie findet ihr das Framework?>,
 <Meldung: Jetzt sehen die Ausgaben auch gut aus.>]
```

Mit der all-Methode von Meldung.objects können wir uns eine Liste [Es handelt sich bei dem Rückgabewert von all nicht um eine Liste im Sinne einer Instanz des Datentyps list. Tatsächlich wird eine Instanz des Django-eigenen Datentyps QuerySet zurückgegeben, der sich nach außen aber fast genauso wie list-Instanzen verhält.] mit allen Meldungen in der Datenbank zurückgeben lassen. Wirklich interessant sind aber die Methoden get und filter des objects-Attributs, mit denen sich gezielt Datensätze ermitteln lassen, die bestimmte Bedingungen erfüllen. Die gewünschten Bedingungen werden bei den Abfragen als Schlüsselwort-Parameter übergeben. Wird mehr als eine Bedingung angegeben, verknüpft Django sie automatisch mit einem logischen UND.

Mit get lassen sich einzelne Datensätze abfragen. Sollten die geforderten Bedingungen auf mehr als einen Datensatz zutreffen, wirft get eine AssertException. Wird kein passender Datensatz gefunden, quittiert get dies mit einem DoesNot Exist-Fehler. Wir nutzen get, um unsere Umfrage-Meldung aus der Datenbank zu lesen:

```
>>> umfrage = Meldung.objects.get(titel="Umfrage zu
Django")
>>> umfrage.kommentar_set.all()
[<Kommentar: Super!>, <Kommentar: Klasse!>]
```

Wie Sie sehen, liest Django den entsprechenden Datensatz nicht nur aus der Datenbank, sondern erzeugt auch eine passende Instanz der dazugehörigen Model-Klasse, die sich anschließend genauso verwenden lässt, als sei sie gerade erst von uns erzeugt worden.

Field Lookups

Mit der Methode filter können wir auch mehrere Datensätze auf einmal auslesen, sofern sie den übergebenen Kriterien entsprechen:

```
>>> Kommentar.objects.filter(meldung__id=2)
[<Kommentar: Super!>, <Kommentar: Klasse!>]
```

Bei dieser Abfrage liefert Django alle Kommentar-Datensätze, die mit einer Mel dung verknüpft sind, deren id-Attribut den Wert 2 hat. Diese Art der Abfrage ist deshalb möglich, weil Django in der Lage ist, Verknüpfungen auch über mehrere Tabellen hinweg zu »folgen«. Der doppelte Unterstrich wird dabei als Trennung zwischen Objekt und Unterobjekt betrachtet, ähnlich dem Punkt in der Python-Syntax. Diese Art der Bedingungsübergabe wird auch von get unterstützt.

Der doppelte Unterstrich kann neben der Abfrage über die Verknüpfungen von verschiedenen Model-Klassen hinweg auch zur Verfeinerung normaler Bedingungen genutzt werden. Dazu wird

einem Schlüsselwortparameter der doppelte Unterstrich, gefolgt von einem speziellen Namen, nachgestellt. Mit dem folgenden `filter`-Aufruf können Sie beispielsweise alle Umfragen ermitteln, deren `text`-Attribut mit der Zeichenfolge `"Jetzt"` beginnt:

```
>>> Meldung.objects.filter(text__startswith="Jetzt")
[<Meldung: Jetzt sehen die Ausgaben auch gut aus.>]
```

Diese Art der verfeinerten Abfrage wird in Django *Field Lookup* (dt. *Feldnachschlagen*) genannt. Alle Field Lookups werden in der Form `attribut__lookup typ=wert` an die Methode `filter` übergeben. Django definiert sehr viele und teilweise spezielle Field-Lookup-Typen, weshalb die folgende Tabelle nur als Einblick zu verstehen ist:

Field-Lookup-Typ	Erklärung
`exact`	Prüft, ob das `attribut` genau gleich `wert` ist. Dies ist das Standardverhalten, wenn kein Field Lookup angegeben wird.
`iexact`	Wie `exact`, aber es wird nicht zwischen Groß- und Kleinschreibung unterschieden.
`contains`	Prüft, ob `attribute` den Wert von `wert` enthält.
`icontains`	Wie `contains`, aber es wird nicht zwischen Groß- und Kleinschreibung unterschieden.
`gt`	Prüft, ob `attribut` größer als `wert` ist. (`gt`: engl. *Greater Than*)
`gte`	Prüft, ob `attribut` größer als oder gleich `wert` ist. (`gte`: engl. *Greater Than or Equal*)
`lt`	Prüft, ob `attribut` kleiner als `wert` ist. (`lt`: engl. *Lower Than*)
`lte`	Prüft, ob `attribut` kleiner als oder gleich `wert` ist. (`lte`: engl. *Lower Than or Equal*)
`in`	Prüft, ob `attribut` in der für `wert` übergebenen Liste ist: `Meldung.objects.filter(id_in=[1, 2])`
`startswith`	Prüft, ob der Wert von `attribut` mit `wert` beginnt.
`istartswith`	Entspricht `startswith`, aber es wird nicht zwischen Groß- und Kleinschreibung unterschieden.
`endswith` und `iendswith`	Wie `startswith` und `istartswith`, aber auf das Ende von `attribut` bezogen.
`range`	Prüft, ob `attribut` in dem Bereich ist, der von dem zweielementigen Tupel `wert` definiert wird: `Meldung.objects.filter(id_range=(1, 3))`

Tabelle 25.2 Eine Übersicht über die wichtigsten Field-Lookup-Typen

Abfragen auf Ergebnismengen

Die `filter`-Methode gibt eine Instanz des Datentyps `QuerySet` zurück. Das Besondere an dem Datentyp `QuerySet` ist, dass man auf seinen Instanzen wiederum die Methoden für den Datenbankzugriff ausführen kann. Auf diese Weise lassen sich sehr komfortabel Teilmengen von Abfrageergebnissen erzeugen.

Beispielhaft ermitteln wir zuerst die Menge aller Kommentare, deren zugehörige Meldung den Titel `"Umfrage zu Django"` hat. Anschließend extrahieren wir aus der Menge die Meldungen, deren Text `"Super!"` lautet:

```
>>> k = Kommentar.objects.filter(meldung__titel="Umfrage
zu Django")
```

```
>>> k
[<Kommentar: Super!>, <Kommentar: Klasse!>]
>>> k.filter(text="Super!")
[<Kommentar: Super!>]
```

Sie sollten sich am besten vor dem Weiterlesen durch »Herumspielen« mit dem Model-API vertraut machen, weil es eine Schlüsselkomponente für den Umgang mit Django darstellt.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

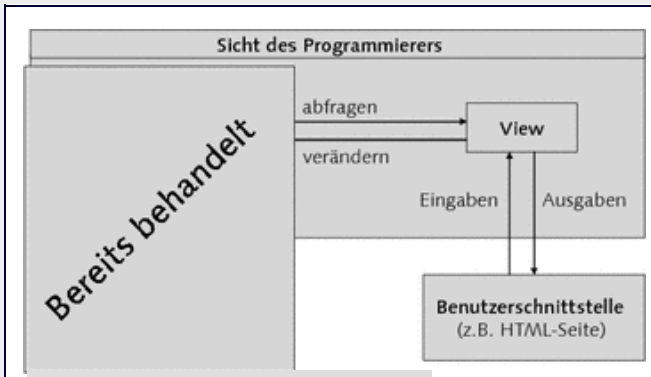
25.6 Unser Projekt wird öffentlich

Bis hierher ist unser Projekt so weit gediehen, dass eine Datenbank mit News-Meldungen und Kommentaren erstellt worden ist. Außerdem können wir die Daten über ein bequemes Webinterface verwalten.

In diesem Kapitel werden wir uns mit dem »Gesicht« unserer Seite beschäftigen, also dem, was der Besucher der Website zu sehen bekommen soll.

Sie erinnern sich sicherlich noch an das Schema zum Aufbau der Model-View-Architektur im Kapitel zur Einführung von Django.

Bildlich gesprochen haben wir uns in Abbildung 25.10 von links nach rechts genau bis zur Mitte vorangearbeitet, da wir uns bereits mit Modellen und Datenbanken, nicht aber mit den Views und der Benutzerschnittstelle beschäftigt haben. Wir werden uns nun um die rechte Seite des Schemas kümmern.



Hier klicken, um das Bild zu vergrößern

Abbildung 25.10 Den linken Teil kennen wir nun

Views

Eine *View* ist eine einfache Python-Funktion, die das zurückgibt, was im Browser des Besuchers angezeigt werden soll. Welche Art von Daten das genau ist, kann frei gewählt werden. Eine View kann beispielsweise HTML-Quelltext zurückgeben, aber auch einfacher Text oder sogar Binärdateien wie Bilder sind durchaus denkbar. In der Regel werden Sie für alles eine eigene View definieren, was Sie auf einer Website anzeigen wollen.

Unser Beispielprojekt wird zwei Views haben: eine zum Anzeigen einer Meldungsübersicht und eine für die Ansicht einer einzelnen Meldung inklusive ihrer Kommentare. Die View-Funktion für die Meldungsübersicht bekommt von uns den Namen `meldungen`, und die andere nennen wir `meldungen_detail`.

Da wir uns in diesem Abschnitt auf den prinzipiellen Umgang mit Views konzentrieren möchten, werden wir eine normale Text-Ausgabe verwenden. Wie Sie auch komfortabel HTML-Quellcode erzeugen können, zeigen wir dann im nächsten Abschnitt.

Die Views einer Applikation werden üblicherweise in der Datei *views.py* abgelegt. Eine *views.py*, die eine View-Funktion `meldungen` für die einfache Textausgabe unserer Meldungen enthält, kann folgendermaßen aussehen:

```
from unser_projekt.news.models import Meldung, Kommentar
from django.http import HttpResponse

def meldungen(request):
    zeilen = []
    for m in Meldung.objects.all():
        zeilen.append("Titel: '%s' vom %s" % (m.titel,
m.zeitstempel))
        zeilen.append("Text: %s" % m.text)
        zeilen.append("")
        zeilen.append("-" * 30)
        zeilen.append("")

    antwort = HttpResponse("\n".join(zeilen))
    antwort["Content-Type"] = "text/plain"
    return antwort
```

Am Anfang binden wir per `import` unsere beiden Model-Klassen aus der `news`-Applikation des Projekts ein. Anschließend importieren wir eine Klasse namens `HttpResponse` aus dem Modul `django.http`, die wir in unseren Views benutzen, um das Ergebnis zurückzugeben.

Die View-Funktion `meldungen` bekommt von Django einen Parameter namens `request` übergeben, mit dem wir auf bestimmte Informationen der Abfrage zugreifen können. Wir werden `request` erst im nächsten Kapitel brauchen.

Innerhalb von `meldungen` verwalten wir eine Liste namens `zeilen`, die alle Textzeilen des Ergebnisses speichert. In einer `for`-Schleife iterieren wir über alle Meldungen in der Datenbank, die wir mit der Model-API auslesen, und fügen jeweils fünf Zeilen für jede Meldung in die Liste `zeilen` ein.

Am Ende erstellen wir eine Instanz des Datentyps `HttpResponse`, dessen Konstruktor wir die Verkettung der Zeilen als Parameter übergeben. Wichtig ist, dass wir über den `[]`-Operator den `"Content-Type"` (dt. *Inhaltstyp*) der Ausgabe mit `"text/plain"` auf einfachen Text setzen, weil es sonst im Browser zu Darstellungsproblemen kommt. [Mit dem `[]`-Operator von `HttpResponse`-Instanzen können beliebige HTTP-Kopfdaten gesetzt werden.]

Bevor wir uns das Ergebnis unserer Bemühungen im Browser ansehen können, müssen wir Django mitteilen, unter welcher Adresse die View zu erreichen sein soll.

Adressen definieren

Wenn Sie schon selbst Webseiten erstellt haben, sind Sie es wahrscheinlich gewohnt, dass Sie Ihre Scripts (beispielsweise *index.php*) und andere Dateien (wie beispielsweise Bilddateien) direkt über deren Adresse auf dem Server ansprechen können. Liegt beispielsweise eine Datei *index.php* im Verzeichnis *meine_seite/scripts/* relativ zum Wurzelverzeichnis des Webservers unter der Adresse http://www.server.de, können Sie es über die Adresse http://www.server.de/meine_seite/scripts/index.php ansprechen.

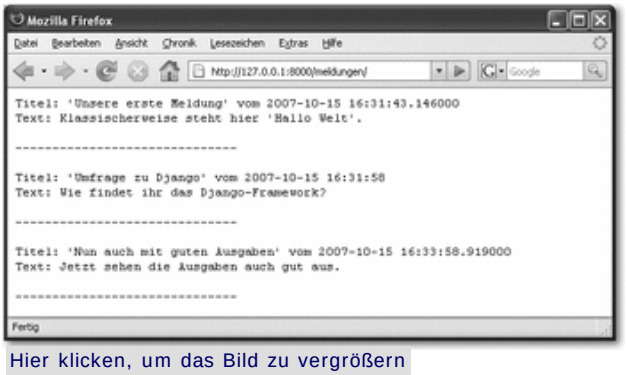
Django geht einen anderen Weg, indem es vollständig von der Ordnerstruktur des Servers abstrahiert: Anstatt die Adressen der realen Dateien auf dem Server für den öffentlichen Zugang zu übernehmen, können Sie selbst angeben, über welche Adresse ein bestimmter Teil der Seite erreichbar sein soll. Dabei ist jede View einzeln ansprechbar und kann mit einer beliebigen Adresse verknüpft werden.

Die Konfiguration der Adressen erfolgt über Regular Expressions [Details zu Regular Expressions (dt. *regulären Ausdrücken*) können Sie in Abschnitt 15.2 nachlesen.] in der Datei *urls.py* des Projekts. In der Datei *urls.py* wird eine Variable namens `urlpatterns` definiert, die in einer `patterns`-Instanz eine Liste mit allen Adressen des Projekts enthält. Die Adressangaben selbst sind Tupel mit zwei Elementen, wobei das erste Element den regulären Ausdruck für die Adresse und das zweite Element den Importnamen der verknüpften View enthält. Eine typische *urls.py*, die auch schon eine Adressangabe für unsere `meldungen`-View enthält, sieht wie folgt aus:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^meldungen/$',
'unser_projekt.news.views.meldungen'),

    # Uncomment this for admin:
    (r'^admin/', include('django.contrib.admin.urls')),
)
```

Der reguläre Ausdruck `r'^meldungen/$'` passt genau auf den String "meldungen". Wird nun eine Seite unseres Projekts aufgerufen, prüft Django der Reihe nach für alle Einträge der `urlpatterns`-Liste, ob der reguläre Ausdruck auf den relativen Pfad der geforderten Adresse passt. Ist dies der Fall, wird die Funktion aufgerufen, die durch den String im zweiten Element des Tupels beschrieben wird. Läge unsere Anwendung auf einem Server mit der Adresse http://www.meinserver.de, würde mit http://www.meinserver.de/meldungen/ die View `meldungen` aufgerufen.

Ergänzen Sie die *urls.py* um die Zeile `(r'^meldungen/$', 'unser_projekt. news.views.meldungen')`, und rufen Sie dann die entsprechende Adresse http://127.0.0.1:8000/meldungen/ in Ihrem Browser auf. Das Ergebnis sollte so aussehen wie in Abbildung 25.11.



Hier klicken, um das Bild zu vergrößern

Abbildung 25.11 Unsere erste eigene Django-Seite im Browser

Nun wollen wir auch eine Adresse und eine View für die

Detailseite der Meldungen definieren.

Parametrisierte Adressen

Wir wollen nun auch die Detailseite jeder Meldung für den Benutzer zugänglich machen. Dabei wäre es äußerst unschön, für jede Meldung eine eigene View-Funktion zu definieren, da wir einerseits den Programmcode mit redundantem Code aufblähen und andererseits die Anzahl möglicher Meldungen dadurch begrenzen würden, wie viele View-Funktionen wir implementieren.

Wesentlich eleganter ist es, wenn wir eine View-Funktion definieren, die jede beliebige Meldung darstellen kann. Welche Meldung konkret angefordert wird, soll dabei über einen Parameter festgelegt werden, der die `id` der gewünschten Meldung enthält.

Django unterstützt die Parameterübergabe für Views über sogenannte *benannte Gruppen*, die wir im Abschnitt über reguläre Ausdrücke (Abschnitt 15.2.1) behandeln. Mit benannten Gruppen können wir Teile aus einem String extrahieren und ihnen Namen geben, wenn ein bestimmter regulärer Ausdruck auf den String passt.

Um die einzelnen Meldungen über Adressen wie http://www.server.de/meldungen/1/ und http://www.server.de/meldungen/2/ erreichbar zu machen, ergänzen wir in der *urls.py* folgende Zeile:

```
(r'^meldungen/(?P<meldungs_id>\d+)/$',
'unser_projekt.news.views.meldung_detail'),
```

Wenn nun ein Besucher der Seite auf die Adresse http://www.server.de/meldungen/1/ zugreift, versucht Django, die Funktion `unser_projekt.news. views.meldung_detail` aufzurufen. Dabei wird zusätzlich zu dem `request`-Parameter ein Schlüsselwortparameter `meldungs_id` mit dem Wert `1` übergeben.

Damit können wir einen ersten View-Prototyp `meldung_detail` in die Datei *views.py* schreiben:

```
from django.http import HttpResponse, Http404
from unser_projekt.news.models import Meldung

def meldungen(request):
    ...

def meldung_detail(request, meldungs_id):
    try:
        m = Meldung.objects.get(id=meldungs_id)
    except Meldung.DoesNotExist:
        raise Http404

    zeilen = [
        "Titel: '%s' vom %s" % (m.titel, m.zeitstempel),
        "Text: %s\n" % m.text,
        "-" * 30 + "",
        "Kommentare:",
        ""]

    zeilen += ["%s: %s" % (k.autor, k.text)
               for k in m.kommentar_set.all()]

    antwort = HttpResponse("\n".join(zeilen))
    antwort["Content-Type"] = "text/plain"
    return antwort
```

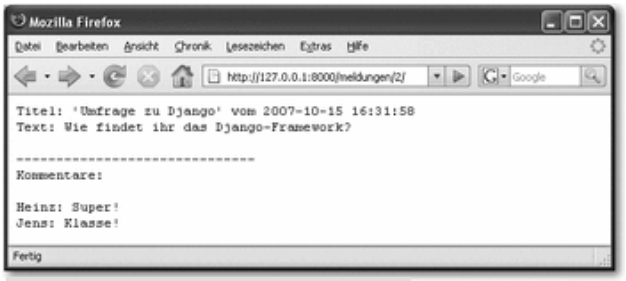
Wir importieren zusätzlich die Exception `Http404`, um einen Fehler an den Browser des Besuchers zu senden, falls er eine nicht vorhandene Meldung aufruft. Als Wert für den Parameter `meldungs_id` bekommen wir bei jedem Aufruf der View den Wert übergeben, der in der Adresse angegeben wurde. In einer `try/except`-Anweisung versuchen wir, die passende Meldung auszugeben, und erzeugen bei Misserfolg den oben genannten `Http404`-Fehler.

Konnte die Meldung erfolgreich aus der Datenbank gelesen werden, speichern wir die Text-Zeilen für die Benutzerseite in der

Liste zeilen und erzeugen außerdem mittels einer List Comprehension Ausgaben für alle Kommentare der Meldung.

Schlussendlich packen wir das Ganze auf gewohnte Weise in eine HttpResponse-Instanz, die wir per return zurückgeben.

Die von meldung_detail erzeugte Ausgabe sieht nun beispielsweise so aus wie in Abbildung 25.12.



Hier klicken, um das Bild zu vergrößern

Abbildung 25.12 Beispiel einer Meldungsdetailseite

Shortcut-Funktionen

Wenn Sie Webanwendungen entwickeln, werden sich in Ihrem Code sehr oft ähnliche Strukturen wiederfinden. Beispielsweise ist es sehr gängig, einen Datensatz aus der Datenbank abzurufen und, wenn dieser nicht existiert, einen Http404-Fehler zu erzeugen.

Damit Sie nicht jedes Mal den kompletten Code eintippen müssen und dadurch Ihren Programmtext künstlich aufblasen, bietet Django sogenannte *Shortcut-Funktionen* an, die häufig benötigte Aufgaben für Sie übernehmen. Alle Shortcut-Funktionen befinden sich im Modul django.shortcuts und können bequem per import eingebunden werden.

Um beispielsweise einen Datensatz aus der Datenbank abzufragen und bei Misserfolg eine Http404-Exception zu werfen, dient die Shortcut-Funktion get_object_or_404. Die Funktion get_object_or_404 hat fast die gleiche Schnittstelle wie die get-Methode der Model-API, mit der einzigen Ausnahme, dass als erster Parameter die Model-Klasse des gesuchten Datensatzes übergeben werden muss.

Damit wird aus der umständlichen try/except-Anweisung eine schlanke Zuweisung (vergessen Sie nicht, get_object_or_404 aus django.shortcuts zu importieren):

```
m = get_object_or_404(Meldung, id=meldungs_id)
```

Django definiert eine Reihe weiterer Shortcut-Funktionen, die wir hier nicht thematisieren werden. In der Dokumentation zu Django findet sich eine ausführliche Beschreibung aller verfügbaren Shortcuts.

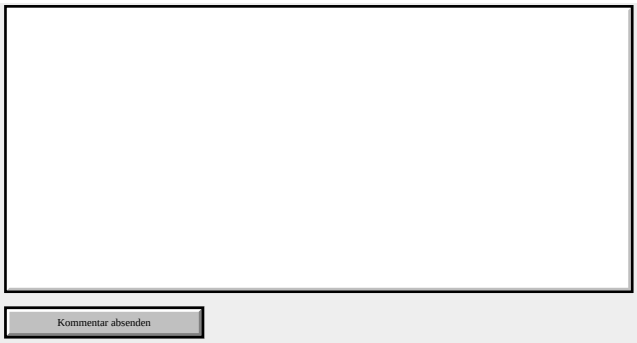
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

25.7 Djangos Template-System

Unsere bisher implementierten Views sind noch alles andere als optimal: Erstens sind sie optisch wenig ansprechend, da nur einfacher Text ausgegeben wird, und außerdem werden sie direkt aus String-Konstanten in der View-Funktion erzeugt. Besonders im zweiten Punkt muss noch nachgebessert werden, da es eines der Hauptziele von Django ist, die Komponenten eines Projekts möglichst unabhängig voneinander zu gestalten. Im Optimalfall kümmert sich die View-Funktion nur um die Verarbeitung der Parameter und die Abfrage und Aufbereitung der Daten. Die Erzeugung der Ausgabe für den Browser sollte einem anderen System übertragen werden, das sich wirklich nur um die Ausgabe kümmert.

Hier kommen sogenannte *Templates* (dt. *Schablonen*) ins Spiel, die darauf spezialisiert sind, aus übergebenen Daten ansprechende Ausgaben zu generieren. Im Prinzip handelt es sich bei Templates um Dateien, die Platzhalter enthalten. Wird ein Template mit bestimmten Werten für die Platzhalter aufgerufen, werden die Platzhalter durch eben diese Werte ersetzt, und als Ergebnis enthält man die gewünschte Ausgabe. Neben einfachen Ersetzungen von Platzhaltern unterstützt das Template-System von Django auch Kontrollstrukturen wie Fallunterscheidungen und Schleifen.

Bevor wir uns mit der Definition von Templates selbst beschäftigen, werden wir das Einbinden von Templates in View-Funktionen besprechen.

Django kapselt das gesamte Template-System in seinem Untermodul `django.template`. Mit der Klasse `loader` dieses Moduls können wir eine Template-Datei laden und daraus ein neues `Template`-Objekt erzeugen. Der Werte für die Platzhalter in dem Template werden über einen sogenannten *Kontext* übergeben, der über die Klasse `Context` erzeugt werden kann.

Einbinden von Templates in View-Funktionen

Angenommen, wir hätten bereits ein Template mit dem Dateinamen *meldungen.html* definiert [Wie Sie angeben können, wo Django nach den Templates suchen soll, wird gleich besprochen.] , dann könnten wir unsere `meldungen`-View folgendermaßen anpassen:

```
from unser_projekt.news.models import Meldung, Kommentar
from django.http import HttpResponse
from django.template import Context, loader
def meldungen(request):
    template = loader.get_template("news/meldungen.html")
    context = Context({"meldungen" :
Meldung.objects.all()})

    return HttpResponse(template.render(context))
```

Durch diese Anpassung ist die View-Funktion `meldungen` auf winzige drei Zeilen geschrumpft. [Man kann sich auf den Standpunkt stellen, dass so einfach gestrickte View-Funktionen ebenfalls überflüssig sind. Django bietet dafür sogenannte ***Generic Views*** (dt. *allgemeine Ansichten*) an. Näheres dazu finden Sie in der Django-Dokumentation.]

Mit der `get_template`-Methode der `loader`-Klasse laden wir das gewünschte Template. Dann erzeugen wir einen Kontext, der die Liste aller Meldungen mit dem Platzhalter `"meldungen"` verknüpft. Die endgültige Ausgabe des Templates für den erzeugten Kontext generieren wir mit der `render`-Methode und übergeben das Ganze als Parameter an `HttpResponse`. Die Änderung des Kontext-Typs nach `"text/plain"` entfällt, da unsere Templates im Folgenden HTML-Code erzeugen werden.

Nun können wir uns mit dem Template *meldungen.html* selbst befassen.

Die Template-Sprache von Django

Django implementiert für die Definition von Templates eine eigene Sprache. Die ist so ausgelegt, dass damit jeder beliebige Ausgabedatentyp erzeugt werden kann, solange er sich als Text ausdrücken lässt. Es bleibt also Ihnen überlassen, ob Sie einfachen Text, HTML-Quelltext, XML-Dokumente oder andere textbasierte Dateitypen generieren.

Bevor Sie allerdings mit Templates arbeiten können, müssen Sie Django mitteilen, wo es nach den Template-Dateien suchen soll. Dazu müssen Sie in der *settings.py* Ihres Projekts die Pfade zu allen Ordnern angeben, unter denen Sie Ihre Templates ablegen wollen. Wir erstellen der Einfachheit halber einen Ordner `templates` in unserem Projektverzeichnis und tragen den Pfad in die *settings.py* ein:

```
TEMPLATE_DIRS = (
    "C:/unser_projekt/templates"
)
```

Beachten Sie dabei, dass Sie für die Trennung der Pfadkomponenten immer den Forward-Slash `/` und nie den Backslash `\` verwenden sollten – selbst dann, wenn Sie unter Windows arbeiten.

Sie sollten der Übersichtlichkeit halber die Templates für jede Applikation Ihres Projekts in einem eigenen Ordner ablegen. Für die Templates unserer News-Applikation legen wir deshalb einen Unterordner namens `news` im Verzeichnis `templates` an.

Nun können wir eine Datei *meldungen.html* anlegen, die folgenden Template-Code enthält: [Bitte beachten Sie, dass hier bewusst wegen der Übersichtlichkeit auf wichtige HTML-Elemente verzichtet wurde, wodurch der HTML-Code nicht mehr den

Standards entspricht. Sie sollten natürlich in Ihren eigenen Programmen nur gültige HTML-Dateien erzeugen.]

```
<h1>News-&Uuml;bersicht</h1>

{% for n in meldungen.objects.all %}
  <div class="kontainer">
    <div class="titelzeile">
      <div class="titel">{{ n.titel|escape }}</div>
      <div class="zeitstempel">{{ n.zeitstempel }}</div>
      <div style="clear: both"></div>
    </div>
    <div class="text">
      {{ n.meldung|escape|linebreaksbr }}
      <div class="link_unten">
        <a href="{{ n.id }}/">Details</a>
      </div>
    </div>
  </div>
{% endfor %}
```

Im Prinzip ist das Template eine einfache HTML-Datei, die durch spezielle Anweisungen der Template-Sprache ergänzt wird. In dem Beispiel wurden alle Stellen, an denen Djangos Template-Sprache zum Einsatz kommt, fett markiert.

Im Folgenden werden wir die markierten Stellen unter die Lupe nehmen.

Variablen ausgeben

Der Zugriff auf Elemente des verwendeten Kontextes geschieht über doppelte geschweifte Klammern. Mit `{{ meldung.titel }}` wird dabei beispielsweise das Attribut `titel` der Kontext-Variable `meldungen` ausgegeben. Wie Sie sehen, kann bei der Ausgabe mit `{{ ... }}` auch der Punkt verwendet werden, um auf die Attribute von Kontext-Variablen zuzugreifen.

Wenn Django bei der Verarbeitung eines Templates auf eine Angabe wie `{{ variable.member }}` stößt, versucht es, die Daten in folgender Weise zu ermitteln:

1. Zuerst wird versucht, wie bei einem Dictionary mit variable["member"] einen Wert zu finden.
2. Schlägt dies fehl, wird versucht, den Wert mit variable.member auszulesen.
3. Wenn auch das nicht funktioniert, versucht Django, die Methode vari able.member() aufzurufen, und schreibt deren Rückgabewert an die Stelle.
4. Bei einem Fehlschlag wird probiert, member als einen Listenindex zu interpretieren, indem mit variable[member] ein Wert gelesen wird. (Dies geht natürlich nur, wenn member eine Ganzzahl ist.)
5. Wenn alle diese Versuche scheitern, nimmt Django den Wert, der in der settings.py für TEMPLATE_STRING_IF_INVALID gesetzt wurde. Standardmäßig ist dies ein leerer String.

Filter für Variablen

Sie können das Ersetzen von Kontextvariablen durch sogenannte *Filter* anpassen. Ein Filter ist eine Funktion, die einen String verarbeiten kann, und wird so verwendet, dass man der Variable bei ihrer Ausgabe einen senkrechten Strich, gefolgt vom Filternamen, nachstellt:

```
{{ variable|filter }}
```

Es ist auch möglich, mehrere Filter hintereinander zu übergeben. Sie werden dabei einfach, durch einen senkrechten Strich getrennt, hintereinander geschrieben:

```
{{ variable|filter1|filter2|filter3 }}
```

In dem Beispiel würde zuerst `filter1` auf den Wert von `variable` angewandt, das Ergebnis an `filter2` übergeben und dessen Rückgabewert schließlich mit `filter3` verarbeitet.

Django implementiert eine ganze Reihe solcher Filter. In der folgenden Tabelle sind die beiden Filter erklärt, die in unserem Beispiel Verwendung finden:

Filter	Bedeutung
`escape`	Ersetzt die Zeichen <, >, &, " und ' durch entsprehende HTML-Kodierungen.
`linebreaksbr`	Ersetzt alle Zeilenvorschübe durch das HTML-Tag , das eine neue Zeile erzeugt.

Tabelle 25.3 Zwei der Filter von Django

Im Übrigen ist es auch möglich, eigene Filter zu definieren. Informationen dazu und eine ausführliche Übersicht mit allen Django-Filtern finden Sie in der Dokumentation.

Tags

Djangos Template-Sprache arbeitet mit sogenannten *Tags* (dt. *Kennzeichnungen*), mit denen Sie den Kontrollfluss innerhalb eines Templates anpassen können. Jedes Tag hat die Form `{% tag_bezeichnung %}`, wobei `tag_bezeichnung` von dem jeweiligen Tag abhängt. Es gibt auch Tags, die einen Block umschließen. Solche Tags haben die folgende Struktur:

```
{% block test_block %}
    Inhalt des Tags
{% endblock }
```

Es existieren Tags, mit denen sich Kontrollstrukturen (wie die bedingte Ausgabe oder die wiederholte Ausgabe eines Blocks) abbilden lassen.

Der `if`-Block dient dazu, einen bestimmten Teil des Templates nur dann auszugeben, wenn eine Bedingung erfüllt ist:

```
{% if besucher.hat_geburtstag % }
    Herzlichen Glückwunsch zum Geburtstag!
{% else %}
    Willkommen auf unserer Seite!
{% endif %}
```

Wenn `besucher.hat_geburtstag` den Wahrheitswert `True` ergibt, wird dem Besucher der Seite zum Geburtstag gratuliert. Ansonsten wird er normal begrüßt, was über den `else`-Zweig festgelegt wird. Natürlich kann der `else`-Zweig auch entfallen.

Als Bedingung können auch komplexe logische Ausdrücke gebildet werden:

```
{% if bedingung1 and bedingung2 and bedingung3%}
    Es gelten bedingung1 und bedingung2 und/oder es gilt
    bedingung3
{% endif %}
```

Neben den Fallunterscheidungen gibt es auch ein Äquivalent zu Python-Schleifen: das `for`-Tag. Das `for`-Tag ist dabei eng an die Syntax von Python angelehnt und kann beispielsweise so aussehen:

```
{% for zahl in ganzzahlen %}
    {{ zahl }} ist eine Ganzzahl.
```

```
{% endfor %}
```

Diese Schleife funktioniert natürlich nur dann, wenn die Kontextvariable ganzzahlen auf ein iterierbares Objekt verweist.

Hätte ganzzahlen den Wert `[1, 2, 3]`, würde das obige Template folgende Ausgabe produzieren:

```
1 ist eine Ganzzahl
2 ist eine Ganzzahl
3 ist eine Ganzzahl
```

Als Letztes werden wir die sogenannte *Vererbung bei Templates* besprechen.

Es kommt häufig vor, dass viele Seiten einer Webanwendung das gleiche Grundgerüst aus beispielsweise Kopfzeile und Navigation besitzen. Wenn aber jede Seite ein eigenes Template hat, müsste dieses Grundgerüst redundant in allen Templates enthalten sein. Dies geht zu Lasten der Wartbarkeit.

Um dieses Problem zu lösen, kann man das Grundgerüst der Seite in einem zentralen Template definieren und von diesem zentralen Template die konkret benötigten Templates ableiten.

Angenommen, das Template in der Datei *basis.html* enthält das Grundgerüst der Webseite, kann ich ein anderes Template mithilfe des `extends`-Tags davon ableiten:

```
{% extends "basis.html" %}
```

Dies hat zur Folge, dass der komplette Inhalt von *basis.html* in das erbende Template eingefügt wird. Damit ein erbendes Template auch den Inhalt der entstehenden Seite selbst bestimmen kann, kann ein Template sogenannte *Blöcke* mit dem `block`-Tag definieren.

Ein Block ist dabei eine Stelle innerhalb eines Templates, die mit einem Namen versehen wird und durch erbende Templates mit konkretem Inhalt versehen werden kann.

Betrachten wir zwei Beispieldateien:

basis.html

```
---------- Kopfzeile ------------
{% block inhalt %}Standardinhalt{% endblock %}
---------- Fußzeile -------------
```

Wenn Sie dieses Template mit Django ausgeben lassen, wird das `block`-Tag einfach ignoriert und durch seinen Inhalt ersetzt:

```
---------- Kopfzeile ------------
Standardinhalt
---------- Fußzeile -------------
```

Interessant wird es dann, wenn wir ein anderes Template von *basis.html* erben lassen:

erbendes_template.html

```
{%extends "basis.html" %}
{% block inhalt %}Hallo, ich habe geerbt!{% endblock}
```

Die Ausgabe von *erbendes_template.html* sieht dann so aus:

```
---------- Kopfzeile ------------
Hallo, ich habe geerbt!
---------- Fußzeile -------------
```

Natürlich ist ein Grundgerüst einer Seite nicht die einzige Anwendung für die Template-Vererbung. Sie können Vererbung immer dann einsetzen, wenn mehrere Seiten auf einer gemeinsamen abstrakten Struktur basieren sollen.

Mit diesem Wissen können wir nun ein ansprechendes HTML-Template-Gefüge für unser Webprojekt erstellen. In einer Datei *basis.html* werden wir das Grundgerüst der Seite mitsamt den CSS-Stylesheets [***Cascading Style Sheets*** (***CSS***) ist eine Formatierungssprache, um beispielsweise HTML-Seiten optisch aufzuwerten.] ablegen. Die Datei *basis.html* hat den folgenden Inhalt, wobei aus Gründen der Übersichtlichkeit auf die Angabe von einigen Teilen verzichtet wurde:

```
<?xml version="1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Unsere Django-Seite</title>
  <link rel="stylesheet" type="text/css" href="styles.css"
/>
  <style type="text/css">
    /* Hier kommen die CSS-Styles hin */
  </style>
</head>
<body>
  <div id="inhalt">
    <h2>{% block titel %}Django Beispielseite{% endblock
%}</h2>

    {% block inhalt %}
    {% endblock %}
  </div>
</body>
</html>
```

Das Template definiert eine einfache HTML-Seite, in der es zwei Template-Blöcke gibt: `titel` und `inhalt`. Diese Blöcke sollen nun von den Templates für die Meldungsübersicht und die Meldungsdetails mit Inhalt gefüllt werden.

Die Datei *meldungen.html*, in der das Template für die Meldungsübersicht definiert werden soll, sieht dann folgendermaßen aus:

```
{% extends "news/basis.html" %}

{% block titel %}News-&Uuml;bersicht{% endblock %}

{% block inhalt %}
    {% for n in object_list %}
        <div class="kontainer">
          <div class="titelzeile">
            <div class="titel">{{ n.titel|escape }}</div>
            <div class="zeitstempel">{{ n.zeitstempel
}}</div>
            <div style="clear: both"></div>
          </div>
          <div class="text">
            {{ n.meldung|escape|linebreaksbr }}
            <div class="link_unten">
              <a href="{{ n.id }}/">Details</a>
            </div>
          </div>
        </div>
    {% endfor %}
{% endblock %}
```

Wenn Sie die Dateien in dem *template/news/*-Verzeichnis im Projektordner gespeichert haben, können Sie das Ergebnis in Ihrem Browser betrachten [Die vollständige *basis.html*-Datei inklusive aller CSS-Styles finden Sie auf der Buch-CD.] (siehe Abbildung 25.13).

Wenn Sie auf dieser Seite den **Details**-Link anklicken, gelangen Sie natürlich weiterhin zu der tristen Textansicht der jeweiligen Meldung. Um dies zu ändern, werden wir auch die View-Funktion `meldung_detail` umstricken:

```
def meldung_detail(request, meldungs_id):
    template =
loader.get_template("news/meldung_detail.html")
```

```
    meldung = get_object_or_404(Meldung, id=meldungs_id)
    kontext = Context({"meldung" : meldung})

    return HttpResponse(template.render(kontext))
```

Hier klicken, um das Bild zu vergrößern

Abbildung 25.13 Schicke HTML-Ausgabe unseres ersten Templates

Ihnen wird sicherlich aufgefallen sein, dass sich die beiden Views `meldungen` und `meldung_detail` strukturell sehr stark ähneln: Zuerst wird ein Template geladen, dann der Kontext über ein Dictionary erzeugt und schließlich ein `HTTPResponse`-Objekt zurückgegeben, das den Rückgabewert von `template.render` enthält.

Um Schreibarbeit zu sparen, bietet Django für solche Fälle eine Shortcut-Funktion `render_to_response` an. Mit `render_to_response` können wir die beiden View-Funktionen noch einmal verkürzen:

```
def meldungen(request):
    return render_to_response("news/meldungen.html",
        {"meldungen" : Meldung.objects.all()})

def meldungen_detail(request, meldungs_id):
    return render_to_response("news/meldung_detail.html",
        {"meldung" : get_object_or_404(Meldung,
id=meldungs_id)})
```

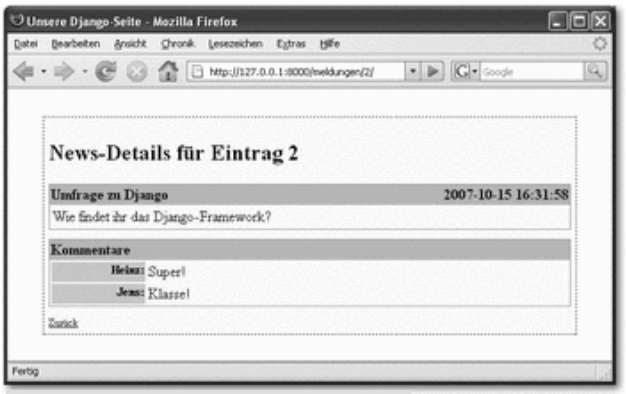
Der Shortcut-Funktion `render_to_response` wird der Pfad zu dem gewünschten Template als erster und das Dictionary mit dem Kontext als zweiter Parameter übergeben.

Um unsere HTML-Ausgabe zu komplettieren, fehlt nur noch das Template für die Detailseite unserer Meldungen:

```
{% extends "news/basis.html" %}

{% block titel %}
  News-Details f&uuml;r Eintrag {{ meldung.id }}
{% endblock %}

{% block inhalt %}
  <div class="kontainer">
    <div class="titelzeile">
      <div class="titel">{{ meldung.titel|escape }}</div>
      <div class="zeitstempel">{{ meldung.zeitstempel
}}</div>
      <div style="clear: both"></div>
    </div>
    <div class="text">
      {{ meldung.text|escape|linebreaksbr }}
    </div>
  </div>

  <div class="kontainer">
    <div class="titelzeile">Kommentare</div>
    {% if meldung.kommentar_set.count %}
      <table>
      {% for k in meldung.kommentar_set.all %}
        <tr class="kommentarzeile">
          <td class="spaltenbezeichner">{{ k.autor }}:</td>
          <td>{{ k.text|escape|linebreaksbr }}</td>
```

```
        </tr>
      {% endfor %}
      </table>
    {% else %}
      Keine Kommentare
    {% endif %}
  </div>

  <div class="link_unten"><a
href="../">Zur&uuml;ck</a></div>
{% endblock %}
```

Im Browser stellt sich das Ganze dann so dar wie in Abbildung 25.14.



Hier klicken, um das Bild zu vergrößern

Abbildung 25.14 Detailseite einer Meldung mit zwei Kommentaren

Wir sind nun so weit, dass wir ansprechende Ausgaben mit wenig Aufwand erzeugen können. Unser Projekt ist damit fast fertiggestellt. Es fehlt nur noch die Möglichkeit für die Besucher der Seite, Kommentare zu den Meldungen abgeben zu können.

Damit werden wir uns im letzten Abschnitt über Django befassen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

25.8 Verarbeitung von Formulardaten

Um eine dynamische Webanwendung wirklich interaktiv werden zu lassen, müssen die Benutzer neben dem einfachen Navigieren über die Seite auch zum Inhalt der Seite beitragen können. Dies geschieht oft über Gästebücher, Foren oder Kommentarfunktionen wie in unserem Beispiel.

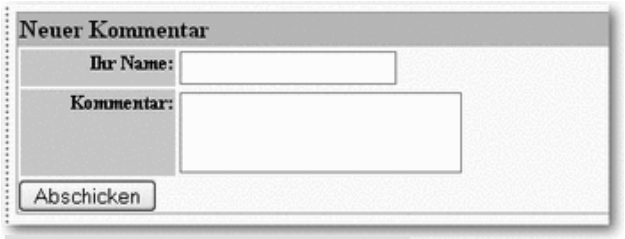
Auf der technischen Seite muss für diese Funktionalitäten eine Schnittstelle existieren, mit der Daten vom Browser des Benutzers an die Serveranwendung übertragen werden können.

Das HTTP-Protokoll bietet für diesen Zweck zwei Methoden für die sogenannte *Argumentübertragung* an: *GET* und *POST*. Der prinzipielle Unterschied zwischen den beiden Übertragungsarten ist, dass mit GET übertragene Daten direkt an die Adresse der jeweiligen Seite angehängt werden und so für den Benutzer unmittelbar sichtbar sind, während die POST-Methode für den Benutzer unsichtbar im Hintergrund Daten übertragen kann.

Beide Methoden arbeiten mit benannten Platzhaltern, die üblicherweise durch HTML-Formulare mit Daten verknüpft werden.

Unser Formular für die Kommentarabgabe

Wir werden auf der Detailseite jeder Meldung ein Formular anbieten, in dem der Besucher neue Kommentare zu der angezeigten Meldung eingeben kann. Das Formular wird zwei Textfelder besitzen: eines für den Namen des Besuchers und eines für den Kommentar selbst. Im Browser soll es später so aussehen wie in Abbildung 25.15.



Hier klicken, um das Bild zu vergrößern

Abbildung 25.15 Unser Kommentarformular

Als Übertragungsart für die eingegebenen Daten wählen wir die POST-Methode, um die Adressleiste nicht zu überfüllen. Das Speichern der neuen Kommentare wird unsere View-Funktion `meldung_detail` übernehmen, der beim Abschicken des Formulars ein GET-Parameter namens `speichere_kommentar` übergeben wird. Außerdem soll das Formular ein Feld für Fehlerausgaben besitzen, falls der Benutzer zu wenige Angaben gemacht hat.

Das Template für die Formulardefinition sieht dann folgendermaßen aus: [Dieser Teil muss unten in der *meldung_detail.html* eingefügt werden. Die vollständige Template-Datei finden Sie auf der Buch-CD.]

```
<div class="kontainer">
  <div class="titelzeile">Neuer Kommentar</div>
  <span class="fehler">{{ fehler }}</span>
  <form method="post" action="?kommentar_speichern">
    <table>
      <tr class="kommentarzeile">
        <td class="spaltenbezeichner">Ihr Name:</td>
        <td><input type="text" name="besuchername"
                   value="{{ bentzername }}"/></td>
      </tr>
      <tr class="kommentarzeile">
        <td class="spaltenbezeichner">Kommentar:</td>
        <td>
          <textarea name="kommentartext">
            {{ kommentar }}
          </textarea>
        </td>
      </tr>
    </table>
    <input type="submit" value="Abschicken" />
  </form>
</div>
```

Wie Sie der Zeile `<form method="post" action="?kommentar_speichern">` entnehmen können, wird beim Versenden des Formulars die Detailseite der Meldung mit dem GET-Parameter `kommentar_speichern` aufgerufen. Außerdem werden die Werte der Textfelder `besuchername` und `kommentartext` als POST-Daten übergeben.

Sollten beim Speichern eines Kommentars Fehler auftreten, kann eine Meldung über die Kontextvariable `fehler` von der View-Funktion gesetzt werden. Damit die Eingaben des Benutzers in einem solchen Fall nicht verloren gehen, können die Textfelder über die Kontextvariablen `benutzername` und `kommentar` von der View mit Initialwerten versehen werden.

Zugriff auf POST- und GET-Variablen in View-Funktionen

Wie Sie bereits wissen, bekommt jede View-Funktion von Django einen Parameter namens `request` übergeben. Dieser Parameter enthält Informationen über den Seitenaufruf und damit insbesondere die GET- und POST-Parameter. Dazu hat `request` zwei Attribute namens `GET` und `POST`, die den Zugriff auf die Parameter über ihre Namen wie in einem Dictionary ermöglichen.

Mit diesem Wissen können wir unsere View-Funktion `model_detail` folgendermaßen erweitern (die noch nicht bekannten Elemente im Listing werden anschließend erläutert):

```
def meldung_detail(request, meldungs_id):
    meldung = get_object_or_404(Meldung, id=meldungs_id)

    if "kommentar_speichern" in request.GET:
```

```
        name = request.POST.get("besuchername", "")
        text = request.POST.get("kommentartext", "")

        if name and text:
            kommentar = meldung.kommentar_set.create(
                autor=name, text=text)
            kommentar.save()
            return HttpResponseRedirect(".")

        else:
            return
render_to_response("news/meldung_detail.html",
                {"meldung" : meldung,
                 "fehler": "Sie m&uuml;ssen Ihren Namen und
\
                           einen Kommentar angeben.",
                 "besuchername" : name, "kommentartext" :
text})

    return render_to_response("news/meldung_detail.html",
        {"meldung" : meldung})
```

Am Anfang der Funktion lesen wir wie gehabt die betreffende Meldung aus der Datenbank oder geben einen `Http404`-Fehler zurück. Anschließend prüfen wir mit dem `in`-Operator, ob `"kommentar_speichern"` per GET übergeben worden ist, um gegebenenfalls einen neuen Kommentar zu speichern. Wurde `"kommentar_speichern"` nicht übergeben, wird der `if`-Block ausgelassen und die Detailseite angezeigt.

Wenn ein neuer Kommentar gespeichert werden soll, lesen wir den eingegebenen Namen und den Kommentartext aus `request.POST`. Anschließend prüfen wir, ob in beide Textfelder etwas eingegeben wurde. Fehlt mindestens eine Angabe, wird im `else`-Zweig die Detailseite erneut angezeigt, wobei ein entsprechender Fehlertext ausgegeben wird. Dadurch, dass wir die Kontext-Variablen `besuchername` und `kommentartext` auf die zuvor übergebenen Werte setzen, gehen eventuell vom Benutzer gemachte Eingaben nicht verloren, sondern erscheinen wieder in den Textfeldern.

Haben die beiden Variablen `name` und `text` korrekte Werte, erzeugen wir ein neues Kommenter-Objekt in der Datenbank. Allerdings benutzen wir in diesem Fall die Klasse `HttpResponseRedirect`, um den Besucher zu der Detailseite weiterzuleiten, anstatt ein Template auszugeben. Der Grund dafür ist einfach: Wenn ein Besucher einen neuen Kommentar verfasst hat und nun wieder auf der Detailseite gelandet ist, könnte er die Aktualisieren-Funktion seines Browsers benutzen, um die Seite neu zu laden. Beim Aktualisieren einer Seite werden aber sowohl GET- als auch POST-Daten erneut übertragen. Deshalb würde bei jeder Aktualisierung derselbe Kommentar noch einmal gespeichert werden. Durch die indirekte Weiterleitung mittels `HttpResponseRedirect` lösen wir dieses Problem, da nun die POST- und GET-Variablen verworfen werden.

Sie können `HttpResponseRedirect` per `import` aus dem Modul `django.http` einbinden.

Nun ist unsere Beispielanwendung vollständig funktionsfähig. Wenn Sie Ihre Django-Kenntnisse vertiefen möchten, um auch die weiterführenden Techniken kennenzulernen, empfehlen wir Ihnen die Lektüre der ausgezeichneten Online-Dokumentation auf der Django-Homepage unter http://www.djangoproject.com.

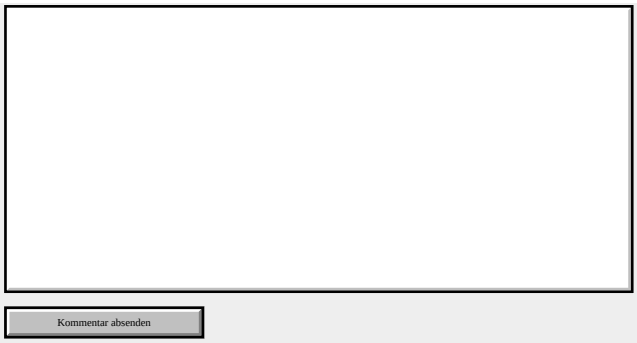
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

26.2 Schreiben von Extensions

Im letzten Kapitel haben wir uns mit einem Ansatz beschäftigt, bestimmte Teile eines Programms in C zu implementieren, in eine dynamische Bibliothek auszulagern und dann aus dem Python-Programm heraus anzusprechen. Das ist, wie Sie sich sicherlich vorstellen können, für eine Anwendung eine tolle Sache, doch gerade beim Schreiben eines Moduls wäre der ctypes-Ansatz aufgrund seiner Einschränkungen an der Schnittstelle nicht besonders praktikabel.

Wir sind im Laufe des Buchs bereits öfters auf Bibliotheken wie `cStringIO` oder `cProfile` gestoßen, bei denen gesagt wurde, dass sie aus Effizienzgründen in C geschrieben wurden. Offensichtlich ist es also möglich und tatsächlich auch durchaus üblich, ein Python-Modul in reinem C zu schreiben und dann ohne ctypes zu verwenden. Solche Module werden *Extentions* (dt. *Erweiterungen*) genannt.

Solche Extensions werden mithilfe der sogenannten *Python API* geschrieben. Mit dieser API ist es möglich, in C beispielsweise Instanzen eines Python-Datentyps zu erzeugen und zu verarbeiten. Dies ist unerlässlich, wenn man bedenkt, dass die in C geschriebene Extension später als vollwertiges Modul auftreten soll und dass auf keinen Fall Probleme an der Schnittstelle, wie

sie beispielsweise mit ctypes auftreten, vorkommen dürfen. Auch das Werfen von Python-Exceptions aus einer C-Extension heraus wird mithilfe der Python API möglich.

Sie werden in diesem Kapitel nur eine knappe Referenz zur Python API finden, da wir hier primär einen praxisorientierten Einstieg in das Schreiben von Extensions bieten möchten. Eine umfassende Referenz finden Sie in der Python-Dokumentation unter dem Stichwort »Python/C API«. Die Dokumentation enthält außerdem ausführliche Informationen zum Schreiben von Erweiterungen bzw. zum Einbetten des Python-Interpreters in ein C-Programm unter dem Stichwort *Extending and Embedding Python*.

top

26.2.1 Ein einfaches Beispiel

In diesem Abschnitt soll eine einfache Beispiel-Extension in C geschrieben werden. Wir nehmen uns vor, eine Extension namens chiffre zu schreiben, die verschiedene Funktionen zur Verschlüsselung eines Strings bereitstellt. Da es sich um ein Beispiel handelt, werden wir uns auf eine einzelne Funktion namens caesar beschränken, die eine Verschiebechiffre, auch »Cäsar-Verschlüsselung« genannt, durchführen soll. Die Funktion soll später folgendermaßen verwendet werden können:

```
>>> chiffre.caesar("HALLOWELT", 13)
'UNYYBJRYG'
```

Dabei entspricht der zweite Parameter der Anzahl Stellen, um die jeder Buchstabe des ersten Parameters verschoben wird.

Im Folgenden werden wir eine Extension schreiben, die das Modul chiffre inklusive der Funktion caesar für ein Python-Programm bereitstellt. Die Quelldatei der Erweiterung lautet *chiffre.c*. Der Quelltext der Erweiterung soll nun Schritt für Schritt besprochen werden.

```
#include <Python.h>

static PyObject *chiffre_caesar(PyObject *self, PyObject
*args);
```

Zunächst wird der Header der Python API eingebunden. Sie finden die Header-Datei *Python.h* im Unterordner *include* Ihrer Python-Installation. Außerdem schreiben wir zu Beginn der Quelldatei den Prototyp der Funktion chiffre_caesar, die später der Funktion caesar des Moduls chiffre entsprechen soll.

Beachten Sie, dass in der Header-Datei *Python.h* einige Präprozessor-Anweisungen enthalten sind, die sich auf andere Header-Dateien auswirken. Aus diesem Grund sollte *Python.h* immer vor den Standard-Headern eingebunden werden.

Dann wird die sogenannte *Method Table* erstellt, die alle Funktionen der Extension auflistet:

```
static PyMethodDef ChiffreMethods[] =
    {
    {"caesar", chiffre_caesar, METH_VARARGS,
     "Perform Caesar cipher encryption."},
    {NULL, NULL, 0, NULL}
    };
```

Jeder Eintrag der Method Table enthält zunächst den Namen, den die Funktion oder Methode in Python tragen soll, dann den Namen der assoziierten C-Funktion, dann die Kennzeichnung der Art der Parameterübergabe und schlussendlich eine Beschreibung der Funktion als String. Das Makro METH_VARARGS besagt, dass alle Parameter, die der Funktion caesar in Python übergeben werden, in der C–Funktion chiffre_caesar in Form eines Tupels ankommen. Dies ist die bevorzugte Art der Parameterübergabe.

Ein mit Nullen gefüllter Eintrag beendet die Method Table.

Es folgt die Initialisierungsfunktion der Erweiterung namens `initchiffre`:

```
PyMODINIT_FUNC initchiffre(void)
    {
    Py_InitModule("chiffre", ChiffreMethods);
    }
```

Sie wird vom Interpreter aufgerufen, wenn das Modul `chiffre` zum ersten Mal eingebunden wird, und hat die Aufgabe, das Modul einzurichten und dem Interpreter die Method Table (`ChiffreMethods`) zu übergeben. Dazu wird die Funktion `Py_InitModule` aufgerufen.

Beachten Sie, dass die Funktion `initchiffre` genannt werden muss, wobei `chiffre` natürlich bei einem anderen Modulnamen entsprechend angepasst werden muss.

Jetzt folgt die Funktion `chiffre_caesar`. Das ist die C-Funktion, die bei einem Aufruf der Python-Funktion `chiffre.caesar` aufgerufen wird.

```
static PyObject *chiffre_caesar(PyObject *self, PyObject
*args)
    {
    char *text, *c, *e;
    PyObject *ergebnis;
    int cipher;

    if(!PyArg_ParseTuple(args, "si", &text, &cipher))
        return NULL;

    ergebnis = PyString_FromStringAndSize(NULL,
strlen(text));
    e = PyString_AsString(ergebnis);

    for(c = text; *c; c++, e++)
        *e = ((*c - 'A' + cipher) % 26) + 'A';

    return ergebnis;
    }
```

Dabei werden als Parameter immer zwei Pointer auf eine `PyObject`-Struktur übergeben. Eine solche `PyObject`-Struktur entspricht ganz allgemein einer Referenz auf ein Python-Objekt. In C werden also alle Instanzen aller Datentypen auf `PyObject`-Strukturen abgebildet. Durch Funktionen der Python-API lassen sich dann datentyp-spezifische Eigenschaften der Instanzen auslesen. Doch kommen wir nun zur Bedeutung der übergebenen Parameter im Einzelnen.

Der erste Parameter, `self`, würde nur dann benötigt, wenn die Funktion `chiffre_caesar` eine Python-Methode implementieren würde, und ist in diesem Beispiel immer `NULL`. Der zweite Parameter, `args`, ist ein Tupel und enthält alle der Python-Funktion übergebenen Parameter. Auf die Parameter kann über die API-Funktion `PyArg_ParseTuple` zugegriffen werden.

Die Funktion `PyArg_ParseTuple` bekommt zunächst den Parameter `args` selbst übergeben und danach einen String, der die Datentypen der enthaltenen Parameter kennzeichnet. `s` steht dabei für einen String und `i` für eine ganze Zahl. Im Folgenden zeigt `text` auf den übergebenen String, während `cipher` den zweiten übergebenen Parameter, die ganze Zahl, enthält. Beachten Sie, dass `text` auf den Inhalt des übergebenen Python-Strings zeigt und aus diesem Grund nicht verändert werden darf.

Nachdem der zu verschlüsselnde Text in `text` und die zu verwendende Anzahl Stellen in `cipher` stehen, kann die tatsächliche Verschlüsselung durchgeführt werden. Dazu wird zunächst ein neuer String mit der Länge des Strings `text` erzeugt. Da dieser String gerade erst erzeugt worden ist und somit keine anderen Referenzen auf ihn verweisen können, dürfen wir im Folgenden in den internen Buffer des Strings schreiben. Zum Verschlüsseln des Strings wird in einer Schleife über alle

Buchstaben des übergebenen Strings iteriert und jeder Buchstabe um die angegebene Anzahl Stellen verschoben. Der auf diese Weise veränderte Buchstabe wird dann in den neu erstellten String geschrieben.

Beachten Sie, dass dieser Algorithmus weder für Kleinbuchstaben noch für Sonderzeichen, sondern ausschließlich für Großbuchstaben funktioniert. Diese Einschränkung erhöht nicht nur die Übersicht, sondern erlaubt es uns später, einen – zugegebenermaßen künstlichen – Fehlerfall zu erzeugen.

26.2.2 Exceptions ▼▲

Wir haben bereits gesagt, dass der in der Funktion `chiffre_caesar` verwendete Algorithmus nur mit Strings arbeiten kann, die allein aus ASCII-Großbuchstaben bestehen. Es wäre natürlich ein Leichtes, die Funktion `chiffre_caesar` dahingehend anzupassen, dass auch Kleinbuchstaben verschlüsselt und Sonderzeichen übersprungen werden. Doch zu Demonstrationszwecken soll in diesem Beispiel stattdessen eine `ValueError`-Exception geworfen werden, wenn der übergebene String nicht ausschließlich aus ASCII-Großbuchstaben besteht.

Eine eingebaute Exception kann mithilfe der Funktion `PyErr_SetString` geworfen werden, wobei der Funktion der Exceptiontyp, in diesem Fall `PyExc_ValueError`, und die Fehlerbeschreibung übergeben wird. Die Funktion `chiffre_caesar` sieht inklusive Fehlerbehandlung so aus:

```
static PyObject *chiffre_caesar(PyObject *self, PyObject
*args)
    {
    char *text, *c, *e;
    PyObject *ergebnis;
    int cipher;

    if(!PyArg_ParseTuple(args, "si", &text, &cipher))
        return NULL;

    ergebnis = PyString_FromStringAndSize(NULL,
strlen(text));
    e = PyString_AsString(ergebnis);

    for(c = text; *c; c++, e++)
        {
        if((*c < 'A') || (*c > 'Z'))
            {
            PyErr_SetString(PyExc_ValueError,
                            "Character out of range");
            Py_DECREF(ergebnis);
            return NULL;
            }

        *e = ((*c - 'A' + cipher) % 26) + 'A';
        }
    return ergebnis;
    }
```

Direkt nach dem Setzen der Exception wird das Makro `Py_DECREF` verwendet. Dieses Makro ist dazu da, den Reference Count der Instanz `ergebnis` zu verringern. Näheres dazu erfahren Sie in Abschnitt 26.2.4.

Nachdem die Extension kompiliert, gelinkt und installiert wurde, (Näheres zu diesen Vorgängen erfahren Sie im nächsten Abschnitt), können wir sie tatsächlich im interaktiven Modus ausprobieren:

```
>>> import chiffre
>>> chiffre.caesar("HALLOWELT", 13)
'UNYYBJRYG'
>>> chiffre.caesar("UNYYBJRYG", 13)
'HALLOWELT'
>>> chiffre.caesar("Hallo Welt", 13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Character out of range
```

Beachten Sie, dass wir die Cäsar-Verschlüsselung bislang immer mit dem Verschiebungsparameter 13 durchgeführt haben. Das

entspricht dem *ROT13*-Algorithmus, der eine relativ große Bekanntheit genießt (was aber nicht heißt, dass er besonders sicher wäre). Der Vorteil von ROT13 ist, dass der verschlüsselte String durch nochmaliges Verschlüsseln mit ROT13 entschlüsselt wird, wie auch im obigen Beispiel zu sehen ist. Selbstverständlich sind aber auch andere Verschiebungsparameter möglich.

26.2.3 Erzeugen der Extension

Da die soeben geschriebene Quelldatei *chiffre.c* in einer ganz bestimmten Art kompiliert und gelinkt werden muss, kommt es uns nur zupass, dass das Paket `distutils` auch das Kompilieren und Linken von Erweiterungen automatisiert.

Beachten Sie, dass zum Kompilieren der Extension ein C-Compiler auf Ihrem System installiert sein muss. Unter Linux wird vom Paket `distutils` der GCC- und unter Windows der MSVC-Compiler von Visual Studio 2003 (sofern es installiert ist) verwendet. Visual Studio kann inzwischen in einer eingeschränkten Version kostenlos von Microsoft bezogen werden. Sollten Sie ein Windows-System einsetzen und Visual Studio 2003 nicht installiert haben, bietet Ihnen das Paket `distutils` an, stattdessen eine MinGW-Installation zu verwenden.

Das Installationsscript *setup.py* sieht in Bezug auf unsere einfache Beispielextension folgendermaßen aus:

```
from distutils.core import setup, Extension

modul = Extension("chiffre", sources=["chiffre.c"])
setup(
     name = "PyChiffre",
     version = "1.0",
     description = "Module for encryption techniques.",
     ext_modules = [modul]
     )
```

Zunächst wird eine Instanz der Klasse `Extension` erzeugt und ihrem Konstruktor der Name der Extension und eine Liste der zugrunde liegenden Quelldateien übergeben. Beim Aufruf der Funktion `setup` wird, abgesehen von den üblichen Parametern, der Schlüsselwortparameter `ext_modules` übergeben. Dort muss eine Liste von `Extension`-Instanzen übergeben werden, die mit dem Installationsscript kompiliert, gelinkt und in die Distribution aufgenommen werden sollen.

Jetzt kann das Installationsscript wie gewohnt ausgeführt werden und kompiliert bzw. installiert die Erweiterung automatisch.

Neben dem Schlüsselwortparameter `sources` können bei der Instanziierung der Klasse `Extension` noch weitere Parameter übergeben werden, die in der folgenden Tabelle kurz erläutert werden.

Parameter	Bedeutung
`include_dirs`	Eine Liste von Verzeichnissen, die für das Kompilieren der Erweiterung benötigte Header-Dateien enthalten
`define_macros`	Eine Liste von Tupeln, über die beim Kompilieren der Erweiterung bestimmte Makros mit bestimmten Werten definiert werden können. Das Tupel muss folgende Struktur haben: `("MAKRONAME", "Wert")`.
`undef_macros`	Eine Liste von Makronamen, die beim Kompilieren nicht definiert sein sollen
`libraries`	Eine Liste von Bibliotheksnamen, gegen die die Erweiterung gelinkt werden soll
`librariy_dirs`	Eine Liste von Verzeichnissen, in denen nach den bei `libraries` angegebenen Bibliotheken gesucht wird

Tabelle 26.2 Schlüsselwortparameter des Extension-Konstruktors

Nachdem die Extension mittels distutils kompiliert und installiert wurde, kann sie in einer Python Shell verwendet werden:

```
>>> import chiffre
>>> chiffre.caesar("HALLOWELT", 13)
'UNYYBJRYG'
>>> chiffre.caesar("UNYYBJRYG", 13)
'HALLOWELT'
```

top

26.2.4 Reference Counting

Wie Sie wissen, basiert die Speicherverwaltung Pythons auf einem sogenannten *Reference-Counting*-Algorithmus. Das bedeutet, dass Instanzen zur Entsorgung freigegeben werden, sobald keine Referenzen mehr auf sie bestehen. Das hat den Vorteil, dass sich der Programmierer nicht um das Freigeben von allokiertem Speicher zu kümmern braucht.

Vermutlich wissen Sie ebenfalls, dass es so etwas wie Reference Counting in C nicht gibt, sondern dass dort der Programmierer für die Speicherverwaltung selbst verantwortlich ist. Wie verträgt es sich damit also, wenn man Python-Instanzen in einem C-Programm verwendet?

Grundsätzlich sollten Sie sich von dem C-Idiom verabschieden, im Besitz einer bestimmten Instanz bzw. in diesem Fall einer PyObject-Struktur zu sein. Vielmehr können Sie allenfalls im Besitz einer Referenz auf eine Instanz bzw. eines Pointers auf eine PyObject-Struktur sein. Damit implementiert die Python API im Grunde das Speichermodell von Python in C. Im Gegensatz zum Speichermodell von Python erhöht bzw. verringert sich der Referenzzähler einer Instanz jedoch nicht automatisch, sondern muss in einer C-Extension explizit mitgeführt werden. Dazu können die Makros Py_INCREF und Py_DECREF der Python API folgendermaßen verwendet werden:

```
PyObject *string = PyString_FromString("Hallo Welt");
Py_INCREF(string);
Py_DECREF(string);
Py_DECREF(string);
```

Zunächst wird mithilfe der Funktion PyString_FromString eine Instanz des Python-Datentyps str erzeugt. In diesem Moment besitzen Sie eine Referenz auf diese Instanz. Der Reference Count ist damit gleich 1. Im Folgenden wird der Reference Count durch die Makros Py_INCREF und Py_DECREF einmal erhöht und zweimal verringert. Am Ende des Beispiels erreicht der Reference Count 0, und der erzeugte String wird der Garbage Collection zum Fraß vorgeworfen.

Das hier besprochene Beispiel ist zugegebenermaßen nicht gerade sinnvoll. Im nächsten Abschnitt wird jedoch ein Beispielprogramm erarbeitet, in dem die Anwendung des Reference Countings in der Praxis zu sehen ist.

Jetzt bliebe nur noch die Frage zu klären, wann Sie den Referenzzähler erhöhen bzw. verringern müssen. Immer dann, wenn Sie in Ihrem Programm eine Instanz eines Python-Datentyps erzeugen und eine oder mehrere Referenzen auf diese Instanz halten, müssen Sie diese Referenzen freigeben, wenn sie nicht mehr benötigt werden. Sollten Sie die Referenzen nicht freigeben, verweilt die Instanz im Speicher, obwohl sie eigentlich nicht mehr benötigt wird. Es handelt sich dann um ein sogenanntes *Memory Leak*, und Memory Leaks sind kein besonders erstrebenswerter Umstand in einem Programm.

Die zweite Möglichkeit sind sogenannte *geliehene Referenzen* (engl. *borrowed references*). Solche Referenzen sind

beispielsweise über Funktionsparameter an Ihr Programm weitergereicht worden, gehören Ihnen aber nicht. Das bedeutet auch, dass Sie solche Referenzen nicht freizugeben brauchen. Ein Beispiel für geliehene Referenzen sind Funktionsparameter, die Sie grundsätzlich nicht freigeben brauchen. Wenn Sie eine geliehene Referenz zu einer eigenen Referenz aufwerten möchten, müssen Sie den Referenzzähler der dahinter liegenden Instanz mittels `Py_INCREF` erhöhen. Das Freigeben von geliehenen Referenzen führt dazu, dass danach möglicherweise auf Speicherbereiche zugegriffen wird, die bereits freigegeben worden sind. Das kann in einigen Fällen gut gehen, führt aber häufig zu einem *Speicherzugriffsfehler*. Ähnlich wie Memory Leaks sollten Sie Speicherzugriffsfehler nach Möglichkeit vermeiden.

Als letzte Möglichkeit können Sie eine vollwertige Referenz in Form eines Rückgabewertes an eine andere Funktion abgeben. Sie brauchen sich also nicht um die Freigabe Ihrer zurückgegebenen Instanzen kümmern.

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

26.3 Python als eingebettete Skriptsprache

In den vorangegangenen Abschnitten haben Sie Möglichkeiten kennengelernt, in C geschriebene Programmteile von einem Python-Programm aus aufzurufen und somit beispielsweise laufzeitkritische Teile in ein C-Programm auszulagern.

In diesem Kapitel soll der entgegengesetzte Weg beschritten werden: Wir möchten Python-Programme aus einem C-Programm heraus ausführen können, Python also als eine Art eingebettete Scriptsprache (engl. *embedded script language*) verwenden. [Ein Beispiel für eine solche eingebettete Skriptsprache ist ***Lua***, die besonders häufig als Skriptsprache für Computerspiele eingesetzt wird.] Auf diese Weise können wir bestimmte Teile des Programms in Python schreiben, für die Python aufgrund seiner Flexibilität einfach besser geeignet ist.

top

26.3.1 Ein einfaches Beispiel

Zum Einstieg soll ein C-Programm erstellt werden, das ein möglichst simples Python-Programm ausführt. Dieses Python-Programm soll lediglich ein bisschen Text und eine Zufallszahl auf dem Bildschirm ausgeben.

Das folgende C-Programm führt ein solches Python-Script aus:

```
#include <Python.h>

const char *programm =
"import random\n"
"print 'Guten Tag, die Zahl ist:', random.randint(0,
100)\n"
"print 'Das war ... Python'\n";

int main(int argc, char *argv[])
    {
    Py_Initialize();
    PyRun_SimpleString(programm);
    Py_Finalize();
    }
```

Zunächst wird die Header-Datei der Python API eingebunden. Sie sehen, dass sowohl zum Erweitern als auch zum Einbetten von Python dieselbe API verwendet wird. Danach wird der String `programm` angelegt, der den Python-Code enthält, der später ausgeführt werden soll.

In der Hauptfunktion `main` wird der Python-Interpreter zuerst durch Aufruf von `Py_Initialize` initialisiert. Danach wird das zuvor im String `programm` abgelegte Python-Script durch Aufruf der Funktion `PyRun_SimpleString` ausgeführt und der Interpreter schlussendlich durch die Funktion `Py_Finalize` wieder beendet.

Statt der Funktion `PyRun_SimpleString` hätte auch die Funktion `PyRun_Simple File` aufgerufen werden können, um den Python-Code aus einer Datei zu lesen.

Wichtig ist, dass dem Compiler das Verzeichnis bekannt ist, in dem die Header-Datei *Python.h* liegt. Außerdem muss das Programm gegen die Python API gelinkt werden. Diese ist als dynamische Bibliothek *python25.lib* im Unterordner *lib* Ihrer Python-Installation zu finden. [Diese Angabe bezieht sich auf Windows. Unter Linux wird die Bibliothek in den meisten Fällen in das Verzeichnis für systemweite Bibliotheken */usr/lib* installiert. Beim Linken Ihres Programms mit dem GCC können Sie die Python 2.5-Bibliothek mit der Option `-lpython2.5` einbinden.] Wenn sowohl das Kompilieren als auch das Linken ohne Probleme vonstatten gegangen ist, werden Sie feststellen, dass das Programm tatsächlich funktioniert:

```
Guten Tag, die Zufallszahl ist: 64
Das war ... Python
```

Sie sehen, dass wir uns für dieses Beispielprogramm einer sehr abstrakten Schnittstelle bedient haben, denn es ist beispielsweise nicht möglich, mit dem Python-Script zu interagieren. Das Python-Script läuft bislang völlig autonom, und es können keine Werte zwischen ihm und dem C-Programm ausgetauscht werden. Aber gerade die Interaktion mit dem Hauptprogramm macht die Qualität einer eingebetteten Scriptsprache aus.

top

26.3.2 Ein komplexeres Beispiel ▼▲

Sie haben sicherlich erkannt, dass das vorangegangene Beispielprogramm noch nicht der Stein der Weisen war. In diesem Abschnitt soll ein besseres, komplexeres Programm entwickelt werden, das dazu in der Lage ist, Funktionen eines Python-Scripts direkt aufzurufen und Werte über die Funktionsschnittstelle zu schicken bzw. entgegenzunehmen. Außerdem soll das C-Programm dazu in der Lage sein, eigene Funktionen zu definieren, die aus dem Python-Script heraus aufgerufen werden können. Sie werden feststellen, dass auch dies dem Schreiben von Erweiterungen stark ähnelt.

Das folgende C-Programm soll ein Python-Script laden, das eine Funktion `entscheide` implementiert. Diese Funktion soll sich für einen von zwei übergebenen Strings entscheiden. Die Funktion

könnte beispielsweise deshalb in ein Python-Script ausgelagert worden sein, weil man es dem Anwender ermöglichen will, die Funktion selbst zu implementieren und das Programm somit an die eigenen Bedürfnisse anzupassen.

Der Quellcode des Beispielprogramms sieht folgendermaßen aus:

```
#include <Python.h>

int main(int argc, char *argv[])
    {
    char *ergebnis;
    PyObject *modul, *funk, *prm, *ret;

    Py_Initialize();
    PySys_SetPath(".");
    modul = PyImport_ImportModule("script");

    if(modul)
        {
        funk = PyObject_GetAttrString(modul,
"entscheide");
        prm = Py_BuildValue("(ss)", "Hallo", "Welt");
        ret = PyObject_CallObject(funk, prm);

        ergebnis = PyString_AsString(ret);
        printf("Das Script hat sich fuer '%s'
entschieden\n",
ergebnis);

        Py_DECREF(prm);
        Py_DECREF(ret);
        Py_DECREF(funk);
        Py_DECREF(modul);
        }
    else
        printf("Fehler: Modul nicht gefunden\n");
    Py_Finalize();
    }
```

In der Hauptfunktion `main` des C-Programms wird zunächst der Python-Interpreter durch Aufruf von `Py_Initialize` initialisiert. Danach wird durch die Funktion `PySys_SetPath` das lokale Programmverzeichnis als einziger Ordner festgelegt, aus dem Module importiert werden können. Beachten Sie, dass dieser Funktionsaufruf sowohl dem C- als auch dem Python-Programm verbietet, globale Module wie beispielsweise `math` einzubinden. Wenn Sie solche Module benötigen, dürfen Sie die `Import`-Pfade nicht, wie es in diesem Beispiel geschehen ist, überschreiben, sondern Sie sollten sich den Pfad mit `Py_GetPath` holen, ihn um das Verzeichnis `.` erweitern und mit `PySys_SetPath` setzen. Beachten Sie, dass das lokale Programmverzeichnis standardmäßig nicht als `import`-Pfad eingetragen ist.

Durch Aufruf der Funktion `PyImport_ImportModule` wird ein Modul eingebunden und als `PyObject`-Pointer zurückgegeben. Beachten Sie, dass, wenn die entsprechenden Pfade festgelegt wurden, sowohl lokale als auch globale Module mit dieser Funktion eingebunden werden können. Nachfolgend prüfen wir, ob das Modul erfolgreich geladen wurde. Bei einem Misserfolg gibt die Funktion `PyImport_ImportModule` wie die meisten anderen Funktionen, die einen `PyObject`-Pointer zurückgeben, `NULL` zurück. Beachten Sie, dass es immer ratsam ist, die zurückgegebenen `PyObject`-Pointer auf `NULL` zu testen. Im Beispielprogramm wurde dies nur exemplarisch bei `modul` gemacht.

Nachfolgend wird durch Aufruf der Funktion `PyObject_GetAttrString` ein Pointer auf die Funktion `entscheide` des Moduls `script` erstellt. Um die Funktion aufrufen zu können, müssen die Funktionsparameter in Form eines Tupels übergeben werden. Dazu erzeugen wir mittels `Py_BuildValue` ein neues Tupel, das die beiden Strings `"Hallo"` und `"Welt"` enthält, von denen die Funktion `entscheide` einen auswählen soll.

Durch Aufruf der Funktion `PyObject_CallObject` wird die Funktion `funk` schlussendlich aufgerufen und ihr Rückgabewert ebenfalls in Form eines Pointers auf `PyObject` zurückgegeben. Da es sich bei dem Rückgabewert um einen String handelt, können wir diesen mittels `PyString_AsString` zu einem C-String konvertieren und dann mit `printf` ausgeben.

Die in diesem Beispiel aufgerufene Python-Funktion `entscheide`

sieht folgendermaßen aus und befindet sich in der Programmdatei *script.py*:

```
def entscheide(a, b):
    return (a if min(a) < min(b) else b)
```

Die Funktion bekommt zwei Strings a und b übergeben und gibt einen der beiden zurück. Die Entscheidung, welcher der beiden Strings zurückgegeben wird, hängt davon ab, in welchem der alphabetisch kleinste Buchstabe enthalten ist.

Im nächsten Beispielprogramm soll es dem Python-Script ermöglicht werden, bestimmte Funktionen des C-Programms aufzurufen. Es soll dem Script also gewissermaßen eine API zur Verfügung gestellt werden, die es verwenden kann. Diese Idee liegt nicht nur gedanklich sehr nah an den in Abschnitt 26.2 besprochenen Extensions, sondern wird auch ganz ähnlich umgesetzt. Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
#include <Python.h>

static PyObject *testfunktion(PyObject *self, PyObject
*args)
    {
    int a, b;
    if(!PyArg_ParseTuple(args, "ii", &a, &b))
        return NULL;
    return Py_BuildValue("i", a + b);
    }

static PyMethodDef MethodTable[] =
    {
    {"testfunktion", testfunktion, METH_VARARGS,
"Testfunktion"},
    {NULL, NULL, 0, NULL}
    };

int main(int argc, char *argv[])
    {
    FILE *f;

    Py_Initialize();
    PySys_SetPath(".");
    Py_InitModule("api", MethodTable);

    f = fopen("script.py", "r");
    PyRun_SimpleFile(f, "script.py");
    fclose(f);

    Py_Finalize();
    }
```

Zunächst wird die Funktion `testfunktion` definiert, die später dem Python-Script zur Verfügung gestellt werden soll. Im Beispiel berechnet die Funktion schlicht die Summe zweier ganzer Zahlen, die ihr als Parameter übergeben werden. Danach wird eine `MethodeTable` erstellt, ganz als würden wir eine Erweiterung schreiben. Und wie bei einer Erweiterung auch, wird die Funktion `test funktion` dem Python-Script später über ein Modul zur Verfügung stehen.

Dieses Modul, im Beispiel `api` genannt, wird durch den Aufruf der Funktion `Py_InitModule` eingerichtet. Schlussendlich brauchen wir nur noch die Funktion `PyRun_SimpleFile` aufzurufen, um das Python-Script *script.py* zu interpretieren. Der Funktion `PyRun_SimpleFile` muss dabei ein geöffnetes Dateiobjekt übergeben werden.

Das Python-Script, das von diesem C-Programm aufgerufen wird, könnte beispielsweise folgendermaßen aussehen:

```
import api
print "Zwei plus zwei ist:", api.testfunktion(2, 2)
```

top

26.3.3 Python-API-Referenz

Nachdem die Python API in den Themenbereichen »Erweitern und

Einbetten von Python« bereits verwendet wurde, soll an dieser Stelle eine kleine Referenz dieser API stehen. Beachten Sie dabei, dass die Python API sehr umfangreich ist und in diesem Abschnitt keinesfalls vollständig behandelt werden kann. Aus diesem Grund beschränken wir uns auf die Beschreibung der Funktionen der Python API, die in den Beispielprogrammen der vorherigen Abschnitte vorgekommen sind.

Die Funktionen sind in alphabetischer Reihenfolge aufgeführt.

PyObject *Py_BuildValue(const char *format, ...)

Erzeugt eine Instanz eines Python-Datentyps mit einem bestimmten Wert. Der String *format* spezifiziert dabei den Datentyp. Die folgende Tabelle listet die wichtigsten Werte für *format* mit ihrer jeweiligen Bedeutung auf.

Formatstring	Beschreibung
`"s"`	Erzeugt eine Instanz des Python-Datentyps `str` aus einem C-String (`char *`).
`"u"`	Erzeugt eine Instanz des Python-Datentyps `unicode` aus einem C-Buffer mit Unicode-Daten (`Py_UNICODE *`).
`"i"`	Erzeugt eine Instanz des Python-Datentyps `int` aus einem C-Integer (`int`).
`"c"`	Erzeugt eine Instanz des Python-Datentyps `str` aus einem C-Zeichen (`char`). Der resultierende String hat die Länge 1.
`"d"`	Erzeugt eine Instanz des Python-Datentyps `float` aus einer C-Gleitkommazahl (`double`). Analog dazu existiert `"d"` für den C-Datentyp `float`.
`"(…)"`	Erzeugt eine Instanz des Python-Datentyps `tuple`. Anstelle der Auslassungszeichen müssen die Datentypen der Elemente des Tupels angegeben werden. `"(iii)"` würde beispielsweise ein Tupel mit drei ganzzahligen Elementen erzeugen.
`"[…]"`	Erzeugt eine Instanz des Python-Datentyps `list`.
`"{…}"`	Erzeugt eine Instanz des Python-Datentyps `dict`.

Tabelle 26.3 Mögliche Angaben im Formatstring

Beachten Sie, dass ein C-String, der an die Funktion `Py_BuildValue` übergeben wird, um einen Python-String zu erzeugen, stets kopiert wird. Das bedeutet insbesondere, dass Sie jeden dynamisch allokierten String wieder freigeben müssen, selbst wenn er an die Funktion `Py_BuiltValue` übergeben wurde.

void Py_Finalize()

Die Funktion `Py_Finalize` deinitialisiert den Python-Interpreter und gibt den vom Interpreter belegten Speicher frei. Diese Funktion sollte beim Embedding des Python-Interpreters aufgerufen werden, wenn der Interpreter nicht mehr benötigt wird.

void Py_INCREF(PyObject *o), void Py_DECREF(PyObject *o)

Die Makros `Py_INCREF` und `Py_DECREF` erhöhen (inkrementieren) bzw. verringern (dekrementieren) den Reference Count der Instanz `o` um 1.

void Py_Initialize()

Die Funktion `Py_Initialize` initialisiert den Python-Interpreter und sollte aufgerufen werden, bevor der Interpreter beim Embedding eingesetzt wird.

PyObject *Py_InitModule(char *name, PyMethodDef *methods))

Erzeugt ein Python-Modul mit dem Namen *name*. Das Modul enthält den Inhalt, der von der Method Table *methods* vorgeschrieben wird. Das erstellte Modul wird in Form eines `PyObject`-Pointers zurückgegeben.

Beachten Sie, dass es sich bei dem zurückgegebenen Pointer um eine geliehene Referenz handelt, dass Sie also den Referenzzähler nicht dekrementieren müssen.

void Py_XINCREF(PyObject *o), void Py_XDECREF(PyObject *o)

Die Makros `Py_XINCREF` und `Py_XDECREF` erhöhen bzw. verringern den Reference Count der Instanz o um 1. Dabei prüfen die Makros vorher, ob für o `NULL` übergeben wurde.

int PyArg_ParseTuple(PyObject *args, const char *format, ...)

Die Funktion `PyArg_ParseTuple` liest die einer Funktion übergebenen Argumente aus und speichert sie in lokale Variablen. Als erster Parameter muss ein Tupel übergeben werden, das die Parameter enthält. Ein solches Tupel bekommt jede Python-Funktion in C übergeben. Beachten Sie, dass mit `PyArg_ParseTuple` nur Positionsargumente ausgelesen werden können.

Der zweite Parameter *format* ist ein String, ähnlich dem Formatstring von `Py_BuildValue` und legt fest, wie viele Parameter ausgelesen werden sollen und welche Datentypen diese haben. Zum Schluss akzeptiert die Funktion `PyArg_Parse Tuple` beliebig viele Pointer auf Variablen, die mit den ausgelesenen Werten gefüllt werden sollen.

Im Erfolgsfall gibt die Funktion `True` zurück. Bei einem Fehler wirft die Funktion eine entsprechende Exception und gibt `False` zurück.

void PyErr_SetString(PyObject *type, PyObject *value)

Die Funktion `PyErr_SetString` wirft eine Python-Exception. Sie bekommt den Typ der auszulösenden Exception als ersten Parameter übergeben. Das kann eine der vordefinierten Standardexceptions, beispielsweise `PyExc_NameError` oder `PyExc_ValueError`, sein. Als zweiter Parameter wird der Wert der Exception übergeben, üblicherweise ein String, der eine Fehlermeldung enthält.

Beachten Sie, dass Sie den Reference Count einer Standardexception nicht erhöhen müssen, wenn Sie sie an `PyErr_SetString` übergeben.

PyObject *PyImport_ImportModule(char *name, PyObject *globals, PyObject *locals, PyObject *fromlist)

Diese C-Funktion lädt ein Python-Modul und ist äquivalent zu einem Aufruf der Built-in Function `__import__` aus Python heraus. Im einfachsten Fall braucht nur der Parameter *name* übergeben werden, der den Namen des zu ladenden Moduls enthält.

Beachten Sie, dass Sie das lokale Programmverzeichnis zuerst mittels `PySys_Set Path` einrichten müssen, bevor lokale Module eingebunden werden können.

void PyMem_Free(void *p)

Die Funktion `PyMem_Free` gibt den allokierten Speicherblock frei, auf den p zeigt.

void *PyMem_Malloc(size_t n)

Die Funktion PyMem_Malloc allokiert einen Speicherbereich der Größe n und gibt einen Pointer auf den allokierten Speicher zurück.

PyObject *PyObject_CallObject(PyObject *callable_object, PyObject *args)

Die Funktion PyObject_CallObject ruft das aufrufbare Objekt *callable_objekt*, beispielsweise also ein Funktions- oder Methodenobjekt, auf und übergibt dabei die Parameter *args*. Wenn keine Parameter übergeben werden sollen, kann für *args* NULL übergeben werden.

Das Ergebnis des Objektaufrufs wird als Rückgabewert zurückgegeben. Bei einem Fehler wird NULL zurückgegeben.

PyObject *PyObject_GetAttrString(PyObject *o, const char *attr_name)

Gibt eine Referenz auf das Attribut mit dem Namen *attr_name* der Instanz *o* zurück. Im Fehlerfall wird NULL zurückgegeben.

int PyRun_SimpleFile(FILE *fp, const char *filename)

Die Funktion PyRun_SimpleFile führt eine Python-Programmdatei aus. Dabei wird der Inhalt der Programmdatei in Form eines geöffneten Datei-Pointers übergeben. Zusätzlich sollte der Dateiname der Programmdatei als zweiter Parameter übergeben werden.

Die Funktion gibt 0 zurück, wenn der Code erfolgreich ausgeführt wurde, und –1, wenn ein Fehler aufgetreten ist.

int PyRun_SimpleString(const char *command)

Die Funktion PyRun_SimpleString verhält sich ähnlich wie PyRun_SimpleFile mit dem Unterschied, dass der auszuführende Python-Code aus dem String *command* statt aus einer Datei gelesen wird.

char *PyString_AsString(PyObject *string)

Gibt den internen C-Buffer des Python-Strings string zurück. Beachten Sie, dass Sie nur dann in den zurückgegebenen Buffer schreiben dürfen, wenn Sie ihn zuvor beispielsweise mit der Funktion PyString_FromStringAndSize selbst erstellt haben.

PyObject *PyString_FromStringAndSize(const char *v, Py_ssize_t len)

Erzeugt eine Instanz des Python-Datentyps str mit der Länge *len* und dem Inhalt des C-Strings *v*. Für *v* kann NULL übergeben werden. Die Funktion gibt eine Referenz auf die erzeugte Instanz zurück.

void PySys_SetPath(char *path)

Über die Funktion PySys_SetPath können die Verzeichnisse festgelegt werden, in denen nach Modulen gesucht wird. Die Funktion entspricht damit dem Schreiben des Strings sys.path in Python. Umgekehrt können die gesetzten Verzeichnisse über die Funktion PySys_GetPath ausgelesen werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

27.2 URLs im Standardbrowser öffnen – webbrowser

Das Modul `webbrowser` dient dazu, eine Website im Standardbrowser des gerade verwendeten Systems zu öffnen. Im Folgenden werden wir die drei wichtigsten Funktionen des Moduls `webbrowser` erklären. Die Beispiele verstehen sich dabei in folgendem Kontext:

```
>>> import webbrowser
```

webbrowser.open(url[, new[, autoraise]])

Öffnet die URL *url* im Standardbrowser des Systems.

Für den Parameter *new* kann eine ganze Zahl zwischen `0` und `2` übergeben werden. Dabei bedeutet ein Wert von `0`, dass die URL nach Möglichkeit in einem bestehenden Browserfenster geöffnet wird, `1`, dass die URL in einem neuen Browserfenster geöffnet werden soll, und `2`, dass die URL nach Möglichkeit in einem neuen Tab eines bestehenden Browserfensters geöffnet werden soll. Der Parameter ist mit `0` vorbelegt.

Wenn für den Parameter *autoraise* `True` übergeben wird, wird versucht, das Browserfenster mit der geöffneten URL in den Vordergrund zu holen. Beachten Sie, dass dies bei vielen Systemen automatisch geschieht.

```
>>> webbrowser.open("http://www.galileo-press.de", 2)
True
```

webbrowser.open_new(url)

Öffnet die URL *url* in einem neuen Fenster des Standardbrowsers.

```
>>> webbrowser.open_new("http://www.galileo-press.de")
True
```

webbrowser.open_new_tab(url)

Öffnet die URL *url* in einem neuen Tab eines bestehenden Fensters des Standardbrowsers. Wenn noch keine Instanz des Browsers gestartet ist, wird die URL in einem neuen Fenster geöffnet.

```
>>> webbrowser.open_new_tab("http://www.galileo-press.de")
True
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

27.3 Funktionsschnittstellen vereinfachen – functools

Im Modul `functools` ist die Funktion `partial` enthalten, mit der sich Funktionsschnittstellen vereinfachen lassen. Betrachten Sie dazu die folgende Funktion:

```
def f(a, b, c, d):
    print "%s %s %s %s" % (a,b,c,d)
```

Die Funktion `f` erwartet viele Parameter, vier an der Zahl. Stellen Sie sich nun vor, wir müssten die Funktion `f` sehr häufig im Programm aufrufen und würden dabei für die Parameter `b`, `c` und `d` immer die gleichen Werte übergeben. Es drängt sich die Frage auf, ob sich solch ein Funktionsaufruf nicht so vereinfachen lässt, dass nur der eigentlich interessante Parameter `a` übergeben werden muss. Und genau hier setzt die Funktion `partial` des Modus `functools` ein.

functools.partial(func[, *args[, **kwargs]])

Die Funktion `partial` bekommt ein Funktionsobjekt übergeben, dessen Schnittstelle vereinfacht werden soll. Zusätzlich werden die feststehenden Positions- und Schlüsselwortparameter der Funktion *func* übergeben.

Die Funktion `partial` gibt jetzt ein neues Funktionsobjekt zurück, dessen Schnittstelle der von *func* entspricht, jedoch um die in *args* (*arguments*) und *kwargs* (*keyword arguments*) angegebenen Parameter erleichtert wurde. Bei einem Aufruf des zurückgegebenen Funktionsobjekts werden diese feststehenden Parameter wieder in die Parameterliste geschrieben.

Dies soll durch das folgende Beispiel anhand der oben definierten Funktion `f` demonstriert werden:

```
>>> def f(a, b, c, d):
...     print "%s %s %s %s" % (a,b,c,d)
...
>>> import functools
>>> f_neu = functools.partial(f, b="du", c="schoene",
d="Welt")
>>> f_neu("Hallo")
Hallo du schoene Welt
>>> f_neu("Tschuess")
Tschuess du schoene Welt
```

Zunächst wird die Funktion `f` definiert, die vier Parameter akzeptiert und diese hintereinander auf dem Bildschirm ausgibt. Da die letzten drei Parameter dieser Schnittstelle in unserem Programm immer gleich sind, möchten wir diese nicht immer wiederholen und vereinfachen die Schnittstelle mittels `partial`.

Dazu wird die Funktion `partial` aufgerufen und das Funktionsobjekt von `f` als erster Parameter übergeben. Danach folgen die drei feststehenden Werte für die Parameter `b`, `c` und `d` in Form von Schlüsselwortparametern. Die Funktion `partial` gibt ein Funktionsobjekt zurück, das der Funktion `f` mit vereinfachter Schnittstelle entspricht. Dieses Funktionsobjekt kann, wie im Beispiel zu sehen, mit einem einzigen Parameter aufgerufen werden, während dieser Parameter daraufhin offensichtlich zusammen mit seinen drei vorgegebenen Parametern beim Funktionsaufruf von `f` ankommt.

Abgesehen von Schlüsselwortparametern können der Funktion `partial` auch Positionsparameter übergeben werden, die dann ebenfalls als solche an die zu vereinfachende Funktion weitergegeben werden. Dabei ist zu beachten, dass die feststehenden Parameter dann am Anfang der Funktionsschnittstelle stehen müssen. Dazu folgendes Beispiel:

```
>>> def f(a, b, c, d):
...     print "%s %s %s %s" % (a,b,c,d)
...
>>> f_neu = functools.partial(f, "Hallo", "du", "schoene")
>>> f_neu("Welt")
Hallo du schoene Welt
>>> f_neu("Frau")
Hallo du schoene Frau
```

Die ersten drei Parameter der Funktion `f` sind immer gleich und sollen mithilfe der Funktion `partial` als Positionsparameter festgelegt werden. Das resultierende Funktionsobjekt `f_neu` kann mit einem Parameter aufgerufen werden, der beim daraus resultierenden Funktionsaufruf von `f` neben den drei festen Parametern als vierter übergeben wird.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

27.4 Versteckte Passworteingaben – getpass

Das Modul getpass ist sehr überschaubar und ermöglicht vor allem das komfortable Einlesen eines Passworts vom Benutzer. Im Folgenden sollen die im Modul getpass enthaltenen Funktionen erklärt werden. Um die Beispiele ausführen zu können, muss zuvor das Modul eingebunden werden:

```
>>> import getpass
```

getpass.getpass([prompt[, stream]])

Die Funktion getpass liest, ähnlich wie etwa raw_input, eine Eingabe vom Benutzer ein und gibt diese als String zurück. Der Unterschied zu raw_input besteht allerdings darin, dass getpass zur Eingabe von Passwörtern gedacht ist. Das bedeutet, dass die Eingabe des Benutzers unter Verwendung von getpass verdeckt geschieht, also in der Konsole nicht angezeigt wird.

Über den optionalen Parameter *prompt* kann der Text angegeben werden, der den Benutzer zur Eingabe des Passworts auffordert. Der Parameter ist mit "Pass word: " vorbelegt. Für den zweiten optionalen Parameter, *stream*, kann ein datei-ähnliches Objekt übergeben werden, in das die Aufforderung *prompt* geschrieben wird. Das funktioniert nur unter Unix-ähnlichen Betriebssystemen, unter Windows wird der Parameter ignoriert.

```
>>> s = getpass.getpass("Ihr Passwort bitte: ")
Ihr Passwort bitte:
>>> print s
Dies ist mein Passwort
```

getpass.getuser()

Die Funktion getuser gibt den Namen zurück, mit dem sich der aktuelle Benutzer im Betriebssystem eingeloggt hat.

```
>>> print getpass.getuser()
'Benutzername'
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

28.2 Python 2.6

Python 2.6 wird der legitime Nachfolger von Python 2.5 und zugleich das letzte abwärtskompatible Python-Release zur Serie 2.x sein. Die finale Version ist für April 2008 geplant. Die Änderungen, die mit Python 2.6 in die Sprache selbst einfließen, sind verglichen mit denen anderer Python-Versionen klein. Im Folgenden fassen wir die wichtigsten Neuerungen kurz zusammen:

- Das Werfen von String-Exceptions wird, nachdem es bereits lange als *deprecated* eingestuft ist, verboten.
- Die `with`-Anweisung wird eingeführt. Näheres zur `with`-Anweisung finden Sie in Abschnitt 13.8.
- Einige Python-3000-Features werden mithilfe des Moduls `__future__` nutzbar sein.

Die wohl größte Änderung in Python 2.6 wird der Python-3000-Kompatibilitätsmodus sein. Wenn dieser Modus aktiviert wurde, gibt Python 2.6 bei der Ausführung eines Programms Warnungen aus, an welchen Stellen der Programmcode Probleme bei der Umstellung auf Python 3000 verursachen könnte.

Es ist als Vorbereitung auf Python 3000 absolut empfehlenswert, Ihre Python-Programme mithilfe des Kompatibilitätsmodus auf Probleme zu überprüfen und diese gegebenenfalls zu beseitigen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name
E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

A.2 Reservierte Wörter

Die folgende Tabelle enthält Wörter, die nicht als Bezeichner verwendet werden dürfen, weil sie einem Schlüsselwort entsprechen.

and	elif	global	or	
assert	else	if	pass	
break	except	import	print	
class	exec	in	raise	yield
continue	finally	is	return	
def	for	lambda	try	
del	from	not	while	

Tabelle A.1 Liste reservierter Wörter

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

A.3 Operatorrangfolge

Die Operatorrangfolge legt fest, wie stark ein Operator im Vergleich zu den anderen bindet. Der zuerst aufgeführte Operator (**) bindet am stärksten, der letzte am schwächsten. Der zweiten Spalte der Tabelle können Sie die übliche Bedeutung des Operators entnehmen, das ist zumeist die Bedeutung des Operators bezogen auf ganze Zahlen.

Operator	Übliche Bedeutung
x ** y	y-te Potenz von x
~x	Bitweises Komplement von x
+x, -x	Positives oder negatives Vorzeichen
x * y	Produkt von x und y
x / y	Quotient von x und y
x % y	Rest bei ganzzahliger Division von x durch y
x // y	Ganzzahlige Division von x durch y
x + y	Summe von x und y
x – y	Differenz von x und y
x << n	Bitweise Verschiebung um n Stellen nach links
x >> n	Bitweise Verschiebung um n Stellen nach rechts
x & y	Bitweises UND zwischen x und y
x ^ y	Bitweises ausschließendes ODER zwischen x und y
x | y	Bitweises nicht ausschließendes ODER zwischen x und y
x < y	Ist x kleiner als y?
x <= y	Ist x kleiner oder gleich y?

`x > y`	Ist x größer als y?
`x >= y`	Ist x größer oder gleich y?
`x <> y`	Ist x ungleich y?
`x != y`	Ist x ungleich y?
`x == y`	Ist x gleich y?
`x is y`	Sind x und y identisch?
`x is not y`	Sind x und y nicht identisch?
`x in y`	Befindet sich x in y?
`x not in y`	Befindet sich x nicht in y?
`not x`	Logische Negierung
`x and y`	Logisches UND
`x or y`	Logisches ODER

Tabelle A.2 Die Operatorrangfolge

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

A.4 Built-in Exceptions

In diesem Abschnitt des Anhangs finden Sie eine Auflistung aller in Python eingebauten Exception-Typen. Die kurze Beschreibung erläutert, in welchen Fällen der jeweilige Exception-Typ auftritt, und gibt somit eine Art Richtlinie, wann Sie einen der eingebauten Exception-Typen in Ihren Programmen verwenden sollten.

Abbildung 13.1 stellt die vollständige Vererbungshierarchie für die eingebauten Exception-Typen dar.

Beachten Sie, dass Sie eigene Exception-Typen von der Basisklasse `Exception` ableiten sollten. Selbstverständlich können Sie auch direkt von einer der eingebauten Exceptions erben, die ihrerseits von `Exception` erben.

Zunächst sehen Sie eine Tabelle mit allen Basisklassen, von denen verschiedene eingebaute Exceptions erben.

Exception	Erklärung
`BaseException`	Die Basisklasse aller eingebauten Exceptions. Diese Klasse stellt einen Großteil der Exception-Funktionalität bereit und wird in Abschnitt 13.1.1 behandelt.
`Exception`	Die Basisklasse aller »normalen« Exceptions. Spezielle Exceptions wie `SystemExit` oder `KeyboardInterrupt` erben nicht von `Exception`. Beachten Sie, dass alle selbst definierten Exception-Typen von `Exception` erben sollten.
`StandardError`	Die Basisklasse aller Exception-Typen, die einen Fehler repräsentieren
`ArithmeticError`	Die Basisklasse aller Exception-Typen, die einen Fehler in einer arithmetischen

	Operation repräsentieren
LookupError	Die Basisklasse aller Exceptions, die auftreten, wenn auf einen nicht vorhandenen Schlüssel oder Index zugegriffen wird
EnvironmentError	Die Basisklasse aller Exception-Typen, die auftreten, wenn ein Fehler außerhalb des Python-Systems passiert – beispielsweise wenn eine Dateioperation fehlschlägt. Das Attribut args der Exception-Instanz referenziert ein Tupel, das Fehlercode und -beschreibung enthält.

Tabelle A.3 Basisklassen für eingebaute Exceptions

Die folgende Tabelle listet alle eingebauten Exception-Typen auf, die von den vorherigen Typen erben und tatsächlich geworfen werden.

Exception	Erklärung
AssertionError	Wird geworfen wenn eine assert-Anweisung fehlschlägt.
AttributeError	Wird geworfen, wenn auf ein nicht existierendes Attribut einer Instanz zugegriffen wird.
EOFError	Wird geworfen wenn die Funktionen input oder raw_input EOF (*end-of-file*) signalisiert bekommen, ohne zuvor Daten eingelesen zu haben.
FloatingPointError	Wird geworfen wenn eine Gleitkommaoperation fehlschlägt.
GeneratorExit	Wird geworfen wenn die Methode close eines Generators gerufen wird. Diese Exception erbt direkt von Exception statt von StandardError.
IOError	Wird geworfen wenn eine I/O-Operation, wie beispielsweise eine Bildschirmaus- oder -eingabe fehlschlägt.
ImportError	Wird geworfen wenn eine import-Anweisung das angegebene Modul nicht finden kann oder der angegebene Name bei einer from/import-Anweisung nicht im Modul enthalten ist.
IndexError	Wird geworfen wenn mit einem Index auf eine Sequenz zugegriffen wird, der außerhalb des gültigen Bereichs liegt.
KeyError	Wird geworfen wenn mit einem Schlüssel auf ein Dictionary zugegriffen wird, der nicht in diesem vorhanden ist.
KeyboardInterrupt	Wird geworfen wenn das Python-Programm per Tastenkombination (üblicherweise mit Strg + C) abgebrochen wird. Diese Exception erbt direkt von BaseException, sodass sie nicht irrtümlich abgefangen werden kann.
MemoryError	Wird geworfen wenn nicht mehr genügend Speicher zur Ausführung einer Operation vorhanden ist.
NameError	Wird aufgerufen, wenn ein unbekannter lokaler oder globaler Bezeichner verwendet wird.

NotImplementedError	Basisklassen werfen diese Exception aus Methoden heraus, die von einer abgeleiteten Klasse überschrieben werden müssen.
OSError	Diese Exception wird vom Modul os geworfen und erbt von EnvironmentError.
OverflowError	Wird geworfen wenn das Ergebnis einer arithmetischen Operation zu groß für den entsprechenden Datentyp ist.
RuntimeError	Wird geworfen wenn ein Fehler aufgetreten ist, der durch keinen der anderen Exception-Typen ausgedrückt wird. Das Attribut args enthält die genaue Fehlermeldung.
StopIteration	Wird von der Methode next eines Iterators geworfen, wenn kein nächstes Element existiert. Diese Exception erbt direkt von Exception statt von StandardError.
SyntaxError	Wird geworfen wenn ein Syntaxfehler in einem Python-Programm, einer exec-Anweisung oder einem Aufruf von eval auftritt. Die Attribute filename, lineno, offset und text spezifizieren den Fehler genauer.
SystemError	Wird geworfen wenn ein interner Fehler im Python-Interpreter auftritt.
SystemExit	Wird von sys.exit geworfen und beendet das laufende Programm, wenn sie nicht abgefangen wird. Eine nicht abgefangene SystemExit-Exception verursacht keinen Traceback. Die Exception erbt direkt von BaseException, sodass sie nicht versehentlich abgefangen werden kann.
TypeError	Wird geworfen wenn eine Operation auf einer Instanz durchgeführt wird, die einen dafür unpassenden Datentyp hat.
UnboundLocalError	Wird geworfen wenn in einer Funktion oder Methode eine lokale Referenz bereits verwendet wird, bevor ihr eine Instanz zugewiesen wurde.
UnicodeError	Wird geworfen wenn bei Encode- und Decode-Vorgängen im Zusammenhang mit Unicode Fehler auftreten.
UnicodeEncodeError	Wird geworfen wenn ein Unicode-spezifischer Fehler bei einem Encode-Vorgang auftritt. Diese Exception erbt von UnicodeError.
UnicodeDecodeError	Wird geworfen wenn ein Unicode-spezifischer Fehler bei einem Decode-Vorgang auftritt. Diese Exception erbt von UnicodeError.
UnicodeTranslateError	Wird geworfen wenn ein Unicode-spezifischer Fehler bei einem Translate-Vorgang auftritt. Diese Exception erbt von UnicodeError.
	Wird geworfen wenn eine Operation auf

`ValueError`	einer Instanz durchgeführt werden soll, die zwar den richtigen Typ, aber einen unpassenden Wert hat.
`WindowsError`	Wird geworfen wenn ein Windows-spezifischer Fehler auftritt. Diese Exception erbt von `OSError`.
`ZeroDivisionError`	Wird bei einer Division durch null geworfen.

Tabelle A.4 Eingebaute Exception-Typen

Die folgende Tabelle listet alle eingebauten Warnungen auf. Das sind spezielle Exceptions, die beispielsweise auf die Verwendung eines veralteten Funktionsparameters hindeuten. Das Auftreten von Warnungen behindert die Ausführung des Programms nicht.

Exception	Erklärung
`Warning`	Basisklasse für alle Warnungen
`UserWarning`	Basisklasse für eigene Warnungen
`DeprecationWarning`	Wird bei der Verwendung eines als veraltet eingestuften Sprachfeatures geworfen.
`PendingDeprecationWarning`	Wird bei der Verwendung eines als zukünftig veraltet eingestuften Sprachfeatures geworfen.
`SyntaxWarning`	Wird bei problematischer Syntax geworfen.
`RuntimeWarning`	Wird bei bedenklichem Laufzeitverhalten geworfen.
`FutureWarning`	Wird bei der Verwendung von Sprachfeatures geworfen, die sich in Zukunft verändern werden.
`ImportWarning`	Wird bei problematischen `import`-Anweisungen geworfen.
`UnicodeWarning`	Wird bei Unicode-spezifischen Problemen geworfen.

Tabelle A.5 Eingebaute Warnungen

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

4.5 Logische Ausdrücke

Es ist möglich, Zahlen miteinander zu vergleichen:

```
>>> 3 < 4
True
```

Hier wird getestet, ob 3 kleiner ist als 4. Auf solche Vergleiche antwortet der Interpreter mit einem *Wahrheitswert*, also mit `True` (dt. *wahr*) oder `False` (dt. *falsch*). Ein Vergleich wird mithilfe eines sogenannten *Vergleichsoperators*, in diesem Fall <, durchgeführt.

Die folgende Liste führt alle relevanten Vergleichsoperatoren auf:

Vergleich	Bedeutung
`3 == 4`	Ist 3 gleich 4? Beachten Sie das doppelte Gleichheitszeichen, das den Vergleich von einer Zuweisung unterscheidet.
`3 != 4`	Ist 3 ungleich 4?
`3 < 4`	Ist 3 kleiner als 4?
`3 > 4`	Ist 3 größer als 4?
`3 <= 4`	Ist 3 kleiner oder gleich 4?
`3 >= 4`	Ist 3 größer oder gleich 4?

Tabelle 4.1 Vergleiche in Python

Allgemein kann für 3 und 4 ein beliebiger arithmetischer Ausdruck eingesetzt werden. Wenn zwei arithmetische Ausdrücke durch einen der obigen Operatoren miteinander verglichen werden, so erzeugt man einen sogenannten *logischen Ausdruck*. Ein solcher könnte also auch folgendermaßen aussehen:

```
(a - 7) < (b * b + 6.5)
```

Neben den bereits eingeführten arithmetischen Operatoren gibt es drei logische Operatoren, mit denen das Ergebnis eines logischen Ausdrucks verändert oder zwei logische Ausdrücke miteinander verknüpft werden können.

Der Operator not kehrt das Ergebnis eines Vergleiches um, macht also aus True False und aus False True. Der Ausdruck not (3 < 4) wäre also das Gleiche wie 3 >= 4:

```
>>> not (3 < 4)
False
>>> not (4 < 3)
True
```

Der Operator and bekommt zwei logische Ausdrücke als Operanden und ergibt nur dann True, wenn sowohl der erste Ausdruck als auch der zweite True ergeben haben. Er entspricht damit der umgangssprachlichen »Und«-Verknüpfung zweier Satzteile. Im Beispiel kann dies so aussehen:

```
>>> (3 < 4) and (5 < 6)
True
>>> (3 < 4) and (4 < 3)
False
```

Der Operator or entspricht dem umgangssprachlichen »oder«. Er bekommt zwei logische Ausdrücke als Operanden und ergibt nur dann False, wenn sowohl der erste Ausdruck als auch der zweite False ergeben haben. Der Operator ergibt also True, wenn mindestens einer seiner Operanden True ergeben hat:

```
>>> (3 < 4) or (5 < 6)
True
>>> (3 < 4) or (4 < 3)
True
>>> (5 > 6) or (4 < 3)
False
```

Wir haben der Einfachheit halber hier nur Zahlen miteinander verglichen. Selbstverständlich macht ein solcher Vergleich nur dann Sinn, wenn komplexere arithmetische Ausdrücke miteinander verglichen werden. Durch die vergleichenden Operatoren und die drei sogenannten *booleschen Operatoren* not, and und or können schon sehr komplexe Vergleiche erstellt werden.

Beachten Sie, dass bei allen Beispielen aus Gründen der Übersicht Klammern gesetzt wurden. Durch Prioritätsregelungen der Operatoren untereinander sind diese überflüssig. Das bedeutet, dass jedes hier vorgestellte Beispiel auch ohne Klammern wie erwartet funktionieren würde. Trotzdem ist es gerade am Anfang sehr sinnvoll, durch Klammerung die Zugehörigkeiten visuell eindeutig zu gestalten.

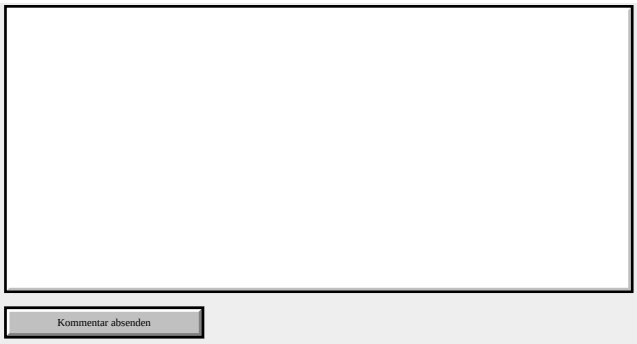
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

8.5 Sequenzielle Datentypen

Unter *sequenziellen Datentypen* wird eine Klasse von Datentypen zusammengefasst, die Folgen von gleichartigen oder verschiedenen *Elementen* verwalten können. Die in sequenziellen Datentypen gespeicherten Elemente haben eine definierte Reihenfolge, und man kann über eindeutige Indizes auf sie zugreifen.

Python stellt die folgenden fünf sequenziellen Typen zu Verfügung: `str`, `unicode`, `list` und `tuple`.

Mithilfe der ersten beiden sequenziellen Datentypen, `str` und `unicode` wird in Python die Arbeit mit Zeichenketten, also Folgen von Buchstaben, ermöglicht, wobei je nach Anwendungsfall einer von ihnen besser geeignet ist. Instanzen des Typs `str` speichern Folgen von Bytes und eignen sich daher besonders für binäre Datenströme, aber auch für Zeichenketten, die nur aus ASCII-Zeichen bestehen. Der Datentyp `unicode` ist für die Speicherung von Text-Strings konzipiert und speichert Folgen von Zeichen in einem speziellen Unicode-Format, das auch die komfortable Verwaltung von speziellen Sonderzeichen wie den deutschen Umlauten oder dem Eurozeichen ermöglicht.

Beide Datentypen sind immutable, ihr Wert kann sich nach der Instanziierung also nicht mehr verändern. Trotzdem können Sie komfortabel mit Strings arbeiten. Bei Änderungen wird nur nicht der Ursprungsstring verändert, sondern stets ein neuer String erzeugt.

Die Typen `list` und `tuple` können Folgen beliebiger Instanzen speichern. Der wesentliche Unterschied zwischen den beiden fast identischen Datentypen ist, dass eine Liste nach ihrer Erzeugung verändert werden kann, während ein Tupel keine Änderung des Anfangsinhalts zulässt: `list` ist ein mutable, `tuple` ein immutable Datentyp.

Für jede Instanz eines sequenziellen Datentyps gibt es einen Grundstock von Operatoren und Methoden, der immer verfügbar ist. Der Einfachheit halber werden wir diesen allgemein am Beispiel von `str`-Instanzen einführen und erst in den folgenden Abschnitten Besonderheiten bezüglich der einzelnen Datentypen aufzeigen.

Für alle sequenziellen Datentypen sind folgende Operationen definiert (`s` und `t` sind hierbei Instanzen desselben sequenziellen Datentyps; `i`, `j`, `k` und `n` sind Ganzzahlen, `x` ist eine Referenz auf eine beliebige Instanz):

Notation	Beschreibung
`x in s`	Prüft, ob x in s enthalten ist. Das Ergebnis ist eine bool-Instanz.
`x not in s`	Prüft, ob x nicht in s enthalten ist. Das Ergebnis ist eine bool-Instanz. Gleichwertig mit `not x in s`.
`s + t`	Das Ergebnis ist eine neue Sequenz, die die Verkettung von s und t enthält.
`s += t`	Erzeugt die Verkettung von s und t und weist sie s zu.
`s * n` oder `n * s`	Liefert eine neue Sequenz, die die Verkettung von n Kopien von s enthält.
`s *= n`	Erzeugt das Produkt `s * n` und weist es s zu.
`s[i]`	Liefert das i-te Element von s.
`s[i:j]`	Liefert den Ausschnitt aus s von i bis j.
`s[i:j:k]`	Liefert den Ausschnitt aus s von i bis j, wobei nur jedes k-te Element beachtet wird.
`len(s)`	Gibt eine Ganzzahl zurück, die die Anzahl der Elemente von s angibt.
`min(s)`	Liefert das kleinste Element von s, sofern eine Ordnungsrelation für die Elemente definiert ist.
`max(s)`	Liefert das größte Element von s, sofern eine Ordnungsrelation für die Elemente definiert ist.

Tabelle 8.13 Methoden der sequenziellen Datentypen

Wie bereits bekannt ist, lässt sich ein neuer String erzeugen, indem man seinen Inhalt in doppelte Hochkommata schreibt:

```
>>> s = "Dies ist unser Teststring"
```

Ist ein Element vorhanden?

Mithilfe von `in` lässt sich ermitteln, ob ein bestimmtes Element in einer Sequenz enthalten ist. Da die Elemente eines Strings Buchstaben sind, können wir mit dem Operator prüfen, ob ein bestimmter Buchstabe in einem String vorkommt. Als Ergebnis wird ein Wahrheitswert geliefert: `True`, wenn das Element vorhanden ist, und `False`, wenn es nicht vorhanden ist. Buchstaben kann man in Python durch Strings der Länge eins abbilden:

```
>>> s = "Dies ist unser Teststring"
>>> "u" in s
True
>>> if "j" in s:
...     print "Juhuu, mein Lieblingsbuchstabe ist enthalten"
... else:
...     print "Ich mag diesen String nicht..."
Ich mag diesen String nicht...
```

Um das Gegenteil – also ob ein Element nicht in einer Sequenz enthalten ist – zu prüfen, dient der `not in`-Operator. Seine Verwendung entspricht der des `in`-Operators, mit dem einzigen Unterschied, dass er das negierte Ergebnis produziert:

```
>>> "a" in "Besuch beim Zahnarzt"
True
>>> "a" not in "Besuch beim Zahnarzt"
False
```

Sie werden sich an dieser Stelle zu Recht fragen, warum für diesen Zweck ein eigener Operator definiert worden ist, wo man doch mit `not` jeden booleschen Wert negieren kann. Folgende Überprüfungen sind vollkommen gleichwertig:

```
>>> "n" not in "Python ist toll"
False
>>> not "n" in "Python ist toll"
False
```

Der Grund für diese scheinbar überflüssige Definition liegt in der besseren Lesbarkeit. `x not in s` liest sich im Gegensatz zu `not x in s` genau wie ein englischer Satz, während die andere Form unnötig kompliziert zu lesen ist. [Zusätzlich muss man für die Interpretation von `not x in s` die Priorität der beiden Operatoren `not` bzw. `in` kennen. Wenn der `not`-Operator stärker bindet, würde der Ausdruck wie `(not x) in s` ausgewertet. Hat `in` eine höhere

Priorität, wäre der Ausdruck wie `not (x in s)` zu behandeln. Tatsächlich bindet `in` stärker als `not`, womit letztere Deutung die richtige ist.]

Verkettung von Sequenzen

Es kommt häufig vor, dass man mehrere Sequenzen aneinanderhängen möchte, um mit dem Ergebnis weiterzuarbeiten. Beispielsweise könnte man den Vor- und den Nachnamen eines Benutzers zu seinem gesamten Namen zusammenfügen, um ihn dann persönlich zu begrüßen. Für solche Zwecke dient der +-Operator, der aus zwei Sequenzen eine neue erzeugt, indem er die beiden verkettet:

```
>>> vorname = "Heinz"
>>> nachname = "Meier"
>>> name = vorname + " " + nachname
>>> name
'Heinz Meier'
```

Eine weitere Möglichkeit, Strings zu verketten, bietet der Operator += für erweiterte Zuweisungen:

```
>>> s = "Musik"
>>> s += "lautsprecher"
>>> s
'Musiklautsprecher'
```

Wiederholung von Sequenzen

Man kann in Python das Produkt einer Sequenz s mit einer Ganzzahl n bilden: `n * s` oder `s * n`. Das Ergebnis ist eine neue Sequenz, die n Kopien von s hintereinander enthält:

```
>>> 3 * "abc"
'abcabcabc'
>>> "xyz" * 5
'xyzxyzxyzxyzxyz'
```

Genau wie bei der Verkettung gibt es auch hier einen Operator für die erweiterte Zuweisung: *=. Da seine Verwendung analog zu += erfolgt, wurde auf ein weiteres platzhungriges Beispiel verzichtet.

Zugriff auf bestimmte Elemente einer Sequenz

Wie eingangs erwähnt wurde, stellen Sequenzen Folgen von Elementen dar. Da diese Elemente in einer bestimmten Reihenfolge gespeichert werden – beispielsweise wäre ein String, bei dem die Reihenfolge der Buchstaben willkürlich ist, wenig sinnvoll –, kann man jedem Element der Sequenz eine ganze Zahl, den sogenannten *Index* zuweisen. Dafür werden alle Elemente der Sequenz fortlaufend von vorn nach hinten durchnummeriert, wobei das erste Element den Index `0` bekommt.

Mit dem `[]`-Operator kann man auf ein bestimmtes Element der Sequenz zugreifen, indem man den entsprechenden Index in die eckigen Klammern schreibt:

```
>>> alphabet = "abcdefghijklmnopqrstuvwxyz"
>>> alphabet[9]
'j'
>>> alphabet[1]
'b'
```

Um komfortabel auf das letzte oder das x-te Element von hinten zugreifen zu können, gibt es eine weitere Indizierung der Elemente von hinten nach vorn. Das letzte Element erhält dabei als Index `-1`, das vorletzte `-2` und so weiter:

```
>>> name = "Python"
>>> name[-2]
'o'
```

Versucht man, mit einem Index auf ein nicht vorhandenes Element zuzugreifen, wird dies mit einem `IndexError` quittiert:

```
>>> zukurz = "Ich bin zu kurz"
>>> zukurz[1337]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Neben dem Zugriff auf einzelne Elemente der Sequenz ist es mit dem `[]`-Operator auch möglich, ganze Teilsequenzen auszulesen. Dies erreicht man

dadurch, dass man den Anfang und das Ende der gewünschten Teilfolge durch einen Doppelpunkt getrennt in die eckigen Klammern schreibt. Der Anfang ist dabei der Index des ersten Elements der gewünschten Teilfolge, und das Ende ist der Index des ersten Elements, das nicht mehr in der Teilfolge enthalten sein soll.

Um im folgenden Beispiel die Zeichenfolge "WICHTIG" aus dem String zu extrahieren, geben wir den Index des großen "W" und den des ersten "s" nach "WICHTIG" an:

```
>>> s = "schrottschrottWICHTIGschrottschrott"
>>> s[14]
'W'
>>> s[21]
's'
>>> s[14:21]
'WICHTIG'
```

Es ist auch möglich, bei diesem sogenannten *Slicing* (dt. *Abschneiden*) positive und negative Indizes zu mischen. Beispielsweise ermittelt der folgende Code-Abschnitt eine Teilfolge ohne das erste und letzte Element der Ursprungssequenz:

```
>>> string = "ameisen"
>>> string[1:-1]
'meise'
```

Aus Bequemlichkeitsgründen können die Indizes weggelassen werden, was dazu führt, dass der maximal bzw. minimal mögliche Wert angenommen wird. Entfällt der Startindex, wird das nullte als erstes Element der Teilsequenz angenommen, und verzichtet man auf den Endindex, werden alle Buchstaben bis zum Ende kopiert. Möchten wir zum Beispiel die ersten fünf Buchstaben eines Strings oder alle ab dem fünften ermitteln, geht das folgendermaßen:

```
>>> s = "abcdefghijklmnopqrstuvwxyz"
>>> s[:5]
'abcde'
>>> s[5:]
'fghijklmnopqrstuvwxyz'
```

Wenn man beide Indizes ausspart (`s[:]`), lässt sich auch eine echte Kopie der Sequenz erzeugen, weil dann alle Elemente vom ersten bis zum letzten kopiert werden. Beachten Sie bitte die unterschiedlichen Ergebnisse der beiden folgenden Code-Ausschnitte:

```
>>> s1 = "Kopier mich!"
>>> s2 = s1
>>> s1 == s2
True
>>> s1 is s2
True
```

Wie erwartet verweisen `s1` und `s2` auf dieselbe Instanz, sind also identisch. Anders sieht es bei dem nächsten Beispiel aus, bei dem eine echte Kopie von "Kopier mich!" im Speicher erzeugt wird. Dies zeigt sich beim Identitätsvergleich mit `is`:

```
>>> s1 = "Kopier mich!"
>>> s2 = s1[:]
>>> s1 == s2
True
>>> s1 is s2
False
```

Slicing bietet noch flexiblere Möglichkeiten, wenn man nicht eine ganze Teilsequenz, sondern nur bestimmte Elemente dieses Teils extrahieren möchte. Mit der *Schrittweite* (hier engl. *step*) lässt sich angeben, wie die Indizes vom Beginn bis zum Ende einer Teilsequenz gezählt werden sollen. Die Schrittweite wird, durch einen weiteren Doppelpunkt abgetrennt, nach der hinteren Grenze angegeben. Eine Schrittweite von 2 sorgt beispielsweise dafür, dass nur jedes zweite Element kopiert wird:

```
>>> ziffern = "0123456789"
>>> ziffern[1:10:2]
'13579'
```

Die Zeichenfolge, die ab dem ersten Element (Achtung: Die Zählweise beginnt bei 0) jedes zweite Element von `ziffern` enthält, ergibt einen neuen String mit den ungeraden Ziffern. Auch bei dieser erweiterten Notation können die Grenzindizes entfallen. Der folgende Code ist also zum vorherigen Beispiel äquivalent:

```
>>> ziffern = "0123456789"
>>> ziffern[1::2]
'13579'
```

Eine negative Schrittweite bewirkt ein Rückwärtszählen vom Start- zum Endindex, wobei in diesem Fall der Startindex auf ein weiter hinten liegendes Element der Sequenz als der Endindex verweisen muss. Mit einer Schrittweite von `-1` lässt sich sehr elegant eine Sequenz »umdrehen«:

```
>>> name = "ytnoM Python"
>>> name[4::-1]
'Monty'
>>> name[::-1]
'nohtyP Monty'
```

Bei negativen Schrittweiten vertauschen sich Anfang und Ende der Sequenz. Deshalb wird in dem Beispiel `name[4::-1]` nicht alles vom vierten bis zum letzten Zeichen, sondern der Teil vom vierten bis zum ersten Zeichen ausgelesen.

Wichtig für den Umgang mit dem Slicing ist die Tatsache, dass zu große oder zu kleine Indizes nicht zu einem `IndexError` führen, wie es beim Zugriff auf einzelne Elemente der Fall ist. Zu große Indizes werden intern durch den maximal, zu kleine durch den minimal möglichen Index ersetzt. Liegen beide Indizes außerhalb des gültigen Bereichs oder ist der Startindex bei positiver Schrittweise größer als der Endindex, wird eine leere Sequenz zurückgegeben:

```
>>> s = "Viel weniger als 1337 Zeichen"
>>> s[5:1337]
'weniger als 1337 Zeichen'
>>> s[-100:100]
'Viel weniger als 1337 Zeichen'
>>> s[1337:2674]
''
>>> s[10:4]
''
```

Länge einer Sequenz

Als *Länge* einer Sequenz ist in Python die Anzahl ihrer Elemente definiert. Sie ist eine ganze Zahl größer oder gleich null und lässt sich mit der Built-in-Function `len` ermitteln:

```
>>> string = "Wie lang bin ich wohl?"
>>> len(string)
22
```

Das kleinste und das größte Element einer Sequenz

Eine sehr häufige Aufgabe innerhalb eines Programms besteht darin, das kleinste beziehungsweise größte Element einer Sequenz zu ermitteln. Aus diesem Grund existieren in Python die Funktionen `min` und `max`, wobei `min` das kleinste und `max` das größte Element zurückgibt. Allerdings machen diese beiden Funktionen nur dann Sinn, wenn eine *Ordnungsrelation* für die Elemente der Sequenz existiert (in Abschnitt 8.3.4 über komplexe Zahlen wird zum Beispiel ein Datentyp ohne Ordnungsrelation beschrieben). Für Buchstaben wird ihre Position im Alphabet als Ordnungsrelation benutzt, solange es sich nur um Großbuchstaben oder nur um Kleinbuchstaben handelt. Beim Vergleichen von Groß- und Kleinbuchstaben untereinander gelten Kleinbuchstaben immer als größer [Falls Sie sich über dieses merkwürdige Verhalten wundern: Die Reihenfolge im Alphabet beschreibt nur einen Teilaspekt der Ordnungsrelation für einzelne Zeichen. Sonderzeichen wie beispielsweise das Leerzeichen lassen sich damit nicht sinnvoll einordnen. Sie werden im Abschnitt über Strings die Hintergründe hierzu kennenlernen.] – `"a"` ist also kleiner als `"z"` und größer als `"A"`:

```
>>> max("wer gewinnt wohl")
'w'
>>> min("zeichenkette")
'c'
```

top

8.5.1 Listen – list ▼▲

In diesem Abschnitt werden Sie den ersten veränderbaren (mutable) Datentyp, die *Liste*, kennenlernen. Anders als bei dem sequenziellen Datentyp `str`, der nur gleichartige Elemente, die Buchstaben, speichern kann, sind Listen für die Verwaltung beliebiger Instanzen auch unterschiedlicher Datentypen geeignet. Eine Liste kann also durchaus Zahlen, Strings oder auch weitere Listen als Elemente enthalten, wodurch sie sehr flexibel anwendbar ist.

Eine neue Liste lässt sich dadurch erzeugen, dass man eine Aufzählung ihrer Elemente in eckige Klammern [] schreibt:

```
>>> l = [1, 0.5, "String", 2]
```

Die Liste l enthält nun zwei Ganzzahlen, eine Gleitkommazahl und einen String.

Da es sich bei dem Listentyp, der innerhalb von Python den Namen list hat, um einen sequenziellen Datentyp handelt, können alle im letzten Abschnitt beschriebenen Methoden und Verfahren auf ihn angewandt werden.

Allerdings kann sich der Inhalt einer Liste auch nach ihrer Erzeugung ändern, weshalb eine Reihe weiterer Operatoren und Methoden für sie verfügbar sind:

Operator	Wirkung
s[i] = x	Das Element von s mit dem Index i wird durch x ersetzt.
s[i:j] = t	Der Teil s[i:j] wird durch t ersetzt. Dabei muss t iterierbar sein.
s[i:j:k] = t	Die Elemente von s[i:j:k] werden durch die von t ersetzt.
del s[i]	Das i-te Element von s wird entfernt.
del s[i:j]	Der Teil s[i:j] wird aus s entfernt. Das ist äquivalent zu s[i:j] = [].
del s[i:j:k]	Die Elemente der Teilfolge s[i:j:k] werden aus s entfernt.

Tabelle 8.14 Operatoren für den Datentyp list

Wir werden diese Operatoren der Reihe nach mit kleinen Beispielen erklären.

Verändern eines Wertes innerhalb der Liste

Man kann Elemente einer Liste durch andere ersetzen, wenn man ihren Index kennt:

```
>>> s = [1, 2, 3, 4, 5, 6, 7]
>>> s[3] = 1337
>>> s
[1, 2, 3, 1337, 5, 6, 7]
```

Diese Methode eignet sich allerdings nicht, um mehr Elemente in die Liste einzufügen. Es können nur bereits bestehende Elemente ersetzt werden, und die Länge der Liste bleibt unverändert.

Ersetzen von Teillisten und Einfügen neuer Elemente

Es ist möglich, eine ganze Teilliste durch andere Elemente zu ersetzen. Dazu schreibt man den zu ersetzenden Teil der Liste wie beim Slicing auf, wobei er aber auf der linken Seite einer Zuweisung stehen muss:

```
>>> einkaufen = ["Brot", "Eier", "Milch", "Fisch", "Mehl"]
>>> einkaufen[1:3] = ["Wasser", "Wurst"]
>>> einkaufen
['Brot', 'Wasser', 'Wurst', 'Fisch', 'Mehl']
```

Die Liste, die eingefügt werden soll, kann auch mehr oder weniger Elemente als der zu ersetzende Teil haben und sogar ganz leer sein.

Man kann wie beim Slicing auch eine Schrittweite angeben, um beispielsweise nur jedes dritte Element der Teilsequenz zu ersetzen. Im nachstehenden Beispiel wird jedes dritte Element der Teilsequenz s[2:11] durch das entsprechende Element aus ["A", "B", "C"] ersetzt:

```
>>> s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> s[2:9:3] = ["A", "B", "C"]
>>> s
[0, 1, 'A', 3, 4, 'B', 6, 7, 'C', 9, 10]
```

Wird eine Schrittweite angegeben, muss die Sequenz auf der rechten Seite der Zuweisung genauso viele Elemente wie die Teilsequenz auf der linken Seite haben. Ist das nicht der Fall, wird ein ValueError erzeugt.

Elemente und Teillisten löschen

Um einen einzelnen Wert aus einer Liste zu entfernen, dient der del-Operator:

```
>>> s = [26, 7, 1987]
>>> del s[0]
>>> s
[7, 1987]
```

Auf diese Weise lassen sich auch ganze Teillisten entfernen:

```
>>> s = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> del s[3:6]
>>> s
[9, 8, 7, 3, 2, 1]
```

Für das Entfernen von Teilen einer Liste wird auch die Schrittfolge der Slicing-Notation unterstützt. Im folgenden Beispiel werden damit alle Elemente mit geradem Index entfernt (Achtung: "a" hat den Index 0):

```
>>> s = ["a","b","c","d","e","f","g","h","i","j"]
>>> del s[::2]
>>> s
['b', 'd', 'f', 'h', 'j']
```

Nachdem nun die Operatoren für Listen behandelt worden sind, wenden wir uns den Methoden einer Liste zu. In der Tabelle sind s und t Listen, i, j und k sind Ganzzahlen, und ist x eine beliebige Instanz:

Methode	Wirkung
s.append(x)	Hängt x ans Ende von s an.
s.extend(t)	Hängt alle Elemente von t ans Ende von s an.
s.count(x)	Gibt an, wie oft das Element x in s vorkommt.
s.index(x[, i[, j]])	Gibt den Index k des ersten Vorkommens von x im Bereich i <= k < j zurück.
s.insert(i, x)	Fügt x an der Stelle i in s ein. Anschließend hat s[i] den Wert von x, wobei alle folgenden Elemente um eine Stelle nach hinten aufrücken.
s.pop([i])	Gibt das i-te Element von s zurück und entfernt es aus s. Ist i nicht angegeben, wird das letzte Element genommen.
s.remove(x)	Entfernt das erste Vorkommen von x aus der Sequenz s.
s.reverse()	Kehrt die Reihenfolge der Elemente in s um.
s.sort([cmp[, key[, reverse]])	Sortiert s.

Tabelle 8.15 Methoden von list-Instanzen

s.append(x)

Mit append kann man eine Liste am Ende um ein weiteres Element erweitern:

```
>>> s = ["Nach mir soll noch ein String stehen"]
>>> s.append("Hier ist er")
>>> s
['Nach mir soll noch ein String stehen', 'Hier ist er']
```

s.extend(t)

Um an eine Liste mehrere Elemente anzuhängen, dient die Methode extend, die ein iterierbares Objekt – beispielsweise eine andere Liste – als Parameter t erwartet. Im Ergebnis werden alle Elemente von *t* an die Liste *s* angehängt:

```
>>> s = [1, 2, 3]
>>> s.extend([4, 5, 6])
>>> s
[1, 2, 3, 4, 5, 6]
```

s.count(x)

Man kann mit count ermitteln, wie oft ein bestimmtes Element *x* in einer Liste enthalten ist:

```
>>> s = [1, 2, 2, 3, 2]
>>> s.count(2)
3
```

s.index(x[, i[, j]])

Mit index kann man die Position eines Elements in einer Liste ermitteln:

```
>>> ziffern = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ziffern.index(3)
2
```

Um die Suche auf einen Teilbereich der Liste einzuschränken, dienen die Parameter *i* und *k*, wobei *i* den ersten Index der gewünschten Teilfolge und *k* den ersten Index hinter der gewünschten Teilfolge angibt:

```
>>> [1, 22, 333, 4444, 333, 22, 1].index(1, 3, 7)
6
```

Ist das Element *x* nicht in *s* oder in der angegebenen Teilfolge enthalten, führt index zu einem ValueError:

```
>>> s = [2.5, 2.6, 2.7, 2.8]
>>> s.index(2.4)
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    s.index(2.4)
ValueError: list.index(x): x not in list
```

s.insert(i, x)

Mit insert kann man an beliebiger Stelle ein neues Element in eine Liste einfügen. Der erste Parameter *i* gibt den gewünschten Index des neuen Elements, der zweite, *x*, das Element selbst an:

```
>>> erst_mit_loch = [1, 2, 3, 5, 6, 7, 8]
>>> erst_mit_loch.insert(3, 4)
>>> erst_mit_loch
[1, 2, 3, 4, 5, 6, 7, 8]
```

Ist der Index *i* zu klein, wird *x* am Anfang von *s* eingefügt, ist er zu groß, wird er wie bei append am Ende angehängt.

s.pop([i])

Das Gegenstück zu insert ist pop. Mit dieser Methode kann man ein beliebiges Element anhand seines Index aus einer Liste entfernen. Ist der optionale Parameter nicht angegeben, so wird das letzte Element der Liste entfernt. Das entfernte Element wird von pop zurückgegeben:

```
>>> s = ["H", "a", "l", "l", "o"]
>>> s.pop()
'o'
>>> s.pop(0)
'H'
>>> s
['a', 'l', 'l']
```

Wird versucht, einen ungültigen Index zu übergeben oder ein Element aus einer leeren Liste zu entfernen, wird ein IndexError erzeugt.

s.remove(x)

Möchte man ein Element mit einem bestimmten Wert aus einer Liste entfernen, egal welchen Index es hat, kann man die Methode remove bemühen. Sie entfernt das erste Element der Liste, das den gleichen Wert wie *x* hat.

```
>>> s = ["H", "u", "h", "u"]
>>> s.remove("u")
>>> s
['H', 'h', 'u']
```

Der Versuch, ein nicht vorhandenes Element zu entfernen, führt zu einem ValueError.

s.reverse()

Mit reverse kann man die Reihenfolge der Elemente einer Liste umkehren:

```
>>> s = [1, 2, 3]
>>> s.reverse()
>>> s
[3, 2, 1]
```

s[::-1]

Im Unterschied zu der Slice-Notation geschieht die Umkehrung »in place«. Es wird also keine neue `list`-Instanz erzeugt, sondern die alte verändert. Da dies weniger Rechenzeit und Speicher kostet, ist `reverse` der Slice-Notation vorzuziehen, wenn man nicht unbedingt eine neue Liste braucht.

s.sort([cmp[, key[, reverse]])

Die komplexeste Methode des `list`-Datentyps ist `sort`, mit der eine Liste nach bestimmten Kriterien sortiert werden kann. Ruft man die Methode ohne Parameter auf, benutzt Python die normalen Vergleichsoperatoren zum Sortieren:

```
>>> l = [4, 2, 7, 3, 6, 1, 9, 5, 8]
>>> l.sort()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Enthält eine Liste Elemente, für die keine Ordnungsrelation definiert ist, wie zum Beispiel `complex`, führt der Aufruf von `sort` ohne Parameter zu einem `TypeError`:

```
>>> lst = [5 + 13j, 1 + 4j, 6 + 2j]
>>> lst.sort()
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    lst.sort()
TypeError: no ordering relation is defined for complex numbers
```

Angenommen, wir wollten eine Liste komplexer Zahlen nach ihrem Imaginärteil sortieren, könnten wir den optionalen Parameter *cmp* benutzen. Die Methode erwartet im Parameter *cmp* eine Referenz auf eine Funktion, die ihrerseits zwei Parameter erwartet und diese vergleicht. Ist der Wert des ersten Parameters kleiner als der des zweiten, soll der Rückgabewert der übergebenen Funktion negativ sein; ist er größer, muss *cmp* eine positive Zahl zurückgeben. Bei gleich einzuordnenden Werten soll die übergebene Funktion `0` zurückgeben. Wir haben bis jetzt noch nicht besprochen, wie eigene Funktionen definiert werden, aber das behandelte Beispiel sollte trotzdem verständlich sein. Wenn Sie genau nachlesen möchten, was es mit Funktionsdefinitionen auf sich hat, können Sie sich im Kapitel 10 informieren.

Der folgende Code-Ausschnitt definiert eine Funktion mit dem Namen `vergleiche_complex`, die zwei `complex`-Instanzen hinsichtlich ihres Imaginärteils vergleicht:

```
>>> def vergleiche_complex(a, b):
...     return a.imag - b.imag
```

Die Funktion erwartet zwei Parameter, `a` und `b`, und nutzt ihre Differenz, um die Imaginärteile zu vergleichen. Wenn `a.imag` größer als `b.imag` ist, wird ihre Differenz positiv, sind sie gleich, ergibt `a.imag - b.imag` den Wert `0`. Ist `b.imag` der größere der beiden Werte, wird eine negative Zahl zurückgegeben. Mit der `return`-Anweisung wird der Rückgabewert an die aufrufende Ebene übergeben. Mithilfe dieser Funktion können wir nun die Liste aus `complex`-Instanzen sortieren:

```
>>> lst = [5 + 13j, 1 + 4j, 6 + 2j]
>>> lst.sort(vergleiche_complex)
>>> lst
[(6+2j), (1+4j), (5+13j)]
```

Wie Sie an der abschließenden Ausgabe sehen können, wurde die Liste korrekt sortiert.

Eine weitere Möglichkeit, die Sortierung anzupassen, bietet der Parameter key, der ebenfalls eine Funktionsreferenz erwartet. Die übergebende Funktion wird vor jedem Vergleich für beide Operanden aufgerufen und sollte deshalb einen Parameter erwarten. Im Ergebnis werden dann nicht die Operanden direkt verglichen, sondern stattdessen die entsprechenden Rückgabewerte der übergebenen Funktion.

Zur Veranschaulichung werden wir das letzte Beispiel unter Verwendung des Parameters *key* implementieren. Die benötigte Funktion muss den Imaginärteil einer übergebenen `complex`-Instanz zurückgeben:

```
>>> def imag_teil(c):
...     return c.imag
```

Es kommt sehr selten vor, dass man sowohl den *cmp*- als auch den *key*-Parameter übergibt, da die Operationen der *key*-Funktion ebenso gut innerhalb

der *cmp*-Funktion erledigt werden können. Deshalb bietet sich eine Übergabe als Schlüsselwortparameter an:

```
>>> lst = [5 + 13j, 1 + 4j, 6 + 2j]
>>> lst.sort(key=imag_teil)
>>> lst
[(6+2j), (1+4j), (5+13j)]
```

Das Ergebnis deckt sich mit unseren Erwartungen.

Der letzte Parameter, *reverse*, erwartet für die Übergabe einen booleschen Wert, der angibt, ob die Reihenfolge der Sortierung umgekehrt werden soll:

```
>>> l = [4, 2, 7, 3, 6, 1, 9, 5, 8]
>>> l.sort(reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Stabile Sortierverfahren

Eine wichtige Eigenschaft von `sort` ist, dass es sich um eine *stabile Sortierung* handelt. Stabile Sortierverfahren zeichnen sich dadurch aus, dass sie beim Sortieren die relative Position gleichwertiger Elemente nicht vertauschen. Stellen Sie sich einmal vor, Sie hätten folgende Namensliste:

Vorname	Nachname
Natalie	Schmidt
Mathias	Schwarz
Florian	Kroll
Ricarda	Schmidt
Helmut	Schmidt
Peter	Kaiser

Tabelle 8.16 Fiktive Namensliste

Nun ist es Ihre Aufgabe, diese Liste nach den Nachnamen zu sortieren. Gruppen mit gleichem Nachnamen sollen nach den jeweiligen Vornamen sortiert werden. Um dieses Problem zu lösen, können Sie die Liste im ersten Schritt nach den Vornamen sortieren, was zu folgender Anordnung führt:

Vorname	Nachname
Florian	Kroll
Helmut	**Schmidt**
Mathias	Schwarz
Natalie	**Schmidt**
Peter	Kaiser
Ricarda	**Schmidt**

Tabelle 8.17 Nach Vornamen sortierte Namensliste

Im Resultat interessieren uns jetzt nur die Positionen der drei Personen, deren Nachname »Schmidt« ist. Würde man einfach alle anderen Namen streichen, wären die Schmidts richtig sortiert, weil ihre relative Position durch den ersten Sortierlauf korrekt hergestellt wurde. Nun kommt die Stabilität der `sort`-Methode zum Tragen, weil dadurch bei einem erneuten Sortierdurchgang nach den Nachnamen diese relative Ordnung nicht zerstört wird. Das Ergebnis sähe am Ende so aus:

Vorname	Nachname
Peter	Kaiser
Florian	Kroll
Helmut	Schmidt
Natalie	Schmidt
Ricarda	Schmidt
Mathias	Schwarz

Tabelle 8.18 Vollständig sortierte Namensliste

Wäre `sort` nicht stabil, so gäbe es keine Garantie dafür, dass Helmut vor Natalie und Ricarda eingeordnet wird.

Wie Sie sehen, ist die sort-Methode extrem flexibel und mächtig. Bei Ihrer Arbeit mit Python werden Sie höchstwahrscheinlich niemals etwas anderes zum Sortieren Ihrer Daten verwenden.

Weitere Eigenschaften von Listen

Im Zusammenhang mit Pythons list-Datentyp ergeben sich ein paar Besonderheiten, die nicht unmittelbar ersichtlich sind.

Zum einen ist list ein veränderbarer Datentyp, und deshalb betreffen Änderungen an einer list-Instanz immer alle Referenzen, die auf sie verweisen. Betrachten wir einmal das folgende Beispiel, in dem der unveränderliche Datentyp str mit list verglichen wird:

```
>>> a = "Hallo "
>>> b = a
>>> b += "Welt"
>>> b
'Hallo Welt'
>>> a
'Hallo '
```

Dieses Beispiel erzeugt einfach eine str-Instanz mit dem Wert "Hallo " und lässt die beiden Referenzen a und b auf sie verweisen. Anschließend wird mit dem Operator += an den String, auf den b verweist, "Welt" angehängt. Wie die Ausgaben zeigen und wie wir es auch erwartet haben, wird eine neue Instanz mit dem Wert "Hallo Welt" erzeugt und b zugewiesen, a bleibt davon unberührt.

Übertragen wir das obige Beispiel auf Listen, ergibt sich ein wichtiger Unterschied:

```
>>> a = [1337]
>>> b = a
>>> b += [2674]
>>> b
[1337, 2674]
>>> a
[1337, 2674]
```

Strukturell gleicht der Code dem str-Beispiel, nur ist diesmal der verwendete Datentyp nicht str, sondern list. Der interessante Teil ist die Ausgabe am Ende, laut der a und b denselben Wert haben, obwohl die Operation nur auf b durchgeführt wurde. Tatsächlich verweisen a und b auf dieselbe Instanz, wovon man sich durch die Built-in Function id überzeugen kann:

```
>>> id(a) == id(b)
True
```

Diese sogenannten *Seiteneffekte* [Seiteneffekte werden im Zusammenhang mit Funktionen in Abschnitt 10.2.4 eine wichtige Rolle spielen.] sollte man bei der Arbeit mit Listen im Hinterkopf behalten. Wenn man sichergehen möchte, dass die Originalliste nicht verändert wird, kann man mithilfe von Slicing eine echte Kopie anlegen:

```
>>> a = [1337]
>>> b = a[:]
>>> b += [2674]
>>> b
[1337, 2674]
>>> a
[1337]
```

In diesem Beispiel wurde die von a referenzierte Liste kopiert und so vor indirekten Manipulationen über b geschützt. Man muss in solchen Fällen die Performance gegen den Schutz vor Seiteneffekten abwägen, da die Kopien der Listen im Speicher erzeugt werden müssen. Das kostet insbesondere bei langen Listen Rechenzeit und Speicherplatz und kann somit das Programm ausbremsen.

Im Zusammenhang mit Seiteneffekten sind auch die Elemente einer Liste interessant: Eine Liste speichert keine Instanzen an sich, sondern nur Referenzen auf sie. Das macht Listen einerseits flexibler und performanter, andererseits aber auch anfällig für Seiteneffekte. Schauen wir uns einmal das folgende, auf den ersten Blick merkwürdig anmutende Beispiel an:

```
>>> a = [[]]
>>> a = 4 * a
>>> a
[[], [], [], []]
>>> a[0].append(10)
>>> a
[[10], [10], [10], [10]]
```

Zu Beginn referenziert a eine Liste, in der eine weitere, leere Liste enthalten ist. Bei der anschließenden Multiplikation mit dem Faktor 4 wird die innere leere Liste nicht kopiert, sondern nur weitere drei Male referenziert. In der Ausgabe sehen wir also viermal dieselbe Liste. Wenn man das verstanden hat, ist es offensichtlich, warum die dem ersten Element von a angehängte 10 auch den anderen drei Listen hinzugefügt wird: Es handelt sich einfach um dieselbe Liste.

Es ist auch durchaus möglich, dass eine Liste sich selbst als Element enthält:

```
>>> a = []
>>> a.append(a)
```

Das Resultat ist eine unendlich tiefe Verschachtelung, da jede Liste wiederum sich selbst als Element enthält. Da nur Referenzen gespeichert werden müssen, verbraucht diese unendliche Verschachtelung nur sehr wenig Speicher und nicht, wie man zunächst vermuten könnte, unendlich viel. Trotzdem bergen solche Verschachtelungen die Gefahr von Endlosschleifen, wenn man die enthaltenen Daten verarbeiten möchte. Stellen Sie sich beispielsweise einmal vor, Sie wollten eine solche Liste auf dem Bildschirm ausgeben. Das würde zu unendlich vielen öffnenden und schließenden Klammern führen und somit den Computer lahmlegen. Trotzdem ist es möglich, solche Listen mit `print` auszugeben. Python überprüft selbstständig, ob eine Liste sich selbst enthält, und gibt dann anstelle von weiteren Verschachtelungen drei Punkte `...` aus:

```
>>> a = []
>>> a.append(a)
>>> a
[[...]]
```

Bitte beachten Sie, dass die Schreibweise mit den drei Punkten kein gültiger Python-Code ist, um in sich selbst verschachtelte Listen zu erzeugen.

Wenn Sie selbst mit Listen arbeiten, die rekursiv sein könnten, sollten Sie Ihre Programme mit Abfragen ausrüsten, um Verschachtelungen von Listen mit sich selbst zu erkennen, damit das Programm bei der Verarbeitung nicht in einer endlosen Schleife stecken bleiben kann.

top

8.5.2 Unveränderliche Listen – tuple ▼▲

Der Datentyp `list` ist sehr flexibel und wird häufig gebraucht. Seine Mächtigkeit und Flexibilität hat aber auch den Nachteil, dass dafür relativ viel Rechenleistung und Speicher benötigt wird. Oft wird gar nicht die Flexibilität einer Liste, sondern nur ihre Fähigkeit, Referenzen auf beliebige Instanzen zu speichern, benötigt. Deshalb existiert in Python neben `list` noch der Datentyp `tuple`, der im Gegensatz zu `list` immutable ist.

Der Datentyp `tuple` bringt keinen Mehrwert in Bezug auf Funktionalität, denn Listen können alles, was `tuple` leistet. Tatsächlich steht für `tuple`-Instanzen nur der Grundstock an Operationen für sequenzielle Datentypen bereit.

Zum Erzeugen neuer `tuple`-Instanzen dienen die runden Klammern, die – wie bei den Listen – durch Kommata getrennt die Elemente des Tupels enthalten:

```
>>> a = (1, 2, 3, 4, 5)
>>> a[3]
4
```

Ein leeres Tupel wird durch zwei runde Klammern `()` ohne Inhalt definiert. Eine Besonderheit ergibt sich für Tupel mit nur einem Element. Würde man versuchen, ein Tupel mit nur einem Element auf die oben beschriebene Weise zu erzeugen, wäre das Literal unter Umständen nicht eindeutig:

```
>>> kein_tuple = (1)
>>> type(kein_tuple)
<type 'int'>
```

Mit `(1)` wird keine neue `tuple`-Instanz erzeugt, weil die Klammer in diesem Kontext schon für die Verwendung in Rechenoperationen für Ganzzahlen verwendet wird. Das Problem wird dadurch umgangen, dass in Literalen für Tupel mit nur einem Element diesem Element ein Komma nachgestellt werden muss:

```
>>> ein_tuple = (1,)
>>> type(ein_tuple)
<type 'tuple'>
```

tuple packing und tuple unpacking

Es ist möglich, die umschließenden Klammern bei einer tuple-Definition entfallen zu lassen. Trotzdem werden die durch Kommata getrennten Referenzen zu einem tuple zusammengefasst, was man *tuple packing* nennt:

```
>>> datum = 26, 7, 1987
>>> datum
(26, 7, 1987)
```

Umgekehrt ist es möglich, die Werte eines Tupels wieder zu entpacken:

```
>>> datum = 26, 7, 1987
>>> (tag, monat, jahr) = datum
>>> tag
26
>>> monat
7
>>> jahr
1987
```

Dieses Verfahren heißt *tuple unpacking*, und auch hier können die umschließenden Klammern entfallen. Durch Kombination von *tuple packing* und *tuple unpacking* ist es sehr elegant möglich, die Werte zweier Variablen ohne Hilfsvariable zu tauschen oder mehrere Zuweisungen in einer Zeile zusammenzufassen:

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Richtig angewandt kann die Nutzung dieses Features zur Lesbarkeit von Programmen beitragen, da das technische Detail der Zwischenspeicherung von Daten hinter die eigentliche Absicht, die Werte zu tauschen, zurücktritt.

Immutable heißt nicht zwingend unveränderlich!

Auch wenn tuple-Instanzen immutable sind, können sich die Werte der enthaltenen Elemente auch nach der Erzeugung ändern. Bei der Erzeugung eines neuen Tupels werden die Referenzen festgelegt, die es speichern soll. Verweist eine solche Referenz auf eine Instanz eines mutable Datentyps, beispielsweise eine Liste, so kann sich dessen Wert trotzdem ändern:

```
>>> a = ([],)
>>> a[0].append("Und sie dreht sich doch!")
>>> a
(['Und sie dreht sich doch!'],)
```

Die Unveränderlichkeit eines Tupels bezieht sich also nur auf die enthaltenen Referenzen und ausdrücklich nicht auf die dahinter stehenden Instanzen.

Dass Tupel immutable sind, ist also keine Garantie dafür, dass sich die Elemente nach der Erzeugung des Tupels nicht mehr verändern.

top

8.5.3 Strings – str, unicode ▲

Dieser Abschnitt behandelt Pythons Umgang mit Zeichenketten und insbesondere die Eigenschaften der dafür bereitgestellten Datentypen str und unicode.

Wie Sie im vorhergehenden Kapitel gelernt haben, handelt es sich bei Strings um Folgen von *Zeichen*. Dies bedeutet, dass alle Operationen für sequenzielle Typen für sie verfügbar sind.

Wir werden uns bis auf Weiteres nur mit str-Instanzen beschäftigen, weil diese anfangs einfacher zu verstehen sind. Aus dem gleichen Grund wird vorerst auf Sonderzeichen wie Umlaute oder das »ß« innerhalb von Strings verzichtet.

Um neue str-Instanzen zu erzeugen, gibt es folgende Literale:

```
>>> string1 = "Ich wurde mit doppelten Hochkommata definiert"
>>> string2 = 'Ich wurde mit einfachen Hochkommata definiert'
```

Der gewünschte Inhalt des Strings wird zwischen die Hochkommata geschrieben, darf allerdings keine Zeilenvorschübe enthalten (im folgenden Beispiel wurde am Ende der ersten Zeile ↵ gedrückt):

```
>>> s = "Erste Zeile
  File "<stdin>", line 1
```

```
    s = "Erste Zeile
                    ^
SyntaxError: EOL while scanning single-quoted string
```

String-Konstanten, die sich auch über mehrere Zeilen erstrecken können, werden durch """ bzw. ''' eingefasst:

```
>>> string3 = """Erste Zeile!
Ui, noch eine Zeile"""
```

Stehen zwei String-Literale unmittelbar oder durch Leerzeichen getrennt hintereinander, werden sie von Python miteinander zu einem String verbunden:

```
>>> string = "Erster Teil" "Zweiter Teil"
>>> string
Erster TeilZweiter Teil
```

Wie Sie im Beispiel sehen können, sind die Leerzeichen zwischen den Literalen bei der Verkettung nicht mehr vorhanden.

Diese Art der Verkettung eignet sich sehr gut, um lange oder unübersichtliche Strings auf mehrere Programmzeilen aufzuteilen, ohne dass die Zeilenvorschübe und Leerzeichen im Resultat gespeichert werden, wie es bei Strings mit """ oder ''' der Fall wäre. Um diese Aufteilung zu erreichen, schreibt man die String-Teile in runde Klammern:

```
>>> a = ("Stellen Sie sich einen schrecklich "
...      "komplizierten String vor, den man "
...      "auf keinen Fall in eine Zeile schreiben "
...      "kann.")
```

Steuerzeichen

Es gibt besondere Text-Elemente, die den Textfluss steuern und sich auf dem Bildschirm nicht als einzelne Zeichen darstellen lassen. Zu diesen sogenannten *Steuerzeichen* zählen unter anderem der Zeilenvorschub, der Tabulator oder der Rückschritt (von engl. *backspace*). Die Darstellung solcher Zeichen innerhalb von String-Literalen erfolgt mittels spezieller Zeichenfolgen, sogenannter *Escape-Sequenzen*. Escape-Sequenzen werden von einem Backslash \ eingeleitet, der von der Kennung des gewünschten Sonderzeichens gefolgt wird. Die Escape-Sequenz "\n" steht beispielsweise für einen Zeilenumbruch:

```
>>> a = "Erste Zeile\nZweite Zeile"
>>> a
'Erste Zeile\nZweite Zeile'
>>> print a
Erste Zeile
Zweite Zeile
```

Beachten Sie bitte den Unterschied zwischen der Ausgabe mit `print` und der ohne `print` im interaktiven Modus. Die `print`-Anweisung setzt die Steuerzeichen in ihre Bildschirmdarstellung um (bei "\n" wird zum Beispiel eine neue Zeile begonnen), wohingegen die Ausgabe ohne `print` ein String-Literal mit den Escape-Sequenzen der Sonderzeichen auf dem Bildschirm anzeigt.

Für Steuerzeichen gibt es in Python die folgenden Escape-Sequenzen:

Escape-Sequenz	Bedeutung
\a	Bell (BEL) erzeugte einen Signalton.
\b	Backspace (BS) setzt die Ausgabeposition um ein Zeichen zurück.
\f	Formfeed (FF) erzeugt einen Seitenvorschub.
\n	Linefeed (LF) setzt die Ausgabeposition in die nächste Zeile.
\r	Carriage Return (CR) setzt die Ausgabeposition an den Anfang der nächsten Zeile.
\t	Horizontal Tab (TAB) hat die gleiche Bedeutung wie die Tabulatortaste.
\v	Vertikaler Tabulator (VT); dient zur vertikalen Einrückung.
\"	Doppeltes Hochkomma
\'	Einfaches Hochkomma
\\	Backslash, der wirklich als solcher in dem String erscheinen soll

Tabelle 8.19 Escape-Sequenzen für Steuerzeichen

Es ist allerdings so, dass Steuerzeichen aus der Zeit stammen, als die Ausgaben hauptsächlich über Drucker erfolgten. Deshalb haben einige dieser Zeichen heute nur noch eine geringe praktische Bedeutung.

Die Escape-Sequenzen für einfache und doppelte Hochkommata sind deshalb notwendig, weil Python diese Zeichen als Begrenzung für String-Literale verwendet. Soll die Art von Hochkomma, die für die Begrenzung eines Strings verwendet wurde, innerhalb dieses Strings als Zeichen vorkommen, muss dort das entsprechende Hochkomma als Escape-Sequenz angegeben werden:

```
>>> a = "Das folgende Hochkomma muss nicht kodiert werden ' "
>>> b = "Dieses doppelte Hochkomma schon \" "
>>> c = 'Das gilt auch in Strings mit einfachen Hochkommata " '
>>> d = 'Hier muss eine Escape-Sequenz benutzt werden \' '
```

Im Abschnitt »Zeichensätze und Sonderzeichen« werden wir auf Escape-Sequenzen zurückkommen, um damit beliebige Sonderzeichen wie Umlaute oder das €–Zeichen zu kodieren.

Das automatische Ersetzen von Escape-Sequenzen ist manchmal lästig, insbesondere dann, wenn sehr viele Backslashs in einem String vorkommen sollen. Für diesen Zweck gibt es in Python die Präfixe `r` oder `R`, die einem String-Literal vorangestellt werden können. Diese Präfixe markieren das Literal als einen sogenannten *Raw-String* (dt. *roh*), was dazu führt, dass alle Backslashs eins zu eins in den Resultat-String übernommen werden:

```
>>> "Ein \tString mit \\ vielen \nEscape-Sequenzen\t"
'Ein \tString mit \\ vielen \nEscape-Sequenzen\t'
>>> r"Ein \tString mit \\ vielen \nEscape-Sequenzen\t"
'Ein \\tString mit \\\\ vielen \\nEscape-Sequenzen\\t'
>>> print r"Ein \tString mit \\ vielen \nEscape-Sequenzen\t"
Ein \tString mit \\ vielen \nEscape-Sequenzen\t
```

Wie Sie an den doppelten Backslashs im Literal des Resultats und der Ausgabe mit `print` sehen können, wurden die Escape-Sequenzen nicht interpretiert.

Stringmethoden

String-Instanzen verfügen zusätzlich zu den Methoden für sequenzielle Datentypen über weitere Methoden, die den Umgang mit Zeichenketten vereinfachen. Aufgrund der großen Anzahl der String-Methoden gibt es anstatt der zusammenfassenden Tabelle aller Methoden mehrere Kategorien, die einzeln erklärt werden.

Wenn im Folgenden von *Whitespaces* gesprochen wird, sind alle Arten von Zeichen zwischen den Wörtern gemeint, die nicht als eigenes Zeichen angezeigt werden. Whitespaces sind folgende Zeichen: das Leerzeichen, der Zeilenvorschub, der vertikale und horizontale Tabulator, Linefeed, Formfeed und Carriage Return.

Trennen von Strings

Um Strings nach bestimmten Regeln in mehrere Teile zu zerlegen, dienen folgende Methoden:

- *s.split([sep[, maxsplit]])*
- *s.rsplit([sep[, maxsplit]])*
- *s.splitlines([keepends])*
- *s.partition(sep)*
- *s.rpartition(sep)*

Die Methoden `split` und `rsplit` zerteilen einen String in seine Wörter und geben diese als Liste zurück. Dabei gibt der Parameter *sep* die Zeichenfolge an, die die Wörter trennt. Mit *maxsplit* kann die Anzahl der Trennungen begrenzt werden. Wird *maxsplit* nicht angegeben, wird der String so oft zerteilt, wie *sep* in ihm vorkommt. Ein gegebenenfalls verbleibender Rest wird als String in die resultierende Liste eingefügt. `split` beginnt mit dem Teilen am Anfang des Strings, während `rsplit` am Ende anfängt:

```
>>> s = "1-2-3-4-5-6-7-8-9-10"
>>> s.split("-")
['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
>>> s.split("-", 5)
['1', '2', '3', '4', '5', '6-7-8-9-10']
>>> s.rsplit("-", 5)
 ['1-2-3-4-5', '6', '7', '8', '9', '10']
```

Folgen mehrere Trennzeichen aufeinander, werden sie nicht zusammengefasst, sondern es wird jedes Mal erneut getrennt:

```
>>> s = "1---2-3"
>>> s.split("-")
['1', '', '', '2', '3']
```

Wird *sep* nicht angegeben, verhalten sich die beiden Methoden anders. Zuerst werden alle Whitespaces am Anfang und am Ende des Strings entfernt, und anschließend wird der String anhand von Whitespaces zerteilt, wobei dieses Mal aufeinanderfolgende Trennzeichen zu einem zusammengefasst werden:

```
>>> s = "  Irgendein \t\t Satz  mit \n\r\t Whitespaces"
>>> s.split()
['Irgendein', 'Satz', 'mit', 'Whitespaces']
```

Der Aufruf von `split` ganz ohne Parameter ist sehr nützlich, um einen Text-String in seine Wörter zu spalten, auch wenn diese nicht nur durch Leerzeichen voneinander getrennt sind.

Die Methode `splitlines` spaltet einen String in seine einzelnen Zeilen auf und gibt eine Liste zurück, die die Zeilen enthält. Dabei werden Unix-Zeilenvorschübe `"\n"`, Windows-Zeilenvorschübe `"\r\n"` und Mac-Zeilenvorschübe `"\r"` als Trennzeichen benutzt:

```
>>> s = "Unix\nWindows\r\nMac\rLetzte Zeile"
>>> s.splitlines()
['Unix', 'Windows', 'Mac', 'Letzte Zeile']
```

Sollen die trennenden Zeilenvorschübe an den Enden der Zeilen erhalten bleiben, muss für den optionalen Parameter *keepends* der Wert `True` übergeben werden.

Die Methode `partition` zerteilt einen String an der ersten Stelle, an der der übergebene Trennstring *sep* auftritt, und gibt ein Tupel zurück, das aus dem Teil vor dem Trennstring, dem Trennstring selbst und dem Teil danach besteht. Die Methode `rpartition` arbeitet genauso, nimmt aber das letzte Vorkommen von *sep* im Ursprungsstring als Trennstelle:

```
>>> s = "www.galileo-computing.de"
>>> s.partition(".")
('www', '.', 'galileo-computing.de')
>>> s.rpartition(".")
('www.galileo-computing', '.', 'de')
```

Suchen von Teilstrings

Um die Position und die Anzahl der Vorkommen eines Strings in einem anderen String zu ermitteln oder Teile eines Strings zu ersetzen, dienen folgende Methoden:

- *s.find(sub[, start[, end]])*
- *s.rfind(sub[, start[, end]])*
- *s.index(sub[, start[, end]])*
- *s.rindex(sub[, start[, end]])*
- *s.count(sub[, start[, end]])*

Die optionalen Parameter *start* und *end* der fünf Methoden dienen dazu, den Suchbereich einzugrenzen. Werden *start* bzw. *end* angegeben, wird nur der Teilstring `s[`*start*`:`*end*`]` betrachtet.

Hinweis

Zur Erinnerung: Beim Slicing eines Strings `s` mit `s[start:end]` wird ein Teilstring erzeugt, der das Element `s[end]` nicht mehr enthält.

Um herauszufinden, ob und, wenn ja, wo ein bestimmter String in einem anderen vorkommt, bietet Python die Methoden `find` und `index` mit ihren Gegenstücken `rfind` und `rindex` an. `find` gibt den Index des ersten Vorkommens von *sub* in `s` zurück, `rfind` entsprechend den Index des letzten Vorkommens. Ist *sub* nicht in `s` enthalten, geben `find` und `rfind` den Wert `-1` zurück:

```
>>> s = "Mal sehen, wo das 'e' in diesem String vorkommt"
>>> s.find("e")
5
>>> s.rfind("e")
29
```

Die Methoden index und rindex arbeiten auf die gleiche Weise, erzeugen aber einen ValueError, wenn *sub* nicht in *s* enthalten ist:

```
>>> s = "Dieser String wird gleich durchsucht"
>>> s.index("wird")
14
>>> s.index("nicht vorhanden")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    s.index("nicht vorhanden")
ValueError: substring not found
```

Der Grund für diese fast identischen Methoden liegt darin, dass sich Fehlermeldungen unter Umständen eleganter handhaben lassen als ungültige Rückgabewerte. [Sie können die Details in Abschnitt 13.1, »Exception Handling«, nachlesen.]

Wie oft ein Teilstring in einem anderen enthalten ist, lässt sich mit count ermitteln:

```
>>> "Fischers Fritze fischt frische Fische".count("sch")
4
```

Ersetzen von Teilstrings

Mit den folgenden Methoden lassen sich bestimmte Teile oder Buchstaben von Strings durch andere ersetzen:

- *s.replace(old, new[, count])*
- *s.lower()*
- *s.upper()*
- *s.swapcase()*
- *s.capitalize()*
- *s.title()*
- *s.expandtabs([tabsize])*

Die Methode replace gibt einen String zurück, in dem alle Vorkommen von *old* durch *new* ersetzt wurden:

```
>>> falsch = "Python ist nicht super!"
>>> richtig = falsch.replace("nicht", "richtig")
>>> richtig
'Python ist richtig super!'
```

Mit dem Parameter *count* kann die Anzahl der Ersetzungen begrenzt werden:

```
>>> s = "Bitte nur die ersten vier e ersetzen"
>>> s.replace("e", "E", 4)
'BittE nur diE ErstEn vier e ersetzen'
```

Die Methode lower ersetzt alle Kleinbuchstaben eines Strings durch die entsprechenden Großbuchstaben und gibt den Ergebnis-String zurück:

```
>>> s = "ERST GANZ GROSS UND DANN GANZ KLEIN!"
>>> s.lower()
'erst ganz gross und dann ganz klein!'
```

Mit upper erreicht man genau den umgekehrten Effekt.

Die Methode swapcase ändert die Groß- bzw. Kleinschreibung aller Buchstaben eines Strings, indem alle Großbuchstaben durch die entsprechenden Kleinbuchstaben und umgekehrt ersetzt werden:

```
>>> s = "wENN MAN IM dEUTSCHEN ALLE wORTE SO SCHRIEBE..."
>>> s.swapcase()
'Wenn man im Deutschen alle Worte so schriebe...'
```

Die Methode capitalize gibt eine Kopie des Ursprungsstrings zurück, wobei das erste Zeichen – sofern möglich – in einen Großbuchstaben umgewandelt wurde:

```
>>> s = "alles klein... noch ;)"
>>> s.capitalize()
'Alles klein... noch ;)'
```

Mit der Methode `title` wird ein String erzeugt, bei dem alle Wörter groß-, aber ihre restlichen Buchstaben kleingeschrieben sind, wie dies bei Titeln üblich ist:

```
>>> s = "nOch BIn iCH eheR weNiGEr alS TITeL gEeiGNEt"
>>> s.title()
'Noch Bin Ich Eher Weniger Als Titel Geeignet'
```

Mit `expandtabs` kann man alle Tabulator-Zeichen (`"\t"`) eines Strings durch Leerzeichen ersetzen lassen. Der optionale Parameter `tabsize` gibt dabei an, wie viele Leerzeichen für einen Tabulator eingefügt werden sollen. Ist `tabsize` nicht angegeben, werden acht Leerzeichen verwendet:

```
>>> s = "\tHier kann Quellcode stehen\n\t\tEine Ebene weiter unten"""
>>> print s.expandtabs(4)
    Hier kann Quellcode stehen
        Eine Ebene weiter unten
```

Entfernen bestimmter Zeichen am Anfang oder am Ende von Strings

Die `strip`-Methoden ermöglichen es, unerwünschte Zeichen am Anfang oder am Ende eines Strings zu entfernen:

- *s.strip([chars])*
- *s.lstrip([chars])*
- *s.rstrip([chars])*

Die Methode `strip` entfernt unerwünschte Zeichen auf beiden Seiten des Strings. `lstrip` entfernt nur die Zeichen auf der linken Seite und `rstrip` nur die Zeichen auf der rechten.

Für den optionalen Parameter *chars* kann ein String übergeben werden, der die Zeichen enthält, die entfernt werden sollen. Wird *chars* nicht angegeben, werden alle Whitespaces entfernt:

```
>>> s = "    \t\n  \rUmgeben von Whitespaces   \t\t\r"
>>> s.strip()
'Umgeben von Whitespaces'
>>> s.lstrip()
'Umgeben von Whitespaces   \t\t\r'
>>> s.rstrip()
'    \t\n  \rUmgeben von Whitespaces'
```

Um beispielsweise alle umgebenden Ziffern zu entfernen, könnte man so vorgehen:

```
>>> ziffern = "0123456789"
>>> s = "3674784673546Versteckt zwischen Zahlen3425923935"
>>> s.strip(ziffern)
'Versteckt zwischen Zahlen'
```

Ausrichten von Strings

Die folgenden Methoden erzeugen einen String mit einer bestimmten Länge und richten den Ursprungsstring darin auf eine bestimmte Weise aus:

- *s.center(width[, fillchar])*
- *s.ljust(width[, fillchar])*
- *s.rjust(width[, fillchar])*
- *s.zfill(width)*

Mit dem Parameter *width* wird die gewünschte Länge des neuen Strings angegeben. Ist die Länge von *s* größer als *width*, wird eine Kopie von *s* zurückgegeben. Die Methode `center` zentriert *s* im neuen String, `ljust` richtet *s* links aus, `rjust` richtet *s* rechts aus. Der optionale Parameter *fillchar* der drei ersten Methoden muss ein String der Länge eins sein und gibt das Zeichen an, das zum Auffüllen bis zur übergebenen Länge verwendet werden soll. Standardmäßig werden Leerzeichen zum Füllen benutzt:

```
>>> s = "Richte mich aus"
>>> s.center(50)
'                  Richte mich aus                 '
>>> s.ljust(50)
'Richte mich aus                                   '
>>> s.rjust(50, "-")
'----------------------------------Richte mich aus'
```

Die Methode `zfill` ist ein Spezialfall von `rjust` und für Strings gedacht, die numerische Werte enthalten. `zfill` erzeugt einen String der Länge *width*, in

dem der Ursprungsstring rechts ausgerichtet ist und die linke Seite mit Nullen aufgefüllt wurde:

```
>>> "13.37".zfill(20)
'00000000000000013.37'
```

String-Tests

Die folgenden Methoden geben einen Wahrheitswert zurück, der aussagt, ob der Inhalt des Strings eine bestimmte Eigenschaft hat. Mit `islower` beispielsweise kann man prüfen, ob alle Buchstaben in *s* Kleinbuchstaben sind.

Methode	Beschreibung
`s.isalnum()`	True, wenn alle Zeichen in *s* Buchstaben oder Ziffern sind
`s.isalpha()`	True, wenn alle Zeichen in *s* Buchstaben sind
`s.isdigit()`	True, wenn alle Zeichen in *s* Ziffern sind
`s.islower()`	True, wenn alle Buchstaben in *s* Kleinbuchstaben sind
`s.isupper()`	True, wenn alle Buchstaben in *s* Großbuchstaben sind
`s.isspace()`	True, wenn alle Zeichen in *s* Whitespaces sind
`s.istitle()`	True, wenn alle Wörter in *s* großgeschrieben sind

Tabelle 8.20 Methoden für einfache String-Tests

Da sich diese Methoden alle sehr ähneln, soll ein Beispiel an dieser Stelle ausreichen:

```
>>> s = "1234abcd"
>>> s.isdigit()
False
>>> s.isalpha()
False
>>> s.isalnum()
True
```

Um zu prüfen, ob ein String mit einer bestimmten Zeichenkette beginnt oder endet, dienen die Methoden `startswith` bzw. `endswidth`:

- *s.startswidth(prefix[, start[, end]])*
- *s.endswidth(suffix[, start[, end]])*

Die optionalen Parameter *start* und *end* können dabei – wie schon bei den Suchen- und Ersetzen-Methoden – die Abfrage auf einen bestimmten Bereich von *s* begrenzen:

```
>>> s = "www.galileo-computing.de"
>>> s.startswith("www.")
True
>>> s.endswith(".de")
True
>>> s.startswith("galileo", 4)
True
```

Verkettung von Elementen in sequenziellen Datentypen

Eine häufige Aufgabe ist es, eine Liste von Strings mit einem Trennzeichen zu verketten. Beispielsweise könnte man die Namen in einer Kontaktliste seines Instant-Messengers durch Kommata getrennt ausgeben wollen. Für diesen Zweck stellt Python die Methode `join` zur Verfügung:

- *s.join(seq)*

Der Parameter *seq* ist dabei ein beliebiges iterierbares Objekt, dessen Elemente alle Strings sein müssen. Die Elemente von *seq* werden mit *s* als Trennzeichen verkettet. Kommen wir auf unser Kontaktlistenbeispiel zurück:

```
>>> kontakt_liste = ["Fix", "Foxy", "Lupo", "Dr. Knox"]
>>> ", ".join(kontakt_liste)
'Fix, Foxy, Lupo, Dr. Knox'
```

Wird für *seq* ein String übergeben, so ist das Ergebnis die Verkettung aller Buchstaben, jeweils durch *s* voneinander getrennt:

```
>>> satz = "Stoiber-Satz"
>>> "...ehm...".join(satz)
'S...ehm...t...ehm...o...ehm...i...ehm...b...ehm...e...ehm...r...ehm...-
...ehm...S...ehm...a...ehm...t...ehm...z'
```

Formatierung

Oft möchte man seine Bildschirmausgaben auf bestimmte Weise anpassen. Um beispielsweise eine dreispaltige Tabelle von Zahlen anzuzeigen, müssen abhängig von der Länge der Zahlen Leerzeichen eingefügt werden, damit die einzelnen Spalten untereinander angezeigt werden. Eine Anpassung der Ausgabe ist auch nötig, wenn man einen Geldbetrag ausgeben möchte, der in einer `float`-Instanz gespeichert ist, die mehr als zwei Nachkommastellen besitzt.

Für die Lösung solcher Probleme gibt es in Python den *Formatierungsoperator* für Strings, das Prozentzeichen %. Er erwartet zwei Operanden: als ersten Operanden einen String, der die Formatierung beschreibt, und als zweiten eine Sequenz, in der die Werte gespeichert sind, die man formatiert ausgeben möchte. Die Formatierungsbeschreibungen in dem String werden durch ein Prozentzeichen eingeleitet, dem im einfachsten Fall ein einzelner Buchstabe folgt. Dieser Buchstabe gibt an, welche Art von Daten man an dieser Stelle einfügen möchte. Ein `d` steht beispielsweise für eine Ganzzahl:

```
>>> "Es ist %d:%d Uhr" % (13, 37)
'Es ist 13:07 Uhr'
```

Es ist auch möglich, einen einzelnen Wert anstatt der Sequenz zu übergeben:

```
>>> x = 10
>>> "Der Wert von x ist %d" % x
'Der Wert von x ist 10'
```

Allerdings muss in jedem Fall die Anzahl der Formatierungsanweisungen mit der Anzahl der Werte übereinstimmen. Werden mehr oder weniger Werte als Formatierungsanweisungen übergeben, führt dies zu einem Fehler:

```
>>> "Nur %d Formatierungsanweisungen" % (1, 1337)
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    "Nur %d Formatierungsanweisungen" % (1, 1337)
TypeError: not all arguments converted during string formatting
```

Das Ergebnis der Formatierung ist ein neuer String, in dem alle Formatierungsanweisungen in dem Ursprungsstring durch die entsprechenden Werte der Sequenz ersetzt wurden.

Es existieren die folgenden Ausgabedatentypen:

Kennung	Beschreibung
d	Ganzzahl mit Vorzeichen
i	Ganzzahl mit Vorzeichen
o	Oktalzahl ohne Vorzeichen
u	Ganzzahl ohne Vorzeichen
x	Hexadezimalzahl ohne Vorzeichen in Kleinbuchstaben
X	Hexadezimalzahl ohne Vorzeichen in Großbuchstaben
e	Fließkommazahl im wissenschaftlichen Format (kleines »e«)
E	Fließkommazahl im wissenschaftlichen Format (großes »E«)
f	Fließkommazahl in Dezimalschreibweise
F	Wie `f`
g	Fließkommazahl in wissenschaftlicher Schreibweise, wenn der Exponent kleiner als –4 ist, sonst in Dezimalschreibweise
G	Wie `g`
c	Zeichen (kann Strings der Länge eins und Ganzzahlen mit ASCII-Codes umwandeln)
r	String (macht aus jeder Instanz einen String mit der Builtin-Function `repr`)
s	String (macht aus jeder Instanz einen String mit der Builtin-Function `str`)
%	Gibt ein Prozentzeichen aus

Tabelle 8.21 Ausgabedatentypen für String-Formatierungen

Zwischen dem Prozentzeichen und dem Ausgabedatentyp können weitere Angaben gemacht werden, um die Ausgabe zu steuern. Betrachten wir ein Beispiel mit allen möglichen Angaben:

```
>>> "%+10.2f" % 123.45678
'   +123.46'
```

Das Pluszeichen + am Anfang ist ein sogenanntes *Umwandlungsflag* und sorgt dafür, dass bei Zahlen das Vorzeichen immer mit ausgegeben wird. Mit der Zahl 10 nach dem Pluszeichen wird angegeben, wie viele Zeichen das Resultat auf jeden Fall haben soll. Wenn die angegebene *Mindestlänge* unterschritten wird, füllt Python von links so lange mit Leerzeichen auf, bis der String lang genug ist. Die letzte Angabe, die aus einem Punkt und einer Zahl .2 besteht, bestimmt die sogenannte *Genauigkeit* der Ausgabe. Ist keine Genauigkeit angegeben, wird für numerische Werte der Standardwert 6 und für nichtnumerische Werte eine unendlich große Zahl verwendet. Bei Fließkommazahlen bestimmt die Genauigkeit die Anzahl der angezeigten Nachkommastellen, bei Zahlen in Exponentialschreibweise die Anzahl geltender Ziffern und bei Strings die maximale Länge der Formatierung.

Python stellt fünf Umwandlungsflags zur Verfügung:

Flag	Bedeutung
#	Setzt die Ausgabe in den alternativen Modus (wird im Folgenden erklärt).
0	Wenn die Mindestlänge bei numerischen Werten unterschritten wird, füllt Python mit der 0 anstatt mit Leerzeichen von links auf.
-	Wird die Mindestlänge unterschritten, wird anstatt von links von rechts aufgefüllt.
(ein Leerzeichen)	Vor einem numerischen Wert mit positivem Vorzeichen wird ein Leerzeichen eingefügt.
+	Vor numerischen Werten wird das Vorzeichen ausgegeben.

Tabelle 8.22 Umwandlungsflags für String-Formatierungen

Es ist möglich, mehrere Umwandlungsflags hintereinander anzugeben:

```
>>> "%0+10.2f" % 123.45678
'+000123.46'
```

Widersprechen sich die angegebenen Umwandlungsflags, wird die Angabe verwendet, die in der obigen Tabelle weiter unten steht: Wenn 0 und das Minuszeichen angegeben sind, wird die 0 ignoriert. Sind das Leerzeichen und + angegeben, wird das Leerzeichen nicht beachtet.

Der durch das Umwandlungsflag # gesetzte *alternative Modus* verändert die Ausgabe einiger Ausgabedatentypen wie folgt:

Kennung	Beschreibung
o	Vor der Oktalzahl wird eine Null (0) eingefügt, wenn dort noch keine stand.
x	Vor der Hexadezimalzahl wird die Zeichenfolge 0x ausgegeben.
X	Wie bei x, aber statt 0x mit 0X
e	Es wird immer ein Dezimalpunkt ausgegeben, auch wenn keine Nachkommastellen folgen.
E	Wie bei e
f	Wie bei e
F	Wie bei e
g	Wie bei e; zusätzlich werden Nullen am Ende nicht entfernt, wie es sonst der Fall wäre.
G	Wie bei g

Tabelle 8.23 Alternativer Modus der String-Formatierung

Die Wirkungsweise des alternativen Modus wird durch die folgenden Beispiele noch einmal verdeutlicht:

```
>>> "%o vs. %#o" % (26, 26)
'32 vs. 032'
>>> "%x vs. %#x" % (26, 26)
'1a vs. 0x1a'
>>> "%5.0e vs. %#5.0e" % (123.4567, 123.4567)
'1e+002 vs. 1.e+002'
>>> "%g vs. %#g" % (1337, 1337)
'1337 vs. 1337.00'
```

Alternative Angabe von Werten

Insbesondere wenn in einem String sehr viele Werte formatiert werden sollen, ist die oben beschriebene Methode, bei der die Werte in einer Sequenz angegeben werden, sehr unübersichtlich. Python bietet deshalb eine Möglichkeit an, die Formatierungsanweisungen mit Namen zu versehen:

```
>>> s = "%(vorname)s %(nachname)s (%(alter)d) sucht..."
>>> s % {"vorname" : "Heinz", "nachname" : "Meier", "alter" : 30}
'Heinz Meier (30) sucht...'
```

Die Namen werden in dem Formatstring direkt hinter dem Prozentzeichen in runden Klammern angegeben.

Dem Formatierungsoperator muss dann anstelle der Wertesequenz eine Zuordnung von Namen und Werten folgen. Diese Zuordnung wird von geschweiften Klammern begrenzt, die eine durch Kommata getrennte Liste von Name-Wert-Paaren umschließen. In den Paaren werden die Namen durch Doppelpunkte von den dazugehörigen Werten getrennt. [Bei diesen Zuordnungen handelt es sich um den Python-Datentyp *Dictionary*, den wir ausführlich in Abschnitt 8.6.1 behandeln.]

Als dritte und letzte Möglichkeit der Werteübergabe ist es möglich, auch die Mindestlänge von dem Format-String in die Wertesequenz zu verlagern. Um dies zu erreichen, wird in den Formatstring anstelle der Mindestlänge ein Sternchen * geschrieben. In der Wertesequenz werden dann zwei Werte für diese Formatierung benutzt: Zuerst wird die Länge und dann der eigentliche Wert gelesen:

```
>>> "Freie Platzwahl: %*d" % (5, 123)
'Freie Platzwahl:   123'
```

Im obigen Beispiel wurde die Zahl 123 auf eine Breite von 5 Stellen formatiert. Man kann die direkte Breitenangabe innerhalb des Format-Strings mit der Angabe innerhalb der Sequenz kombinieren, wie das folgende Beispiel demonstriert:

```
>>> "Im Format-String: %6d. In der Sequenz: %*d" % (11, 3, 7)
'Im Format-String:     11. In der Sequenz:   7'
```

Hier wurde die Zahl 11 direkt über den Formatstring auf eine Mindestlänge von sechs Zeichen formatiert, während die 7 durch die in dem Tupel angegebene Zahl 3 formatiert wurde.

Zeichensätze und Sonderzeichen

Bisher haben wir uns der Einfachheit halber nur mit Strings beschäftigt, die keine Sonderzeichen (wie Umlaute oder das €-Zeichen) beinhalten. Die Besonderheiten beim Umgang mit solchen Zeichen liegen zum Teil an der geschichtlichen Entwicklung der Zeichenkodierung. Deshalb werden wir diese im Folgenden kurz umreißen.

Zuerst müssen wir eine Vorstellung davon entwickeln, wie ein Computer intern mit Zeichenketten umgeht. Generell lässt sich sagen, dass der Computer eigentlich überhaupt keine Zeichen kennt, da sich in seinem Speicher nur Zahlen befinden. Um trotzdem Bildschirmausgaben zu produzieren oder andere Operationen mit Zeichen durchzuführen, hat man deshalb Übersetzungstabellen, die sogenannten *Codepages* (dt. *Zeichensatztabellen*) definiert, die jedem Buchstaben eine bestimmte Zahl zuordnen. Der bekannteste und wichtigste Zeichensatz ist durch die *ASCII-Tabelle* [American Standard Code for Information Interchange (dt. *Amerikanische Standardkodierung für den Informationsaustausch*)] festgelegt.

Durch diese Zuordnung werden neben den Buchstaben und Ziffern auch Satz- und einige Sonderzeichen abgebildet. Außerdem existieren nicht druckbare Steuerzeichen, wie der Tabulator oder der Zeilenvorschub. Die ASCII-Tabelle ist eine 7-Bit-Zeichenkodierung, das bedeutet, dass von jedem Buchstaben 7Bit Speicherplatz belegt werden. Es können also 27 = 128 verschiedene Zeichen abgebildet werden. Die Definition des ASCII-Zeichensatzes orientiert sich am Alphabet der englischen Sprache, das insbesondere keine Umlaute wie »ä« oder »ü« enthält. Um auch solche Sonderzeichen in Strings abspeichern zu können, erweiterte man den ASCII-Code, indem man den Speicherplatz für ein Zeichen um ein Bit auf 28 = 256 Möglichkeiten erhöhte, was 128 Plätze für weitere Sonderzeichen bot. Welche Interpretation konkret für diese weiteren Plätze verwendet wird, hängt von der verwendeten Codepage ab und unterscheidet sich in der Regel zwischen verschiedenen Plattformen.

Pythons `str`-Datentyp implementiert einen solchen 8-Bit-String und ist im Prinzip nichts anderes als eine Kette von Bytes. Um den Zahlenwert eines

Zeichens zu ermitteln, gibt es in Python die Built-in-Function ord, die als einzigen Parameter einen String der Länge eins erwartet:

```
>>> ord("j")
106
>>> ord("[")
91
```

Umgekehrt liefert die Built-in-Function chr das zu einem Byte gehörige Zeichen:

```
>>> chr(106)
'j'

>>> chr(91)
'['
```

Die Beispiele oben beziehen sich nur auf Zeichen mit Ordnungszahlen, die kleiner als 128 sind und damit noch im ASCII-Bereich liegen. Interessanter ist das folgende Beispiel:

```
>>> ord("ä")
228
```

Auf dem Computer, der dieses Beispiel ausgeführt hat, läuft eine Version von Microsoft Windows für Westeuropa, die standardmäßig eine Codepage mit dem Namen »Windows-1252« verwendet. »Windows-1252« bildet alle wichtigen Zeichen für Westeuropa, das Eurozeichen inbegriffen, ab. Wenn Sie das Beispiel ausführen und eine andere Zahl als 228 auf dem Bildschirm sehen, liegt das einfach daran, dass Ihr Computer eine andere Codepage als »Windows-1252« verwendet.

Wir haben uns bereits während der Einführung zu Strings mit Escape-Sequenzen beschäftigt. In Bezug auf Sonderzeichen spielen sie eine zentrale Rolle:

```
>>> "Überprüfung der Änderungen"
'\xdcberpr\xfcfung der \xc4nderungen'
```

Was auf den ersten Blick kryptisch erscheint, hat eine einfache Struktur: Wie Sie bereits wissen, wird durch den Backslash \ innerhalb von String-Literalen eine Escape-Sequenz eingeleitet. Die Escape-Sequenz mit der Kennung x ermöglicht es, einzelne Bytes in str-Instanzen direkt zu kodieren. Sie erwartet eine zweistellige Hexadezimalzahl als Parameter, die direkt hinter das x geschrieben wird. Der Wert dieses Parameters gibt den Zahlenwert des Bytes an, im Beispiel also 0xdc = 220 ("Ü"), 0xfc = 252 ("ü") und 0xc4 = 196 ("Ä"). Diese Zahlen hat Python der aktuellen Codepage entnommen, in der sie genau den angegebenen Zeichen entsprechen:

```
>>> print chr(220), chr(252), chr(196)
Ü ü Ä
```

Diese Kodierung von Sonderzeichen hat den Vorteil, dass der Quelltext nur aus normalen ASCII-Zeichen besteht und beim Abspeichern und Verteilen nicht mehr auf die verwendete Codepage geachtet werden muss.

Allerdings bringt eine solche Kodierung zwei wichtige Nachteile mit sich: Zum einen ist die Anzahl möglicher Zeichen auf 256 begrenzt, und zum anderen muss jemand, der einen so kodierten String verarbeiten will, wissen, welche Codepage verwendet wurde, weil sich viele Codepages widersprechen. Den zweiten Nachteil kann man eher als Schönheitsfehler betrachten, da eine einfache Lösung darin besteht, einfach zu jedem String die verwendete Kodierung mit anzugeben. Ein wirklicher Mangel ist dagegen die Begrenzung der Zeichenanzahl. Stellen Sie sich mal einen String vor, der eine Ausarbeitung über Autoren aus verschiedenen Sprachräumen mit Originalzitaten enthält: Sie würden aufgrund der vielen verschiedenen Alphabete sehr schnell an die Grenze der 8-Bit-Kodierung stoßen und könnten das Werk nicht digitalisieren. Oder stellen Sie sich vor, Sie wollen einen Text in chinesischer Sprache kodieren, was durch die über 10.000 Schriftzeichen unmöglich würde.

Ein naheliegender Lösungsansatz für dieses Problem bestand darin, den Speicherplatz pro Zeichen zu erhöhen, was aber neue Nachteile mit sich brachte. Verwendet man beispielsweise 16 Bits für jedes einzelne Zeichen, ist die Anzahl der Zeichen immer noch auf 65.536 begrenzt, und man muss davon ausgehen, dass die Sprachen sich weiterentwickeln werden und somit auch diese Anzahl einmal nicht mehr ausreichen wird. [Es ist tatsächlich so, dass 16 Bit schon heute nicht mehr ausreichen, um alle Zeichen der menschlichen Sprache zu kodieren.] Außerdem würde sich im 16-Bit-Beispiel der Speicherplatzbedarf für einen String verdoppeln, weil für jedes Zeichen doppelt

so viele Bits wie bei erweiterter ASCII-Kodierung verwendet würden, und das, obwohl ein Großteil aller Texte hauptsächlich aus einer kleinen Teilmenge aller vorhandenen Zeichen besteht. Die einfache Speicherplatzerhöhung für jedes einzelne Zeichen ist also keine wirkliche Lösung, denn das Problem wird irgendwann wieder auftreten, wenn die neu gesetzte Schranke erneut überschritten wird. Außerdem wird unnötig Speicherplatz vergeudet.

Eine langfristige Lösung für das Kodierungsproblem wurde schließlich durch den Standard namens *Unicode* erarbeitet, der variable Kodierungslängen für einzelne Zeichen vorsieht. Im Prinzip ist Unicode eine riesige Tabelle, die jedem bekannten Zeichen eine Zahl, den sogenannten *Codepoint* zuweist. Diese Tabelle wird vom *Unicode Consortium*, einer gemeinnützigen Institution, gepflegt und ständig erweitert. Codepoints werden in der Regel als »U+*x*« geschrieben, wobei *x* die hexadezimale Repräsentation des Codepoints ist. Das wirklich Neue an Unicode ist das Verfahren UTF (Unicode Transformation Format), das Codepoints durch Byte-Folgen unterschiedlicher Länge darstellen kann. Es gibt verschiedene dieser Transformationsformate, aber das wichtigste und am weitesten verbreitete ist UTF-8. UTF-8 verwendet bis zu 7 Byte, um ein einzelnes Zeichen zu kodieren, wobei die tatsächliche Länge von der Häufigkeit des Zeichens in Texten abhängt. So lassen sich zum Beispiel alle Zeichen des ASCII-Standards mit jeweils einem Byte kodieren, das zusätzlich den gleichen Zahlenwert wie die entsprechende ASCII-Kodierung des Zeichens hat. Durch dieses Vorgehen wird erreicht, dass jeder mit ASCII kodierte String auch gültiger UTF-8-Code ist: UTF-8 ist zu ASCII abwärtskompatibel. Wie das technisch genau realisiert worden ist, soll uns an dieser Stelle nicht weiter beschäftigen, sondern uns interessiert in erster Linie, wie wir Unicode mit Python nutzen können.

Python bietet dem Programmierer neben dem bereits bekannten Datentyp `str`, der Strings als Ketten von Bytes abstrahiert, den Datentyp `unicode`, der Strings als Ketten von Zeichen darstellt, wobei jedes beliebige Unicode-Zeichen unterstützt wird. Um eine `unicode`-Instanz zu erzeugen, dient ein normales String-Literal, dem ein `u` bzw. ein `U` vorangestellt wird:

```
>>> mein_unicode = u"Hallo Welt"
>>> type(mein_unicode)
<type 'unicode'>
>>> print mein_unicode
Hallo Welt
```

`unicode`-Instanzen haben alle Methoden, die Sie schon von `str` kennen, und sind dementsprechend genauso komfortabel nutzbar:

```
>>> u = u"Überprüfung der Änderungen"
>>> print u.upper()
ÜBERPRÜFUNG DER ÄNDERUNGEN
```

Wie bereits von `str` bekannt, erzeugt die Ausgabe ohne `print` im interaktiven Modus auch für `unicode`-Instanzen ein Literal, in dem alle Sonderzeichen als Escape-Sequenzen kodiert sind, was zur Folge hat, dass für das Literal nur ASCII-Zeichen verwendet werden:

```
>>> u"Überprüfung der Änderungen"
u'\xdcberpr\xfcfung der \xc4nderungen'
```

Die durch `\x` eingeleitete Escape-Sequenz wird innerhalb von `unicode`-Literalen nur für die ersten 256 Codepoints verwendet, damit die Literale genauso wie normaler ASCII-Code aussehen. Um einen beliebigen Codepoint direkt mit seinem Zahlenwert anzugeben, dient die Escape-Sequenz, die mit `\u` eingeleitet wird. Hinter das `\u` wird die hexadezimal kodierte Zahl des Codepoints geschrieben. Das Eurozeichen »€« hat beispielsweise den Codepoint »U+20ac« und lässt sich folgendermaßen maskieren:

```
>>> s = u"\u20ac"
>>> print s
€
```

Der Datentyp `unicode` eignet sich wunderbar für die Arbeit mit Text-Strings in Python-Programmen und vereinfacht dabei den Umgang mit internationalen Schriftzeichen ernorm. Allerdings gibt es einige Besonderheiten, die bei der Verwendung von `unicode` beachtet werden müssen. Unicode abstrahiert von Bytes zu Zeichen, was für den Programmierer angenehmer ist, auf Maschinenebene aber den Nachteil mit sich bringt, dass solche Strings nicht einfach in Byte-Ketten gespeichert werden können. Möchte man aber beispielsweise Daten auf der Festplatte ablegen, sie über das Netzwerk versenden oder mit anderen Programmen austauschen, ist man auf die Gegebenheiten der Maschine und damit auch die Byte-Ketten beschränkt. Es muss also Möglichkeiten geben, aus einem abstrakten `unicode`-String eine konkrete Byte-Folge zu erzeugen und umgekehrt. `unicode`-Instanzen haben

encode

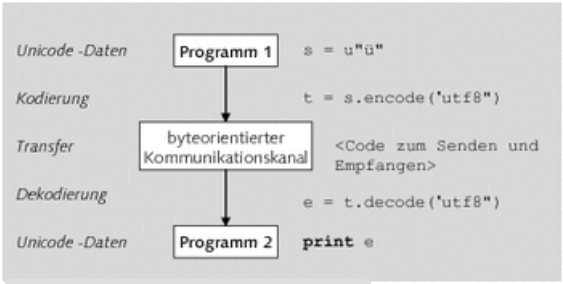
eine Methode , die als Parameter den Namen der gewünschten Kodierung enthält, zum Beispiel "utf8". Das Ergebnis dieser Umwandlung ist eine str-Instanz, die die Repräsentation des Strings in der übergebenen Kodierung enthält. Um aus einer kodierten str-Instanz wieder ein unicode-Objekt zu machen, dient die Methode decode. Sie erwartet als Parameter den Namen der Kodierungsvorschrift, die beim Erzeugen des Strings verwendet wurde:

```
>>> s_unicode = u"Überprüfung der Änderungen in \u20ac"
>>> s
u'\xdcberpr\xfcfung der \xc4nderungen in \u20ac'
>>> s_utf8 = s.encode("utf8")
>>> s_utf8
'\xc3\x9cberpr\xc3\xbcfung der \xc3\x84nderungen in \xe2\x82\xac'
>>> t = s_utf8.decode("utf8")
>>> t
u'\xdcberpr\xfcfung der \xc4nderungen in \u20ac'
```

Im Beispiel erzeugen wir zuerst die unicode-Instanz s_unicode, die neben drei direkt eingegebenen Sonderzeichen auch ein maskiertes Eurozeichen enthält. Anschließend nutzen wir die Methode encode, um die UTF-8-Repräsentation von s_unicode zu ermitteln und mit der Referenz s_utf8 zu verknüpfen. In der Ausgabe von s_utf8 sehen wir, dass für die Kodierung der Umlaute zwei und für die des Eurozeichens sogar drei Bytes verwendet wurden. Am Ende erhalten wir eine neue unicode-Instanz, die den gleichen Inhalt hat wie s_unicode, indem wir s_utf8 mithilfe von decode als UTF-8-String interpretieren.

Innerhalb eines einzelnen Programms ist es wenig sinnvoll, unicode-Strings erst zu kodieren und dann wieder zu dekodieren, da man intern sehr bequem mit ihnen arbeiten kann. Wichtig wird die Kodierung erst, wenn man die enthaltenen Daten senden oder speichern möchte, wobei der Kommunikationskanal oder das Speichermedium nur mit Bytes arbeiten kann.

Folgendes Schema veranschaulicht den Transfer von Unicode mithilfe von Kodierung und Dekodierung:



Hier klicken, um das Bild zu vergrößern

Abbildung 8.6 Schematische Darstellung eines Unicode-Transfers

Angenommen, Programm 1 erzeugt einen unicode-String s, der zum Beispiel ein »ü« enthält. Nun soll diese Zeichenkette über eine Netzwerkverbindung, die nur Bytefolgen übertragen kann, an Programm 2 gesendet werden. Dazu wird s zuerst in sein UTF-8-Äquivalent überführt und dann – wie genau ist hier nicht wichtig – über das Netzwerk an Programm 2 gesendet, wo es wieder dekodiert und anschließend verwendet werden kann.

Als Faustregeln für den Umgang mit dem Datentyp unicode kann man sich Folgendes merken:

- Benutzen Sie nur für Binärdaten str.
- Verwenden Sie für alle Textdaten, die das Programm verwendet, unicode-Instanzen.
- Kodieren Sie unicode-Daten beim Speichern oder beim Datenversand zu anderen Programmen nach einer bestimmten Methode.
- Gewinnen Sie beim Lesen und Empfangen der Daten mit dem entsprechenden Kodierungsverfahren wieder die unicode-Instanzen zurück.

Wenn man diese Regeln konsequent einhält, kann das Programm mit beliebigen Sonderzeichen umgehen, ohne dass besondere Anpassungen notwendig werden, weil sich eine unicode-Instanz genauso handhaben lässt wie ein str-Objekt. Dadurch wird nicht nur die Übersetzung, sondern auch der allgemeine Umgang mit Textdaten vereinfacht, weil sich der Programmierer nicht mehr mit den Beschränkungen der Maschine beschäftigen muss. Er muss nur dafür Sorge tragen, dass die Schnittstellen nach außen enkodierte Daten bereitstellen.

Codecs

Bis jetzt sind wir nur mit den beiden Kodierungsverfahren »Windows-1252« und »UTF-8« in Berührung gekommen. Es gibt neben diesen beiden noch eine ganze Reihe weiterer Verfahren, von denen Python viele von Haus aus unterstützt. Jede dieser Kodierungen hat in Python einen String als Namen, den man der encode-Methode übergeben kann. Die folgende Tabelle zeigt exemplarisch ein paar dieser Namen. Alle Kodierungen, die Python unterstützt, können Sie dem Anhang entnehmen.

Name in Python	Eigenschaften
"ascii"	Kodierung mithilfe der ASCII-Tabelle. Englisches Alphabet, Ziffern, Satzzeichen und Steuerzeichen. Ein Byte pro Zeichen.
"utf8"	Kodierung für alle Unicode-Codepoints. Abwärtskompatibel zu ASCII. Variable Anzahl Bytes pro Zeichen.
"cp1252"	Kodierung für Westeuropa, die von Windows verwendet wird. Zusätzlich zu den ASCII-Zeichen Unterstützung für europäische Sonderzeichen, insbesondere das Eurozeichen. Abwärtskompatibel zu ASCII. Ein Byte pro Zeichen.

Tabelle 8.24 Drei der von Python unterstützten Encodings

Wenn Sie nun versuchen, einen unicode-String mit einem Kodierungsverfahren zu enkodieren, das nicht für alle in dem String enthaltenen Zeichen geeignet ist, führt dies zu einem Fehler (U+03a9 ist der Codepoint des großen Omega):

```
>>> s = u"\u03a9"
>>> print s

>>> s.encode("cp1252")
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    t = s.encode("cp1252")
  File "C:\Python25\lib\encodings\cp1252.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character  '\u03a9' in
position 0: character maps to <undefined>
```

Wie aus dem Beispiel ersichtlich ist, unterstützt »Windows-1252« das Omega-Zeichen nicht, weshalb das Enkodieren mit einer Fehlermeldung quittiert wird. Es ergibt sich ein Problem, wenn man mit Kodierungen arbeitet, die nicht jedes beliebige Zeichen verarbeiten können: Man kann nie sicher sein, dass die beispielsweise vom Benutzer eingegebenen Daten unterstützt werden, und läuft deshalb Gefahr, bei der Verarbeitung sein Programm abstürzen zu lassen. Um dieses Problem zu umgehen, bieten die Methoden encode und decode einen optionalen Parameter namens *errors* an, der die Vorgehensweise in solchen Fehlerfällen definiert. Für *errors* können die folgenden Werte übergeben werden: [Dabei handelt es sich um spezielle Formatierungen zur Darstellung von Sonderzeichen in XML-Dateien. Näheres zu XML-Dateien erfahren Sie in Abschnitt 19.2.]

Wert	Bedeutung
"strict"	Standardeinstellung. Jedes nicht kodierbare Zeichen führt zu einem Fehler.
"ignore"	Nicht kodierbare Zeichen werden ignoriert.
"replace"	Nicht kodierbare Zeichen werden durch einen Platzhalter ersetzt: Beim Enkodieren durch das Fragezeichen "?", beim Dekodieren durch das Unicode-Zeichen U+FFFD.
"xmlcharrefreplace"	Nicht kodierbare Zeichen werden durch ihre XML-Entität ersetzt.[9] (Nur bei encode möglich.)
"backslashreplace"	Nicht kodierbare Zeichen werden durch eine Escape-Sequenz ersetzt. (Nur bei encode möglich.)

Tabelle 8.25 Werte für den *errors*-Parameter von encode und decode

Wir betrachten das letzte Beispiel mit anderen Werten für *errors*:

```
>>> s = u"\u03a9"
>>> print s

>>> s.encode("cp1252", "replace")
'?'
>>> s.encode("cp1252", "xmlcharrefreplace")
'&#937;'
>>> s.encode("cp1252", "backslashreplace")
'\\u03a9'
```

Damit es erst gar nicht nötig wird, Kodierungsprobleme durch diese Hilfsmittel

zu umgehen, sollten Sie nach Möglichkeit immer zu allgemeinen Kodierungsverfahren wie UTF-8 greifen.

Encoding-Deklaration

Damit Sonderzeichen nicht nur innerhalb von Strings, sondern auch in Kommentaren geschrieben werden dürfen, muss im Kopf einer Python-Programmdatei eine sogenannte *Encoding-Deklaration* stehen. Dies ist eine Zeile, die das Encoding kennzeichnet, in dem die Programmdatei gespeichert wurde.

Das ist nur dann wichtig, wenn Sie in der Programmdatei Buchstaben oder Zeichen verwendet haben, die nicht im englischen Alphabet enthalten sind. Ein Encoding ermöglicht es dem Python-Interpreter dann, diese Zeichen korrekt zuzuordnen. Eine Encoding-Deklaration sieht folgendermaßen aus und steht in der Regel direkt unter der Shebang-Zeile [Die Bedeutung einer Shebang-Zeile wurde in Abschnitt 3.2.1 geklärt.] bzw. in der ersten Zeile der Programmdatei:

```
# -*- coding: cp1252 -*-
```

In diesem Fall wurde das Windows-Encoding cp1252 verwendet.

Beachten Sie, dass aus Gründen der Übersichtlichkeit in keinem Beispielprogramm des Buchs eine Encoding-Deklaration enthalten ist. Das bedeutet aber ausdrücklich nicht, dass der Einsatz einer Encoding-Deklaration grundsätzlich falsch wäre.

Die in diesem Buch vorgestellten Beispielprogramme enthalten nicht nur keine Encoding-Deklaration, sondern sind auch ohne sie lauffähig.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

10.5 Anonyme Funktionen

Mithilfe des Schlüsselwortes lambda können kleine, anonyme Funktionen erstellt werden. Solche Funktionen werden üblicherweise für häufig auftretende Berechnungen verwendet, um sich alle Vorteile einer echten Funktion zu erhalten, diese gleichzeitig aber nicht aufwendig definieren zu müssen. Eine anonyme Funktion wird zum Beispiel folgendermaßen erzeugt:

```
f = lambda x: x * 3 +7
```

Auf das Schlüsselwort lambda folgen eine Parameterliste und ein Doppelpunkt. Hinter dem Doppelpunkt muss ein beliebiger arithmetischer oder logischer Ausdruck stehen, der nach seiner Auswertung im Rückgabewert der Funktion mündet. Beachten Sie, dass die Beschränkung auf einen arithmetischen Ausdruck zwar die Verwendung von Kontrollstrukturen ausschließt, nicht aber die Verwendung einer *conditional expression*.

Eine lambda-Form ergibt ein Funktionsobjekt und kann, wie im Beispiel geschehen, referenziert werden. Der Aufruf der Funktion läuft wie gewohnt ab:

```
r = f(10)
```

Der Rückgabewert wäre in diesem Fall 37. Betrachten wir noch ein etwas komplexeres Beispiel einer anonymen Funktion mit drei Parametern:

```
f = lambda x, y, z: (x - y) * z
```

Beachten Sie, dass Sie im »Funktionskörper« keine Kontrollstrukturen verwenden dürfen, es muss ein rein

arithmetischer Ausdruck sein. Jede lambda-Form kann ebenso durch eine »echte« Funktion ersetzt werden. Das entsprechende Gegenstück zum obigen Beispiel sähe so aus:

```
def f(x, y, z):
    return (x - y) * z
```

Anonyme Funktionen können auch aufgerufen werden, ohne sie vorher referenzieren zu müssen. Dazu muss der lambda-Ausdruck in Klammern gesetzt werden:

```
(lambda x, y, z: (x - y) * z)(1, 2, 3)
```

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

13.5 Iteratoren

Sie sind bei der Lektüre dieses Buchs schon oft mit dem Begriff »iterierbares Objekt« konfrontiert worden, wobei Ihnen bisher nur gesagt wurde, dass Sie solche Instanzen beispielsweise mit einer `for`-Schleife durchlaufen oder bestimmten Funktionen, wie `list`, als Parameter übergeben konnten. In diesem Kapitel werden wir uns nun endlich mit den Hintergründen und Funktionsweisen dieser Objekte befassen.

Ein sogenannter *Iterator* ist eine Abstraktionsschicht, die es ermöglicht, auf die Elemente einer Sequenz über eine standardisierte Schnittstelle zuzugreifen. Bisher mussten Sie für den Zugriff auf die Elemente einer Sequenz oder eines Dictionarys immer eine Referenz auf den Container, also die `list`- oder `dict`-Instanz, sowie den Index des jeweiligen Elements benutzen. Dies hatte den Nachteil, dass man dafür immer die Art der Indizes kennen musste, die die Datenstruktur anbot, weshalb man den Code für jeden Datentyp anpassen musste. Nun ist aber insbesondere das Durchlaufen aller Elemente einer Sequenz oder eines anderen Objekts, das mehrere Elemente speichert, eine Operation, die unabhängig von dem jeweiligen Datentyp immer

auf das Gleiche hinausläuft. Um beispielsweise alle Elemente einer Sequenz auszugeben, benötige ich nacheinander Zugriff auf die Elemente, wobei es mir egal sein kann, ob dieser nun über numerische Indizes oder irgendeine andere Art von Schlüsseln bereitgestellt wird.

Deshalb wurden Iteratoren eingeführt, mit denen der jeweilige Datentyp sich selbst um die Bereitstellung der Elemente kümmert und die konkrete Implementation hinter einer einheitlichen Schnittstelle versteckt. Die dazu festgelegte Schnittstelle heißt *Iterator-Protokoll* und ist folgendermaßen definiert:

Jede iterierbare Instanz muss eine parameterlose `__iter__`-Methode implementieren, die ein *Iterator-Objekt* zurückgibt. Das Iterator-Objekt muss ebenfalls eine `__iter__`-Methode besitzen, die einfach eine Referenz auf das Objekt selbst zurückgibt. Außerdem muss es eine `next`-Methode aufweisen, die bei jedem Aufruf das nächste Element des zu durchlaufenden Containers zurückgibt. Ist das Ende der Iteration erreicht, muss die `next`-Methode die `StopIteration`-Exception mittels `raise` werfen.

Auf der anderen Seite, also da, wo die Iteration selbst stattfindet, muss, um das Durchleufen zu beginnen, mittels der Built-in Function iter eine Referenz auf den Iterator ermittelt werden. `iter(objekt)` ruft dabei die `__iter__`-Methode der Instanz `objekt` auf und reicht das Ergebnis als Rückgabewert an die aufrufende Ebene weiter. Von der zurückgegebenen Iterator-Instanz kann dann so lange die `next`-Methode aufgerufen werden, bis diese die `StopIteration`-Exception wirft.

Um mehr Licht in diese abstrakte Beschreibung zu bringen, werden wir eine Klasse entwickeln, die uns über die Fibonacci-Folge iterieren lässt. Die Fibonacci-Folge ist eine Folge aus ganzen Zahlen, wobei jedes Element *f(n)* durch die Summe seiner beiden Vorgänger *f(n-2) + f(n-1)* berechnet werden kann. Die beiden ersten Elemente werden per Definition auf *f(1) = f(2) = 1* gesetzt. Der Anfang der unendlichen Folge ist in der nachstehenden Tabelle gezeigt:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f(n)	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Tabelle 13.1 Die ersten 14 Elemente der Fibonacci-Folge

Die Folge kann unter anderem dazu verwendet werden, die idealisierte Entwicklung von Kaninchenpopulationen zu berechnen. Außerdem konvergiert der Quotient von aufeinanderfolgenden Elementen für große *n* gegen den goldenen Schnitt (= 1,618…), einem Verhältnis, das sich sehr oft in der Natur findet.

```
class Fibonacci(object):
    def __init__(self, max_n):
        self.MaxN = max_n
        self.N = 0
        self.A = 0
        self.B = 0

    def __iter__(self):
        self.N = 0
        self.A = 0
        self.B = 1
        return self

    def next(self):
        if self.N < self.MaxN:
            self.N += 1
            self.A, self.B = self.B, self.A + self.B
            return self.A
        else:
            raise StopIteration
```

Unsere Klasse `Fibonacci` erwartet als Parameter für ihren Konstruktor die Nummer des Elements, nach dem die Iteration stoppen soll. Diese Nummer speichern wir in dem privaten Attribut `MaxN` und zählen dann mit dem Attribut `N`, wie viele Elemente bereits zurückgegeben wurden. Um uns zwischen den `next`-

Aufrufen die aktuelle Position in der Folge zu merken und um das nächste Element berechnen zu können, speichern wir das zuletzt zurückgegebene Element und seinen Nachfolger in den Attributen `A` und `B` der `Fibonacci`-Klasse. Wir werden keine separate Iterator-Klasse definieren und lassen deshalb die `__iter__`-Methode eine Referenz auf die `Fibonacci`-Instanz selbst, also `self`, zurückgeben. Außerdem müssen beim Beginn des Durchlaufens die Speicher für das letzte nächste Element mit ihren Anfangswerten `0` bzw. `1` belegt und der `N`-Zähler auf `0` gesetzt werden. Die `next`-Methode kümmert sich um die Berechnung des aktuellen Elements der Folge und aktualisiert die Zwischenspeicher und den Zähler. Ist das Ende der gewünschten Teilfolge erreicht, wird `StopIteration` geworfen.

Die Klasse lässt sich nun mit allen Konstrukten verarbeiten, die das Iterator-Protokoll unterstützen, wie beispielsweise die `for`-Schleife und die Built-in Functions `list` oder `sum`:

```
>>> for f in Fibonacci(14):
        print f,
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> list(Fibonacci(16))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
610, 987]
>>> sum(Fibonacci(60))
4052739537880L
```

Mit einer kleinen Subklasse von `Fibonacci` können wir auch einen Iterator erzeugen, der uns die Verhältnisse zweier aufeinanderfolgender Fibonacci-Zahlen durchlaufen lässt. Dabei sieht man sehr schnell, dass sich die Quotienten dem goldenen Schnitt nähern. Die Subklasse muss nur die `next`-Methode der Fibonacci-Klasse überschreiben und dann anstatt der Folgenelemente die Quotienten zurückgeben. Dabei kommt es uns zugute, dass wir in dem Attribut `B` bereits den Wert des nächsten Elements im Voraus berechnen. Die Implementation sieht dann folgendermaßen aus:

```
class GoldenerSchnitt(Fibonacci):
    def next(self):
        Fibonacci.next(self)
        return float(self.B) / self.A
```

Die Konvertierung von `self.B` in eine Gleitkommazahl ist deshalb notwendig, damit keine Ganzzahldivision durchgeführt wird und die Nachkommastellen nicht verloren gehen. Schon die ersten vierzehn Elemente dieser Folge lassen die Konvergenz erkennen. (Der goldene Schnitt, bis auf sechs Nachkommastellen gerundet, lautet 1,618034.)

```
>>> for g in GoldenerSchnitt(14):
    print "%.6f" % g,
1.000000 2.000000 1.500000 1.666667 1.600000 1.625000
1.615385
1.619048 1.617647 1.618182 1.617978 1.618056 1.618026
1.618037
```

Es ist durchaus üblich, die `__iter__`-Methode eines iterierbaren Objekts als Generator zu implementieren. Im Falle unserer Fibonacci-Folge läuft diese Technik auf wesentlich eleganteren Code hinaus, weil wir uns nun nicht mehr den Status des Iterators zwischen den `next`-Aufrufen merken müssen und auch die explizite Definition von `next` entfällt:

```
class Fibonacci2(object):
    def __init__(self, max_n):
        self.MaxN = max_n

    def __iter__(self):
        n = 0
        a, b = 0, 1
        for n in xrange(self.MaxN):
            a, b = b, a + b
            yield a
```

Instanzen der Klasse Fibonacci2 würden sich bei der Iteration genau wie die Lösung ohne Generator-Ansatz verhalten:

```
>>> list(Fibonacci2(10))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Allerdings ließe sich die Klasse GoldenerSchnitt nicht mehr so einfach als Subklasse von Fibonacci2 implementieren, da die Zwischenspeicherung der Werte und auch die next-Methode nun in dem Generator gekapselt sind.

Benutzung von Iteratoren

Nun haben Sie gelernt, wie Sie eine gültige Iterator-Schnittstelle in ihren eigenen Klassen definieren können. Wir werden diese Thematik nun von der anderen Seite betrachten und uns damit beschäftigen, wie die Benutzung dieser Iterator-Schnittstelle aussieht, damit Sie auch Funktionen schreiben können, die nicht Listen oder andere Sequenzen, sondern beliebige iterierbare Instanzen verarbeiten können.

Wir betrachten zu diesem Zweck eine einfache for-Schleife und werden dann hinter die Kulissen schauen, indem wir eine äquivalente Schleife ohne for programmieren werden, die das Iterator-Protokoll explizit benutzt:

```
>>> for i in xrange(10):
        print i,
0 1 2 3 4 5 6 7 8 9
```

Wie Sie bereits wissen, benötigen wir zum Durchlaufen einer Sequenz das dazugehörige Iterator-Objekt. Dieses liefert uns die Built-in Function iter, die, wie schon in der Einleitung erklärt, die __iter__-Methode des übergebenen Objekts aufruft:

```
>>> iter(xrange(10))
<rangeiterator object at 0x01446608>
```

Über die next-Methode das Iterator-Objekts können wir nun der Reihe nach alle Elemente ermitteln:

```
>>> i = iter(xrange(3))
>>> i.next()
0
>>> i.next()
1
>>> i.next()
2
>>> i.next()
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    i.next()
StopIteration
```

Wird i.next nach dem Zurückgeben des letzten Elements erneut aufgerufen, wirft die Methode erwartungsgemäß die StopIteration-Exception. Wenn wir diese Exception mit einer try/except-Anweisung abfangen, können wir die for-Schleife folgendermaßen nachbauen:

```
>>> i = iter(xrange(10))
>>> while True:
        try:
            print i.next(),
        except StopIteration:
            break
0 1 2 3 4 5 6 7 8 9
```

Natürlich soll dieses Beispiel keine Aufforderung sein, in Zukunft keine for-Schleifen mehr zu benutzen. Das Ziel unserer Bemühungen war es, Ihnen ein besseres Verständnis für die Benutzung von Iteratoren zu geben. Die for-Schleife in Python ist natürlich nicht wie in dem Beispiel implementiert, sondern in eine optimierte Routine des Python-Interpreters ausgelagert. Dadurch

erlaubt der Iterator-Ansatz auch eine Geschwindigkeitssteigerung, weil die Iteration durch eine maschinennahe C-Schleife übernommen werden kann.

Die for-Schleife kann im Übrigen auch über einen Iterator selbst iterieren und muss diesen nicht selbst erzeugen. Die folgenden beiden Schleifen sind also äqivalent:

```
>>> for i in xrange(3):
        print i,
0 1 2
>>> for i in iter(xrange(3)):
        print i,
0 1 2
```

Dass for dabei, wie in der alternativen while-Schleife verdeutlicht, noch einmal selbst iter aufruft, ist insofern kein Problem, als die __iter__-Methode eines Iterator-Objekts eine Referenz auf das Objekt selbst zurückgeben muss. Ist a ein Iterator-Objekt, so gilt immer a == iter(a), wie das folgende Beispiel noch einmal verdeutlicht:

```
>>> a = iter(xrange(10))  # einen xrange-Iterator erzeugen
>>> a == iter(a)
True
```

Im Gegensatz dazu muss die __iter__-Methode eines iterierbaren Objekts weder eine Referenz auf sich selbst noch immer dieselbe Iterator-Instanz zurückgeben:

```
>>> a = xrange(10)  # ein iterierbares Objekt erzeugen
>>> iter(a) == iter(a)
False
```

Im Umkehrschluss bedeutet dies, dass die Built-in Function iter bei Aufrufen für dasselbe iterierbare Objekt verschiedene Iteratoren zurückgeben kann.

Dieses Verhalten kann zu relativ schwierig auffindbaren Fehlern führen. Stellen Sie sich einmal vor, Sie lesen eine Textdatei ein, die eine bestimmte Schlüsselzeile enthält. Alles, was vor dieser Schlüsselzeile steht, ist für Ihr Programm vollkommen uninteressant, denn Sie interessieren sich nur für den dahinterstehenden Teil. Da Sie bereits wissen, dass man über die Zeilen einer Datei mittels einer eleganten for-Schleife iterieren kann, könnten Sie auf folgende Scheinlösung kommen:

```
datei = open("textdatei.txt", "r")
for zeile in datei:
    if zeile.strip() == "Schlüsselzeile":
        break

for zeile in datei:
    print zeile
```

Der Grund, warum mit diesem Miniprogramm nicht nur die interessanten Zeilen hinter der "Schlüsselzeile", sondern alle Zeilen der Datei ausgegeben werden, liegt darin, dass beide Schleifen jeweils ihren eigenen Iterator für die Datei erzeugt haben. Deshalb wurde zu Beginn der zweiten for-Schleife die Leseposition innehalb der Datei wieder an den Anfang gesetzt und somit alles ausgegeben. Mit ein paar kleinen Änderungen können wir die Schleifen aber dazu zwingen, sich einen Iterator zu teilen, und erreichen damit das gewünschte Verhalten:

```
datei = open("textdatei.txt", "r")
datei_iterator = iter(datei)
for zeile in datei_iterator:
    if zeile.strip() == "Schlüsselzeile":
        break

for zeile in datei_iterator:
    print zeile
```

Dadurch, dass die impliziten iter-Aufrufe am Anfang der beiden for-Schleifen nun Referenzen auf denselben Iterator zurückgeben, erscheinen nur die interessanten Informationen auf dem Bildschirm. Es kann also in manchen Fällen durchaus sinnvoll sein, explizit Iteratoren zu erzeugen und mit diesen zu arbeiten.

Nachteile von Iteratoren gegenüber dem direkten Zugriff über Indizes

Neben den schon angesprochenen Vorteilen, dass einmal geschriebener Code für alle Datentypen, die das Iterator-Interface implementieren, gilt und dass durch die maschinennahe Implementation der Schnittstelle die Ausführung der Programme beschleunigt werden kann, gibt es auch Nachteile.

Iteratoren eignen sich hervorragend, um alle Elemente einer Sequenz zu durchlaufen und dies einheitlich für alle Container-Datentypen umzusetzen. Mit Indizes ist aber auch möglich, in beliebiger Reihenfolge auf die Elemente zuzugreifen und ihre Werte zu verändern, was mit dem Iterator-Ansatz nicht möglich ist.

Insofern lassen sich die Indizes nicht vollständig durch Iteratoren ersetzen, sondern werden für Spezialfälle durch sie ergänzt.

Alternative Definition für iterierbare Objekte

Neben der oben beschriebenen Definition für iterierbare Objekte gibt es noch eine weitere Möglichkeit, eine Klasse iterierbar zu machen. Da es bei sehr vielen Folgen und Containern möglich ist, die Elemente einfach durchzunummerieren und über ganzzahlige Indizes anzusprechen, haben sich die Python-Entwickler dazu entschlossen, dass ein Objekt schon dann iterierbar ist, wenn man seine Elemente über die __getitem__-Methode, also den []-Operator, ansprechen kann. Ruft man die Built-in Function iter mit einer solchen Instanz als Parameter auf, kümmert Python sich um die Erzeugung des Iterators. Bei jedem Aufruf der next-Methode des erzeugten Iterators wird die __getitem__-Methode der iterierbaren Instanz aufgerufen, wobei immer eine Ganzzahl als Parameter übergeben wird. Die Zählung der übergebenen Indizes beginnt bei 0 und endet erst, wenn die __getitem__-Methode einen IndexError produziert, sobald ein ungültiger Index übergeben wurde.

Beispielsweise könnte eine Klasse zum Iterieren über die ersten max_n Quadratzahlen folgendermaßen aussehen, wenn sie zudem noch das Bestimmen ihrer Länge mittels len unterstützt:

```
class Quadrate(object):
    def __init__(self, max_n):
        self.MaxN = max_n

    def __getitem__(self, index):
        index += 1 # 0*0 ist nicht sehr interessant...
        if index > len(self) or index < 1:
            raise IndexError
        return index*index

    def __len__(self):
        return self.MaxN
```

Zur Demonstration dieses versteckten Iterators lassen wir uns eine Liste mit den ersten zwanzig Quadratzahlen ausgeben:

```
>>> list(Quadrate(20))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169,
196, 225, 256, 289, 324, 361, 400]
```

Es bleibt noch zu sagen, dass diese Art von Iterator-Definition nur in seltenen Fällen benutzt werden sollte, da sie einerseits wenig elegant und andererseits meistens langsamer als eine explizite Implementation des Iterator-Protokolls ist.

Funktionsiteratoren

Die letzte Möglichkeit, in Python auf Iteratoren zurückzugreifen, stellen sogenannte *Funktionsiteratoren* dar. Funktionsiteratoren sind Objekte, die eine bestimmte Funktion so lange aufrufen, bis diese einen bestimmten Wert, den *Terminator* der Folge, zurückgibt. Einen Funktionsiterator kann man mit der Built-in Function `iter` erzeugen, wobei als erster Parameter eine Referenz auf die Funktion, über die man iterieren möchte, und als zweiter Parameter der Wert des Terminators übergeben wird.

iter(funktion, terminator)

Ein gutes Beispiel ist die Methode `readline` des `file`-Objekts, die so lange den Wert der nächsten Zeile zurückgibt, bis das Ende der Datei erreicht wurde. Wenn sich keine weiteren Daten mehr hinter der aktuellen Leseposition der `file`-Instanz befinden, gibt `readline` einen leeren String zurück. Läge eine Datei namens *freunde.txt*, die die vier Namen `"Lucas"`, `"Florian"`, `"Lars"` und `"John"` in je einer separaten Zeile enthält, so könnten wir folgendermaßen über sie iterieren und würden die nachstehende Ausgabe erhalten:

```
>>> datei = open("freunde.txt")
>>> for zeile in iter(datei.readline, ""):
        print zeile.strip(),
Lucas Florian Lars John
```

Anmerkung

Dieses Beispiel dient nur der Veranschaulichung von Funktionsiteratoren. Über die Zeilen einer Datei können Sie natürlich auch weiterhin direkt mit

```
>>> for zeile in datei:
...     print zeile.strip(),
```

iterieren.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

15.5 Dateiinterface für Strings – StringIO

Das Modul `StringIO` der Standardbibliothek ermöglicht es, eine Ausgabe, die eigentlich in eine Datei gehen sollte, in einen String umzulenken. Dabei wird eine Instanz der im Modul enthaltenen Klasse `StringIO` erzeugt, die sich wie ein Dateiobjekt verhält. Daten, die in dieses Pseudo-Dateiobjekt geschrieben werden, werden als String in der `StringIO`-Instanz gespeichert und können später im Programm ausgelesen werden.

Das ist beispielsweise dann nützlich, wenn eine Funktion ein geöffnetes Dateiobjekt als Parameter erwartet, um bestimmte Daten abzuspeichern, Sie diese Daten aber lieber als String vorliegen haben würden. Hier kann in den meisten Fällen problemlos eine `StringIO`-Instanz übergeben werden, sodass die geschriebenen Daten nachher im Programm weiterverwendet werden können.

```
>>> import StringIO
>>> pseudodatei = StringIO.StringIO()
```

Dem Konstruktor kann ein optionaler String übergeben werden, der dann dem anfänglichen Inhalt der simulierten Datei entspricht. Von nun an kann die zurückgegebene Instanz, referenziert durch `pseudodatei`, wie ein Dateiobjekt verwendet

werden:

```
>>> pseudodatei.write("Hallo Welt")
>>> print >> pseudodatei, " Hallo Welt"
```

Neben der Funktionalität eines Dateiobjekts bietet eine Instanz der Klasse `StringIO` eine zusätzliche Methode namens `getvalue`, durch die auf den gespeicherten String zugegriffen werden kann:

```
>>> pseudodatei.getvalue()
'Hallo Welt Hallo Welt\n'
```

Wie ein Dateiobjekt auch sollte eine `StringIO`-Instanz durch Aufruf der Methode `close` geschlossen werden, wenn sie nicht mehr gebraucht wird:

```
>>> pseudodatei.close()
```

Das Verwenden der Klasse `StringIO` kann bei intensivem Gebrauch recht langsam sein. Aus diesem Grund existiert eine schnellere Implementation von `String IO` namens `cStringIO`, die dafür allerdings einige Einschränkungen mit sich bringt. So können beispielsweise keine eigenen Klassen von `cStringIO` abgeleitet werden, und es dürfen keine Unicode-Strings verwendet werden. Außerdem erwirkt ein bei der Instanziierung von `cStringIO` übergebener String, dass das entstehende Dateiobjekt schreibgeschützt ist. Die Klasse `cStringIO` kann durch Einbinden des gleichnamigen Moduls verwendet werden.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

17.5 Kommandozeilenparameter – optparse

Im vorletzten Abschnitt wurde gesagt, dass man über `sys.argv` auf die Kommandozeilenparameter zugreifen kann, die beim Aufruf des Programms übergeben wurden. Das ist richtig und funktioniert. Das Modul `optparse` erlaubt Ihnen jedoch einen wesentlich komfortableren Umgang mit Kommandozeilenparametern.

Doch zunächst möchten wir uns allgemein mit der Thematik der Kommandozeilenparameter befassen. Bislang wurden hier ausschließlich Konsolenprogramme behandelt, das heißt Programme, die eine rein textbasierte Schnittstelle zum Benutzer haben. Solche Programme werden üblicherweise aus einer Konsole, auch *Shell* genannt, gestartet. Eine Konsole ist beispielsweise die »Eingabeaufforderung« unter Windows.

Unter Windows wird ein Python-Programm aus der Eingabeaufforderung heraus gestartet, indem in das Programmverzeichnis gewechselt und dann der Name der Programmdatei eingegeben wird. Hinter dem Namen können jetzt zum einen sogenannte *Optionen* und zum anderen sogenannte *Argumente* übergeben werden:

- Ein Argument wird einfach hinter den Namen der Programmdatei geschrieben. Um einen Vergleich zu Funktionsparametern zu ziehen, könnte man von *positional arguments* sprechen. Das bedeutet vor allem, dass die Argumente anhand ihrer Reihenfolge zugeordnet werden. Ein Programmaufruf mit drei Argumenten könnte beispielsweise folgendermaßen aussehen:

```
programm.py karl 1337 heinz
```

- Neben den Argumenten können Optionen übergeben werden. Optionen sind, wie der Name sagt, optional und deshalb *keyword arguments*. Das bedeutet, dass jede Option einen Namen hat und über diesen angesprochen wird. Beim Programmaufruf müssen Optionen vor den Argumenten geschrieben und jeweils durch einen Bindestrich eingeleitet werden. Dann folgen der Optionsname, ein Leerzeichen und der gewünschte Wert. Ein Programmaufruf mit Optionen und Argumenten könnte also folgendermaßen aussehen:

```
programm.py -a karl -b heinz -c 1337 hallo welt
```

  In diesem Fall existieren drei Optionen namens a, b und c mit den Werten "karl", "heinz" und 1337. Zudem wurden zwei Argumente angegeben, die Strings "hallo" und "welt".

  Neben diesen parameterbehafteten Optionen gibt es parameterlose Optionen, die mit einem Flag vergleichbar sind. Das bedeutet, dass sie entweder vorhanden (aktiviert) oder nicht vorhanden (deaktiviert) sind:

```
programm.py -a -b 1 hallo welt
```

  In diesem Fall handelt es sich bei a um eine parameterlose Option.

Im Weiteren soll die Verwendung des Moduls optparse anhand zweier Beispiele besprochen werden.

top

17.5.1 Taschenrechner – ein einfaches Beispiel ▼▲

Das erste Beispiel soll ein einfacher Taschenrechner sein, bei dem sowohl die Rechenoperation als auch die Operanden über Kommandozeilenparameter angegeben werden. Das Programm soll folgendermaßen aufgerufen werden können:

```
calc.py -o plus 7 5
calc.py -o minus 13 29
calc.py -o mal 4 11
calc.py -o geteilt 3 2
```

Das bedeutet, dass über die Option -o eine Rechenoperation festgelegt werden kann, die auf die beiden folgenden Argumente angewendet wird. Wenn die Option nicht angegeben wurde, sollen die Argumente addiert werden.

Zu Beginn des Programms muss die Klasse OptionParser des Moduls optparse eingebunden und instanziiert werden:

```
from optparse import OptionParser
parser = OptionParser()
```

Jetzt können durch die Methode add_option der OptionParser-

Instanz erlaubte Optionen hinzugefügt werden. In unserem Fall ist es nur eine:

```
parser.add_option("-o", "--operation", dest="operation")
```

Der erste Parameter der Methode gibt den Kurznamen der Option an. Jede Option ist auch mit einer ausgeschriebenen Version des Namens verwendbar, sofern diese Alternative durch Angabe des zweiten Parameters gegeben ist. In diesem Fall wären die Optionen `-o` und `--operation` gleichbedeutend. Der letzte Parameter, ein *keyword argument* wohlgemerkt, gibt an, unter welchem Namen der Wert der Option später im Programm verfügbar gemacht werden soll.

Nachdem alle Optionen hinzugefügt worden sind, wird die Methode `parse_args` aufgerufen, die die Kommandozeilenparameter ausliest und in der gewünschten Form aufbereitet.

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

Die Methode gibt ein Tupel mit zwei Werten zurück: zum einen eine Instanz, die alle übergebenen Optionen enthält (`optionen`), und zum anderen eine Liste mit allen weiteren Argumenten (`args`).

Um korrekt arbeiten zu können, müssen dem Taschenrechner-Programm exakt zwei Argumente übergeben worden sein, was wir an dieser Stelle im Quelltext überprüfen. Wenn die Anzahl der Argumente ungleich zwei ist, kann keine Berechnung durchgeführt werden und das Programm beendet sich:

```
if len(args) != 2:
    parser.error("Es werden exakt zwei Argumente erwartet")
```

Für Fehler, die aufgrund falscher oder fehlender Kommandozeilenparameter auftreten, eignet sich die Methode `error` der `OptionParser`-Instanz, die eine entsprechende Fehlermeldung ausgibt und das Programm beendet.

Als Nächstes legen wir ein Dictionary an, das alle möglichen Rechenoperationen als Schlüssel und die dazugehörigen Berechnungsfunktionen als jeweiligen Wert enthalten. Die Schlüssel sind dieselben, die über die Option `-o` angegeben werden können, sodass wir anhand des bei der Option übergebenen Strings direkt auf die zu verwendende Berechnungsfunktion schließen können:

```
calc = {
       "plus" : lambda a, b: a + b,
       "minus" : lambda a, b: a - b,
       "mal" : lambda a, b: a * b,
       "geteilt" : lambda a, b: a / b,
       None : lambda a, b: a + b
       }
```

Prinzipiell braucht jetzt nur noch der Wert ausgelesen werden, der mit der Option `-o` übergeben wurde. Der Zugriff auf eine Option ist anhand der von `parse_args` zurückgegebenen Instanz `optionen` relativ einfach, da jede Option unter ihrem gewählten Namen als Attribut dieser Instanz verfügbar ist. Der von uns gewählte Name für die Option `-o` war `operation`.

```
op = optionen.operation
if op in calc:
    print "Ergebnis:", calc[op](float(args[0]),
float(args[1]))
else:
    parser.error("%s ist keine Operation" % op)
```

Beachten Sie, dass im Falle einer nicht angegebenen Option das entsprechende Attribut nicht etwa nicht vorhanden ist, sondern lediglich `None` referenziert. Da `None` im Dictionary `calc` als Schlüssel geführt wird und auf die Berechnungsfunktion der

Addition verweist, werden die beiden Argumente in einem solchen Fall schlicht zusammengezählt.

17.5.2 Weitere Verwendungsmöglichkeiten

In diesem Abschnitt soll das Beispielprogramm des letzten Abschnitts dahingehend erweitert werden, dass weitere Verwendungsmöglichkeiten des Moduls optparse hervorgehoben werden. Hier sehen Sie zunächst den Quellcode des veränderten Beispielprogramms:

```
from optparse import OptionParser

parser = OptionParser("calc2.py [Optionen] Operand1
Operand2")
parser.add_option("-o", "--operation", dest="operation",
                  help="Rechenoperation")
parser.add_option("-v", "--verbose", action="store_true",
                  dest="verbose", default=False,
                  help="Schwafelmodus")

(optionen, args) = parser.parse_args()
if len(args) != 2:
    parser.error("Es werden exakt zwei Argumente erwartet")

calc = {
       "plus" : lambda a, b: a + b,
       "minus" : lambda a, b: a - b,
       "mal" : lambda a, b: a * b,
       "geteilt" : lambda a, b: a / b,
       None : lambda a, b: a + b
       }

if optionen.verbose:
    print "Das Ergebnis wird berechnet"
op = optionen.operation
if op in calc:
    print "Ergebnis:", calc[op](float(args[0]),
float(args[1]))
else:
    parser.error("%s ist keine Operation" % op)
```

Zunächst einmal werden Sie feststellen, dass bei der Instanziierung von Option Parser ein String übergeben wurde. Zusätzlich haben auch die Aufrufe der Methode add_option ein weiteres *keyword argument* namens help spendiert bekommen. Diese Angaben sind zwar nicht notwendig, sollten jedoch getätigt werden, da die OptionParser-Instanz aus den dort übergebenen Strings automatisch eine Hilfeseite generiert, wenn das Programm mit den Optionen -h oder --help gestartet wird. In dieser Hilfeseite wird die Verwendung des Programms kurz umrissen. Dazu gehört eine Auflistung aller möglichen Optionen, jeweils mit einem kurzen erläuternden Satz. Für das obige Beispiel sieht der Hilfetext folgendermaßen aus:

```
Usage: calc2.py [Optionen] Operand1 Operand2

Options:
  -h, --help            show this help message and exit
  -o OPERATION, --operation=OPERATION
                        Rechenoperation
  -v, --verbose         Schwafelmodus
```

Zusätzlich zu der bereits im ursprünglichen Beispielprogramm vorhandenen Option -o wurde eine weitere Option namens -v bzw. --verbose angelegt. Viele bekannte Programme verwenden die Option -v als Schalter, um das Programm in eine Art geschwätzigen Zustand zu versetzen. Das bedeutet, dass auch nicht essenzielle Statusmeldungen auf dem Bildschirm ausgegeben werden. Diese Funktionalität soll auch unser Beispielprogramm bekommen. Der geschwätzige Modus soll aktiviert werden, wenn -v oder --verbose angegeben wurden, und sonst deaktiviert bleiben. Programmintern sollte die Option daher als boolescher Wert ankommen.

Dazu werden der Methode add_option zwei weitere *keyword arguments* übergeben. Zum einen wird der Parameter action auf "store_true" gesetzt, was bedeutet, dass das dazugehörige Attribut verbose auf True gesetzt wird, wenn die Option -v vorhanden ist. Analog dazu wäre auch "store_false" für den

umgekehrten Fall möglich gewesen.

Der zweite neue Parameter ist die Angabe eines Default-Wertes. Das ist der Wert, den das dazugehörige Attribut vebose annimmt, wenn die Option -v nicht vorhanden ist.

Der resultierende boolesche Wert optionen.verbose wird im Programm abgefragt, und dann wird eventuell eine zugegebenermaßen sinnlose Statusmeldung ausgegeben. Die Ausgabe des Programms sieht bei Angabe der Option -v folgendermaßen aus:

```
Das Ergebnis wird berechnet
Ergebnis: 19.0
```

Damit wäre der grundlegende Funktionsumfang von optparse erläutert.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

19.5 Das Tabellenformat CSV – csv

Ein sehr verbreitetes Import- und Exportformat für Datenbanken und Tabellenkalkulationen ist das *CSV-Format* (für *Comma Separated Values*). CSV-Dateien sind Textdateien, die zeilenweise Datensätze enthalten. Innerhalb der Datensätze sind die einzelnen Werte durch ein Trennzeichen wie beispielsweise das Komma voneinander getrennt, daher auch der Name.

Eine CSV-Datei, die Informationen zu Personen speichert und das Komma als Trennzeichen nutzt, könnte beispielsweise so aussehen:

```
vorname,nachname,geburtsdatum,wohnort,haarfarbe
Daniel,Zakowski,29.11.1987,Dinslaken,Schwarz
David,Schönauer,10.09.1988,Aachen,Braun
Sebastian,Sentner,06.09.1987,Sydney,Dunkelblond
Jan,Fitzke,13.09.1987,Köln,Schwarz
Lucas,Hövelmann,25.03.1988,Canberra,Hellrot
```

Die erste Zeile enthält die jeweiligen Spaltenköpfe, und alle folgenden Zeilen enthalten die eigentlichen Datensätze.

Leider existiert kein Standard für CSV-Dateien, sodass sich beispielsweise das Trennzeichen von Programm zu Programm unterscheiden kann. Dieser Umstand erschwert es, CSV-Dateien von verschiedenen Quellen zu lesen, da immer auf das besondere Format der exportierenden Anwendung eingegangen werden muss.

Um trotzdem mit CSV-Dateien der verschiedensten Formate umgehen zu können, stellt Python das Modul `csv` zur Verfügung. Das `csv`-Modul implementiert `reader`- und `writer`-Klassen, die den Lese- bzw. Schreibzugriff auf CSV-Daten kapseln. Mithilfe

sogenannter *Dialekte* kann dabei das Format der Datei angegeben werden. Standardmäßig gibt es vordefinierte Dialekte für die CVS-Dateien, die von Microsoft Excel generiert werden. Außerdem stellt das Modul eine Klasse namens `Sniffer` (dt. *Schnüffler*) bereit, die den Dialekt einer Datei erraten kann.

Eine Liste aller definierten Dialekte kann man mit `csv.list_dialects` erhalten:

```
>>> import csv
>>> csv.list_dialects()
['excel-tab', 'excel']
```

reader-Objekte

Mithilfe von `reader`-Objekten können CSV-Dateien gelesen werden. Der Konstruktor sieht dabei folgendermaßen aus:

csv.reader(csvfile[, dialect][, fmtparam])

Der Parameter *csvfile* muss eine Referenz auf ein für den Lesezugriff geöffnetes Dateiobjekt sein, aus dem die Daten gelesen werden sollen. Die Dateiobjekte sollten dabei im Binärmodus (`"rb"`) geöffnet worden sein, weil es sonst auf manchen Plattformen Probleme geben kann.

Mit *dialect* kann angegeben werden, in welchem Format die zu lesende Datei geschrieben wurde. Dazu kann als *dialect* ein String übergeben werden, der in der Liste enthalten ist, die `csv.list_dialects` zurückgibt. Alternativ kann eine Instanz der Klasse `Dialect` angegeben werden, die wir später besprechen werden. Standardmäßig wird der Wert `"excel"` für *dialect* verwendet, wobei die damit kodierten Dateien das Komma als Trennzeichen verwenden.

Der Platzhalter *fmtparam* steht nicht für einen einzelnen Parameter, sondern für Schlüsselwortparameter, die übergeben werden können, um den Dialekt ohne Umweg über die `Dialect`-Klasse festzulegen. Ein Beispiel, bei dem wir auf diese Weise das Semikolon als Trennzeichen zwischen den einzelnen Werten festlegen, sieht folgendermaßen aus:

```
>>> reader = csv.reader(open("datei.csv", "rb"),
delimiter=";")
```

Wir werden uns später ausführlich mit Dialekten beschäftigen.

Die `reader`-Instanzen implementieren das Iterator-Protokoll und lassen sich deshalb komfortabel mit einer `for`-Schleife verarbeiten. Im folgenden Beispiel lesen wir die CSV-Datei mit den Personen aus der Einleitung:

```
>>> reader = csv.reader(open("namen.csv", "rb"))
>>> for row in reader:
        print row
['vorname', 'nachname', 'geburtsdatum', 'wohnort',
'haarfarbe']
['Daniel', 'Zakowski', '29.11.1987', 'Dinslaken',
'Schwarz']
['David', 'Sch\xf6nauer', '10.09.1988', 'Aachen', 'Braun']
['Sebastian', 'Sentner', '06.09.1987', 'Sydney',
'Dunkelblond']
['Jan', 'Fitzke', '13.09.1987', 'K\xf6ln', 'Schwarz']
['Lucas', 'H\xf6velmann', '25.03.1988', 'Canberra',
'Hellrot']
```

Wie Sie sehen, gibt uns der `reader` für jede Zeile eine Liste mit den Werten der einzelnen Spalten zurück. Wichtig ist dabei, dass die Spaltenwerte immer als Strings zurückgegeben werden.

Neben dem Standard-`reader`, der Listen zurückgibt, existiert noch der sogenannte `DictReader`, der für jede Zeile ein Dictionary erzeugt, das den Spaltenköpfen die Werte der jeweiligen Zeile zuordnet.

Unser letztes Beispiel verändert sich durch die Verwendung von DictReader wie folgt, wobei wir nur die ersten beiden Datensätze ausgeben, um Platz zu sparen:

```
>>> reader = csv.DictReader(open("namen.csv", "rb"))
>>> for row in reader:
        print row
{'nachname': 'Zakowski', 'geburtsdatum': '29.11.1987',
'wohnort':
'Dinslaken', 'vorname': 'Daniel', 'haarfarbe': 'Schwarz'}
{'nachname': 'Sch\xf6nauer', 'geburtsdatum': '10.09.1988',
'wohnort': 'Aachen', 'vorname': 'David', 'haarfarbe':
'Braun'}
```

writer-Objekte

Der Konstruktor der writer-Klasse erwartet die gleichen Parameter wie der Konstruktor der reader-Klasse, mit der Ausnahme, dass das für *csvfile* übergebene Dateiobjekt für den Schreibzugriff im Binärmodus ("wb") geöffnet worden sein muss.

csv.reader(csvfile[, dialect][, fmtparam])

Das resultierende writer-Objekt hat die beiden Methoden writerow und writerows, mit denen sich einzelne bzw. mehrere Zeilen auf einmal in die CSV-Datei schreiben lassen:

```
>>> writer = csv.writer(open("autos.csv", "wb"))
>>> writer.writerow(["marke", "modell", "leistung_in_ps"])
>>> daten =(
    ["Volvo", "P245", "130"], ["Ford", "Focus", "90"],
    ["Mercedes", "CLK", "250"], ["Audi", "A6", "350"],
    )
>>> writer.writerows(daten)
```

In dem Beispiel erzeugen wir eine neue CSV-Datei mit dem Namen "autos.csv". Mit der writerow-Methode schreiben wir die Spaltenköpfe in die erste Zeile der neuen Datei und mit writerows anschließend vier Beispieldatensätze.

Analog zur DictReader-Klasse existiert auch eine DictWriter-Klasse, die sich fast genauso wie die normale writer-Klasse erzeugen lässt, außer dass neben dem Dateiobjekt noch eine Liste mit den Spaltenköpfen übergeben werden muss. Für ihre writerow- und writerows-Methoden erwarten DictWriter-Instanzen Dictionarys als Parameter. Das folgende Beispiel erzeugt die gleiche CSV-Datei wie das letzte:

```
>>> writer = csv.DictWriter(open("autos.csv", "wb"),
                    ["marke", "modell", "leistung_in_ps"])

>>> writer.writerow({"marke" : "marke", "modell" :
"modell",
                     "leistung_in_ps" : "leistung_in_ps"})
>>> daten = ({"marke" : "Volvo","modell" : "P245",
              "leistung_in_ps" : "130"},
             {"marke" : "Ford", "modell" : "Focus",
              "leistung_in_ps": "90"},
             {"marke" : "Mercedes", "modell" : "CLK",
              "leistung_in_ps" : "250"},
             {"marke" : "Audi", "modell" : "A6",
              "leistung_in_ps": "350"})
>>> writer.writerows(daten)
```

Die merkwürdige Zeile mit writerow ist notwendig, um die Spaltenköpfe zu schreiben, da dies nicht automatisch geschieht.

Dialect-Objekte

Die Instanzen der Klasse csv.Dialect dienen dazu, den Aufbau von CSV-Dateien zu beschreiben. Sie sollten Dialect-Objekte nicht direkt erzeugen, sondern stattdessen die Funktion csv.register_dialect verwenden. Mit register_dialect können Sie einen neuen Dialekt erzeugen und mit einem Namen versehen. Dieser Name kann dann später als Parameter an die Konstruktoren der reader- und writer-Klassen übergeben werden. Außerdem ist jeder registrierte Name in der von csv.get_dialects zurückgegebenen Liste enthalten.

Die Funktion `register_dialect` hat folgende Schnittstelle:

csv.register_dialect(name[, dialect][, fmtparam])

Der Parameter *name* muss dabei ein String sein, der den neuen Dialekt identifizieren soll. Mit *dialect* kann ein bereits bestehendes `Dialect`-Objekt übergeben werden, das dann mit dem entsprechenden Namen verknüpft wird.

Am wichtigsten ist der Platzhalter *fmtparam*, der für eine Reihe optionaler Schlüsselwortparameter steht, die den neuen Dialekt beschreiben. Es sind die in der folgenden Tabelle aufgeführten Parameter erlaubt:

Name	Bedeutung
`delimiter`	Trennzeichen zwischen den Spaltenwerten. Der Standardwert ist das Komma (`,`).
`quotechar`	Zeichen, um Felder zu umschließen, die besondere Zeichen wie das Trennzeichen oder den Zeilenumbruch enthalten. Der Standardwert sind die doppelten Anführungszeichen (`"`).
`doublequote`	Ein boolescher Wert, der angibt, wie das für `quotechar` angegebene Zeichen innerhalb von Feldern selbst maskiert werden soll. Hat `doublequote` den Wert `True`, so wird `quotechar` zweimal hintereinander eingefügt. Ist der Wert von `doublequote` `False`, wird stattdessen das für `escapechar` angegebene Zeichen vor `quotechar` geschrieben. Standardmäßig hat `doublequote` den Wert `True`.
`escapechar`	Ein Zeichen, das benutzt wird, um das Trennzeichen innerhalb von Spaltenwerten zu maskieren, sofern `quoting` den Wert `QUOTE_NONE` hat. Bei einem `dobulequote`-Wert von `False` wird `escapechar` außerdem für die Maskierung von den `quotechar` verwendet. Standardmäßig ist die Maskierung deaktiviert und `escapechar` hat den Wert `None`.
`lineterminator`	Zeichen, das zum Trennen der Zeilen benutzt wird. Standardmäßig ist es auf `"\r\n"` gesetzt. Bitte beachten Sie, dass diese Einstellung nur den Writer betrifft. Alle `reader`-Objekte bleiben von der `lineterminator`-Einstellung unbeeinflusst und verwenden immer `"\r"`, `"\n"` oder die Kombination aus beiden als Zeilentrennzeichen.
`quoting`	Gibt an, ob und wann Spaltenwerte mit `quotechar` umschlossen werden sollen. Gültige Werte sind: `QUOTE_ALL` – Alle Spaltenwerte werden umschlossen. `QUOTE_MINIMAL` – Nur die Felder mit speziellen Zeichen wie Zeilenvorschüben oder dem Trennzeichen für Spaltenwerte werden umschlossen. `QUOTE_NONNUMERIC` – Beim Schreiben werden alle nicht-numerischen Felder von `quotechar` umschlossen. Beim Lesen werden alle nicht

	umschlossenen Felder automatisch nach `float` konvertiert. `QUOTE_NONE` – Keine Umschließung mit `quotechar` wird vorgenommen. Standardmäßig ist `quoting` auf `QUOTE_MINIMAL` eingestellt.
`skipinitialspace`	Ein boolescher Wert, der angibt, wie mit führenden Whitespaces in einem Spaltenwert verfahren werden soll. Eine Einstellung auf `True` bewirkt, dass alle führenden Whitespaces ignoriert werden; bei einem Wert von `False` wird der komplette Spalteninhalt gelesen und zurückgegeben. Der Standardwert ist `False`.

Tabelle 19.9 Schlüsselwortparameter für register_dialect

Wir wollen als Beispiel einen neuen Dialekt namens "mein_dialekt" registrieren, der als Trennzeichen den Tabulator verwendet und alle Felder mit Anführungszeichen umschließt:

```
>>> csv.register_dialect("mein_dialekt", delimiter="\t",
quoting=csv.QUOTE_ALL)
```

Diesen neuen Dialekt können wir nun dem Konstruktor unserer `reader`- und `writer`-Klassen übergeben und auf diese Weise unsere eigenen CSV-Dateien schreiben und lesen.

<< zurück <top> vor >>

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

20.5 E-Mail ▼

In diesem Abschnitt werden wir Module der Standardbibliothek vorstellen, die es ermöglichen, mit einem E-Mail-Server zu kommunizieren, das heißt E-Mails von diesem abzuholen bzw. E-Mails über den Server zu versenden.

Das Versenden einer E-Mail geschieht über einen sogenannten SMTP-Server, mit dem über ein gleichnamiges Protokoll kommuniziert werden kann. Im ersten Unterabschnitt werden wir deshalb das Modul `smtplib` der Standardbibliothek vorstellen, das genau dieses Kommunikationsprotokoll implementiert.

Für das Herunterladen einer empfangenen E-Mail gibt es zwei verbreitete Möglichkeiten: das POP3- und das IMAP4-Protokoll. Beide können mit dem jeweiligen Modul `poplib` bzw. `imaplib` verwendet werden.

Im letzten Abschnitt soll das Modul `email` der Standardbibliothek besprochen werden, das es über die MIME-Kodierung ermöglicht, beliebige Dateien (üblicherweise Bilder oder Dokumente) mit der E-Mail zu versenden.

top

▼▲

20.5.1 SMTP – smtplib

Das sogenannte *SMTP-Protokoll* (für *Simple Mail Transfer Protocol*) wird zum Versenden einer E-Mail über einen SMTP-Server verwendet. Das SMTP-Protokoll ist ähnlich wie FTP ein textbasiertes, lesbares Protokoll. Ursprünglich bot das SMTP-Protokoll keine Möglichkeit zur Authentifizierung des angemeldeten Benutzers durch Benutzername und Passwort beispielsweise. Dies war bei der rasanten Entwicklung des Internets ziemlich schnell nicht mehr tragbar, und so wurde das SMTP-Protokoll um den *ESMTP-Standard* (*Extended SMTP*) erweitert.

Ähnlich wie im Abschnitt über das FTP-Protokoll möchten wir hier zunächst eine Übersicht über die wichtigsten SMTP-Befehle geben. Die Befehle sind in der folgenden Tabelle in der Reihenfolge ihrer Benutzung in einer SMTP-Sitzung aufgelistet und werden jeweils mit einem kurzen Satz erklärt.

Befehl	Beschreibung
`HELO`	Startet eine SMTP-Sitzung.
`EHLO`	Startet eine ESMTP-Sitzung
`MAIL FROM`	Leitet das Absenden einer E-Mail ein. Diesem Kommando wird die Absenderadresse beigefügt.
`RCPT TO`	Fügt einen Empfänger der E-Mail hinzu. (RCPT steht für »recipient«, dt. »*Empfänger*«.)
`DATA`	Mit diesem Kommando wird der Inhalt der E-Mail angegeben und die Mail schlussendlich verschickt.
`QUIT`	Beendet die SMTP- bzw. ESMTP-Sitzung.

Tabelle 20.6 SMTP-Befehle

Wie schon die `ftplib` enthält das Modul `smtplib` im Wesentlichen nur eine Klasse namens `SMTP`. Über diese Klasse läuft, nachdem sie instanziiert wurde, alle weitere Kommunikation mit dem Server. Der Konstruktor der Klasse `SMTP` hat folgende Schnittstelle:

smtplib.SMTP([host[, port[, local_hostname]]])

Erzeugt eine Instanz der Klasse `SMTP`. Optional können hier bereits die Verbindungsdaten zum SMTP-Server übergeben werden. Beachten Sie, dass Sie den Port nur explizit anzugeben brauchen, wenn er sich vom SMTP-Standardport 25 unterscheidet.

Als dritter Parameter kann der Domainname des lokalen Hosts übergeben werden. Dieser wird dem SMTP-Server als Identifikation im ersten gesendeten Kommando übermittelt. Wenn der Parameter *local_hostname* nicht angegeben wird, wird versucht, den lokalen Hostnamen automatisch zu ermitteln.

Die Klasse SMTP

Im Folgenden sollen die wichtigsten Methoden der `SMTP`-Klasse erläutert werden. Um die Beispiele dieses Abschnitts nachvollziehen zu können, muss das Modul `smtplib` eingebunden werden und eine Instanz der Klasse `SMTP` mit dem Namen `s` existieren. Für die meisten der Beispiele muss die `SMTP`-Instanz zusätzlich mit einem Server verbunden und angemeldet sein.

s.connect([host[, port]])

Verbindet zum SMTP-Server *host* mit der Portnummer *port*. Diese Methode sollte nicht aufgerufen werden, wenn bei der Instanziierung der `SMTP`-Klasse bereits Verbindungsdaten übergeben wurden. Wenn keine Verbindung zum SMTP-Server aufgebaut werden kann, wird eine Exception geworfen.

```
>>> s.connect("smtp.knallhart.de")
(220, 'Die Botschaft des Servers')
```

s.login(user, password)

Diese Methode ermöglicht es, sich beim SMTP-Server mit dem Benutzernamen *user* und dem Passwort *password* einzuloggen, sofern der Server dies verlangt.

```
>>> s.login("Benutzername", "Passwort")
(235, '2.0.0 Authentication successful')
```

Im Fehlerfall können folgende Exceptions geworfen werden:

Exception	Beschreibung
SMTPHeloError	Der SMTP-Server hat nicht oder nicht richtig auf das Begrüßungskommando HELO geantwortet.
SMTPAuthenticationError	Die angegebene Benutzername/Passwort-Kombination wurde vom SMTP-Server nicht akzeptiert.
SMTPError	Es wurde keine Möglichkeit gefunden, eine Authentifizierung bei diesem SMTP-Server durchzuführen.

Tabelle 20.7 Mögliche Exceptions beim Login

s.sendmail(from_addr, to_addr, msg[, mail_options, rctp_options])

Durch Aufruf der Methode sendmail kann eine E-Mail über den SMTP-Server versendet werden. Beachten Sie, dass die SMTP-Instanz dafür an einem SMTP-Server angemeldet und zumeist auch authentifiziert sein muss.

Die ersten beiden Parameter enthalten die E-Mail-Adressen des Absenders (*from_addr*) bzw. eine Liste der E-Mail-Adressen der Empfänger (*to_addr*). Als E-Mail-Adresse wird dabei ein String des folgenden Formats bezeichnet:

```
Vorname Nachname <em@il.addr>
```

Alternativ kann auch nur die E-Mail-Adresse an sich im String stehen.

Als dritter Parameter, *msg*, wird der Text der E-Mail übergeben. Beachten Sie, dass weitere Angaben wie beispielsweise der Betreff der E-Mail innerhalb des E–Mail-Bodys definiert werden. Wie so etwas genau aussieht und welche Möglichkeiten Python bietet, diesen Header komfortabel zu erzeugen, erfahren Sie in Abschnitt 20.5.4.

Die Methode sendmail gibt stets ein Dictionary zurück, in dem alle Empfänger, die vom SMTP-Server zurückgewiesen wurden, als Schlüssel enthalten sind und der jeweilige Error-Code mit Fehlerbezeichnung als Wert aufgeführt ist. Wenn alle Empfänger die E-Mail bekommen haben, ist das zurückgegebene Dictionary leer.

Im Fehlerfall wirft die Methode sendmail folgende Exceptions:

Exception	Beschreibung
SMTPHeloError	Der SMTP-Server hat nicht oder nicht richtig auf das Begrüßungskommando HELO geantwortet.
SMTPRecipientsRefused	Alle Empfänger wurden vom SMTP-Server zurückgewiesen. Das heißt, dass die E-Mail tatsächlich an niemanden verschickt wurde. Als Attribut enthält die Exception ein Dictionary, das demjenigen ähnelt, das von sendmail im Erfolgsfall zurückgegeben wird.
SMTPSenderRefused	Der angegebene Absender wurde vom SMTP-Server zurückgewiesen.
SMTPDataError	Der Server hat mit einem unerwarteten Fehler geantwortet.

Tabelle 20.8 Mögliche Exceptions beim Login

Beachten Sie, dass der Text einer E-Mail nur aus ASCII-Zeichen bestehen darf. Um auch andere Zeichen und insbesondere auch Binärdaten verschicken zu können, bedient man sich der sogenannten MIME-Kodierung, die wir im Kapitel email behandeln werden.

Über die optionalen Parameter *mail_options* und *rctp_options* kann je eine Liste von Strings übergeben werden, die Optionen des ESMTP-Standards (*Extended SMTP*) enthalten. Die für *mail_options* übergebenen Optionen werden dem Kommando MAIL FROM angefügt; hier wäre beispielsweise die Option "8bitmime" sinnvoll. Alle für *rctp_options* übergebenen Optionen werden dem Kommando RCPTTO angehängt, wo beispielsweise die Option "DSN" verwendet werden könnte.

Welche Optionen im ESMTP-Standard definiert werden und was diese bedeuten, soll nicht Thema dieses Buches sein. Wenn Sie sich dafür interessieren, fühlen Sie sich dazu ermutigt, unter den genannten Stichwörtern im Internet zu recherchieren.

s.quit()

Beendet die Verbindung zum SMTP-Server.

Beispiel

Nachdem die wichtigsten Methoden einer SMTP-Instanz erläutert wurden, folgt nun ein kleines Beispiel, in dem zu einem SMTP-Server verbunden wird, um zwei E-Mails an verschiedene Empfänger zu verschicken:

```
>>> smtp = smtplib.SMTP("smtp.hostname.de")
>>> smtp.login("benutzername", "passwort")
(235, '2.0.0 Authentication successful')
>>> smtp.sendmail(
...                "Peter Kaiser <p@penguin-p.de>",
...                "Johannes Ernesti <je@revelation-soft.de>",
...                "Dies ist der Text")
{}
>>> smtp.sendmail(
...                "Peter Kaiser <p@penguin-p.de>",
...                ["je@revelation-soft.de", "p@penguin-p.de"]
...                "Dies ist der Text")
{}
>>> smtp.quit()
```

Bei der ersten E-Mail wurden die vollen Namen des Absenders bzw. des Empfängers angegeben. Das zweite Beispiel zeigt, dass auch die E-Mail-Adresse allein reicht und wie eine E-Mail an mehrere Empfänger versandt werden kann.

top

20.5.2 POP3 – poplib ▼▲

Nachdem anhand der smtplib erläutert wurde, wie E-Mails über einen SMTP-Server versandt werden können, soll das Thema dieses Kapitels das Modul poplib der Standardbibliothek sein. Dieses Modul implementiert das POP3-Protokoll (*Post Office Protocol Version 3*). Bei POP3 handelt es sich um ein Protokoll, um auf einen POP3-Server zuzugreifen und dort gespeicherte E-Mails einzusehen und abzuholen. Das POP3-Protokoll steht damit in Konkurrenz zu IMAP4, dessen Benutzung mit der imaplib das Thema des nächsten Abschnitts sein soll. Die folgende Tabelle listet die wichtigsten POP3-Kommandos mit ihrer Bedeutung auf. Die Befehle stehen dabei in der Reihenfolge, wie sie in einer üblichen POP3-Sitzung verwendet werden.

Befehl	Beschreibung
USER	Überträgt den Benutzernamen zur Authentifizierung auf dem Server.
PASS	Überträgt das Passwort zur Authentifizierung auf dem Server.
STAT	Liefert den Status des Posteingangs, beispielsweise die Anzahl der neu eingegangenen E-Mails.
LIST	Liefert Informationen zu einer bestimmten E-Mail des Posteingangs.
RETR	Überträgt eine bestimmte E-Mail.
DELE	Löscht eine bestimmte E-Mail.
	Löschvorgänge werden gepuffert und erst am Ende der Sitzung

RSET	ausgeführt. Mit diesem Kommando können alle anstehenden Löschvorgänge widerrufen werden.
QUIT	Beendet die POP3-Sitzung.

Tabelle 20.9 POP3-Befehle

Wie bereits beim Modul smtplib ist im Modul poplib im Wesentlichen die Klasse POP3 enthalten, die instanziiert werden muss, bevor Operationen auf einem POP3-Server durchgeführt werden können. Die Schnittstelle des Konstruktors sieht folgendermaßen aus:

poplib.POP3(host[, port])

Erzeugt eine Instanz der Klasse POP3. Dem Konstruktor wird der Hostname des POP3-Servers übergeben, zu dem verbunden werden soll. Optional kann ein Port angegeben werden, wenn dieser sich vom voreingestellten Standardport 110 unterscheidet.

Die Klasse POP3

Im Folgenden sollen die wichtigsten Methoden der Klasse POP3 beschrieben werden. Die Funktionsnamen entsprechen im Wesentlichen den POP3-Befehlen, die sie senden. Um die in diesem Abschnitt vorgestellten Beispiele ausführen zu können, muss zum einen das Modul poplib eingebunden sein und zum anderen eine Instanz der Klasse POP3 mit dem Namen pop existieren. Beachten Sie, dass diese Instanz für die meisten Beispiele mit einem POP3-Server verbunden und bei diesem authentifiziert sein muss.

pop.user(username)

Übermittelt den Benutzernamen *username* an den POP3-Server. Die Antwort des Servers ist in der Regel ein String, in dem er ein Passwort fordert. Das angeforderte Passwort kann jetzt durch einen Aufruf der Methode pass_ übermittelt werden. Beachten Sie, dass ein falscher Benutzername zunächst nicht zu einer Exception führt.

```
>>> pop.user("Benutzername")
'+OK Password required.'
```

pop.pass_(password)

Übermittelt das Passwort *password* an den POP3-Server. Nachdem das Passwort vom Server akzeptiert worden ist, darf auf den Posteingang zugegriffen werden. Dieser ist bis zum Aufruf von quit für andere Login-Versuche gesperrt.

```
>>> pop.pass_("Passwort")
'+OK logged in.'
```

Im Falle einer fehlgeschlagenen Authentifizierung wird eine poplib.error_proto-Exception geworfen.

pop.stat()

Gibt den Status des Posteingangs zurück. Das Ergebnis ist ein Tupel mit zwei ganzen Zahlen: der Anzahl der enthaltenen Nachrichten und der Größe des Posteingangs.

```
>>> pop.stat()
(1, 623)
```

In diesem Fall befindet sich eine E-Mail im Posteingang, und die Gesamtgröße des Posteingangs beläuft sich auf 623 Byte.

pop.list([which])

Gibt eine Liste der im Posteingang liegenden Mails zurück. Der Rückgabewert dieser Methode ist ein Tupel der folgenden Form:

```
(antwort, ["mailID laenge", ...], datlen)
```

Dabei enthält das Tupel als erstes Element den Antwortstring des Servers und als zweites Element eine Liste von Strings, die je für eine E-Mail des Posteingangs stehen. Im String sind zwei Angaben enthalten. Die Angabe `mailID` ist die laufende Nummer der Mail, eine Art Index, und `laenge` ist die Gesamtgröße der Mail in Byte. In Bezug auf den Index sollten Sie beachten, dass alle E-Mails auf dem Server fortlaufend von `1` an indiziert werden und nicht, wie bei einer Python-Liste beispielsweise, mit `0` beginnend.

Das erste Element des Tupels (`antwort`) enthält dabei nicht den vollständigen Antwortstring des Servers, denn die Informationen, die zum zweiten Element des Tupels aufbereitet wurden, wurden aus `antwort` entfernt. Um dennoch die komplette Länge der Serverantwort berechnen zu können, existiert das dritte Element des Tupels (`datlen`). Dieses referenziert die Länge des Datenbereichs der Antwort des Servers. Damit entspräche `len(antwort) + datlen` der Gesamtgröße des vom Server tatsächlich gesendeten Antwortstrings.

Über den optionalen Parameter *which* kann die laufende Nummer einer E-Mail angegeben werden, über die nähere Informationen zurückgegeben werden sollen. In diesem Fall gibt die Methode einen String des Formats `"+OK mailID laenge"` zurück. Es ist also mit dieser Methode nur möglich, die Länge einer bestimmten E-Mail in Byte herauszufinden, da die ID ja bereits bekannt ist. Wenn eine ungültige ID für `which` übergeben wurde, wird eine `poplib.error_proto`-Exception geworfen.

```
>>> pop.list()
('+OK […]', ['1 623'], 7)
>>> pop.list(1)
'+OK 1 623'
```

pop.retr(which)

Greift auf die Mail mit der laufenden Nummer *which* zu und gibt ihren Inhalt in Form des folgenden Tupels zurück:

```
(antwort, zeilen, laenge)
```

Das erste Element des Tupels entspricht dem Antwortstring des Servers. An zweiter Stelle steht eine Liste von Strings, die je eine Zeile der E-Mail inklusive des E–Mail-Headers enthalten. Das letzte Element des Tupels ist die Größe der E-Mail in Byte.

```
>>> pop.retr(1)
('+OK 623 octets follow.', […], 623)
```

Anstelle des Auslassungszeichens stünde eine Liste von Strings, die die Zeilen der vollständigen E-Mail enthält.

pop.dele(which)

Löscht die Mail mit der laufenden Nummer *which* vom POP3-Server. Beachten Sie, dass die meisten Server solche Befehle puffern und erst nach Aufruf der Methode `quit` tatsächlich ausführen.

```
>>> pop.dele(1)
'+OK Deleted.'
```

pop.rset()

Ein Aufruf dieser Methode veranlasst, dass alle anstehenden Löschvorgänge verworfen werden.

```
>>> pop.rset()
'+OK Resurrected.'
```

pop.quit()

Beendet die Verbindung zum POP3-Server. Bei den meisten Servern werden erst jetzt alle anstehenden Löschvorgänge durchgeführt.

```
>>> pop.quit()
'+OK Bye-bye.'
```

Beispiel

Nachdem die wichtigsten Methoden einer POP3-Instanz erklärt wurden, werden wir hier in einem kleinen Beispiel das Modul poplib dazu verwenden, alle Mails von einem POP3-Server abzuholen und auf dem Bildschirm anzuzeigen:

```
import poplib

pop = poplib.POP3("pop.hostname.de")
pop.user("benutzername")
pop.pass_("passwort")

for i in xrange(1, pop.stat()[0]+1):
    for zeile in pop.retr(i)[1]:
        print zeile
    print "***"

pop.quit()
```

Zunächst wird eine Instanz der Klasse POP3 erzeugt und das Programm meldet sich mit den Methoden user und pass_ beim POP3-Server an. Der Ausdruck pop.stat()[0] liefert die Zahl der Mails, die sich im Posteingang befinden. In der for-Schleife werden also alle Mail-Indizes durchlaufen. Beachten Sie dazu, dass die Indizes der E-Mails im Posteingang mit 1 beginnen.

In der inneren Schleife wird die jeweils aktuelle Mail mit dem Index i durch Aufruf der Methode retr heruntergeladen. Das zweite Element, also das mit dem Index 1 des von dieser Methode zurückgegebenen Tupels, enthält eine Liste mit allen Zeilen des Mail-Inhalts. Diese Liste wird in der Schleife durchlaufen, und es wird jeweils die aktuelle Zeile ausgegeben.

Beachten Sie, dass im Beispielprogramm aus Gründen der Übersichtlichkeit auf jegliche Fehlerbehandlung verzichtet wurde. In einem fertigen Programm müssten Sie auf jeden Fall prüfen, ob die Verbindung zum Server hergestellt werden konnte und ob die Authentifizierung erfolgreich war.

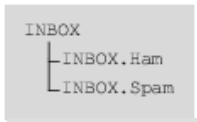
Nachdem eine E-Mail vollständig durchlaufen worden ist, werden drei Sternchen ausgegeben, die damit als eine Art Trennlinie zwischen den Mails fungieren.

top

20.5.3 IMAP4 – imaplib

Das Modul imaplib stellt die Klasse IMAP4 zur Verfügung, mit deren Hilfe man eine Verbindung zu einem IMAP4-Server herstellen und mit diesem kommunizieren kann. Das IMAP4-Protokoll (*Internet Message Access Protocol 4*) ist ähnlich wie das POP3-Protokoll zur Verwaltung von E-Mails auf einem Mailserver gedacht. Anders als bei dem bekannteren Protokoll POP3 verbleiben die E-Mails bei IMAP4 zumeist auf dem Mailserver, was den Vorteil hat, dass man von überall – beispielsweise auch von einem Internet-Cafe im Urlaub aus – vollen Zugriff auf alle archivierten E-Mails hat. Heutzutage bieten die meisten E-Mail-Anbieter sowohl einen POP3- als auch einen IMAP4-Zugang an. Im Vergleich zu POP3 unterstützt IMAP4 Kommandos zur komfortablen Verwaltung der Mails auf dem Server. So können beispielsweise Unterordner angelegt werden.

Im Gegensatz zu den bisherigen Protokollen wie FTP oder POP3 ist IMAP4 mit einem hohen Funktionsumfang ausgestattet, und obwohl das Protokoll immer noch auf lesbaren Textnachrichten basiert, ist es zu komplex, um es im Stil der bisherigen Abschnitten mit einem kurzen Text und einer Tabelle ausreichend zu beschreiben. Grundsätzlich kann aber gesagt werden, dass das IMAP4-Protokoll umfassende Unterstützung zur Verwaltung der E-Mails bereitstellt. So lassen sich diese beispielsweise in verschiedene sogenannte *Mailboxen* einsortieren. Dabei kann man sich eine Mailbox als eine Art Verzeichnis vorstellen, das E-Mails enthalten kann, wie ein Ordner Dateien enthält. Die Mailbox-Struktur des verwendeten Beispielservers sieht folgendermaßen aus:



Hier klicken, um das Bild zu vergrößern

Abbildung 20.3 Mailbox-Struktur des Beispielservers

Es existieren eine übergeordnete Mailbox namens *INBOX* sowie zwei untergeordnete Mailboxen namens *INBOX.Ham* und *INBOX.Spam*.

Um eine Verbindung zu einem IMAP4-Server herstellen zu können, muss eine Instanz der Klasse IMAP4 erzeugt werden. Der Konstruktor dieser Klasse hat folgende Schnittstelle:

imaplib.IMAP4([host[, port]])

Erzeugt eine Instanz der Klasse IMAP4. Optional kann direkt nach der Instanziierung automatisch eine Verbindung zu einem IMAP4-Server mit dem Hostnamen *host* unter Verwendung des Ports *port* aufgebaut werden. Wenn der Parameter *port* nicht angegeben wurde, wird der IMAP4-Standardport 143 verwendet.

Die Klasse IMAP4

Nachdem eine Instanz der Klasse IMAP4 erzeugt wurde, stellt diese verschiedene Methoden bereit, um mit dem verbundenen Server zu kommunizieren. Jede Methode, die ein IMAP4-Kommando repräsentiert, gibt ein Tupel der folgenden Form zurück:

```
(Status, [Daten, …])
```

Dabei steht im resultierenden Tupel für *Status* entweder "OK" oder "NO", je nachdem, ob die Operation erfolgreich verlaufen oder fehlgeschlagen ist. Das zweite Element des Tupels ist eine Liste, die die Daten enthält, die der Server als Antwort geschickt hat. Diese Daten können entweder ebenfalls ein String oder ein Tupel sein. Wenn es sich um ein Tupel handelt, verfügt dieses über zwei Elemente:

```
(Header, Daten)
```

Beide Elemente dieses Tupels sind Strings. Im Folgenden sollen die wichtigsten Methoden einer IMAP4-Instanz erläutert werden. Die Beispiele setzen zumeist eine verbundene IMAP4-Instanz im voraus:

```
>>> import imaplib
>>> im = imaplib.IMAP4("imap.test.de")
```

In den meisten Fällen muss die IMAP4-Instanz zudem beim Server eingeloggt sein, was durch Aufruf der Methode login geschieht.

im.login(user, password)

Sendet Benutzername und Passwort an den verbundenen IMAP4-Server.

```
>>> im.login("Benutzername", "Passwort")
('OK', ['LOGIN Ok.'])
```

im.logout()

Beendet die Verbindung zum IMAP4-Server.

```
>>> im.logout()
('BYE', ['Courier-IMAP server shutting down'])
```

im.select([mailbox[, readonly]])

Wählt eine Mailbox aus, um weitere Operationen auf dieser durchführen zu können. Dabei wird als erster Parameter der Name der

auszuwählenden Mailbox übergeben. Wenn für den Parameter *readonly* 1 übergeben wird, ist die gewählte Mailbox bei diesem Zugriff schreibgeschützt und kann somit nicht verändert werden. Die Funktion `select` gibt die Anzahl der E-Mails zurück, die sich in der gewählten Mailbox befinden.

```
>>> im.select("INBOX")
('OK', ['2'])
```

Beachten Sie, dass keine Exception geworfen wird, wenn die gewünschte Mailbox nicht existiert, sondern dass der Fehler anhand des Rückgabewertes ausgemacht werden muss:

```
>>> im.select("INBOX.NichtExistent")
('NO', ['Mailbox does not exist, or must be subscribed to.'])
```

im.close()

Schließt die momentan ausgewählte Mailbox.

```
>>> im.close()
('OK', ['mailbox closed.'])
```

im.list([directory[, pattern]])

Gibt die Namen aller Mailboxen zurück, die sich im Ordner *directory* befinden und auf *pattern* passen. Wenn der Parameter *directory* nicht übergeben wird, werden Mailboxen des Hauptordners zurückgegeben. Sollte der zweite Parameter *pattern* nicht übergeben werden, so werden alle im jeweiligen Ordner enthaltenen Mailboxen zurückgegeben. Der Parameter *pattern* muss ein String sein und enthält üblicherweise Fragmente eines Mailbox-Namens inklusive Platzhalter (`*`).

```
>>> im.list(".", "*Ham")
('OK', ['(\\HasNoChildren) "." "INBOX.Ham"'])
>>> im.list(".", "*am")
('OK', ['(\\HasNoChildren) "." "INBOX.Ham"',
'(\\HasNoChildren) "." "INBOX.Spam"'])
>>> im.list(".", "*")
('OK', ['(\\HasNoChildren) "." "INBOX.Ham"',
'(\\HasNoChildren) "." "INBOX.Spam"',
'(\\Unmarked \\HasChildren) "." "INBOX"'])
>>> im.list(".", "NichtVorhandeneMailbox")
('OK', [None])
```

Jeder Eintrag der Liste ist ein String und enthält drei, jeweils durch ein Leerzeichen voneinander getrennte Informationen: die sogenannten *Flags* der Mailbox in Klammern, das Verzeichnis der Mailbox und der Mailbox-Name jeweils in doppelten Anführungsstrichen. Aus den Flags kann man beispielsweise die Information entnehmen, ob eine Mailbox untergeordnete Mailboxen enthält (`\HasChildren`) oder nicht (`\HasNoChildren`).

im.fetch(message_set, message_parts)

Lädt Teile der E-Mails vom Server herunter. Der Parameter *message_set* muss ein String sein, der die Mail-IDs der E-Mails enthält, die herunterzuladen sind. Dabei können diese entweder einzeln im String vorkommen (`"1"`), als Bereich (`"1:4"` für Mail Nr.1 bis 4), als Liste von Bereichen (`"1:4,7:9"` für Mail Nr.1 bis 4 und Nr.7 bis 9) oder als Bereich mit unbestimmter oberer Grenze (`"3:*"` für alle Mails ab Mail Nr.3).

Wenn andere Methoden der `IMAP4`-Klasse über einen Parameter *message_set* verfügen, so ist damit stets ein String im oben beschriebenen Format gemeint.

Der zweite Parameter *message_parts* kennzeichnet, welche Teile der angegebenen E-Mails heruntergeladen werden sollen. Ein Wert von `"(RCF822)"` bedeutet, die gesamte Mail, also inklusive des Mail-Headers herunterzuladen. Bei einem Wert von `"(BODY[TEXT])"` würde hingegen nur der Text und bei `"(BODY[HEA DER])"` nur der Header der E-Mail heruntergeladen.

Ein Aufruf der Methode `fetch` funktioniert nur, wenn zuvor eine Mailbox mittels `select` ausgewählt wurde.

```
>>> im.fetch("1", "(BODY[TEXT])")
('OK', [('1 (BODY[TEXT] {29}',
'Dies ist eine Testnachricht\r\n'), ')'])
>>> im.fetch("1:2", "(BODY[TEXT])")
('OK', [('1 (BODY[TEXT] {29}',
'Dies ist eine Testnachricht\r\n'), ')',
('2 (BODY[TEXT] {25}',
'Noch eine Testnachricht\r\n'), ')'])
```

Im Falle einer nicht vorhandenen Mail-ID wird keine Exception geworfen, sondern schlicht ein leeres Ergebnis zurückgegeben. Wenn die ID ungültig ist, kommt eine entsprechende Fehlermeldung zurück.

```
>>> im.fetch("100", "(BODY[TEXT])")
('OK', [None])
>>> im.fetch("KeineID", "(BODY[TEXT])")
('NO', ['Error in IMAP command received by server.'])
```

im.create(mailbox)

Erstellt eine neue Mailbox namens *mailbox*.

```
>>> im.create("INBOX.Hallo")
('OK', ['"INBOX.Hallo" created.'])
```

im.delete(mailbox)

Löscht die Mailbox *mailbox*.

```
>>> im.delete("INBOX.Hallo")
('OK', ['Folder deleted.'])
```

im.rename(old_mailbox, new_mailbox)

Benennt die Mailbox *old_mailbox* in *new_mailbox* um.

```
>>> im.rename("INBOX.Hallo", "INBOX.Ciao")
('OK', ['Folder renamed.'])
```

im.copy(message_set, new_mailbox)

Kopiert die E-Mails *message_set* in die Mailbox *new_mailbox*.

im.search(charset, criterion[, ...])

Sucht innerhalb der ausgewählten Mailbox nach E-Mails, die auf die angegebenen Kriterien passen. Als Kriterium *criterion* kann entweder der String `"ALL"` (alle Mails erfüllen dieses Kriterium) oder ein String des Formats `"(FROM \"Johannes\")"` verwendet werden. Das zweite Kriterium ist für alle Mails erfüllt, die von einem gewissen Johannes geschrieben wurden.

Der Parameter *charset* spezifiziert das Encoding von *criterion* in Form eines Strings. Üblicherweise wird der Parameter *charset* nicht benötigt und `None` übergeben.

Die Funktion `search` gibt die IDs der gefundenen E-Mails in Form einer Liste zurück.

```
>>> im.search(None, "(FROM \"Johannes\")")
('OK', ['1 2 3'])
>>> im.search(None, "(FROM \"Johann\")")
('OK', ['1 2 3'])
>>> im.search(None, "(FROM \"Johanninski\")")
('OK', [''])
```

Beispiel

Im folgenden Beispielprogramm soll das Modul `imaplib` dazu verwendet werden, zu einem IMAP4-Server zu verbinden und alle enthaltenen E-Mails einer bestimmten Mailbox anzuzeigen. Dabei soll dem Benutzer die Möglichkeit gegeben werden, die Mailbox zu wählen.

Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
import imaplib

im = imaplib.IMAP4("Hostname")
im.login("Benutzername", "Passwort")

print "Vorhandene Mailboxen:"
for mb in im.list()[1]:
    name = mb.split("\".\"")[-1]
    print " - %s" % name.strip(" \"")

mb = raw_input("Welche Mailbox soll angezeigt werden: ")
im.select(mb)
status, daten = im.search(None, "ALL")
for mailnr in daten[0].split():
    typ, daten = im.fetch(mailnr, "(RFC822)")
    print "%s\n+++\n" % daten[0][1]

im.close()
im.logout()
```

Zunächst wird eine Instanz der Klasse `IMAP4` erzeugt und zu einem IMAP4-Server verbunden. Dann werden mithilfe der Methode `list` alle im Hauptordner des IMAP4-Kontos vorhandenen Mailboxen durchlaufen und die Namen der Mailboxen auf dem Bildschirm angezeigt. Dabei ist zu beachten, dass die Methode `list` die Namen der Mailboxen mit zusätzlichen Informationen zurückgibt. Diese Informationen müssen herausgefiltert werden, bevor der Mailboxname angezeigt werden kann. Nachdem die Namen angezeigt wurden, wird der Benutzer dazu aufgefordert, einen der angegebenen Mailbox-Namen auszuwählen.

Die vom Benutzer ausgewählte Mailbox wird dann mithilfe der Methode `select` auch auf dem Server ausgewählt. Der danach aufgerufenen Methode `search` wird der String `"ALL"` übergeben, was den Mailserver dazu veranlasst, Daten über alle E-Mails der ausgewählten Mailbox zurückzugeben.

Danach iterieren wir in einer `for`-Schleife über die Liste von Mail-IDs, die `search` zurückgegeben hat, und laden die jeweilige Mail mittels `fetch` vollständig herunter. Die heruntergeladene Mail wird auf dem Bildschirm ausgegeben.

Schlussendlich schließen wir die ausgewählte Mailbox und beenden die Verbindung mit dem Server.

Beachten Sie auch bei diesem Beispielprogramm, dass keine Fehlerbehandlung durchgeführt wurde. In einem fertigen Programm sollten sowohl die Verbindungsanfrage als auch das Login und insbesondere auch die Benutzereingabe überprüft werden.

top

20.5.4 Erstellen komplexer E-Mails – email

In den vorherigen Kapiteln wurde besprochen, wie E-Mails über einen SMTP-Server versendet und von einem POP3- oder IMAP4-Server heruntergeladen werden können. Trotz alledem bleibt eine Frage weiterhin offen: Wie Sie wissen, basiert das Senden und Empfangen von E-Mails auf reinen ASCII-Protokollen. Das bedeutet vor allem, dass mit diesen Protokollen keine Binärdaten verschickt werden können. Außerdem sind Sonderzeichen, die nicht dem 7-Bit-ASCII-Standard entsprechen, problematisch.

Um also solche Zeichen oder Binärdaten verschicken zu können, wurde der sogenannte MIME-Standard (*Multipurpose Internet Mail Extension*) entwickelt, der Sonderzeichen und Binärdaten so kodiert, dass sie als eine Folge reiner ASCII-Zeichen versandt werden können. Durch eine solche Form der Kodierung steigt allerdings die Größe der zu übermittelnden Daten. Zudem definiert der MIME-Standard verschiedene Dateitypen und legt eine Syntax fest, mit der Dateianhänge einem bestimmten Dateityp zugeordnet werden, sodass die Dateien beim Empfänger leichter verarbeitet werden können.

Das `email`-Paket ist sehr mächtig, weswegen hier nur ein Teil seiner Funktionalität besprochen werden kann. Zunächst werden wir uns darum kümmern, wie eine simple ASCII-Mail mittels `email` erstellt werden kann. Darauf aufbauend werden wir zu komplexeren MIME-kodierten Mails übergehen.

Erstellen einer einfachen E-Mail

Als Basisklasse für eine neue E-Mail dient die Klasse Message des Moduls email.message. Das folgende Beispielprogramm zeigt, wie sie zu verwenden ist:

```
from email.message import Message

msg = Message()
msg.set_payload("Dies ist meine selbst erstelle E-Mail.")
msg["Subject"] = "Hallo Welt"
msg["From"] = "Donald Duck <don@ld.de>"
msg["To"] = "Onkel Dagobert <d@gobert.de>"

print msg.as_string()
```

Zunächst wird eine Instanz der Klasse Message erzeugt. Der Konstruktor dieser Klasse erwartet keine Argumente. Durch die Methode set_payload (dt. *Nutzlast*) wird der E-Mail ein Text hinzugefügt.

Jetzt fehlt nur noch der E-Mail-Header. Um diesen hinzuzufügen, kann die Message-Instanz wie ein Dictionary angesprochen werden. Auf diese Weise werden die einzelnen Teile des Headers hinzugefügt. Wichtig sind dabei "Subject" für den Betreff, "From" für den Absender und "To" für den Empfänger der Mail.

Zu guter Letzt wird die entstandene E-Mail durch die Methode as_string in einen String geschrieben und ausgegeben. Dieser String könnte der Methode sendmail des Moduls smtplib übergeben und somit als E-Mail verschickt werden. Die Ausgabe des Beispielprogramms, also die erzeugte E-Mail, sieht folgendermaßen aus:

```
Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>

Dies ist meine selbst erstelle E-Mail.
```

Erstellen einer E-Mail mit Anhängen

Wir haben angekündigt, dass es das Paket email ermöglicht, Binärdaten per E–Mail zu verschicken. Dafür ist das Modul email.mime zuständig. Das folgende Beispielprogramm soll eine E-Mail erstellen und eine Bilddatei als Anhang einfügen.

```
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
from email.mime.text import MIMEText

msg = MIMEMultipart()
msg["Subject"] = "Hallo Welt"
msg["From"] = "Donald Duck <don@ld.de>"
msg["To"] = "Onkel Dagobert <d@gobert.de>"

text = MIMEText("Dies ist meine selbst erstelle E-Mail.")
msg.attach(text)

f = open("buch.png")
bild = MIMEImage(f.read())
f.close()
msg.attach(bild)

print msg.as_string()
```

Zunächst wird eine Instanz der Klasse MIMEMultipart erzeugt. Diese repräsentiert eine E-Mail, die MIME-kodierte Binärdaten enthalten kann. Wie im vorherigen Beispiel werden Betreff, Absender und Empfänger nach Art eines Dictionarys hinzugefügt.

Danach wird eine Instanz der Klasse MIMEText erzeugt, die den reinen Text der E–Mail enthalten soll. Diese Instanz wird mithilfe der Methode attach an die MIMEMultipart-Instanz angehängt.

Genauso wird mit dem Bild verfahren: Es wird eine Instanz der Klasse MIMEImage erzeugt und mit den Binärdaten des Bildes gefüllt. Danach wird sie mittels attach an die E-Mail angefügt.

Schlussendlich wird die MIMEMultipart-Instanz durch Aufruf der Methode as_string in einen String konvertiert, der so als reine ASCII-E-Mail versendet werden kann. Der angefügte Anhang wird von E-Mail-Programmen als Grafik erkannt und dann dementsprechend präsentiert.

Die Ausgabe des Beispiels sieht in etwa so aus:

```
Content-Type: multipart/mixed;
boundary="===========0094312333=="
```

```
MIME-Version: 1.0
Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>

--===============0094312333==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Dies ist meine selbst erstelle E-Mail.
--===============0094312333==
Content-Type: image/png
MIME-Version: 1.0
Content-Transfer-Encoding: base64

iVBORw0KGgoAAAANSUhEUgAAASwAAAD8CAIAAABCVg65AAAACXBIWXMAACcQAAAnE
B3RJTUUH1wkMERU1+MuwjgAAIABJREFUeNrsfXecJWWV9nPet+rezt2ThzDkqCBLB
[…]
--===============0094312333==--
```

Sie sehen, dass sowohl der Text als auch das Bild in ähnlicher Form kodiert wurden. Die Aufbereitung der beiden Sektionen zum Textteil der E-Mail und zu einem Bild im Anhang erledigt Ihr Mail-Programm. Das mime-Paket bietet auch eine entsprechende Funktionalität an, auf die wir noch zu sprechen kommen werden.

Hier wurden nur MIMEText und MIMEImage besprochen. Im Folgenden sind alle verfügbaren MIME-Datentypen aufgelistet:

- email.mime.application.MIMEApplication für ausführbare Programme
- email.mime.audio.MIMEAudio für Sounddateien
- email.mime.image.MIMEImage für Grafikdateien
- email.mime.message.MIMEMessage für Message-Instanzen
- email.mime.image.MIMEText für reinen Text

Beim Instanziieren all dieser Klassen müssen die jeweiligen Binärdaten bzw. der Text, den die entsprechende Instanz enthalten soll, als erster Parameter des Konstruktors übergeben werden. Wichtig ist noch, dass alle hier vorgestellten Klassen von der Basisklasse Message abgeleitet sind, also über die Methoden dieser Basisklasse verfügen.

Internationale Zeichensätze

Bisher wurde besprochen, wie der MIME-Standard dazu verwendet werden kann, Binärdaten im Anhang einer E-Mail zu versenden. Beim Text der E-Mail waren wir aber bislang auf die Zeichen des 7-Bit-ASCII-Standards beschränkt. Die Frage ist, wie ein spezielles Encoding innerhalb einer E-Mail verwendet werden kann. Auch dies ermöglicht der MIME-Standard. Das folgende Beispielprogramm soll eine einfache E-Mail erstellen, deren Text ein Euro-Zeichen enthält.

```
from email.mime.text import MIMEText

text = u"39,90\u20AC"
msg = MIMEText(text.encode("cp1252"), _charset="cp1251")
msg["Subject"] = "Hallo Welt"
msg["From"] = "Donald Duck <don@ld.de>"
msg["To"] = "Onkel Dagobert <d@gobert.de>"

print msg.as_string()
```

Als Erstes erzeugen wir einen Unicode-String, der das Euro-Zeichen enthält, das nicht im ASCII-Standard enthalten ist. Nachfolgend wird der Unicode-String ins Windows-Encoding cp1252 kodiert und bei der Instanziierung der Klasse MIME Text übergeben. Das verwendete Encoding muss dem Konstruktor ebenfalls über den Parameter _charset bekannt gemacht werden. Der nun folgende Teil des Programms ist bereits von den anderen Beispielen her bekannt.

Der MIME-kodierte Text, den das Beispielprogramm ausgibt, sieht folgendermaßen aus:

```
Content-Type: text/plain; charset="cp1252"
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>

MzksOTCA
```

Dabei entspricht MzksOTCA der MIME-Kodierung des Texts 39,90€.

Es kann durchaus vorkommen, dass auch Einträge im Header der E-Mail Sonderzeichen enthalten. Solche können mithilfe der Klasse Header kodiert werden:

```
from email.mime.text import MIMEText
from email.header import Header
msg = MIMEText("Hallo Welt")
msg["Subject"] = Header("39,90\u20AC", "cp1252")
[…]
```

Eine E-Mail einlesen

Zum Schluss möchten wir noch ein kurzes Beispiel dazu geben, dass eine abgespeicherte E-Mail auch wieder eingelesen und automatisch zu der bislang besprochenen Klassenstruktur aufbereitet werden kann. Dazu folgendes Beispiel:

```
import email

mail = """Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>

Hallo Welt
"""

msg = email.message_from_string(mail)
print msg["From"]
```

Im Beispielprogramm ist eine E-Mail in Form eines Strings vorhanden und wird durch die Funktion message_from_string eingelesen. Diese Funktion gibt eine vollwertige Message-Instanz zurück, wie die darauf folgende print-Ausgabe beweist:

```
Donald Duck <don@ld.de>
```

Alternativ hätte auch die Funktion message_from_file verwendet werden können, um die E-Mail aus einer Datei zu lesen. Dieser Funktion hätten wir dann ein geöffnetes Dateiobjekt übergeben müssen.

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

21.5 Automatisiertes Testen ▼

Pythons Standardbibliothek stellt zwei Module zur testgetriebenen Entwicklung (engl. *test-driven development*) bereit. Unter *testgetriebener Entwicklung* versteht man eine Art der Programmierung, bei der viele kleine Abschnitte des Programms, sogenannte *Units*, durch automatisierte Testdurchläufe auf Fehler geprüft werden. Bei der testgetriebenen Entwicklung wird das Programm nach kleineren, in sich geschlossenen Arbeitsschritten so lange verbessert, bis es wieder alle bisherigen und alledazu gekommenen Tests besteht. Auf diese Weise können sich durch das Hinzufügen von neuem Code keine Fehler in alten, bereits getesteten Code einschleichen.

In Python ist das möglicherweise bekannte Konzept der Unit Tests im Modul `unittest` implementiert. Das Modul `doctest` ermöglicht es, Testfälle innerhalb eines Docstrings, beispielsweise einer Funktion, unterzubringen. Im Folgenden werden wir uns zunächst mit dem Modul `doctest` beschäftigen, um danach zum Modul `unittest` voranzuschreiten.

top

21.5.1 Testfälle in Docstrings – doctest ▼▲

Das Modul `doctest` erlaubt es, Testfälle innerhalb des Docstrings einer Funktion, Methode, Klasse oder eines Moduls zu erstellen, die beim Aufruf der im Modul `doctest` enthaltenen Funktion `testmod` getestet werden. Die Testfälle innerhalb eines Docstrings werden dabei nicht in einer komplizierten neuen Definitionssprache verfasst, sondern können direkt aus einer Sitzung im interaktiven Modus in den Docstring kopiert werden.

Beachten Sie bei der Verwendung von Doctests, dass Docstrings auch bzw. hauptsächlich für die Dokumentation beispielsweise einer Funktion gedacht sind. Aus diesem Grund sollten Sie die Testfälle im Docstring möglichst einfach und lehrreich halten, sodass der resultierende Docstring auch in Dokumentationen Ihres Programms verwendet werden kann.

Das folgende Beispiel soll die Verwendung des Moduls `doctest` anhand der Funktion `fak` erläutern, die die Fakultät einer ganzen Zahl berechnen und zurückgeben soll.

```
import doctest
import math

def fak(n):
    """
        Berechnet die Fakultaet einer ganzen Zahl.

        >>> fak(5)
        120
        >>> fak(10)
        3628800
        >>> fak(20)
        2432902008176640000L

        Es muss eine positive ganze Zahl uebergeben werden.

        >>> fak(-1)
        Traceback (most recent call last):
        ...
        ValueError: Keine negativen Zahlen!
        >>> fak(1.5)
        Traceback (most recent call last):
        ...
        ValueError: Keine Gleitkommazahlen!
    """
    res = 1
    for i in xrange(2, n+1):
        res *= i
    return res

if __name__ == "__main__":
    doctest.testmod()
```

Im Docstring der Funktion `fak` steht zunächst ein erklärender Text. Dann folgt, durch eine leere Zeile davon abgetrennt, ein Auszug aus Pythons interaktivem Modus, in dem Funktionsaufrufe von `fak` mit ihren Rückgabewerten stehen. Diese Testfälle werden beim Ausführen des Tests nachvollzogen und entweder für wahr oder für falsch befunden.

Auf diese einfachen Fälle folgen, jeweils durch eine Leerzeile eingeleitet, ein weiterer erklärender Text sowie zwei Ausnahmefälle, in denen eine negative Zahl bzw. eine Gleitkommazahl übergeben wurden. Beachten Sie, dass Sie den Stacktrace eines auftretenden Tracebacks im Docstring weglassen können. Auch die im Beispiel stattdessen geschriebenen Auslassungszeichen sind optional.

Die beiden letzten Testfälle wurden in der Funktion noch nicht berücksichtigt, sodass diese im nun durchzuführenden Test fehlschlagen sollten. Um den Test zu starten, muss die Funktion `testmod` des Moduls `doctest` aufgerufen werden. Aufgrund der `if`-Abfrage

```
if __name__ == "__main__":
    doctest.testmod()
```

wird diese Funktion immer dann aufgerufen, wenn die Programmdatei direkt ausgeführt wird. Der Test wird hingegen nicht durchgeführt, wenn die Programmdatei von einem anderen Python-Programm als Modul eingebunden wird.

Im provozierten Fehlerfall lautet das Testresultat folgendermaßen. Zur Rettung des Regenwaldes haben wir uns hier auf das Abdrucken eines der fehlgeschlagenen Testfälle beschränkt.

```
**********************************************************************
File "fak.py", line 21, in __main__.fak
Failed example:
    fak(1.5)
Expected:
    Traceback (most recent call last):
    ...
    ValueError: Keine Gleitkommazahlen!
Got:
    1
**********************************************************************
```

```
1 items had failures:
   2 of   5 in __main__.fak
***Test Failed*** 2 failures.
```

Wenn wir die Funktion dahingehend korrigieren, dass sie im Falle unpassender Parameter die erwarteten Exceptions wirft:

```
def fak(n):
    """
    """ […]
    if n < 0:
        raise ValueError("Keine negativen Zahlen!")

    if math.floor(n) != n:
        raise ValueError("Keine Gleitkommazahlen!")

    res = 1
    for i in xrange(2, n+1):
        res *= i
    return res
```

werden bei erneutem Durchführen des Tests keine Fehler mehr angezeigt. Um genau zu sein: Es wird überhaupt nichts angezeigt. Das liegt daran, dass generell nur fehlgeschlagene Testfälle auf dem Bildschirm ausgegeben werden. Sollten Sie auch auf die Ausgabe geglückter Testfälle bestehen, können Sie die Programmdatei mit der Option `-v` (für *verbose*) starten.

Bei der Verwendung von Doctests sollten Sie unbedingt beachten, dass die in den Docstrings geschriebenen Vorgaben Zeichen für Zeichen mit den Ausgaben der intern ausgeführten Testfälle verglichen werden. Dabei sollten Sie immer im Hinterkopf behalten, dass die Ausgaben bestimmter Datentypen nicht immer gleich sind. So stehen beispielsweise die Schlüssel/Wert-Paare eines Dictionarys in keiner garantierten Reihenfolge, sodass Sie innerhalb eines Doctests nie ein Dictionary als Ergebnis ausgeben sollten. Des Weiteren gibt es Informationen, die vom Interpreter oder anderen Gegebenheiten abhängen, beispielsweise entspricht die Identität einer Instanz intern ihrer Speicheradresse und wird sich deswegen natürlich beim Neustart des Programms ändern.

Eine weitere Besonderheit, auf die Sie achten müssen, ist, dass eine Leerzeile in der erwarteten Ausgabe einer Funktion durch den String `<BLANKLINE>` gekennzeichnet werden muss, da eine Leerzeile als Trennung zwischen Testfällen und Dokumentation fungiert.

Flags

Um einen Testfall genau an Ihre Bedürfnisse anzupassen, können Sie sogenannte *Flags* vergeben. Das sind Einstellungen, die Sie aktivieren oder deaktivieren können. Ein Flag wird in Form eines Kommentars hinter den Testfall im Docstring geschrieben. Wird das Flag von einem Plus (+) eingeleitet, wird es aktiviert, bei einem Minus (-) deaktiviert. Bevor wir zu einem konkreten Beispiel kommen, sollen die beiden wichtigsten Flags eingeführt werden.

Flag	Bedeutung
`ELLIPSIS`	Wenn dieses Flag gesetzt ist, kann die Angabe `...` für eine beliebige Ausgabe einer Funktion verwendet werden. So können veränderliche Angaben wie Speicheradressen oder Ähnliches in größeren Ausgaben überlesen werden.
`NORMALIZE_WHITESPACES`	Wenn dieses Flag gesetzt ist, werden Whitespace-Zeichen nicht mit in den Ergebnisvergleich einbezogen. Das ist besonders dann interessant, wenn man ein langes Ergebnis auf mehrere Zeilen umbrechen möchte.

Tabelle 21.4 Doctest-Flags

In einem einfachen Beispiel möchten wir den Doctest der bereits bekannten Fakultätsfunktion um die Berechnung der Fakultät einer relativ großen Zahl erweitern. Da es müßig wäre, alle Stellen des Ergebnisses im Doctest anzugeben, soll die Zahl mithilfe des Flags

ELLIPSIS gekürzt angegeben werden:

```
def fak(n):
    """
        Berechnet die Fakultaet einer ganzen Zahl.

        >>> fak(1000) # doctest: +ELLIPSIS
        402387260077093773543702...000L
    """
    res = 1
    for i in xrange(2, n+1):
        res *= i
    return res

if __name__ == "__main__":
    doctest.testmod()
```

Das Setzen des Flags wurde fett hervorgehoben.

Bleibt noch zu sagen, dass insbesondere die Funktion testmod eine Fülle von Möglichkeiten bietet, die Testergebnisse im Programm zu verwenden oder den Prozess des Testens an Ihre Bedürfnisse anzupassen. Sollten Sie daran interessiert sein, bietet sich die Python-Dokumentation an, in der die Funktion besprochen wird.

top

21.5.2 Unit Tests – unittest

Das zweite Modul zur testgetriebenen Entwicklung heißt unittest und ist ebenfalls in der Standardbibliothek enthalten. Das Modul unittest implementiert die Funktionalität des aus Java bekannten Moduls JUnit, das den De-facto-Standard zur testgetriebenen Entwicklung in Java darstellt.

Der Unterschied zum Modul doctest besteht darin, dass die Testfälle bei unit test außerhalb des eigentlichen Programmcodes in einer eigenen Programmdatei in Form von regulärem Python-Code definiert werden. Das vereinfacht die Ausführung der Tests und hält die Programmdokumentation sauber. Umgekehrt ist mit dem Erstellen der Testfälle allerdings mehr Aufwand verbunden.

Um einen neuen Testfall bei unittest zu erstellen, muss eine von der Basisklasse unittest.TestCase abgeleitete Klasse erstellt werden, in der einzelne Testfälle als Methoden implementiert sind. Die folgende Klasse implementiert die gleichen Testfälle, die wir im vorherigen Kapitel mit dem Modul doctest durchgeführt haben. Dabei muss die zu testende Funktion fak in der Programmdatei *fak.py* implementiert sein, die von unserer Test-Programmdatei als Modul eingebunden wird.

```
import unittest
import fak

class MeinTest(unittest.TestCase):

    def testberechnung(self):
        self.failUnlessEqual(fak.fak(5), 120)
        self.failUnlessEqual(fak.fak(10), 3628800)
        self.failUnlessEqual(fak.fak(20), 2432902008176640000L)

    def testausnahmen(self):
        self.failUnlessRaises(ValueError, fak.fak, -1)
        self.failUnlessRaises(ValueError, fak.fak, 1.5)

if __name__ == "__main__":
    unittest.main()
```

Es wurde eine Klasse namens MeinTest erzeugt, die von der Basisklasse unit test.TestCase erbt. In der Klasse MeinTest wurden zwei Testmethoden namens testberechnung und testausnahmen implementiert. Beachten Sie, dass der Name solcher Testmethoden mit test beginnen muss, damit sie später auch tatsächlich zum Testen gefunden und ausgeführt werden.

Innerhalb der Testmethoden werden die Methoden failUnlessEqual bzw. fail UnlessRaises verwendet, die den Test fehlschlagen lassen, wenn die beiden angegebenen Werte nicht gleich sind bzw. wenn die angegebene Exception nicht geworfen wurde.

Um den Testlauf zu starten, wird die Funktion unittest.main aufgerufen. Die Fallunterscheidung

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

```
    unittest.main()
```

bewirkt, dass der Unit Test nur durchgeführt wird, wenn die Programmdatei direkt ausgeführt wird, und ausdrücklich nicht, wenn die Programmdatei als Modul in ein anderes Python-Programm importiert wurde. Die aufgerufene Funktion `unittest.main` erzeugt, um den Test durchzuführen, Instanzen aller Klassen, die im aktuellen Namensraum existieren und von `unittest.TestCase` erben. Dann werden alle Methoden dieser Instanzen aufgerufen, deren Namen mit `test` beginnen.

Die Ausgabe des Beispiels lautet im Erfolgsfall:

```
..
----------------------------------------------------------------
-------
Ran 2 tests in 0.000s

OK
```

Dabei stehen die beiden Punkte zu Beginn für zwei erfolgreich durchgeführte Tests. Ein fehlgeschlagener Test würde durch ein `F` gekennzeichnet.

Im Fehlerfall wird die genaue Bedingung angegeben, die zum Fehler geführt hat:

```
.F
======================================================================
FAIL: testberechnung (__main__.MeinTest)
----------------------------------------------------------------
-------
Traceback (most recent call last):
  File "testen.py", line 7, in testberechnung
    self.failUnlessEqual(fak.fak(5), 12)
AssertionError: 120 != 12

----------------------------------------------------------------
-------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Die Klasse TestCase

An dieser Stelle sollen die wichtigsten Methoden der Klasse `TestCase` des Moduls `unittest` besprochen werden. Die hier vorgestellten Methoden können entweder in einer von `TestCase` abgeleiteten Klasse implementiert werden oder können für die Tests selbst aufgerufen werden.

Im Folgenden sei *tc* eine Instanz der Klasse `TestCase`.

tc.setUp()

Die Methode `setUp` wird vor jedem Aufruf einer der implementierten Testmethoden aufgerufen und kann somit für den Test benötigten Intialisierungscode enthalten. Eine in der Methode `setUp` geworfene Exception wird als Error in den Testbericht eingetragen, und der Test wird abgebrochen.

tc.tearDown()

Die Methode `tearDown` wird nach jedem Aufruf einer der implementierten Testmethoden aufgerufen und kann somit abschließenden Code enthalten. Eine in der Methode `tearDown` geworfene Exception wird als Error in den Testbericht eingetragen.

Die nun folgenden Methoden können innerhalb der Testmethoden verwendet werden, um die Testbedingungen festzulegen. Beachten Sie, dass es jeweils zwei äquivalente Methoden gibt, eine `assert`- und eine `fail`-Methode. Der Unterschied liegt allein in der Namensgebung, in der die Bedeutung der Methode für den Testfall entweder positiv (`assert`-Methoden) oder negativ (`fail`-Methoden) formuliert ist.

tc.assert_(expr[, msg]), tc.failUnless(expr[, msg])

Der Test schlägt fehl, wenn *expr* `False` ergibt. Der optionale Parameter *msg* kennzeichnet die Beschreibung, die für diesen Fehler angezeigt werden soll. Jede der kommenden Methoden verfügt über diesen

Parameter und verwendet ihn mit der gleichen Bedeutung. Wir werden daher im Folgenden nicht mehr auf den Parameter *msg* eingehen.

tc.assertEqual(first, second[, msg]), tc.failUnlessEqual(first, second[, msg])

Der Test schlägt fehl, wenn die Parameter *first* und *second* nicht gleich sind. Beachten Sie, dass auf Gleichheit und nicht auf Identität geprüft wird.

tc.assertNotEqual(first, second[, msg]), tc.failIfEqual(first, second[, msg])

Der Test schlägt fehl, wenn die Parameter *first* und *second* gleich sind. Beachten Sie, dass auf Gleichheit und nicht auf Identität geprüft wird.

tc.assertAlmostEqual(first, second[, places[, msg]])
tc.failUnlessAlmostEqual(first, second[, places[, msg]])

Rundet *first* und *second* auf *places* Stellen genau und vergleicht die beiden Werte. Der Test schlägt fehl, wenn die gerundeten Werte nicht gleich sind.

tc.assertNotAlmostEqual(first, second[, places[, msg]]),
tc.failIfAlmostEqual(first, second[, places[, msg]])

Rundet *first* und *second* auf *places* Stellen genau und vergleicht die beiden Werte. Der Test schlägt fehl, wenn die gerundeten Werte gleich sind.

tc.assertRaises(exception, callable, ...), tc.failUnlessRaises(exception, callable, ...)

Der Test schlägt fehl, wenn die Funktion *callable* nicht die Exception *exception* wirft, falls sie mit den Parametern aufgerufen wird, die anstelle der Auslassungszeichen (...) stehen.

tc.failIf(expr[, msg])

Der Test schlägt fehl, wenn *expr* `True` ergibt.

tc.fail([msg])

Der Test schlägt bedingungslos fehl.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

23.5 C

Grundsätzlich gilt, dass ein in Python geschriebener Programmteil, auch wenn er noch so optimiert wurde, in puncto Geschwindigkeit niemals ein vergleichbares C-Programm schlagen kann. Aus diesem Grund sollten Sie an einer laufzeitkritischen Stelle so oft wie möglich auf Algorithmen zurückgreifen, die in C implementiert wurden. So lohnt es sich beispielsweise immer, eine Built-in Function einzusetzen, anstatt den entsprechenden Algorithmus selbst zu implementieren. Beachten Sie dazu, dass Teile der Standardbibliothek in C implementiert sind. Üblicherweise erkennen Sie solche Module an dem Präfix `c`, wie beispielsweise bei den Modulen `cStringIO` oder `cProfile`.

In Kapitel 26 wird behandelt, wie Sie eigene Module oder Programmteile in C schreiben können.

Sie könnten eine `for`-Schleife zum Beispiel möglicherweise durch einen Aufruf der Built-in Function `map` ersetzen. Beachten Sie, dass eine List Comprehension ähnlich laufzeiteffizient ist wie der Aufruf der Funktion `map`.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

24.5 Überblick über das Qt-Framework

Nachdem das Kapitel über grafische Benutzeroberflächen mit einem praxisorientierten Einstieg in das Qt-Framework begonnen hat, möchten wir in diesem Abschnitt einen Überblick darüber geben, wie das Framework an sich aufgebaut ist und welchen Funktionsumfang es tatsächlich hat. Sie sollten dieses Kapitel also lesen, um sich später im Qt-Framework zurechtzufinden. Wir beginnen mit einer Übersicht über die Namensräume des Qt-Frameworks. Zwei dieser Namensräume, nämlich `PyQt4.QtGui` und `PyQt4.QtCore`, haben wir bereits im letzten Abschnitt verwendet. Allgemein kapseln diese Namensräume bestimmte Teile der Klassenbibliothek Qt in thematisch zusammengehörige Gruppen.

Beachten Sie, dass Qt für die Programmiersprache C++ entwickelt wird, die bei Weitem nicht über eine so umfangreiche Standardbibliothek verfügt, wie sie bei Python vorhanden ist. Aus diesem Grund sind im Qt-Framework viele Klassen enthalten, die nichts mit grafischen Benutzeroberflächen zu tun haben, sondern verschiedene Zwecke, wie beispielsweise den Zugriff auf eine Datenbank oder das Einlesen von XML-Daten, haben. Viele dieser nicht GUI-spezifischen Klassen sind bei Python bereits durch die Standardbibliothek abgedeckt und daher im Zusammenhang mit PyQt weniger interessant. Trotzdem listet die folgende Tabelle der Vollständigkeit halber alle Namensräume auf, aus denen das Qt-Framework besteht.

Namensraum	Beschreibung
`PyQt4`	Enthält alle Namensräume des Qt-Frameworks
`PyQt4.QtCore`	Enthält alle nicht-GUI-bezogenen Klassen des Qt-Frameworks, die eine Kern-Funktionalität implementieren und in vielen Situationen benötigt werden. Beispielsweise sind in diesem Namensraum Klassen für Threads, reguläre Ausdrücke oder Unicode enthalten.
`PyQt4.QtCore.Qt`	Dieser Namensraum gehört zwar zu `PyQt4.QtCore`, ist aber so wichtig, dass er getrennt besprochen werden soll. Im Namensraum `PyQt4.QtCore.Qt` sind alle symbolischen Konstanten enthalten, die in Qt verwendet werden. So könnte beispielsweise die symbolische Konstante `PyQt4.QtCore.Qt.Vertical` verwendet werden, um einem Fortschrittsbalken eine vertikale Ausrichtung zu geben.
`PyQt4.QtGui`	Enthält alle Klassen des Qt-Frameworks, die sich auf die grafische Benutzeroberfläche beziehen.
`PyQt4.QtNetwork`	Enthält alle Klassen, die zur Netzwerkkommunikation benötigt werden.
`PyQt4.QtOpenGL`	Enthält Klassen, die zur Darstellung von 3D-Szenen mit OpenGL verwendet werden können.
`PyQt4.QtScript`	Enthält Klassen, die eine Scripting-Funktionalität für Qt-Programme mittels JavaScript bereitstellen. Dies ist im Zusammenhang mit Python eher uninteressant.
	Enthält Klassen, die zum Umgang mit SQL-

PyQt4.QtSql	Datenbanken gedacht sind.
PyQt4.QtTest	Enthält Klassen zum Durchführen eines Unit Tests in der Qt-Umgebung. Dabei kann ein virtueller Benutzer simuliert werden, sodass die grafische Benutzeroberfläche automatisch verschiedenen Tests unterzogen werden kann.
PyQt4.QtXml	Enthält Klassen, die zum Umgang mit XML-Daten gedacht sind.
PyQt4.Qt	Enthält alle Klassen des Qt-Frameworks. Durch Einbinden dieses Namensraums können Sie die vollständige Funktionalität von Qt nutzen, ohne sich Gedanken über das Importieren des jeweiligen Namensraums machen zu müssen. Beachten Sie aber, dass dann auch das vollständige Qt-Framework geladen und im Speicher gehalten wird.

Tabelle 24.2 Namensräume des Qt-Frameworks

<< zurück <top> vor >>

Inhaltsverzeichnis

Download:
- ZIP, ca. 4,8 MB

Buch bestellen

Ihre Meinung?

Python von Peter Kaiser, Johannes Ernesti

Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

25.5 Djangos Administrationsoberfläche

Eine der zeitaufwendigsten Aufgaben bei der Erstellung einer Webanwendung ist die Entwicklung einer sogenannten *Administrationsoberfläche*, kurz *ACP* (engl. *Admin Control Panel*). ACPs sind Werkzeuge, die den Betreibern einer Seite das nachträgliche Verändern der Seiteninhalte ermöglichen, ohne den Programmcode zu verändern oder direkt auf die Datenbank zuzugreifen.

Beispielsweise sollte ein ACP einer Nachrichtenseite neue Meldungen hinzufügen und alte bearbeiten oder löschen können. Ebenso wäre eine Administrationsoberfläche für die Erstellung von Umfragen zu aktuellen Themen denkbar.

Im Prinzip ist ein ACP eine eigene Webanwendung, mit der sich alle Daten der zu administrierenden Anwendung bearbeiten lassen. Dementsprechend hoch ist auch der Entwicklungsaufwand.

Die gute Nachricht für Sie als angehenden Django-Entwickler ist, dass Sie sich in Ihren Projekten nur wenig um die Programmierung von ACPs kümmern müssen. Django erstellt fast vollautomatisch eine komfortable und zweckmäßige Administrationsoberfläche, sodass Sie von der lästigen Eigenimplementation verschont bleiben. Sie müssen nur kleine Änderungen an Ihrem Datenmodell und der Konfiguration des Projekts vornehmen.

Zuerst müssen Sie die Applikation mit dem Namen `"django.contrib.admin"` in das `INSTALLED_APPS`-Tupel der *settings.py* Ihres Projekts einfügen, die Django bereits mitliefert:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'unser_projekt.news',
```

```
    'django.contrib.admin'
)
```

Da die Administrationsapplikation zusätzliche Tabellen in der Datenbank benötigt, müssen Sie noch einmal das *manage.py*-Script mit dem Parameter syncdb starten:

```
$ python manage.py syncdb
```

In der *models.py* muss jede Modellklasse, für die es ein ACP geben soll, in ihrem Körper eine Klasse namens `Admin` definieren, die im einfachsten Fall nur ein `pass` enthält. Im ersten Schritt machen wir nur die `Meldung`-Klasse über das ACP zugänglich: [Lassen Sie sich nicht durch das fehlende `(object)` verwirren. Django verwendet noch *old-style classes*.]

```
class Meldung(models.Model):
    ...
    class Admin:
        pass
```

Die letzte zur Einrichtung eines ACPs notwendige Änderung betrifft die Datei *urls.py*, in der wir das Kommentarzeichen für eine bestimmte Zeile entfernen müssen. Die fertige *urls.py* sieht dann so aus:

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    # Uncomment this for admin:
    (r'^admin/', include('django.contrib.admin.urls')),
)
```

Auch wenn Sie jetzt wahrscheinlich noch nicht verstehen, was der Inhalt der Datei konkret bedeutet, ist das kein Problem. Wir werden sehr bald die Funktionsweise dieser Datei im Detail behandeln.

Nun können wir bewundern, was Django als Administrationsoberfläche für unsere Anwendung gezaubert hat. Dazu starten Sie einfach den Entwicklungswebserver mit `python manage.py runserver` und besuchen mit Ihrem Webbrowser die Adresse http://127.0.0.1:8000/admin. Beim Login-Prompt geben Sie die Benutzerdaten ein, die Sie beim ersten Lauf von `python manage.py syncdb` angegeben haben. Anschließenden präsentiert sich Ihnen eine schicke Administrationsmaske:

Hier klicken, um das Bild zu vergrößern

Abbildung 25.7 Startseite des Django-ACPs

Uns interessiert hier nur die unterste der drei Sektionen, mit deren Hilfe wir unsere News-Meldungen bearbeiten können. Wenn Sie auf »Meldungs« [Django bildet den Plural immer durch Anhängen eines »s«. Dies ist manchmal nicht sehr sinnvoll, wie obiges Beispiel zeigt.] klicken, gelangen Sie zur Übersicht aller gespeicherten Meldungen (siehe Abbildung 25.8).

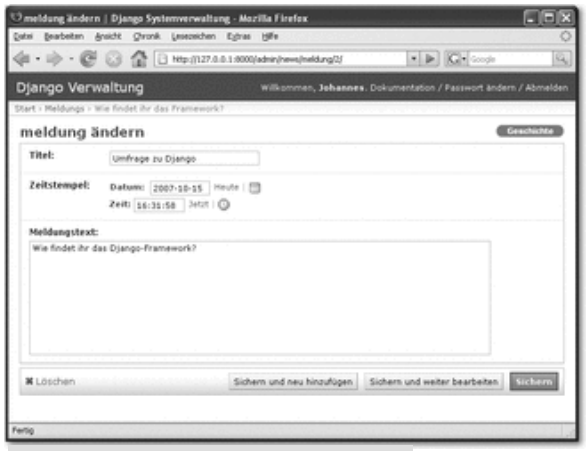
Hier klicken, um das Bild zu vergrößern

Abbildung 25.8 Alle gespeicherten Meldungen

Über die Administrationsoberfläche können Sie jede Meldung editieren, neue Meldungen hinzufügen und Meldungen löschen. Wie Sie sehen können, hat Django für die Anzeige der Meldungen die `__unicode__`-Methode des Models verwendet.

Da die Handhabung des ACPs sehr einfach ist, wollen wir uns als Beispiel mit dem Verändern einer Meldung begnügen.

Wir klicken beispielsweise auf die Meldung mit dem Text »Wie findet ihr das Framework?« und gelangen so zur Editieransicht. Dort werden wir als Demonstration den Text in »Wie findet ihr das Django-Framework?« umändern (siehe Abbildung 25.9).



Hier klicken, um das Bild zu vergrößern

Abbildung 25.9 Editieren ener Meldung

Nach dem Klick auf die Schaltfläche **Sichern** in der Ecke rechts unten ist die Meldung mit ihrem neuen Text in der Datenbank gespeichert, und Sie gelangen wieder zur Meldungsübersicht.

Als interessantes Feature speichert Django zu jedem Datensatz seine »Veränderungsgeschichte«, also wann wer welche Änderung an den Daten vorgenommen hat. Dieses Extra ist vor allem dann nützlich, wenn viele Leute Zugang zu der Administrationsseite haben und man den Überblick über die vorgenommenen Änderungen behalten möchte. Die Veränderungen zu einem bestimmten Datensatz können über die Schaltfläche **Geschichte** abgerufen werden.

Sie sollten sich vor Augen halten, welchen großen Nutzen dieser Service von Django hat: Aus einer einfachen Model-Definition werden die gesamte Datenbank und sogar das dazugehörige Verwaltungswerkzeug generiert. Diese Arbeit hätte Sie sonst Stunden gekostet.

Natürlich bietet das Django-ACP unzählige weitere Konfigurationsmöglichkeiten, die im Rahmen dieses Buchs nicht besprochen werden können. Wir verweisen Sie wieder einmal auf die sehr gute Dokumentation der Django-Homepage (http://www.djangoproject.com), wenn Sie weitere Informationen

suchen.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

27.5 Kommandozeilen-Interpreter – cmd

Das Modul cmd bietet eine einfache und abstrahierte Schnittstelle zum Schreiben eines zeilenorientierten Kommandointerpreters. Unter einem zeilenorientierten Kommandointerpreter versteht man eine interaktive Konsole, in der zeilenweise Kommandos eingegeben und direkt nach Bestätigung der Eingabe interpretiert werden. Der interaktive Modus ist ein bekanntes Beispiel für solch einen Kommandointerpreter. In einem eigenen Projekt ließe sich cmd beispielsweise für eine Administrator-Konsole verwenden.

Im Modul cmd ist die Klasse Cmd enthalten, die als Basisklasse für eigene Kommandointerpreter verwendet werden kann und dafür ein grobes Gerüst bereitstellt. Da Cmd als Basisklasse gedacht ist, macht es keinen Sinn, die Klasse direkt zu instanziieren. Im folgenden Beispielprojekt wird die Klasse Cmd verwendet, um eine rudimentäre Konsole zu erstellen. Die Konsole soll die beiden Kommandos date und time verstehen und jeweils das aktuelle Datum bzw. die aktuelle Uhrzeit ausgeben.

```
import cmd
import time

class MeineKonsole(cmd.Cmd):

    def __init__(self):
        cmd.Cmd.__init__(self)
        self.prompt = ">>> "

    def do_date(self, prm):
        d = time.localtime()
        print "Heute ist der %d.%d.%d" % (d[2], d[1],
d[0])
        return False

    def help_date(self):
        print "Gibt das aktuelle Datum aus"

    def do_time(self, prm):
        z = time.localtime()
        print "Es ist %d:%d:%d Uhr" % (z[3], z[4], z[5])
        return False

    def do_exit(self, prm):
```

```
        print "Auf Wiedersehen"
        return True
```

Im Beispiel wurde die Klasse MeineKonsole definiert, die von cmd.Cmd abgeleitet ist. Im Konstruktor der Klasse wird die Basisklasse initialisiert und das Attribut self.prompt gesetzt. Dieses Attribut stammt von der Basisklasse und referenziert den String, der zur Eingabe eines Kommandos auffordern soll.

Unsere Konsole soll insgesamt drei Kommandos unterstützen: date zum Ausgeben des aktuellen Datums, time zum Ausgeben der aktuellen Uhrzeit und exit zum Beenden der Konsole. Um ein Kommando in einer cmd.Cmd-Konsole zu implementieren wird einfach eine Methode do_kommando angelegt, wobei kommando durch den Namen des jeweiligen Kommandos ersetzt werden muss. In diesem Sinne finden Sie in der Klasse MeineKonsole die Methoden do_date, do_time und do_exit für die drei verfügbaren Kommandos. Jede dieser Methoden wird aufgerufen, wenn das Kommando vom Benutzer eingegeben wurde, und bekommt als einzigen Parameter prm den String übergeben, den der Benutzer hinter das Kommando geschrieben hat. Die Beispielimplementation der Methoden ist denkbar einfach und braucht in dieser Stelle nicht näher erläutert zu werden.

Wichtig ist aber, dass eine Kommandomethode anhand des Rückgabewerts angibt, ob die Konsole nach diesem Kommando noch weitere Kommandos annehmen soll. Wenn die Methode False zurückgibt, werden weitere Kommandos entgegengenommen. Bei einem Rückgabewert von True wird die Kommandoschleife beendet. Beachten Sie, dass der Rückgabewert False einiger Methoden im obigen Beispiel eigentlich überflüssig ist, da eine Funktion oder Methode ohne Rückgabewert implizit None zurückgibt und der Wahrheitswert von None False ist. Dennoch ist die entsprechende return-Anweisung zu Demonstrationszwecken im Quellcode enthalten.

Zusätzlich zu den Kommandomethoden existiert eine Methode help_date als Beispielimplementation der interaktiven Hilfe, die die Klasse cmd.Cmd bereitstellt. Gibt der Benutzer der Konsole ein Fragezeichen oder den Befehl help, gefolgt von einem der verfügbaren Kommandonamen, ein, wird die Methode help_kommando mit dem entsprechenden Kommandonamen aufgerufen. Diese gibt dann einen kurzen erklärenden Text zu dem jeweiligen Kommando aus.

Um den obigen Code zu einem vollwertigen Programm zu machen, muss die Klasse instanziiert werden und die Kommandoschleife durch Aufruf der Methode cmdloop gestartet werden:

```
konsole = MeineKonsole()
konsole.cmdloop()
```

Nach dem Starten des Programms wird der Benutzer durch Ausgabe des Prompts >>> dazu aufgefordert, ein Kommando einzugeben. Eine Beispielsitzung in unserer Konsole sieht folgendermaßen aus:

```
>>> date
Heute ist der 25.9.2007
>>> time
Es ist 18:13:23 Uhr
>>> time
Es ist 18:13:26 Uhr
>>> exit
Auf Wiedersehen
```

Die Hilfetexte der Kommandos können folgendermaßen angezeigt werden:

```
>>> help date
Gibt das aktuelle Datum aus
>>> ? date
Gibt das aktuelle Datum aus
>>> help
```

```
Documented commands (type help <topic>):
========================================
date

Undocumented commands:
======================
exit  help  time
```

Der Befehl help ohne Parameter gibt eine Liste aller dokumentierten und undokumentierten Kommandos Ihrer Konsole aus.

Das hier vorgestellte Beispielprogramm versteht sich als eine einfache Möglichkeit, die Klasse Cmd zu verwenden. Neben der hier vorgestellten Funktionalität bietet die Klasse eine Fülle von Möglichkeiten, um das Verhalten der Konsole genau an Ihre Bedürfnisse anzupassen. Sollte Ihr Interesse an dem Modul cmd geweckt sein, bietet sich die Python-Dokumentation als Referenz für alle Methoden und Attribute der Klasse an.

Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar

Kommentar absenden

<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de

Python von Peter Kaiser, Johannes Ernesti
Das umfassende Handbuch - Aktuell zu Python 2.5

Python
gebunden, mit CD
819 S., 39,90 Euro
Galileo Computing
ISBN 978-3-8362-1110-9

top

A.5 Built-in Functions

Python enthält eine Menge Built-in Function, die aus didaktischen Gründen an verschiedenen Stellen des Buchs eingeführt wurden. Deshalb gibt es im Buch bislang keine Liste aller Built-in Functions. Die folgende *Tabelle* listet alle Built-in Functions mitsamt einer kurzen Beschreibung und einem Vermerk auf, wo die Funktion ausführlich behandelt wird. Beachten Sie, dass die Funktionen in dieser Tabelle ohne Parametersignaturen angegeben werden.

Built-in Function	Beschreibung	Erklärt auf
`__import__`	Bindet ein Modul oder Paket ein.	S. 226
`abs`	Berechnet den Betrag einer Zahl.	S. 195
`all`	Prüft, ob alle Elemente einer Sequenz `True` ergeben.	S. 195
`any`	Prüft, ob mindestens ein Element einer Sequenz `True` ergibt.	S. 196
`bool`	Erzeugt einen booleschen Wert.	S. 196
`callable`	Gibt an, ob eine Instanz aufrufbar ist.	–
`chr`	Gibt das Zeichen mit einem bestimmten ASCII-Code zurück.	S. 196
`classmethod`	Ähnlich wie `staticmethod`, aber die mit `classmethod` dekorierten Methoden erwarten als ersten Parameter die Klasse, mit der sie aufgerufen werden. Dieser Parameter heißt in der Regel `cls` und ist mit dem `self` von gewöhnlichen Methoden vergleichbar.	–
`cmp`	Vergleicht zwei Werte miteinander.	S. 196
`complex`	Erzeugt eine komplexe Zahl.	S. 197

delattr	Löscht ein bestimmtes Attribut einer Instanz.	–
dict	Erzeugt ein Dictionary.	S. 197
dir	Gibt eine Liste aller Attribute eines Objekts zurück.	–
divmod	Gibt ein Tupel mit dem Ergebnis einer Ganzzahldivision und dem Rest zurück. divmod(a, b) ist äquivalent zu (a // b, a % b)	S. 198
eval	Wertet einen Python-Ausdruck aus.	S. 306
execfile	Führt eine Python-Programmdatei aus.	–
file	Erzeugt ein Dateiobjekt.	S. 198
filter	Ermöglicht es, bestimmte Elemente einer Liste herauszufiltern.	S. 199
float	Erzeugt eine Gleitkommazahl.	S. 199
frozenset	Erzeugt eine unveränderliche Menge.	S. 199
getattr	Gibt ein bestimmtes Attribut einer Instanz zurück.	–
globals	Gibt ein Dictionary mit allen Referenzen des globalen Namensraums zurück.	S. 200
hasattr	Überprüft, ob eine Instanz über ein bestimmtes Attribut verfügt.	–
hash	Gibt den Hash-Wert einer Instanz zurück.	S. 200
help	Startet die eingebaute interaktive Hilfe von Python.	S. 200
hex	Gibt den Hexadezimalwert einer ganzen Zahl in Form eines Strings zurück.	S. 201
id	Gibt die Identität einer Instanz zurück.	S. 201
input	Liest einen Python-Ausdruck vom Benutzer ein.	S. 201
int	Erzeugt eine ganze Zahl mit begrenztem Zahlenraum.	S. 202
isinstance	Prüft, ob ein Objekt Instanz einer bestimmten Klasse ist.	–
issubclass	Prüft, ob eine Klasse von einer bestimmten Basisklasse erbt.	–
iter	Erzeugt ein Iterator-Objekt.	S. 299, 303
len	Gibt die Länge einer bestimmten Instanz zurück.	S. 202
list	Erzeugt eine Liste.	S. 202
locals	Gibt ein Dictionary zurück, das alle Referenzen des lokalen Namensraums enthält.	S. 202
long	Erzeugt eine ganze Zahl mit unbegrenztem Zahlenraum.	S. 203
map	Wendet eine Funktion auf jedes Element einer Liste an.	S. 203
max	Gibt das größte Element einer Sequenz zurück.	S. 204
min	Gibt das kleinste Element einer Sequenz zurück.	S. 205
object	Gibt die einfachste Form einer Instanz zurück, wenn sie aufgerufen wird. Außerdem dient object als Basisklasse für alle new-style classes.	S. 234, 249
oct	Gibt den Oktalwert einer ganzen Zahl in Form eines Strings zurück.	S. 205

open	Erzeugt ein Dateiobjekt.	S. 175, 205
ord	Gibt den Unicode-Code eines bestimmten Zeichens zurück.	S. 205
pow	Führt eine Potenzoperation durch.	S. 206
property	Erzeugt ein managed attribute.	S. 245
range	Erzeugt eine Liste mit fortlaufend numerischen Elementen.	S. 65 ,206, 299
raw_input	Liest einen String vom Benutzer ein.	S. 207
reduce	Reduziert die Werte einer Sequenz mithilfe einer Funktion auf einen einzigen Wert.	S. 207
reload	Bindet ein bereits importiertes Modul erneut ein.	S. 227
repr	Gibt eine String-Repräsentation einer Instanz zurück.	S. 208
reversed	Erzeugt einen Iterator, der ein iterierbares Objekt rückwärts durchläuft.	S. 209
round	Rundet eine Zahl.	S. 209
set	Erzeugt ein Set.	S. 155, 209
setattr	Setzt ein bestimmtes Attribut einer Instanz auf einen bestimmten Wert.	–
sorted	Erzeugt eine sortierte Liste aus den Elementen eines iterierbaren Objekts.	S. 209
staticmethod	Erzeugt eine statische Methode.	S. 246
str	Erzeugt einen String.	S. 210
sum	Gibt die Summe aller Elemente einer Sequenz zurück.	S. 211
super	Gibt die Superklasse eines Typs zurück.	–
tuple	Erzeugt ein Tupel.	S. 211
type	Gibt den Datentyp einer Instanz zurück.	S. 71, 211
unichr	Gibt das Unicode-Zeichen mit einem bestimmten Unicode-Code zurück.	–
unicode	Erzeugt einen Unicode-String.	S. 212
vars	Gibt ein Dictionary zurück, das den aktuellen Kontext enthält.	–
xrange	Erzeugt einen Iterator, der über fortlaufend numerische Werte iteriert.	S. 212
zip	Fasst mehrere Sequenzen zu Tupeln zusammen, um sie beispielsweise mit einer for-Schleife zu durchlaufen.	S. 212

Tabelle A.6 Built-in Functions in Python

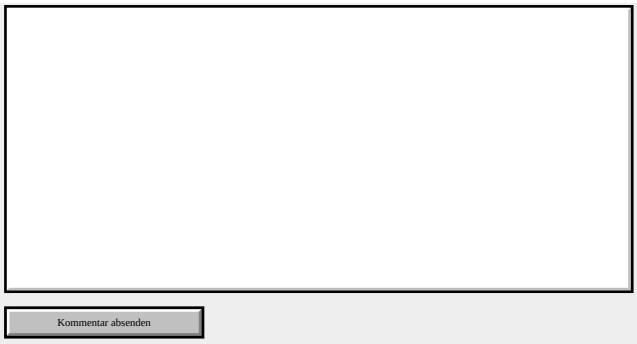
Ihr Kommentar

Wie hat Ihnen das <openbook> gefallen? Wir freuen uns immer über Ihre freundlichen und kritischen Rückmeldungen.

Name

E-Mail

Ihr Kommentar



<< zurück <top> vor >>

[Galileo Computing]

Galileo Press, Rheinwerkallee 4, 53227 Bonn, Tel.: 0228.42150.0, Fax 0228.42150.77, info@galileo-press.de