A Byte of Python

Deutsche Übersetzung

Swaroop C H

A Byte of Python: Deutsche Übersetzung

Swaroop C H
Deutsche Übersetzung: Bernd Hengelein, Lutz Horn, Bernhard Krieger, Christoph Zwerschke

Version 1.20, deutsche Übersetzung

Zusammenfassung

Dieses Buch wird Ihnen dabei helfen, die Programmiersprache Python zu lernen, unabhängig davon, ob Sie ein erfahrener Programmierer sind oder ob Sie zum ersten Mal mit Computern zu tun haben.

Die Zusammenarbeit an der deutschen Übersetzung dieses Buchs wurde von BerliOS [http://developer.berlios.de] freundlich unterstützt.

Inhaltsverzeichnis

Tabellenverzeichnis

Liste der Beispiele

Vorwort

Python ist eine der wenigen Programmiersprachen, die zugleich einfach und mächtig sind. Das ist sowohl für Anfänger als auch Experten gut, und - noch wichtiger - es macht Spaß damit zu programmieren. Das Ziel dieses Buches ist es, Ihnen beim Lernen dieser wunderbaren Programmiersprache zu helfen und Ihnen zu zeigen, wie sich Dinge schnell und leicht erledigen lassen - in der Tat 'Das perfekte Gegengift für Ihre Programmierprobleme'.

Für wen ist dieses Buch gedacht?

Dieses Buch dient als Handbuch oder Unterrichtskurs für die Programmiersprache Python. Es ist hauptsächlich für Anfänger gedacht, ist aber auch für erfahrene Programmierer nützlich.

Die Zielsetzung ist es, dass Sie Python mit diesem Buch lernen können, selbst wenn Ihr einziges Wissen über Computer darin besteht, wie man eine Textdatei abspeichert. Falls Sie bereits über Programmiererfahrung verfügen, können Sie aber ebenfalls Python mit diesem Buch lernen.

Falls Sie bereits über Programmiererfahrung verfügen, werden Sie sicher die Unterschiede zwischen Python und Ihrer bevorzugten Programmiersprache interessieren - ich habe viele solcher Unterschiede hervorgehoben. Eine Warnung jedoch: Python wird bald Ihre Lieblingsprogrammiersprache sein!

Geschichtsstunde

Ich habe zuerst mit Python angefangen, als ich ein Installationsprogramm für meine Software Diamond [http://www.g2swaroop.net/software/] schreiben musste, um dessen Installation zu vereinfachen. Ich musste mich zwischen Python- und Perl-Bindings für die Qt-Bibliothek entscheiden. Ich forschte ein wenig im Web nach und stieß auf einen Artikel, in dem Eric S. Raymond, der berühmte und respektierte Hacker, darüber schrieb, wie Python seine bevorzugte Programmiersprache geworden war. Ich fand auch heraus, dass die PyQT-Bindings verglichen mit Perl-Qt sehr gut waren. Daher entschied ich, dass Python die Sprache für mich sein würde.

Dann begann ich, nach einem guten Pythonbuch zu suchen. Ich konnte keines finden! Ich fand zwar einige O'Reilly-Bücher, aber sie waren entweder zu teuer oder eher Nachschlagewerke als Einführungskurse. Also begnügte ich mich mit der Dokumentation, die mit Python mitgeliefert wird. Sie war jedoch zu kurz und zu wenig umfangreich. Sie gab einen guten Einblick in Python, war aber nicht vollständig. Ich kam mit ihr zurecht, weil ich bereits über Programmiererfahrung verfügte, aber sie war ungeeignet für Anfänger.

Ungefähr sechs Monate nach meiner ersten Berührung mit Python installierte ich die damals aktuelle Version 9.0 von Red Hat Linux und spielte mit KWord herum. Ich war beeindruckt und bekam plötzlich die Idee, etwas über Python zu schreiben. Ich begann, einige Seiten zu schreiben, aber aus diesen wurden schnell 30 Seiten. Dann machte ich mir ernsthaft Gedanken darüber, wie ich das Material in eine Buchform bringen könnte. Nach *vielen* Neufassungen erreichte es einen Zustand, in dem es eine nützliche Anleitung für das Lernen von Python war. Ich betrachte dieses Buch als meinen Beitrag und meinen Tribut an die Open-Source-Gemeinde.

Dieses Buch begann als meine persönliche Notizsammlung zu Python, und ich betrachte es immer noch als eine solche, obwohl ich mir Mühe gegeben habe, das Buch auch für andere brauchbar und interessant zu gestalten :)

Entsprechend dem wahren Geist des Open-Source-Gedankens habe ich viele konstruktive Hinweise, viel Kritik und Rückmeldungen von enthusiastischen Lesern erhalten, die mir sehr bei der Verbesserung des Buchs geholfen haben.

Status des Buches

Dieses Buch ist eine **ständige Baustelle**. Viele Kapitel werden dauernd verändert und verbessert. Trotzdem ist das Buch um vieles reifer geworden. Sie sollten in der Lage sein, mit diesem Buch Python zu lernen. Bitte teilen Sie mir mit, wenn Sie irgendeinen Teil des Buchs fehlerhaft oder unverständlich finden.

Weitere Kapitel sind für die Zukunft geplant, etwa über wxPython, Twisted und vielleicht auch Boa Constructor.

Status der deutschen Übersetzung

Vorliegend finden Sie eine vollständige deutsche Übersetzung der Version 1.20 des Buchs.

Offizielle Website

Die offizielle Website des Buches ist www.byteofpython.info [http://www.byteofpython.info]. Auf der Website können Sie das gesamte Buch lesen, die letzte Version herunterladen, und mir auch Rückmeldungen schicken.

Website des Übersetzungs-Projekts

Die deutsche Übersetzung des Buches wurde im BerliOS [http://developer.berlios.de/]-Projekt abop-german [https://developer.berlios.de/projects/abop-german/] erstellt. Auf der Projektseite haben Sie Zugriff auf die deutsche Übersetzung in verschiedenen Formaten.

Lizenz

Rückmeldung

Ich habe mir große Mühe gegeben, das Buch so interessant und genau wie möglich zu machen. Falls Sie trotzdem Material finden, das inkonsistent oder ungenau ist, oder das einfach nur verbessert werden muss, dann informieren Sie mich bitte, so dass ich entsprechende Verbesserungen vornehmen kann. Sie können mich unter `<swaroop@byteofpython.info>` erreichen.

Rückmeldung zur deutschen Übersetzung

Falls Sie Rückmeldungen zur deutschen Übersetzung dieses Buches haben, falls Sie einen Tippfehler gefunden haben oder sonst etwas beitragen möchten, nehmen Sie bitte über die Website des Über-

setzungsprojekts abop-german [https://developer.berlios.de/projects/abop-german/] Kontakt mit den Übersetzern auf.

Etwas zum Nachdenken

> Es gibt zwei Arten, Software zu entwerfen: Eine ist, es so einfach zu machen, dass es offensichtlich keine Mängel gibt; die andere ist, es so kompliziert zu machen, dass es keine offensichtlichen Mängel gibt.
>
> —C. A. R. Hoare

> Erfolg im Leben ist eine Sache, die weniger mit Talent und Gelegenheit zu tun hat, als mit Konzentration und Beharrlichkeit.
>
> —C. W. Wendte

Kapitel 1. Einführung

Einführung

Python ist eine der wenigen vorhandenen Programmiersprachen, die zu Recht behaupten können, sowohl **einfach** als auch **mächtig** zu sein. Sie werden positiv überrascht darüber sein, wie einfach es in Python ist, sich auf die Lösung eines Problems zu konzentrieren, anstatt sich ständig mit den Tücken der Syntax und Struktur der Programmiersprache befassen zu müssen.

Offiziell wird Python wie folgt vorgestellt:

> Python ist eine einfach zu erlernende, mächtige Programmiersprache. Sie hat effiziente Datenstrukturen hoher Abstraktion und einen einfachen, aber effektiven Ansatz zur objektorientierten Programmierung. Die elegante Syntax und die dynamische Typisierung von Python machen sie zusammen mit der Lauffähigkeit in einem Interpreter zur idealen Sprache zum Schreiben von Programmskripts und für die schnelle Anwendungsentwicklung in vielen Bereichen auf den meisten Plattformen.

Ich werde die meisten dieser Eigenschaften detailliert im nächsten Abschnitt besprechen.

Anmerkung

Guido van Rossum, der Schöpfer der Sprache Python, benannte die Sprache nach der BBC-Show "Monty Python's Flying Circus". Er ist kein besonderer Freund von Schlangen, die ihre Beutetiere dadurch töten, dass sie ihre langen Körper um sie schlingen und sie erdrücken.

Eigenschaften von Python

Einfach	Python ist eine einfache und minimalistische Sprache. Ein gutes Python-Programm zu lesen kommt einem fast so vor, als würde man die englische Programmbeschreibung im Klartext lesen, wenngleich in einem sehr eingeschränktem englischen Wortschatz. Diese pseudocode-artige Wesen von Python ist eine ihrer größten Stärken. Dadurch wird es Ihnen ermöglicht, sich auf die Lösung Ihres Problems zu konzentrieren, anstatt auf die Sprache selbst.
Einfach zu lernen	Wie Sie sehen werden, ist es extrem einfach, in Python einzusteigen. Python hat, wie bereits erwähnt, eine außerordentlich einfache Syntax.
Freier und offener Quelltext	Python ist ein Beispiel für FLOSS (Free/Libre and Open Source Software - Freie Software mit offengelegtem Quelltext). Einfach gesagt können Sie frei Kopien solcher Software weitergeben, ihren Quelltext lesen, sie verändern und Teile von ihr in neuer Software verwenden. Außerdem werden Sie darüber aufgeklärt, dass Sie all dies machen dürfen. FLOSS basiert auf dem Konzept einer Gemeinschaft, die Wissen austauscht. Das ist einer der Gründe, warum Python so gut ist - es wurde von einer Gemeinschaft geschaffen und ständig verbessert, die einfach nur ein immer besseres Python haben möchte.
Hochsprache	Wenn Sie Programme in Python schreiben, müssen Sie sich nie um die "niedrigen Dinge" kümmern, d.h. solche Details wie etwa die Verwaltung des von Ihrem Programm verwendeten Speichers.
Portierbar	Aufgrund seiner Open-Source-Natur wurde Python auf viele Plattformen portiert (d.h. verändert, um auf ihnen zu laufen).

Alle Ihre Python-Programme können auf diesen Plattformen laufen, ohne dass Sie irgendwelche Änderungen an ihnen vornehmen müssen, vorausgesetzt, Sie sind vorsichtig genug, systemabhängige Funktionen zu vermeiden.

Sie können Python auf folgenden Plattformen verwenden: Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE und sogar PocketPC!

Interpretiert

Das bedarf einer kleinen Erklärung.

Ein in einer kompilierten Sprache wie C oder C++ geschriebenes Programm wird aus der Quellsprache in eine Sprache übersetzt, die Ihr Computer versteht (Binärcode, also Nullen und Einsen). Dazu wird ein Compiler mit verschiedenen Schaltern und Optionen verwendet. Wenn Sie das resultierende Programm ausführen, kopiert eine Software, Linker oder Loader genannt, das Programm von der Festplatte in den Hauptspeicher des Computers und führt es dort aus.

Im Gegensatz dazu muss ein in Python geschriebenes Programm nicht in Binärcode kompiliert werden. Es genügt, das Programm direkt als Quellcode *auszuführen*. Python wandelt den Quellcode automatisch in eine Zwischenform um, die Bytecode genannt wird, übersetzt diese in die Sprache Ihres Computers und führt das Ergebnis aus. All das führt dazu, dass die Benutzung von Python viel leichter gemacht wird, da Sie sich nicht um das Kompilieren des Programms, das korrekte Linken der benötigten Bibliotheken usw. kümmern müssen. Ihre Python-Programme werden dadurch auch einfacher portierbar, da Sie lediglich Ihr Python-Programm auf einen anderen Computer kopieren und es dort einfach funktioniert!

Objektorientiert

Python unterstützt sowohl die prozedural orientierte Art des Programmierens, als auch das objektorientierte Programmieren. In *prozeduralen* Sprachen wird ein Programm um Prozeduren oder Funktionen herum aufgebaut, die lediglich Stücke von wieder verwendbarem Programmcode sind. In der objektorientierten Programmierweise wird es um Objekte herum aufgebaut, in denen Daten und Funktionalität vereint sind. Die Weise, in der objektorientierten Programmierung in Python realisiert ist, ist sehr mächtig und zugleich einfach, insbesondere im Vergleich mit Sprachen wie C++ oder Java.

Erweiterbar

Falls es nötig ist, dass ein kritisches Stück Code besonders schnell ausgeführt wird, oder wenn Sie möchten, dass einige Algorithmen nicht offen lesbar sind, können Sie solche Teile Ihres Programms in C oder C++ schreiben und dann diese in Ihrem Python-Programm verwenden.

Einbettbar

Sie können umgekehrt Python in Ihre C- oder C++-Programme einbetten, um Ihren Programmbenutzern Skripting (z.B. mit Python-Makros) zu ermöglichen.

Umfangreiche Bibliotheken

Die Python-Standardbibliothek ist in der Tat riesig. Sie liefert Hilfsfunktionen in vielen verschiedenen Bereichen: Reguläre Ausdrücke, Dokumentationserzeugung, Testen von Programmeinheiten, Threading, Datenbanken, Webbrowser, CGI,

FTP, E-Mail, XML, XML-RPC, HTML, WAV-Dateien, Kryptographie, GUI (grafische Benutzeroberflächen), Tk und systemabhängige Dinge. Denken Sie daran: Dies alles ist überall verfügbar, wo Python installiert ist. Das wird als die "*Batteries included*"-Philosophie von Python bezeichnet ("Batterien gehören zum Lieferumfang").

Neben der Standardbibliothek gibt es verschiedene weitere Bibliotheken von hoher Qualität, z.B. wxPython [http://www.wxpython.org], Twisted [http://www.twistedmatrix.com/products/twisted], Python Imaging Library [http://www.pythonware.com/products/pil/] und viele andere.

Zusammenfassung

Python ist in der Tat eine aufregende und mächtige Sprache. Sie hat die richtige Kombination von Leistung und Funktionsumfang, die das Schreiben von Python-Programmen zugleich einfach und zu einem Vergnügen macht.

Warum nicht Perl?

Perl ist, falls Sie es nicht bereits kennen, eine andere sehr populäre interpretierte Open-Source-Programmiersprache.

Falls Sie jemals versucht haben, ein großes Programm in Perl zu schreiben, würden Sie sich diese Frage selbst beantworten! Anders ausgedrückt: Perl-Programm sind einfach, wenn sie klein sind, Perl glänzt, wenn es um kleine Hacks oder um Skripte geht, mit denen einfach etwas erledigt werden soll. Sie werden jedoch schnell unhandlich, sobald Sie mit dem Schreiben von größeren Programmen beginnen. Ich sage das aus der Erfahrung heraus, große Perl-Programme für Yahoo! geschrieben zu haben.

Verglichen mit Perl sind Python-Programme auf alle Fälle einfacher, sauberer, leichter zu schreiben und deswegen verständlicher und leichter wartbar. Ich bewundere Perl und verwende es täglich für verschiedene Dinge. Aber wenn ich ein neues Programm schreibe, denke ich immer zuerst in Python-Begriffen, weil das für mich so natürlich geworden ist. Perl war so vielen Hacks und Veränderungen ausgesetzt, dass es sich wie ein einziger großer, höllischer Hack ausnimmt. Traurigerweise scheint das kommende Perl 6 hier keine Verbesserungen zu bringen.

Der einzige und sehr bedeutsamste Vorteil, den ich bei Perl sehe, ist seine riesige CPAN [http://cpan.perl.org]-Bibliothek - Das "Comprehensive Perl Archive Network". Wie der Name vermuten lässt, handelt es sich dabei um eine ungeheure Sammlung von Perl-Modulen, atemberaubend durch seine reine Größe und Tiefe - Sie können nahezu alles, was mit einem Computer möglich ist, mit Hilfe dieser Module tun. Ein Grund dafür, dass Perl mehr Module als Python hat, liegt darin, dass Perl schon länger existiert als Python. Vielleicht sollte ich einen Hackathon zum Portieren von Perl-Modulen nach Python auf comp.lang.python [http://groups.google.com/groups?q=comp.lang.python] vorschlagen :)

Außerdem wird die neue virtuelle Maschine Parrot [http://www.parrotcode.org] entwickelt, um das komplett neu entworfene Perl 6 und auch Python und andere interpretierte Sprachen wie Ruby, PHP und Tcl ausführen zu können. Das bedeutet, dass Sie *vielleicht* in der Zukunft all diese Perl-Module auch in Python verwenden können, Sie also das beste beider Welten haben werden: die mächtige CPAN-Bibliothek kombiniert mit der mächtigen Programmiersprache Python. Allerdings werden wir einfach abwarten müssen, wie sich dies entwickelt.

Was Programmierer sagen

Vielleicht interessiert es Sie, was prominente Hacker wie Eric S. Raymond über Python sagen:

- **Eric S. Raymond** ist der Autor des Buchs "Die Kathedrale und der Basar [http://www.phonesoft.com/RaymondCathedralBazaar/catb_g.0.html]" und derjenige, der den Begriff "Open Source" geprägt hat. Er sagt, dass Python seine bevorzugte Programmiersprache geworden ist [http://www.linuxjournal.com/article.php?sid=3882]. Sein Artikel inspirierte mich dazu, mich zum ersten Mal mit Python zu befassen.

- **Bruce Eckel** ist der Autor der berühmten Bücher "Thinking in Java [http://www.java-net.de/thinking_in_java/]" und "Thinking in C++ [http://java.freeq.de]" (" In C++ denken"). Er sagt, dass er mit keiner Sprache produktiver sei als mit Python. Python ist für ihn wohl die einzige Sprache, die sich darauf konzentriert, die Arbeit für den Programmierer einfacher zu machen. Lesen Sie das vollständige Interview [http://www.artima.com/intv/aboutme.html] für mehr Details.

- **Peter Norvig** ist ein bekannter Lisp [http://www.norvig.com/python-lisp.html]-Autor und Direktor für Suchqualität bei Google (vielen Dank an Guido van Rossum für diesen Hinweis). Er sagt, dass Python immer ein integraler Bestandteil von Google gewesen ist. Sie können diese Aussage überprüfen, wenn Sie sich die Google-Jobs [http://www.google.com/jobs/]-Seite anschauen, auf der Python-Kenntnisse als Vorraussetzung für Softwareentwickler genannt werden.

- **Bruce Perens** ist Mitbegründer von OpenSource.org [http://opensource.org] und des UserLinux [http://www.userlinux.com]-Projekts. Das Ziel von UserLinux ist es, eine standardisierte Linux-Distribution zu schaffen, die von verschiedenen Herstellern unterstützt wird. Python hat Mitbewerber wie Perl und Ruby im Wettbewerb um die wesentliche von UserLinux unterstützte Programmiersprache geschlagen.

Kapitel 2. Python installieren

Für Linux/BSD-Anwender

Falls Sie eine GNU/Linux-Distribution wie Fedora Core, SuSE oder Mandrake verwenden oder ein BSD-System wie FreeBSD, dann ist Python vermutlich bereits auf Ihrem System installiert.

Um zu überprüfen, ob Python auf Ihrem GNU/Linux-System installiert ist, öffnen Sie eine Kommandozeilenumgebung (z.B. konsole oder gnome-terminal) und geben Sie das Kommando **python -V** ein:

```
$ python -V
Python 2.3.4
```

Anmerkung

$ ist die Eingabeaufforderung des Kommandozeilenumgebung. Ihr genaues Aussehen hängt von den Einstellungen Ihres Betriebssystems ab. Wir werden sie allgemein nur als $ darstellen.

Falls Sie eine Versionsinformation wie die oben dargestellte sehen, ist Python bereits installiert.

Wenn Sie jedoch eine solche Meldung sehen:

```
$ python -V
bash: python: command not found
```

dann ist Python nicht installiert. Das ist zwar sehr unwahrscheinlich, aber möglich.

In diesem Fall gibt es zwei Möglichkeiten, um Python auf Ihrem System zu installieren.

- Installieren Sie die Binärpakete mit Hilfe des Paketmanagers Ihres Betriebssystems, wie z.B. yum unter Fedora Core, urpmi unter Mandrake Linux, yast (rpm) unter SuSE-Linux apt-get unter Debian GNU/Linux, pkg_add unter FreeBSD usw. Achten Sie darauf, dass Sie für diese Methode eine Internetverbindung oder ein entsprechendes Installationsmedium (CD, DVD) zur Verfügung haben müssen.

  Alternativ dazu können Sie die Binärpakete auch zuerst aus dem Internet herunterladen, auf Ihren Rechner kopieren und dort installieren.

- Die andere Möglichkeit ist, Python aus dem unter www.python.org/download/ [http://www.python.org/download/] verfügbaren Quellcode zu übersetzen und zu installieren. Die genaue Anleitung hierfür finden Sie ebenfalls auf dieser Webseite.

Für Windows-Anwender

Gehen Sie nach www.python.org/download [http://www.python.org/download/] und laden Sie die neueste Version des Windows-Installationsprogramms herunter. Dieses ist nur etwa 10 MB groß und, verglichen mit anderen Programmiersprachen, sehr kompakt. Die Installation läuft genauso wie bei jeder anderen Windows-Software.

Achtung

Wenn Sie während der Installation gefragt werden, ob Sie *optionale* Komponenten abwählen möchten, so tun Sie das nicht! Einige dieser Komponenten können für Sie sehr nützlich sein, insbesondere IDLE.

Eine interessante Tatsache ist, dass etwa 70% aller Python-Downloads durch Windows-Benutzer geschehen. Allerdings ist dieser Prozentsatz dadurch verzerrt, das fast alle GNU/Linux-Benutzer Python sowieso schon automatisch auf ihrem System installiert haben.

Python in der Windows-Eingabeaufforderung verwenden

Wenn Sie in der Lage sein wollen, Python von der Windows-Eingabeaufforderung aus laufen zu lassen, dann müssen Sie die Umgebungsvariable PATH entsprechend setzten.

Bei Windows 2000, XP und 2003 klicken Sie hierzu auf Systemsteuerung -> System -> Erweitert -> Umgebungsvariablen. Klicken Sie auf die Variable namens **Path** in dem unteren Bereich für 'Systemvariablen', dann wählen Sie Bearbeiten und fügen Sie **;C:\Python23** am Ende der bereits vorhandenen Zeile an. Natürlich sollten Sie den richtigen Verzeichnisnamen verwenden, wenn Sie Python in einem anderen Verzeichnis installiert haben.

Bei älteren Windows-Versionen müssen Sie die folgende Zeile in der Datei `C:\AUTOEXEC.BAT` hinzufügen: '**PATH=%PATH%;C:\Python23**' (ohne die Anführungszeichen) und das System neu starten. Bei Windows NT müssen Sie stattdessen die Datei `AUTOEXEC.NT` benutzen.

Zusammenfassung

Unter GNU/Linux wird Python vermutlich bereits auf Ihrem System installiert sein. Ansonsten können Sie es mit Hilfe der Paketmanagementsoftware Ihrer Distribution installieren. Unter Windows müssen Sie für die Installation lediglich das Installationsprogramm herunterladen und ausführen. Ab jetzt gehen wir davon aus, dass Python auf Ihrem System installiert ist.

Als Nächstes werden wir unser erstes Pythonprogramm schreiben.

Kapitel 3. Erste Schritte

Einführung

Jetzt werden wir sehen, wie man das traditionelle "Hallo Welt"-Programm in Python zum Laufen bekommt. Sie lernen dadurch, wie man Python-Programme schreibt, sie abspeichert und ausführt.

Mit Python haben Sie zwei verschiedene Möglichkeiten, Ihr Programm auszuführen - entweder über die interaktive Kommandozeile des Interpreters, oder indem Sie eine Quelldatei verwenden. Wir werden jetzt beide Varianten kennen lernen.

Benutzung der Interpreter-Eingabeaufforderung

Starten Sie die Eingabeaufforderung des Python-Interpreters durch Eingabe von **python** auf der Kommandozeile. Geben Sie nun `print 'Hallo Welt'` ein und drücken Sie die **Enter**-Taste. Als Ausgabe sollten Sie die Wörter `Hallo Welt` sehen.

Wenn Sie Windows benutzen, können Sie den Interpreter von der Eingabeaufforderung aus Windows heraus starten, wenn die Umgebungsvariable `PATH` korrekt gesetzt ist. Alternativ dazu kann auch das Programm IDLE verwendet werden. IDLE ist die Abkürzung für *Integrated DeveLopment Environment* (Integrierte Entwicklungsumgebung). Zum Starten klicken Sie auf Start -> Programme -> Python 2.3 -> IDLE (Python GUI). Linux-Benutzer können IDLE ebenfalls verwenden.

Beachten Sie, dass die Zeichen >>> die Eingabeaufforderung für Python-Anweisungen darstellen.

Beispiel 3.1. Die Eingabeaufforderung des Python-Interpreters

```
$ python
Python 2.3.4 (#1, Oct 26 2004, 16:42:40)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hallo Welt'
Hallo Welt
>>>
```

Beachten Sie, dass Python Ihnen das Ergebnis der Eingabe sofort ausgibt, nachdem Sie eine einzelne Python-*Anweisung* eingegeben haben. Der Befehl `print` wird verwendet, um einen beliebigen Wert auszugeben, der diesem Befehl übergeben wird. Hier wird der Text `Hallo Welt` übergeben und auf dem Bildschirm ausgegeben.

Beenden der Python-Eingabeaufforderung

Wenn Sie IDLE verwenden oder in einer Linux/BSD-Kommandozeilen-Umgebung arbeiten, beenden Sie die Kommandozeile mit der Tastenkombination **Strg**+**d**. Für den Fall, dass Sie die Windows Eingabeaufforderung verwenden, drücken Sie **Strg**+**z** gefolgt von der **Enter**-Taste.

Die Auswahl des Editors

Bevor wir anfangen, Quelldateien für Python-Programme zu schreiben, brauchen wir einen Texteditor zum Editieren der Quelldateien. Die Wahl des Editors ist wirklich eine äußerst wichtige Angelegen-

heit. Sie sollten Ihren Editor genauso sorgfältig auswählen, wie ein Auto, das Sie kaufen wollen. Mit einem guten Editor fällt es Ihnen leichter, Python-Programme zu schreiben. Um beim Vergleich mit einem Auto zu bleiben: Er macht Ihre Reise bequemer und Sie werden Ihr Ziel wesentlich schneller und sicherer erreichen.

Eine der wichtigsten Anforderungen ist **Syntax-Hervorhebung**. Hierdurch werden die verschiedenen Teile Ihres Python-Programms farblich hervorgehoben, so dass Sie Ihr Programm *sehen* und sich den Ablauf vorstellen können.

Wenn Sie Windows verwenden, empfehle ich Ihnen, IDLE zu verwenden. IDLE bieten Ihnen neben Syntax-Hervorhebung und anderen Dingen auch die Möglichkeit, Ihr Programm direkt in IDLE auszuführen. Eine wichtige Anmerkung: **verwenden Sie nicht Notepad** (den Standard-Editor unter Windows) - er unterstützt weder Syntax-Hervorhebung noch das Einrücken von Text, was für uns ebenfalls sehr wichtig ist, wie wir später noch sehen werden. Gute Editoren wie IDLE (oder auch VIM) helfen Ihnen automatisch bei diesen Vorgängen.

Als Benutzer von Linux/FreeBSD haben Sie eine große Auswahl an Editoren. Wenn Sie ein erfahrener Programmierer sind, haben Sie bestimmt schon VIM oder Emacs verwendet. Das sind natürlich zwei der leistungsfähigsten Editoren, von deren Verwendung Sie bei der Entwicklung Ihrer Python-Programme profitieren werden. Ich persönlich verwende für die meisten meiner Programme VIM. Als Programmieranfänger kann ich Ihnen auch Kate empfehlen. Wenn Sie sich die Zeit nehmen können, den Umgang mit VIM oder Emacs zu erlernen, empfehle ich Ihnen, sich mit einem der beiden vertraut zu machen, da Sie langfristig davon profitieren werden.

Wenn Sie sich noch nicht für einen Editor entscheiden konnten, dann hilft Ihnen vielleicht diese ausführliche Liste von Editoren für Python [http://www.python.org/cgi-bin/moinmoin/PythonEditors]. Sie können auch eine IDE (*Integrated Development Environment*, Integrierte Entwicklungsumgebung) für Python verwenden. Einzelheiten dazu finden Sie in der umfangreichen Liste von IDEs, die Python unterstützen [http://www.python.org/cgi-bin/moinmoin/IntegratedDevelopmentEnvironments]. Der Einsatz einer IDE kann für größere Python-Programme durchaus sinnvoll sein.

Nochmals zur Wiederholung: Bitte verwenden Sie einen geeigneten Editor - dies erleichtert das Schreiben von Python-Programmen enorm und macht einfach mehr Spaß.

Die Verwendung von Quelldateien

Jetzt aber zurück zum Programmieren. Traditionell ist das erste Programm, das Sie schreiben und ausführen, wenn Sie eine neue Programmiersprache lernen, das "Hallo Welt"-Programm - beim Ausführen macht es nichts anderes als 'Hallo Welt' zu sagen. Simon Cozens [1] nennt es die "traditionelle Beschwörung der Programmiergötter, damit sie einem helfen mögen, die Programmiersprache besser zu lernen" :).

Starten Sie den von Ihnen bevorzugten Editor, geben Sie das folgende Programm ein, und speichern Sie es unter dem Namen `hallowelt.py`

Beispiel 3.2. Gebrauch einer Quelldatei (hallowelt.py [code/hallowelt.py])

```
#!/usr/bin/python

print 'Hallo Welt'
```

Um das Programm auszuführen, öffnen Sie eine Kommandozeilen-Umgebung (Linux-Terminal oder DOS-Eingabeaufforderung) und geben Sie den Befehl **python *`hallowelt.py`*** ein. Wenn Sie IDLE

[1] einer der führenden Perl6/Parrot-Hacker und Autor des fantastischen Buchs "Beginning Perl"

benutzen, verwenden Sie das Menü Edit -> Run Script oder die Tastenkombination **Strg**+**F5**. Unten sehen Sie die Ausgabe des Programms.

Ausgabe

```
$ python hallowelt.py
Hallo Welt
```

Sie sehen die gleiche Ausgabe wie oben gezeigt? Glückwunsch! - Sie haben gerade Ihr erstes Python-Programm erfolgreich ausgeführt.

Sollten Sie eine Fehlermeldung erhalten, überprüfen Sie bitte, ob Sie es *genau so* wie oben angegeben abgetippt haben. Beachten Sie, dass Python zwischen Groß- und Kleinschreibung unterscheidet. D.h. `print` ist nicht das Gleiche wie `Print`. Stellen Sie auch sicher, dass sich keine Leerzeichen oder Tabulatorzeichen vor dem ersten Zeichen einer Zeile befinden - wir werden später sehen, warum das wichtig ist.

So funktioniert es

Sehen wir uns die ersten beiden Zeilen des Programms an. Hierbei handelt es sich um *Kommentare* - alles was rechts des Zeichens # steht, ist ein Kommentar und dient hauptsächlich als Hinweis für Leser des Programms.

Außer im Spezialfall der ersten Zeile interpretiert Python die Kommentare nicht weiter. Diese erste Zeile wird *Shebang-Zeile* genannt - immer wenn Ihre Quelldatei mit den Zeichen `#!` und der Pfadangabe für ein Programm beginnt, weiß Ihr Linux/Unix-System, dass das Programm mit diesem Interpreter gestartet werden soll, wenn Sie es *ausführen*. Eine ausführliche Erklärung dazu folgt im nächsten Kapitel. Unabhängig von der verwendeten Plattform können Sie das Programm immer ausführen, indem Sie den Interpreter direkt in der Kommandozeile mit angeben, wie zum Beispiel **python *`hallowelt.py`***.

Wichtig

Verwenden Sie Kommentare auf sinnvolle Weise in Ihrem Programm, um wichtige Details zu erklären - dadurch können Leser Ihres Programms leichter verstehen, wie es funktioniert. Vergessen Sie nicht, dass Sie selbst in sechs Monaten dieser Leser sein könnten!

Nach den Kommentaren folgt eine Python-*Anweisung* - diese gibt den Text `'Hallo Welt'` aus. `print` ist eigentlich ein Operator und `'Hallo Welt'` wird als String (Zeichenkette) bezeichnet. Keine Bange, wir werden diese Fachausdrücke später ausführlich besprechen.

Eine Anmerkung für deutsche Anwender

Das deutsche Alphabet besitzt ja bekanntlich zusätzlich zum englischen Alphabet, das dem so genannten ASCII-Zeichensatz zugrunde liegt, auch noch die Umlaute ä, ö, ü sowie das ß, das sich trotz Rechtschreibreform hartnäckig hält. Diese Zeichen kommen im normalen ASCII-Zeichensatz nicht vor und führen daher des Öfteren zu Problemen, da sie speziell kodiert werden müssen, wofür es verschiedene Standards gibt.

Auch in Python-Programmen machen diese und andere Sonderzeichen Probleme, wenn Sie keine Kodierung angeben. In der Sprache Python an sich kommen zwar keine derartigen Sonderzeichen vor, aber Sie werden sie eventuell in Kommentaren oder innerhalb von String-Konstanten (Zeichenketten) in Python verwenden wollen.

Wollen Sie ein Python-Programm wie das folgende ausführen (beachten Sie das ä im Wort 'Käse'),

```
#!/usr/bin/python

print 'Käse!' # ein Umlaut
```

dann werden neuere Python-Versionen die folgende Warnung ausgeben:

```
sys:1: DeprecationWarning: Non-ASCII character '\xe4'
in file kaese.py on line 3, but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details
Käse!
```

Diese Warnung besagt, dass Sie ein Sonderzeichen eingegeben haben, das nicht im ASCII-Zeichensatz vorhanden ist, aber nicht dazu angegeben haben, welche Kodierung für Sonderzeichen gelten soll.

Windows-Benutzer können dieses Problem lösen, indem sie die Datei z. B. mit Notepad, dem Standard-Editor unter Windows, laden und dann mit 'Speichern unter...' in der Kodierung UTF-8 abspeichern. Das Python-Programm bekommt dadurch eine (normalerweise nicht sichtbare) Kennung am Dateianfang, wodurch das System und auch Python bescheid weiß, dass in dieser Datei die UTF-8-Kodierung verwendet wird. Mit dieser Kodierung können Sie sämtliche Zeichen des universellen Unicode-Zeichensatzes speichern, darunter natürlich auch die deutschen Umlaute.

Eine andere Möglichkeit, die Kodierung anzugeben, wird unter der Internetadresse erläutert, die in der Warnmeldung angegeben ist. Sie besteht darin, am Dateianfang eine weitere 'magische' Kommentarzeile einzufügen, in der die Kodierung angegeben wird:

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

print 'Käse!' # kein Problem mehr
```

Neben der universellen Kodierung `utf-8` ist im westeuropäischen Sprachraum auch `latin-1` (auch als `iso-8859-1` bekannt) weit verbreitet; diese Kodierung war auch der Standard in Python bis Version 2.1.

Um nicht durch diese Kodierungsproblematik vom Wesentlichen abzulenken, haben wir bei den Beispielprogrammen Umlaute vermieden bzw. mit ASCII-Zeichen umschrieben. Denken Sie aber daran, dass Sie hier keineswegs eingeschränkt sind, sondern nach Angabe einer Kodierung nach Herzenslust nicht nur Umlaute verwenden dürfen, sondern auch kyrillische oder beliebig exotische Zeichen, wie sie z. B. in der Kodierung UTF-8 im Unicode-Zeichenvorrat reichlich vorhanden sind.

Ausführbare Python-Programme

Dieser Abschnitt trifft nur für Linux/Unix-Benutzer zu. Aber auch Windows-Benutzer könnte es interessieren, was es mit der ersten Zeile des Programms auf sich hat. Mit dem Befehl **chmod** ändern wir

zuerst die Berechtigung des Programms so, dass es ausgeführt werden kann. Danach *starten* wir das Programm.

```
$ chmod a+x hallowelt.py
$ ./hallowelt.py
Hallo Welt
```

Der Befehl **chmod** wird hier verwendet, um die Zugriffsrechte der Datei so zu ändern, dass alle Benutzer des Systems (*a*) die Berechtigung zum Ausführen (*x*) des Programms bekommen. Danach wird das Programm sofort gestartet, indem wir den Pfad der Quelldatei eingeben. Die Zeichenfolge `./` gibt an, dass sich die Datei im aktuellen Verzeichnis befindet.

Das Ganze macht noch mehr Spaß, wenn Sie die Datei in `hallowelt` umbenennen und mit **./hallowelt** starten. Es funktioniert immer noch, da das System weiß, dass es das Programm mit dem Interpreter ausführen muss, der in der ersten Zeile der Quelldatei angegeben ist.

Sie können das Programm jetzt laufen lassen, solange Sie den genauen Pfad des Programms kennen - aber vielleicht wollen Sie das Programm aus jedem beliebigen Verzeichnis heraus starten können? Um dies zu erreichen, können Sie das Programm in einem der Verzeichnisse speichern, die in der Umgebungsvariable `PATH` aufgeführt sind. Immer wenn Sie irgendein Programm starten, sucht das System in jedem Verzeichnis, das in der `PATH`-Variable vorkommt, und führt es dann dort aus. Wir können unser Programm auch von überall erreichbar machen, indem wir die Quelldatei einfach in eines der in `PATH` aufgeführten Verzeichnisse kopieren.

```
$ echo $PATH
/opt/mono/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/swaroop/bin
$ cp hallowelt.py /home/swaroop/bin/hallowelt
$ hallowelt
Hallo Welt
```

Um die `PATH`-Variable anzuzeigen, verwenden wir den Befehl **echo**. Durch Voranstellen von `$` weiß der Kommandozeilen-Interpreter, dass wir uns für den Wert der Variable interessieren. Wie wir sehen, ist `/home/swaroop/bin` ein Verzeichnis, das in der Variablen PATH vorkommt. **swaroop** ist dabei der Benutzer, den ich auf meinem System verwende. Üblicherweise haben Sie auf Ihrem System ein entsprechendes Verzeichnis für Ihren Benutzernamen. Sie können auch wahlweise ein anderes Verzeichnis zu der `PATH`-Variablen hinzufügen. Dazu geben Sie den Befehl **PATH=$PATH:/home/swaroop/meinverzeichnis** als Kommandozeile ein, wobei `'/home/swaroop/meinverzeichnis'` für das Verzeichnis stehen soll, das Sie hinzufügen wollen.

Dieses Verfahren ist sehr hilfreich, wenn Sie nützliche Skripte schreiben wollen, die Sie als Programm von überall und jederzeit ausführen können wollen. Sie können auf diese Weise Ihren eigenen **cd**-Befehl oder irgendeinen anderen Linux- oder DOS-Befehl selbst schreiben.

Achtung

Im Zusammenhang mit Python bedeuten die Begriffe Programm, Skript oder Software alle das Gleiche.

Wenn Sie Hilfe brauchen

Mit der eingebauten Hilfe (`help`) bekommen Sie zu jeder Funktion oder jeder Anweisung in Python eine kurze Information angezeigt. Das ist vor allem sehr nützlich, wenn Sie die Kommandozeile des

Interpreters verwenden. Wenn Sie z. B. `help(str)` eingeben, wird Ihnen die Hilfe zu der Klasse `str` angezeigt. Diese Klasse wird dazu verwendet um Text (Strings, Zeichenketten) in Ihrem Programm zu speichern. Was man unter einer Klasse zu verstehen hat, wird im Kapitel über objektorientierte Programmierung ausführlich behandelt.

Anmerkung

Durch Drücken der Taste **q** verlassen Sie die Hilfe.

Analog dazu können Sie sich für fast alles, was es in Python gibt, Informationen ausgeben lassen. Durch den Aufruf von `help()` können Sie zum Beispiel mehr über den Hilfe-Befehl `help` selbst erfahren.

Wenn Sie Hilfe zu Operatoren wie `print` benötigen, müssen Sie die Umgebungsvariable `PYTHONDOCS` entsprechend setzen. Unter Linux/Unix können Sie dies einfach mit dem **env**-Befehl erledigen.

```
$ env PYTHONDOCS=/usr/share/doc/python-docs-2.3.4/html/ python
Python 2.3.4 (#1, Oct 26 2004, 16:42:40)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> help('print')
```

Wie Sie sehen, habe ich `'print'` in Anführungszeichen gesetzt, damit Python versteht, dass ich Hilfe zu 'print' haben möchte und nicht etwa mit print etwas ausgeben will.

Beachten Sie, dass der Pfad, den ich angegeben habe, der Pfad für Fedora Core 3 Linux ist - bei anderen Linux-Distributionen oder -Versionen kann der Pfad variieren.

Zusammenfassung

Sie können jetzt schon Python-Programme schreiben, abspeichern und ausführen lassen. Nachdem Sie nun zu einem Python-Benutzer geworden sind, wollen wir ein paar weitere Konzepte der Programmiersprache Python kennen lernen.

Kapitel 4. Die Grundlagen

Es ist natürlich ein bisschen wenig, wenn man nur 'Hallo Welt' ausgeben kann. Sie wollen mehr als das tun können - Sie wollen Eingaben entgegennehmen, diese verarbeiten und daraus irgendwelche Ausgaben produzieren. Wir können dies in Python mithilfe von Konstanten und Variablen erreichen.

Literale

Beispiele für Literale sind Zahlen wie `5`, `1.23`, `9.25e-3` oder Zeichenketten (Strings) wie `'Dies ist ein String'` oder `"Das ist auch ein String!"`. Sie werden Literale oder Direkt-Wert-Konstanten genannt, weil sie für Konstanten stehen, die *direkt* und wortwörtlich den Wert bezeichnen, für den sie stehen. Die Zahl `2` steht immer für sich selbst und nichts anderes - sie ist eine Konstante, weil ihr Wert nicht geändert werden kann.

Zahlen

Es gibt vier Zahlentypen in Python - Ganzzahlen (int), lange Ganzzahlen (long), Fließkommazahlen (float) und komplexe Zahlen (complex).

- Beispiele für Ganzzahlen sind `2` oder `-100`, also Zahlen ohne Nachkommastellen.
- Lange Ganzzahlen sind einfach betragsmäßig größere Ganzzahlen. Im Gegensatz zu den normalen Ganzzahlen ist ihre Größe prinzipiell nicht eingeschränkt (d.h. eigentlich nur durch die Größe des Hauptspeichers beschränkt).
- Beispiele für Fließkommazahlen sind `3.23` oder `52.3E-4`. Die Schreibweise mit dem `E` bezeichnet Zehnerpotenzen, `52.3E-4` steht hier also für $52.3 * 10^{-4}$.
- Beispiele für komplexe Zahlen sind `(-5 + 4j)` oder `(2.3 - 4.6j)`. Beachten Sie, dass die imaginäre Einheit hierbei nicht wie in der Mathematik üblich mit `i`, sondern in der Schreibweise der Elektrotechniker mit `j` bezeichnet wird.

Strings

Ein String, d.h. eine Zeichenkette, denkt man sich in Python als eine *Sequenz* (Abfolge) von einzelnen *Zeichen*.

Ich kann Ihnen fast garantieren, dass Sie in nahezu jedem von Ihnen geschriebenen Python-Programm auch Strings benutzen werden. Bitte beachten Sie daher die folgenden Hinweise zum Gebrauch von Strings in Python:

Benutzung einzelner Anführungszeichen (')

Sie können Strings durch Benutzung einzelner Anführungszeichen angeben, so wie `'Berufen Sie sich nur auf mich'`. Sämtlicher Leerraum, d. h. Leerzeichen und Tabulatoren, wird genauso beibehalten.

Benutzung doppelter Anführungszeichen (")

Strings in doppelten Anführungszeichen verhalten sich genauso wie Strings in einfachen Anführungszeichen. Ein Beispiel hierfür: `"Wie heißen's doch gleich?"`.

Benutzung dreifacher Anführungszeichen (''' oder """)

Sie können mehrzeilige Strings angeben, indem Sie dreifache Anführungszeichen verwenden. Innerhalb der dreifachen Anführungszeichen können Sie einfache und doppelte Anführungszeichen nach Belieben benutzen. Ein Beispiel:

```
'''Dies ist ein mehrzeiliger String. Dies ist die erste Zeile.
Dies ist die zweite Zeile.
"Wie heißen's doch gleich?", fragte ich ihn.
Er sagte: "Mein Name ist Bond, James Bond."
'''
```

Maskierungscodes

Angenommen, Sie wollen einen String eingeben, der ein einzelnes Anführungszeichen (`'`) enthält. Wie würden Sie diesen angeben? Nehmen Sie zum Beispiel den String `Wie heißen's doch gleich?`. Sie können Ihn nicht als `'Wie heißen's doch gleich?'` eingeben, weil Python dann durcheinander käme und nicht mehr wüsste, wo der String anfängt und wo er aufhört. Daher müssen Sie irgendwie angeben, dass dieses einzelne Anführungszeichen nicht das Ende des Strings bedeutet. Dies kann mit Hilfe eines so genannten *Maskierungscodes* geschehen. Sie geben ein einzelnes Anführungszeichen dabei als `\'` an - beachten Sie dabei den umgekehrten Schrägstrich ("backslash"). Nun können Sie den String als `'Wie heißen\'s doch gleich?'` eingeben.

Eine andere Möglichkeit, diesen speziellen String anzugeben, wäre `"Wie heißen's doch gleich?"`, d. h. unter Benutzung doppelter Anführungszeichen. Gleichermaßen müssen Sie einen Maskierungscode benutzen, um ein doppeltes Anführungszeichen als solches innerhalb eines Strings zu benutzen, der von doppelten Anführungszeichen umschlossen ist. Außerdem müssen Sie den umgekehrten Schrägstrich selber innerhalb eines Strings mittels des Maskierungscodes `\ \` eingeben.

Was machen Sie, wenn Sie einen zweizeiligen String eingeben wollen? Eine Möglichkeit besteht darin, wie oben dreifache Anführungszeichen zu benutzen. Eine andere Möglichkeit ist, hierfür den Maskierungscode für einen Zeilenvorschub zu benutzen - `\n`, um anzugeben, dass an dieser Stelle eine neue Zeile anfängt. Ein Beispiel hierfür ist `Dies ist die erste Zeile\nDies ist die zweite Zeile`. Ein weiterer nützlicher Maskierungscode, den man kennen sollte, ist der für den Tabulator - `\t`. Es gibt viele weitere Maskierungscodes, von denen ich hier nur die nützlichsten aufgeführt habe.

Eine Sache, die man sich merken sollte, ist, dass ein einzelner umgekehrter Schrägstrich am Ende der Zeile anzeigt, dass der String auf der nächsten Zeile weitergeht, jedoch ohne dass ein Zeilenvorschub hinzugefügt wird. Zum Beispiel ist

```
"Dieses war der erste Streich. \
Doch der zweite folgt sogleich."
```

gleichbedeutend mit `"Dieses war der erste Streich. Doch der zweite folgt sogleich."`

"Rohe" Zeichenketten

Wenn Sie einen String angeben müssen, bei dem keine Zeichen besonders ausmaskiert werden sollen, dann müssen Sie ihn als eine *rohe* Zeichenkette (*raw string*) eingeben, indem Sie dem String ein `r` oder ein `R` voranstellen. Ein Beispiel ist `r"Zeilenvorschübe kennzeichnet man mit \n"`.

Unicode-Strings

Unicode ist ein Standard, der es einem erlaubt, beliebige internationale Zeichen in seinen Texten zu verwenden. Wenn Sie einen Text in Ihrer Heimatsprache wie Deutsch, Hindi oder Arabisch schreiben wollen, benötigen Sie einen Text-Editor, der entweder mit Ihren länderspezifischen Zeichensätzen umgehen kann, die z.B. die deutschen Umlaute enthalten, oder mit Unicode, das alle diese länderspezifischen Zeichen enthält. Analog ist es auch in Python möglich, Unicode-Text zu verarbeiten - das Einzige, was Sie hierfür tun müssen ist, dem String ein `u` oder ein `U` voranzustellen, um ihn zu einem Unicode-String zu machen. Beispiel: `u"Dies ist ein Unicode-String."`

Denken Sie daran, Unicode-Strings zu benutzen, wenn Sie Text-Dateien bearbeiten, besonders, wenn Sie wissen, dass die Datei Text enthält, der in anderen Sprachen als Englisch geschrieben wurde.

Strings sind unveränderlich

Das bedeutet, dass man einen String, wenn man ihn einmal erzeugt hat, nicht mehr ändern kann. Das sieht zwar zuerst ziemlich problematisch aus, ist es aber in Wirklichkeit nicht. Wir werden in den verschiedenen Programmbeispielen, die wir später durchgehen werden, sehen, warum dies keine Beschränkung darstellt.

Verkettung von String-Literalen

Wenn man zwei Strings direkt hintereinander schreibt, werden sie automatisch von Python verkettet. Zum Beispiel wird `'Wie heißen\'s' 'doch gleich?'` automatisch in `"Wie heißen's doch gleich?"` umgewandelt.

Anmerkung für C/C++-Programmierer

In Python gibt es keinen eigenen Datentyp `char` für einzelne Zeichen. Es gibt hierfür keine echte Notwendigkeit und ich bin sicher, dass Sie ihn nicht vermissen werden.

Anmerkung für Perl/PHP-Programmierer

Denken Sie daran, dass Strings in einfachen und doppelten Anführungszeichen gleich sind - sie unterscheiden sich in keinster Weise.

Anmerkung, wenn Sie reguläre Ausdrücke verwenden

Verwenden Sie immer rohe Zeichenketten, wenn Sie mit regulären Ausdrücken hantieren, weil sie sonst ständig Zeichen mit umgekehrten Schrägstrichen ausmaskieren müssen und dabei schnell den Überblick verlieren. Zum Beispiel können Rückwärtsverweise als `'\\1'` oder als `r'\1'` eingegeben werden.

Variablen

Immer nur Literale zu verwenden würde schnell langweilig werden - wir brauchen eine Möglichkeit, beliebige Informationen zu speichern und sie auch zu verarbeiten. An dieser Stelle treten die *Variablen* auf den Plan. Variablen sind genau so, wie sie heißen - ihr Wert ist variabel, d.h. man kann alles Mögliche in einer Variable abspeichern. Variablen sind einfach ein Bereich im Speicher Ihres Computers, in welchem Sie Informationen ablegen. Anders als bei Literalen braucht man eine Möglichkeit, auf diese Variablen zuzugreifen. Deswegen gibt man ihnen Namen.

Bezeichner

Variablennamen sind Beispiele für Bezeichner. *Bezeichner* sind Namen, die vergeben werden, um *irgendetwas* zu identifizieren. Es gibt einige Regeln, die Sie bei Bezeichnern berücksichtigen müssen:

- Das erste Zeichen eines Bezeichners muss ein (großer oder kleiner) Buchstabe des (englischen) Alphabets oder ein Unterstrich sein ('_').
- Der Rest des Bezeichners kann aus (großen oder kleinen englischen) Buchstaben, Unterstrichen ('_') oder Ziffern (0-9) bestehen. Denken Sie also daran, dass deutsche Umlaute und andere Sonderzeichen in Bezeichnern nicht zulässig sind.
- Bei Bezeichnern spielt die Groß-/Kleinschreibung eine Rolle. So bezeichnen zum Beispiel `meinname` und `meinName` **nicht** die gleichen Variablen. Beachten Sie, dass das `n` in ersterem klein geschrieben ist, während das `N` in letzterem groß geschrieben wurde.
- Beispiele *zulässiger* Bezeichner sind `i`, `__mein_name`, `name_23`, `a1b2_c3` oder `Kaese`.
- Beispiele *unzulässiger* Bezeichner sind `2dinge`, `leer zeichen`, `mein-name` oder `käse`.

Datentypen

Variablen können verschiedene Arten von Werten enthalten, die **Datentypen** genannt werden. Die grundlegenden Datentypen sind Zahlen und Strings, die wir bereits besprochen haben. In späteren Kapiteln werden wir sehen, wie wir unter Benutzung von Klassen unsere eigenen Datentypen erzeugen können.

Objekte

Denken Sie daran, dass in einem Python-Programm alles als ein *Objekt* angesehen wird. Das ist in einem allgemeinen Sinn gemeint. Anstatt zu sagen 'das *Dings*', sagen wir 'das *Objekt*'.

Anmerkung für objekt-orientierte Programmierer

Python ist stark objekt-orientiert in dem Sinne, dass alles ein Objekt ist, einschließlich Zahlen, Strings und sogar Funktionen.

Wir werden nun sehen, wie wir Variablen zusammen mit Literalen benutzen können. Um dies auszuprobieren, können Sie das folgende Beispiel als Programm abpeichern und laufen lassen.

Wie man Python-Programme schreibt

Um ein Python-Programm abzuspeichern und laufen zu lassen, gehen wir im Folgenden immer wie folgt vor:

1. Starten Sie Ihren Lieblings-Editor.
2. Geben Sie den Programm-Code aus dem Beispiel ein.
3. Speichern Sie das Programm als Datei mit dem Dateinamen, der im Kommentar angegeben ist. Ich halte mich an die Vereinbarung, nach der alle Python-Programme mit der Datei-Endung `.py` gespeichert werden.
4. Starten Sie den Python-Interpreter mit dem Befehl **python *`programm.py`*** oder benutzen Sie IDLE, um Ihre Programme laufen zu lassen. Unter Linux/Unix können Sie auch die bereits erläuterte Methode für ausführbare Dateien verwenden.

Beispiel 4.1. Benutzung von Variablen und Literalen (var.py [code/var.py])

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

```
i = 5
print i
i = i + 1
print i

s = '''Dies ist ein mehrzeiliger String.
Dies ist die zweite Zeile.'''
print s
```

Ausgabe

```
$ python var.py
5
6
Dies ist ein mehrzeiliger String.
Dies ist die zweite Zeile.
```

So funktioniert es

Und so funktioniert dieses Programm: Zuerst weisen wir der Variable `i` die als Literal angegebene Zahl `5` zu, indem wir den Zuweisungsoperator (`=`) verwenden. Diese Zeile wird eine Anweisung genannt, weil sie angibt, dass etwas getan werden soll. In diesem Fall wird der Variablenname `i` mit dem Wert `5` verbunden. Als Nächstes geben wir den Wert von `i` unter Benutzung der `print`-Anweisung aus, die uns, kaum überraschend, einfach den Wert der Variable auf den Bildschirm ausgibt.

Danach addieren wir `1` zu dem in der Variable `i` gespeicherten Wert und speichern das Ergebnis zurück in die Variable. Wir geben es dann wieder aus und erhalten wie erwartet den Wert `6`.

Entsprechend weisen wir der Variable `s` den String-Literal in dreifachen Anführungszeichen zu und geben ihn dann aus.

Anmerkung für C/C++-Programmierer

Variablen werden benutzt, indem man ihnen einfach einen Wert zuweist. Es wird keine Deklaration oder Datentyp-Definition benötigt/verwendet.

Logische und physikalische Zeilen

Eine physikalische Zeile ist das, was man beim Schreiben eines Programms als Zeile *sieht*. Eine logische Zeile ist das, was Python als eine einzelne Anweisung *ansieht*. Python nimmt zunächst einmal implizit an, dass jede *physikalische Zeile* einer *logischen Zeile* entspricht.

Ein Beispiel für eine logische Zeile ist eine Anweisung wie `print 'Hallo Welt'` - wenn sie in einer Zeile für sich steht (wie man sie im Editor sieht), dann entspricht sie auch einer physikalischen Zeile.

Hierdurch legt es Python nahe, pro Zeile nur eine einzige Anweisung zu schreiben, wodurch der Code auch lesbarer wird.

Wenn Sie mehr als eine logische Zeile in einer einzigen physikalischen Zeile angeben wollen, dann müssen Sie dies explizit durch ein Semikolon (`;`) kennzeichnen, mit dem Sie das Ende einer logischen Zeile/Anweisung markieren. Zum Beispiel bewirkt

```
i = 5
print i
```

genau das Gleiche wie

```
i = 5;
print i;
```

oder das Gleiche wie

```
i = 5; print i;
```

oder auch

```
i = 5; print i
```

Ich möchte jedoch die **ausdrückliche Empfehlung** aussprechen, dass Sie dabei bleiben, **eine einzelne logische Zeile pro physikalischer Zeile** zu schreiben. Man sollte das Semikolon also möglichst vermeiden, um den Code übersichtlich und lesbar zu halten. Tatsächlich habe ich *noch nie* ein Semikolon in einem Python-Programm benutzt oder auch nur gesehen. Umgekehrt sollten Sie mehr als eine physikalische Zeile für eine einzelne logische Zeile nur dann benutzen, wenn die logische Zeile wirklich sehr lang ist.

Im Folgenden ein Beispiel dafür, wie man eine logische Zeile schreiben kann, die mehrere physikalische Zeilen umspannt. Dies wird als **explizite Zeilenverkettung** bezeichnet.

```
s = 'Dies ist ein String. \
Hier geht der String weiter.'
print s
```

Dies führt zu folgender Ausgabe:

```
Dies ist ein String. Hier geht der String weiter.
```

Entsprechend ist

```
print \
i
```

das gleiche wie

```
print i
```

Manchmal geht Python implizit von einer Zeilenverkettung aus, so dass man keinen umgekehrten Schrägstrich verwenden muss. Dies ist der Fall, wenn die logische Zeile runde, eckige oder geschweifte Klammern enthält, und vor dem Zeilenende Klammern noch nicht geschlossen wurden. Dies wird **implizite Zeilenverkettung** genannt. Sie können sehen, wie dies funktioniert, wenn wir in späteren Kapiteln Programme schreiben werden, die Listen verwenden.

Einrückung

Leerraum spielt in Python eine Rolle. Vor allem der **Leerraum am Zeilenanfang ist wichtig**. Man meint diesen Leerraum, wenn man von der **Einrückung** einer Zeile spricht. Der Leerraum (Leerzeichen und Tabulatoren) am Anfang der logischen Zeile wird benutzt, um die Einrückungstiefe der logischen Zeile zu bestimmen, die wiederum benutzt wird, um die Gruppierung der Anweisungen zu bestimmen.

Das bedeutet, dass Anweisungen, die zusammen gehören, die gleiche Einrückungstiefe haben **müssen**. Jeder solche Abschnitt von Anweisungen mit der gleichen Einrückungstiefe wird ein **Block** genannt. Wir werden bald in den folgenden Kapiteln Beispiele dafür sehen, warum Blöcke wichtig sind.

Sie sollten im Kopf behalten, dass falsche Einrückung zu Fehlern führen kann. Zum Beispiel:

(leerzeichen.py [code/leerzeichen.py])

```
#!/usr/bin/python

i = 5
 print 'Der Wert ist', i # Fehler! Leerzeichen am Zeilenanfang
print 'Ich wiederhole, der Wert ist', i
```

Wenn Sie dies laufen lassen, erhalten Sie die folgende Fehlermeldung:

```
  File "leerzeichen.py", line 4
    print 'Der Wert ist', i # Fehler! Leerzeichen am Zeilenanfang
    ^
SyntaxError: invalid syntax
```

Beachten Sie das Leerzeichen, das da am Anfang der zweiten Zeile steht. Der Fehler, auf den Python hinweist, sagt uns, dass die Syntax des Programms unzulässig ist, d.h. dass das Programm nicht sauber

geschrieben wurde. Das bedeutet für Sie, dass Sie *nicht nach Belieben neue Blöcke von Anweisungen anfangen können* (außer dem Hauptblock natürlich, den Sie die ganze Zeit verwenden). Die Fälle, in denen Sie neue Blöcke benutzen können, werden in späteren Kapiteln ausgeführt, wie dem Kapitel über den Kontrollfluss.

So macht man Einrückungen

Benutzen Sie **auf keinen Fall** ein Mischmasch von Tabulatoren und Leerzeichen zur Einrückung, weil es nicht sauber plattformübergreifend funktioniert, da die Breite eines Tabulators in Leerzeichen nicht eindeutig für alle Plattformen definiert ist. Meine *eindringliche Empfehlung* ist, dass Sie entweder einen *einzelnen Tabulator* oder *zwei oder vier Leerzeichen* pro Einrückungs-Ebene benutzen.

Wählen Sie einen dieser drei Einrückungsstile. Vor allem aber, wählen und benutzen Sie diesen Stil **einheitlich**, d.h. benutzen Sie *ausschließlich* diesen einen Einrückungsstil.

Zusammenfassung

Nachdem wir nun viele konkrete Details im Einzelnen durchgesprochen haben, können wir uns jetzt interessanteren Dingen widmen, wie den Anweisungen, die den Kontrollfluss steuern. Sie sollten sich aber zuerst mit den in diesem Kapitel besprochenen Dingen wirklich vertraut gemacht haben.

Kapitel 5. Operatoren und Ausdrücke

Einführung

Die meisten Anweisungen, die Sie schreiben, werden **Ausdrücke** enthalten. Ein einfaches Beispiel für einen Ausdruck ist `2 + 3`. Ein Ausdruck setzt sich aus Operatoren und Operanden zusammen.

Operatoren bieten eine bestimmte Funktionalität und können durch Symbole wie `+`, oder besondere Schlüsselwörter ausgedrückt werden. Sie operieren stets auf irgendwelchen Daten, die man *Operanden* nennt. In unserem Beispiel sind `2` und `3` die Operanden.

Operatoren

Lassen Sie uns einen kurzen Blick auf die Operatoren und ihre Verwendung werfen:

Tipp

Die Ausdrücke aus den Beispielen können mit dem Interpreter interaktiv ausgewertet werden. Um z.B. den Ausdruck `2 + 3` zu testen, geben Sie folgendes in der Kommandozeile des Interpreters ein:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Tabelle 5.1. Operatoren und ihre Verwendung

Operator	Name	Erklärung	Beispiele
+	Plus	Addiert bzw. verkettet die beiden Objekte	`3 + 5` ergibt `8`. `'a' + 'b'` ergibt `'ab'`.
-	Minus	Ergibt entweder eine negative Zahl, oder das Ergebnis einer Subtraktion zweier Zahlen	`-5.2` ergibt eine negative Zahl. `50 - 24` ergibt `26`.
*	Multiplikation	Multipliziert zwei Zahlen miteinander oder gibt eine mehrfache Wiederholung eines Strings (einer Sequenz) zurück.	`2 * 3` ergibt `6`. `'la' * 3` ergibt `'lalala'`.
**	Potenz	Gibt die Potenzierung x hoch y zurück	`3 ** 4` ergibt `81` (d.h. `3 * 3 * 3 * 3`)
/	Division	Dividiere x durch y	`4/3` ergibt `1` (die Division von Ganzzahlen ergibt wieder eine Ganzzahl). `4.0/3` oder `4/3.0` ergibt `1.3333333333333333`
//	Ganzzahlige Division	Liefert das Ergebnis der ganzzahligen Division zurück	`4 // 3.0` ergibt `1.0`

Operator	Name	Erklärung	Beispiele
%	Modulo	Liefert den Rest bei einer ganzzahligen Division zurück	`8%3` ergibt `2`. `-25.5%2.25` ergibt `1.5`.
<<	Bitweises Linksschieben	Verschiebt das Bitmuster des linken Operanden um die angegebene Anzahl von Bit-Positionen nach links (jede Zahl wird im Speicher durch Bits, d.h. die Binärzeichen 0 und 1 repräsentiert).	`2 << 2` ergibt `8`. `2` wird durch das Bitmuster `10` repräsentiert. Eine Verschiebung um 2 Bits nach links ergibt `1000`, was wiederum der Dezimalzahl `8` entspricht.
>>	Bitweises Rechtsschieben	Verschiebt das Bitmuster des linken Operanden um die angegebene Anzahl von Bit-Positionen nach rechts.	`11 >> 1` ergibt `5` - `11` wird durch das Bitmuster `1011` repräsentiert. Eine Verschiebung um 1 Bit nach rechts ergibt `101`, was wiederum der Dezimalzahl `5` entspricht.
&	Bitweises UND	Die Bitmuster der beiden Zahlen werden mit UND verknüpft.	`5 & 3` ergibt `1`.
|	Bitweises ODER	Die Bitmuster der beiden Zahlen werden mit ODER verknüpft	`5 | 3` ergibt `7`
^	Bitweises XOR	Die Bitmuster der beiden Zahlen werden mit XOR (exklusivem ODER) verknüpft	`5 ^ 3` ergibt `6`
~	Bitweises NICHT	Ergibt die bitweise Negation des Operanden. Dies wird auch als das Einerkomplement bezeichnet. Das Einerkomplement einer Zahl x ist gleich -(x+1).	`~5` ergibt `-6`.
<	Kleiner als	Das Ergebnis zeigt an, ob x kleiner als y ist. Alle Vergleichsoperatoren liefern als Rückgabewert 1 für wahr und 0 für falsch. Diese Werte entsprechen den speziellen logischen Konstanten `True` bzw. `False`. Beachten Sie die Großschreibung dieser Konstanten.	`5 < 3` ergibt `0` (d.h. `False`) und `3 < 5` ergibt `1` (d.h. `True`). Vergleiche können beliebig hintereinander geschaltet werden: 3 < 5 < 7 ergibt `True`.
>	Größer als	Das Ergebnis zeigt an, ob x größer als y ist	`5 > 3` ist wahr, ergibt also `True`. Falls beide Operanden Zahlen sind, werden sie vor dem Vergleich zuerst in einen gemeinsamen Datentyp umgewandelt. Andernfalls erhält man immer `False`.
<=	Kleiner gleich	Das Ergebnis zeigt an, ob x kleiner als oder gleich y ist	`x = 3; y = 6; x <= y` ergibt `True`.
>=	Größer gleich	Das Ergebnis zeigt an, ob x größer als oder gleich y ist	x = 4; y = 3; x >= 3 ergibt `True`.
==	Gleichheit	Prüft die beiden Objekte auf Gleichheit	`x = 2; y = 2; x == y` ergibt `True`. `x = 'str'; y = 'stR'; x == y` ergibt `False`. `x = 'str'; y = 'str'; x == y` ergibt `True`.

Operator	Name	Erklärung	Beispiele
!=	Ungleichheit	Prüft die beiden Objekte auf Ungleichheit	`x = 2; y = 3; x != y` ergibt `True`.
not	Logisches NICHT	Bewirkt eine Invertierung des Wahrheitswertes: Wenn x `True` ist, ist das Ergebnis `False`. Wenn x `False` ist, ist das Ergebnis `True`.	`x = True; not x` ergibt `False`.
and	Logisches UND	`x and y` ergibt `x`, wenn `x` als Wahrheitswert interpretiert gleich `False` ist. Andernfalls ist das Ergebnis der Wert von of `y`. Das Ergebnis entspricht daher einer logischen UND-Verknüpfung.	`x = False; y = True; x and y` ergibt `False`, da `x False` ist. In diesem Fall wird der Python-Interpreter die Variable `y` gar nicht erst in die Auswertung mit einbeziehen, da der Ausdruck bereits falsch ist (da `x False` ist). Dies wird eine "Kurzschlussauswertung" genannt: Sie liefert das richtige Ergebnis, ohne dass `y` ausgewertet werden muss (bedenken Sie, dass an der Stelle von `y` auch ein sehr komplizierter Ausdruck stehen kann, dessen Berechnung lange Zeit in Anspruch nehmen würde oder zu einem Fehler führen könnte - durch die verkürzte Auswertung wird dies vermieden). So liefert z.B. `0 and 1/0` den Wert 0 zurück, da die Zahl 0 dem Wahrheitswert `False` entspricht. Der Ausdruck `1/0` wird in diesem Fall nicht ausgewertet und führt daher auch nicht zu einer Fehlermeldung.
or	Logisches ODER	`x or y` ergibt `x`, wenn `x` als Wahrheitswert interpretiert gleich `True` ist. Andernfalls ist das Ergebnis der Wert von `y`. Das Ergebnis entspricht daher einer logischen ODER-Verknüpfung.	`x = True; y = False; x or y` ergibt `True`. Hier wird ebenfalls das Prinzip der Kurzschlussauswertung verwendet. So liefert z.B. `3 and 1/0` ohne Fehlermeldung den Wert 123 zurück, da die Zahl 3 dem Wahrheitswert `True` entspricht. Die Auswertung von `3/4 and 1/0` führt jedoch zu einer Fehlermeldung, da die Division der Ganzzahlen den Wert 0 ergibt, was logisch gleich `False` ist.

Auswertungsreihenfolge der Operatoren

Wird bei einem Ausdruck wie z.B. `2 + 3 * 4` zuerst die Addition oder die Multiplikation durchgeführt? Wie wir aus der Schulmathematik wissen, kommt die Multiplikation zuerst - d.h. der Operator für die Multiplikation hat gegenüber dem Operator für die Addition eine höhere Priorität.

Die nachfolgende Tabelle listet alle Operatoren in Python in aufsteigender Reihenfolge ihrer Priorität auf, von der niedrigsten Priorität (am schwächsten bindend) zur höchsten (am stärksten bindend). Mit anderen Worten wird der Python-Interpreter bei einem Ausdruck zuerst die Operatoren auswerten, die in der Tabelle weiter unten stehen, bevor er die Operatoren auswertet, die weiter oben stehen.

Die Tabelle soll hier nur zur Vervollständigung dienen (Sie finden diese auch im Python-Referenzhandbuch). Ich rate Ihnen, Operatoren und Operanden mit Klammern zu gruppieren, um die Reihenfolge der Auswertung zu verdeutlichen. Das Programm wird dadurch auch lesbarer. Der Ausdruck `2 + (3 * 4)` ist z.B. viel klarer als `2 + 3 * 4`. Wie überall gilt es auch bei Klammern auf eine vernünftige Verwendung zu achten und diese wegzulassen, wo sie nicht benötigt werden (wie bei `2 + (3 + 4)`).

Tabelle 5.2. Auswertungsreihenfolge der Operatoren

Operator	**Bedeutung**
lambda	Lambda-Funktion
or	Logisches ODER
and	Logisches UND
not x	Logisches NICHT
in, not in	Mitgliedschaftstest (bei Sequenzen)
is, is not	Test auf Identität
<, <=, >, >=, !=, ==	Vergleiche
|	Bitweises ODER
^	Bitweises XOR
&	Bitweises UND
<<, >>	Bitweise Verschiebungen
+, -	Addition und Subtraktion
*, /, %	Multiplikation, Division und Rest
+x, -x	Positives und negatives Vorzeichen
~x	Bitweises NICHT
**	Potenzierung
x.attribut	Attributreferenzierung
x[index]	Indexierung
x[index:index]	Teilbereich einer Sequenz
f(parameter ...)	Funktionsaufruf
(ausdruck, ...)	Klammerung oder Bildung von Tupeln
[ausdruck, ...]	Bildung von Listen
{schluessel:wert, ...}	Bildung von Dictionaries
`ausdruck, ...`	Umwandlung in Stringdarstellung

In der Tabelle tauchen einige bisher noch nicht besprochene Operatoren auf, die in späteren Kapiteln erläutert werden.

Operatoren mit *gleicher Priorität* sind in der obigen Tabelle in der gleichen Zeile aufgeführt. Die Operatoren `+` und `-` haben z.B. die gleiche Priorität.

Auswertungsreihenfolge

Normalerweise werden die Operatoren in der Reihenfolge wie in der Tabelle angegeben ausgewertet. Sie können diese Reihenfolge jedoch durch die Verwendung von Klammern abändern. Wenn Sie etwa möchten, dass bei einem Ausdruck eine Addition vor einer Multiplikation ausgewertet wird, schreiben Sie `(2 + 3) * 4`.

Assoziativität

Operatoren mit gleicher Priorität werden innerhalb eines Ausdrucks von links nach rechts ausgewertet. Zum Beispiel wird `2 + 3 + 4` wie `(2 + 3) + 4` ausgewertet. Manche Operatoren, wie z.B. der Zuweisungsoperator, sind rechtsassoziativ. D.h. der Ausdruck `a = b = c` wird wie `a = (b = c)` behandelt.

Ausdrücke

Verwendung von Ausdrücken

Beispiel 5.1. Verwendung von Ausdrücken

(ausdruck.py [code/ausdruck.py])

```
#!/usr/bin/python

laenge = 5
breite = 2

flaeche = laenge * breite
print 'Die Flaeche ist', flaeche
print 'Der Umfang ist', 2 * (laenge + breite)
```

Ausgabe

```
$ python ausdruck.py
Die Flaeche ist 10
Der Umfang ist 14
```

So funktioniert es

Die Länge und die Breite des Rechtecks werden in zwei gleichnamigen Variablen gespeichert. Diese werden dann in den Ausdrücken verwendet, um die Fläche und den Umfang zu berechnen. Wir speichern das Ergebnis des Ausdrucks `laenge * breite` in der Variablen `flaeche` und geben diese mit der `print`-Anweisung auf dem Bildschirm aus. Im zweiten Fall übergeben wir den Wert des Ausdrucks `2 * (laenge + breite)` der `print`-Anweisung direkt als Parameter.

Beachten Sie auch, wie Python automatisch versucht, die Ausgabe lesbar zu gestalten. Obwohl wir zwischen `'Die Flaeche ist'` und der Variablen `flaeche` kein Leerzeichen eingefügt haben, fügt Python dieses für uns ein, so dass wir eine schöne, klare Ausgabe erhalten. Dadurch ist auch unser Programmcode leichter zu lesen, da wir uns keine Sorgen um die Formatierung der Ausgabe machen müssen. Das ist ein Beispiel dafür, wie Python Programmierern das Leben erleichtert.

Zusammenfassung

In diesem Abschnitt haben wir gesehen, wie Operatoren, Operanden und Ausdrücke verwendet werden - sie sind die Basisbausteine für jedes Programm. Als Nächstes werden wir sie zusammen mit Anweisungen in unseren Programmen verwenden.

Kapitel 6. Kontrollfluss

Einführung

In den Programmen, die wir bisher gesehen haben, gab es immer eine Folge von Anweisungen, die Python getreu in der angegebenen Reihenfolge ausgeführt hat. Was aber, wenn man den Programmfluss, d.h. die Reihenfolge, in der die Anweisungen ausgeführt werden, ändern will? Eine Anforderung könnte zum Beispiel sein, dass das Programm eine Entscheidung treffen soll und je nach Situation verschiedene Dinge tun soll, z.B. 'Guten Morgen' oder 'Guten Abend' ausgeben, je nach Tageszeit.

Wie Sie wahrscheinlich schon vermutet haben, kann dies mittels Kontrollfluss-Anweisungen erreicht werden. Es gibt in Python drei Kontrollfluss-Anweisungen - `if`, `for` und `while`.

Die if-Anweisung

Die `if`-Anweisung wird benutzt, um eine Bedingung abzuprüfen. Falls diese Bedingung erfüllt ist, wird ein Block von Anweisungen ausgeführt (der so genannte *if-Block*), andernfalls wird ein anderer Block von Anweisungen ausgeführt (der so genannte *else-Block*). Die `else`-Klausel ist dabei optional.

Gebrauch der if-Anweisung

Beispiel 6.1. Gebrauch der if-Anweisung (if.py [code/if.py])

```
#!/usr/bin/python

zahl = 23
geraten = int(raw_input('Geben Sie eine ganze Zahl ein: '))

if geraten == zahl:
 print 'Glueckwunsch, Sie haben es erraten.' # ein neuer Block beginnt hier
 print "(Aber Sie gewinnen leider keinen Preis!)" # und hier hoert er auf
elif geraten < zahl:
 print 'Nein, die Zahl ist etwas hoeher.' # noch ein Block
 # Sie koennen in dem Block tun, was sie wollen ...
else:
 print 'Nein, die Zahl ist etwas niedriger.'
 # hierhin gelangt man, wenn geraten > zahl ist

print 'Fertig.'
# Diese letzte Anweisung wird immer am Ende nach der if-Anweisung ausgefuehrt
```

Output

```
$ python if.py
Geben Sie eine ganze Zahl ein: 50
Nein, die Zahl ist etwas niedriger.
```

```
Fertig.
$ python if.py
Geben Sie eine ganze Zahl ein: 22
Nein, die Zahl ist etwas hoeher.
Fertig.
$ python if.py
Geben Sie eine ganze Zahl ein: 23
Glueckwunsch, Sie haben es erraten.
(Aber Sie gewinnen leider keinen Preis!)
Fertig.
```

So funktioniert es

In diesem Programm nehmen wir als Benutzereingabe einen Versuch entgegen, eine geheime Zahl zu erraten, und überprüfen, ob diese Zahl erraten wurde. Wir setzen die Variable `zahl` auf irgendeine von uns ausgesuchte ganze Zahl, etwa `23`. Dann nehmen wir den Rateversuch des Benutzers mit Hilfe der Funktion `raw_input()` entgegen. Funktionen sind nichts weiter als wieder verwendbare Programmteile. Wir werden mehr darüber im nächsten Kapitel erfahren.

Wir übergeben der eingebauten Funktion `raw_input` einen String, den diese auf dem Bildschirm ausgibt, bevor sie auf eine Benutzereingabe wartet. Sobald wir etwas eingegeben und die **Enter**-Taste gedrückt haben, liefert die Funktion die Eingabe zurück, die im Fall der Funktion `raw_input` ein String ist. Wir verwandeln dann diesen String mittels `int` in eine Ganzzahl, die wir in der Variablen `geraten` abspeichern. Eigentlich ist `int` eine Klasse, aber alles was Sie momentan wissen müssen, ist, dass sie hiermit einen String in eine Ganzzahl umwandeln können (unter der Annahme, dass der String eine gültige Ganzzahl als Text enthält).

Als Nächstes vergleichen wir die vom Benutzer geratene Zahl mit der von uns ausgewählten Zahl. Wenn sie gleich sind, geben wir eine Erfolgsmeldung aus. Beachten Sie, dass wir verschiedene Einrücktiefen benutzen, um Python klarzumachen, welche Anweisung zu welchem Block gehört. Darum sind Einrückungen in Python so wichtig und man muss auf den Leerraum am Zeilenanfang achten (im Gegensatz zu anderen Programmiersprachen, wo Blöcke üblicherweise durch irgendeine Art von Klammern oder Schlüsselworte wie BEGIN..END markiert werden). Ich hoffe, dass Sie sich an die Regel 'einen Tabulator pro Einrückungstiefe' halten, erinnern Sie sich?

Beachten Sie, dass die `if`-Anweisung einen Doppelpunkt am Ende erfordert - hiermit wird in Python grundsätzlich angezeigt, dass ein neuer Anweisungsblock folgt.

Falls die beiden Zahlen nicht gleich waren, überprüfen wir, ob die geratene Zahl kleiner als unsere Zahl ist, und falls dies der Fall ist, informieren wir den Benutzer, dass unsere Zahl ein wenig höher ist. Was wir hier verwendet haben, ist eine `elif`-Klausel, mit der man zwei verschachtelte `if else`-`if else`-Anweisungen als eine kombinierte `if-elif-else`-Anweisung schreiben kann. Dadurch wird das Programm vereinfacht und die nötige Einrückungstiefe reduziert.

Die `elif`- und `else`-Anweisungen müssen ebenfalls einen Doppelpunkt am Ende der logischen Zeile haben, gefolgt von dem jeweiligen Anweisungsblock (natürlich mit der richtigen Einrückungstiefe).

Sie können innerhalb des if-Blocks einer `if`-Anweisung eine weitere `if`-Anweisung unterbringen und so weiter - dies nennt man eine verschachtelte `if`-Anweisung.

Denken Sie daran, dass die `elif`- und `else`-Teile optional sind. Eine minimale gültige `if`-Anweisung ist:

```
if True:
```

```
 print 'Ja, es ist wahr.'
```

Nachdem Python die gesamte `if`-Anweisung sowie die zugehörigen `elif`- und `else`-Klauseln abgearbeitet hat, fährt es mit der nächsten Anweisung in dem Block fort, der die `if`-Anweisung enthält. In diesem Fall ist es der Hauptblock, in dem die Programmausführung beginnt, und die nächste Anweisung lautet `print 'Fertig.'`. Danach erkennt Python das Programmende und beendet die Programmausführung.

Dies ist zwar ein sehr kleines Programm, aber ich habe dennoch auf eine Menge Dinge hingewiesen, die Sie selbst in diesem kleinen Programm beachten sollten. All dies ist im Grunde unmittelbar einleuchtend (und für Leute mit C/C++-Hintergrund überraschend einfach) und wenn Sie sich diese Dinge einmal bewusst klar gemacht haben, werden Sie sich schnell mit ihnen anfreunden, und sie werden ihnen bald völlig 'natürlich' vorkommen.

Anmerkung für C/C++-Programmierer

In Python gibt es keine `switch`-Anweisung. Sie können eine `if..elif..elif..else`-Anweisung benutzen, um das Gleiche zu erreichen, und oft stattdessen auch eine *Abbildung* (ein Dictionary) benutzen.

Die while-Anweisung

Die `while`-Anweisung ermöglicht es Ihnen, einen Anweisungsblock wiederholt auszuführen, solange eine zugehörige Bedingung erfüllt ist. Eine `while`-Anweisung ist ein Beispiel für eine so genannte *Schleifen*-Anweisung. Die `while`-Anweisung kann eine optionale `else`-Klausel haben.

Gebrauch der while-Anweisung

Beispiel 6.2. Gebrauch der while-Anweisung (while.py [code/while.py])

```
#!/usr/bin/python

zahl = 23
weiter = True

while weiter:
 geraten = int(raw_input('Geben Sie eine ganze Zahl ein: '))

 if zahl == geraten:
  print 'Glueckwunsch, Sie haben es erraten.'
  weiter = False # das fuehrt zum Ende der while-Schleife
 elif geraten < zahl:
  print 'Nein, die Zahl ist etwas hoeher.'
 else:
  print 'Nein, die Zahl ist etwas niedriger.'
else:
 print 'Die while-Schleife wurde beendet.'
 # Hier koennte man noch weitere Dinge tun.

print 'Fertig.'
```

Ausgabe

```
$ python while.py
Geben Sie eine ganze Zahl ein: 50
Nein, die Zahl ist etwas niedriger.
Geben Sie eine ganze Zahl ein: 22
Nein, die Zahl ist etwas hoeher.
Geben Sie eine ganze Zahl ein: 23
Glueckwunsch, Sie haben es erraten.
Die while-Schleife wurde beendet.
Fertig.
```

So funktioniert es

In diesem Programm spielen wir immer noch das Ratespiel, aber mit der Verbesserung, dass der Benutzer nun so lange weiter raten kann, bis er richtig geraten hat - man muss das Programm nicht mehr für jeden Rateversuch neu starten, wie wir es zuvor getan haben. Hier zeigt sich sehr gut der Nutzen einer `while`-Anweisung.

Wir verschieben die `raw_input`- und `if`-Anweisung in die `while`-Schleife und setzen die Variable `weiter` vor dem Betreten der while-Schleife auf `True` (logisch wahr). In der Schleife wird zuerst geprüft, ob die Variable `weiter` immer noch als logischer Wert wahr ist, und wenn dies der Fall ist, dann wird der zugehörige *while-Block* ausgeführt. Nachdem dieser Block ausgeführt wurde, wird die Bedingung erneut geprüft, die in diesem Fall aus der Variablen `weiter` besteht. Wenn Sie weiterhin logisch wahr ist, dann wird der while-Block wieder ausgeführt, ansonsten wird der optionale else-Block ausgeführt und danach die nächste Anweisung.

Der `else`-Block wird ausgeführt, wenn die `while`-Bedingung als logischer Wert falsch (`False`) wird - dies kann sogar bereits beim ersten Mal passieren, wenn die Bedingung überprüft wird. Wenn eine `while`-Schleife eine `else`-Klausel hat, dann wird diese stets ausgeführt, es sei denn, dass die Schleife eine `break`-Anweisung besitzt (darauf kommen wir in diesem Kapitel noch zu sprechen) oder dass die `while`-Schleife endlos läuft, ohne jemals abzubrechen.

Die Werte `True` und `False` werden *Wahrheitswerte* genannt. Es handelt sich um den so genannten logischen oder booleschen Datentyp, und Sie können sich diese Werte als gleichbedeutend mit den Werten `1` bzw. `0` denken, und in dem Programm auch `1` statt `True` bzw. `0` statt `False` schreiben, aber dann würde nicht ganz deutlich, dass die Variable `weiter` eigentlich einen Wahrheitswert repräsentiert, und nicht eine Zahl.

Solange keine `break`-Anweisung im Spiel ist, ist der `else`-Block eigentlich überflüssig, denn man kann in dem Fall die darin befindlichen Anweisungen einfach im gleichen Block, in dem sich die `while`-Anweisung befindet, hinter die `while`-Schleife schreiben um das gleiche Ergebnis zu erzielen.

Anmerkung für C/C++-Programmierer

Denken Sie daran, dass man unter Python bei einer `while`-Schleife auch eine `else`-Klausel zur Verfügung hat.

Die for-Schleife

Die `for..in`-Anweisung ist eine weitere Schleifen-Anweisung, die eine *Iteration* über eine Sequenz von Objekten durchführt, d.h. sie durchläuft jedes Objekt der Sequenz. Wir werden in späteren Kapiteln Genaueres über Sequenzen erfahren. Was Sie momentan wissen müssen, ist nur, dass eine Sequenz einfach eine geordnete Abfolge von einzelnen Objekten ist.

Gebrauch der for-Anweisung

Beispiel 6.3. Gebrauch der for-Anweisung (for.py [code/for.py])

```
#!/usr/bin/python

for i in range(1, 5):
 print i
else:
 print 'Die for-Schleife ist zu Ende.'
```

Ausgabe

```
$ python for.py
1
2
3
4
Die for-Schleife ist zu Ende.
```

So funktioniert es

In diesem Programm geben wir eine *Sequenz* von Zahlen aus. Wir generieren diese Sequenz von Zahlen mittels der eingebauten Funktion `range`.

In diesem Beispiel übergeben wir ihr zwei Zahlen, und `range` gibt eine Sequenz von Zahlen zurück, die von der ersten bis zur zweiten angegebenen Zahl läuft. Zum Beispiel ergibt `range(1,5)` die Sequenz `[1, 2, 3, 4]`. Normalerweise benutzt `range` eine Schrittweite von 1. Wenn wir der Funktion `range` eine dritte Zahl übergeben, dann wird diese als Schrittweite benutzt. Zum Beispiel ergibt `range(1,5,2)` die Sequenz `[1,3]`. Denken Sie daran, dass sich der Bereich *bis zur* zweiten Zahl erstreckt, diese aber **nicht** mit einschließt.

Die `for`-Schleife durchläuft dann diesen Zahlenbereich - `for i in range(1,5)` ist gleichbedeutend mit `for i in [1, 2, 3, 4]`, was bewirkt, dass der Variablen `i` der Reihe nach jede Zahl (allgemein jedes Objekt) in der Sequenz zugewiesen wird, und dann der Anweisungsblock für jeden dieser Werte von `i` ausgeführt wird. In diesem Fall geben wir im Anweisungsblock einfach diesen Wert aus.

Beachten Sie, dass der `else`-Teil optional ist. Wenn er hinzugefügt wird, wird er immer einmal nach dem Ende der `for`-Schleife ausgeführt, außer wenn die Schleife durch eine break-Anweisung abgebrochen wird.

Denken Sie daran, dass die `for..in`-Schleife mit jeder beliebigen Sequenz funktioniert. In diesem Fall haben wir eine mit der eingebauten `range`-Funktion gebildete Liste von Zahlen, aber wir können ganz allgemein jede Art von Sequenz mit beliebig gearteten Objekten benutzen. Wir werden dieser Idee in späteren Kapiteln noch einmal genauer nachgehen.

Anmerkung für C/C++/Java/C#-Programmierer

Die `for`-Schleife in Python unterscheidet sich radikal von der `for`-Schleife in C/C++. C#-Programmierer werden bemerken, dass die `for`-Schleife in Python der `foreach`-Schleife

in C# ähnlich ist. Java-Programmierer werden bemerken, dass sie `for (int i : IntArray)` in Java 1.5 entspricht.

Wo man `for (int i = 0; i < 5; i++)` in C/C++ schreibt, schreibt man in Python einfach `for i in range(0,5)`. Wie Sie sehen, ist die `for`-Schleife in Python einfacher, ausdrucksstärker und weniger fehleranfällig.

Die break-Anweisung

Die `break`-Anweisung wird benutzt, um aus einer Programmschleife vorzeitig *auszubrechen*, d.h. die Ausführung der Schleifenanweisung zu beenden, auch wenn die Schleifenbedingung noch nicht `False` geworden ist, oder die Sequenz in einer `for`-Schleife noch nicht vollständig abgearbeitet wurde.

Eine wichtige Anmerkung ist, dass wenn man aus einer `for`- oder `while`-Schleife *ausbricht*, ein eventuell zur Schleife gehörender `else`-Block **nicht** ausgeführt wird.

Gebrauch der break-Anweisung

Beispiel 6.4. Gebrauch der break-Anweisung (break.py [code/break.py])

```
#!/usr/bin/python

while True:
 s = raw_input('Geben Sie etwas ein: ')
 if s == 'ende':
  break
 print 'Die Laenge des Strings ist', len(s)
print 'Fertig.'
```

Ausgabe

```
$ python break.py
Geben Sie etwas ein: Programmieren mit Elan
Die Laenge des Strings ist 22
Geben Sie etwas ein: und die Arbeit wird getan,
Die Laenge des Strings ist 26
Geben Sie etwas ein: willst du Spass haben daran:
Die Laenge des Strings ist 28
Geben Sie etwas ein:   Nimm Python!
Die Laenge des Strings ist 13
Geben Sie etwas ein: ende
Fertig.
```

So funktioniert es

In diesem Programm nehmen wir in einer Schleife Benutzereingaben entgegen und geben die Länge jeder Eingabe jedes Mal aus. Wir haben eine besondere Bedingung vorgesehen, unter der das Pro-

gramm beendet werden soll, indem wir überprüfen, ob die Benutzereingabe `'ende'` ist. Wir beenden das Programm, in dem wir aus der Schleife *ausbrechen* und damit das Programmende erreichen.

Die Länge des Eingabestrings kann ermittelt werden, indem man die eingebaute Funktion `len` benutzt.

Denken Sie daran, dass die `break`-Anweisung auch bei `for`-Schleifen möglich ist.

G2s poetisches Python

Die Eingabe, die ich hier benutzt habe, ist ein kleines Gedicht, das ich **G2s poetisches Python** genannt habe und das sogar in einer deutschen Fassung existiert:

```
Programmieren mit Elan
und die Arbeit wird getan,
willst du Spass haben daran:
        Nimm Python!
```

Die continue-Anweisung

Die `continue`-Anweisung wird benutzt, um Python mitzuteilen, dass es die restlichen Anweisungen in der aktuellen Schleife überspringen soll und direkt mit dem nächsten Schleifendurchlauf *fortfahren* soll.

Gebrauch der continue-Anweisung

Beispiel 6.5. Gebrauch der continue-Anweisung (continue.py [code/continue.py])

```
#!/usr/bin/python

while True:
 s = raw_input('Geben Sie etwas ein: ')
 if s == 'ende':
  break
 if len(s) < 3:
  continue
 print 'Die Laenge der Eingabe ist ausreichend.'
 # Verarbeite die Eingabe hier irgendwie...
```

Ausgabe

```
$ python continue.py
Geben Sie etwas ein: a
Geben Sie etwas ein: 12
Geben Sie etwas ein: abc
Die Laenge der Eingabe ist ausreichend.
Geben Sie etwas ein: ende
```

So funktioniert es

In diesem Programm nehmen wir Benutzereingaben entgegen, die wir aber nur dann verarbeiten, wenn sie mindestens drei Zeichen lang sind. Wir benutzen daher die `len`-Funktion, um die Länge der Eingabe zu bestimmen, und wenn die Länge weniger als 3 Zeichen beträgt, überspringen wir die restlichen Anweisungen im while-Block, indem wir die `continue`-Anweisung benutzen. Andernfalls werden die restlichen Anweisungen der while-Schleife ausgeführt, und wir können die Eingabe auf irgendeine Weise verarbeiten.

Beachten Sie, dass die `continue`-Anweisung auch bei einer `for`-Schleife funktioniert.

Zusammenfassung

Wir haben gesehen, wir man die drei Kontrollfluss-Anweisungen - `if`, `while` und `for` - im Zusammenspiel mit den beiden zugehörigen Anweisungen `break` und `continue` verwendet. Diese Anweisungen gehören zu den Teilen der Sprache Python, die man am häufigsten braucht und daher ist es wesentlich, mit ihnen vertraut zu werden.

Als Nächstes sehen wir, wie wir Funktionen erzeugen und benutzen können.

Kapitel 7. Funktionen

Einführung

Funktionen sind wieder verwendbare Teile eines Programms. Sie ermöglichen es Ihnen, einem Block von Anweisungen einen Namen zu geben. Sie können dann den Block irgendwo in Ihrem Programm beliebig oft laufen lassen, indem Sie seinen Namen benutzen. Man spricht hier auch davon, eine Funktion *aufzurufen*. Wir haben bereits viele eingebauten Funktionen wie etwa `len` oder `range` verwendet.

Funktionen werden mit dem Schlüsselwort `def` **definiert**. Diesem Schlüsselwort folgt ein *Bezeichner* als Name für die Funktion. Danach kommt ein Klammerpaar, welches einige Variablennamen enthalten kann. Die Zeile endet mit einem Doppelpunkt. Danach kommt der Block mit den Anweisungen, die zu der Funktion gehören. An einem Beispiel sehen wir, wie einfach das eigentlich ist:

Definition einer Funktion

Beispiel 7.1. Definition einer Funktion (funktion1.py [code/funktion1.py])

```
#!/usr/bin/python

def sagHallo():
 print 'Hallo Welt!' # Block gehoert zur Funktion
# Ende der Funktion

sagHallo() # Aufruf der Funktion
```

Ausgabe

```
$ python funktion1.py
Hallo Welt!
```

So funktioniert es

Wir definieren eine Funktion `sagHallo`, indem wir die oben erläuterte Syntax verwenden. Diese Funktion nimmt keine Parameter entgegen, weswegen keine Variablen zwischen den Klammern angegeben sind. Parameter dienen einer Funktion einfach als Eingabewerte. Auf diese Weise können wir ihr verschiedene Werte übergeben und entsprechende Ausgaben als Ergebnis zurückbekommen.

Funktionsparameter

Eine Funktion kann als Parameter Werte verarbeiten, die Sie der Funktion übergeben, damit die Funktion unter Benutzung dieser Werte irgendetwas *tun* kann. Diese Parameter verhalten sich genauso wie gewöhnliche Variablen, nur dass deren Werte nicht innerhalb der Funktion zugewiesen werden, sondern beim Aufruf der Funktion definiert werden.

Parameter werden innerhalb des Klammernpaars der Funktionsdefinition festgelegt und durch Kommas voneinander getrennt. Wenn wir die Funktion später aufrufen, legen wir die Werte auf die gleiche Weise fest. Beachten Sie aber die Terminologie - während die in der Funktionsdefinition angegebenen Namen *Parameter* heißen, werden die Werte, die Sie beim Funktionsaufruf übergeben, *Argumente* genannt.

Funktionsparameter benutzen

Beispiel 7.2. Funktionsparameter benutzen (funk_param.py [code/funk_param.py])

```
#!/usr/bin/python

def printMax(a, b):
 if a > b:
  print a, 'ist Maximalwert'
 else:
  print b, 'ist Maximalwert'

printMax(3, 4) # uebergebe Zahlen direkt als Literale

x = 5
y = 7

printMax(x, y) # uebergebe Variablen als Argumente
```

Ausgabe

```
$ python funk_param.py
4 ist Maximalwert
7 ist Maximalwert
```

So funktioniert es

Wir definieren hier eine Funktion `printMax` mit zwei Variablen `a` und `b`. Wir berechnen die größere Zahl indem wir eine einfache `if..else`-Anweisung verwenden, und geben dann die größere Zahl mit `print` aus.

Beim ersten Aufruf der Funktion `printMax` übergeben wir die Zahlen direkt als Argumente. Beim zweiten Aufruf benutzen wir hierfür Variablen. `printMax(x, y)` bewirkt, dass der Wert des Argumentes `x` dem Parameter `a` zugewiesen wird, und der Wert des Argumentes `y` dem Parameter `b`. Für die Funktion `printMax` ist es egal, auf welche Weise man ihr die Argumente übergibt, sie arbeitet immer gleich.

Lokale Variablen

Wenn wir Variablen innerhalb einer Funktion definieren, sind sie in keinster Weise mit anderen Variablen des gleichen Namens verbunden, die außerhalb der Funktion benutzt werden, d.h. die Variablennamen sind für die Funktion *lokal* gültig. Dies nennt man den *Geltungsbereich* einer Variablen. Alle

Variablen haben als Geltungsbereich den Block, in dem sie erklärt werden, beginnend an der Stelle, wo der Name definiert wird.

Benutzung lokaler Variablen

Beispiel 7.3. Benutzung lokaler Variablen (funk_lokal.py [code/funk_lokal.py])

```
#!/usr/bin/python

def funk(x):
 print 'x ist', x
 x = 2
 print 'Lokales x ist jetzt', x

x = 50
funk(x)
print 'x ist immer noch', x
```

Ausgabe

```
$ python funk_lokal.py
x ist 50
Lokales x ist jetzt 2
x ist immer noch 50
```

So funktioniert es

Wenn wir das erste mal in der Funktion den *Wert* mit dem Namen `x` verwenden, benutzt Python hierfür den Wert des in der Funktion angegeben gleichnamigen Parameters.

Als nächstes weisen wir `x` den Wert `2` zu. Der Name `x` ist innerhalb unserer Funktion lokal gültig. Wenn wir also den Wert von `x` in der Funktion verändern, wird hierdurch das im Hauptblock definierte `x` nicht beeinflusst.

Mit der `print`-Anweisung am Ende bestätigen wir, dass der Wert von `x` im Hauptblock tatsächlich unverändert geblieben ist.

Benutzung der global-Anweisung

Wenn Sie einem außerhalb der Funktion definierten Namen einen Wert zuweisen wollen, dann müssen Sie Python mitteilen, dass der Name nicht lokal ist, sondern *global*. Wir können dies mit der `global`-Anweisung erreichen. Es ist unmöglich, ohne die `global`-Anweisung einer außerhalb der Funktion definierten Variablen einen Wert zuzuweisen.

Sie können die Werte außerhalb der Funktion definierter Variablen benutzen (vorausgesetzt, dass es innerhalb der Funktion keine gleichnamige Variable gibt). Dies wird jedoch nicht empfohlen und sollte vermieden werden, da es einem Leser des Programms nicht sofort klar ist, wo diese Variable definiert wurde. Wenn man die `global`-Anweisung benutzt, ist damit sofort klar, dass die Variable in einem äußeren Block definiert wurde.

Beispiel 7.4. Benutzung der global-Anweisung (funk_global.py [code/funk_global.py])

```
#!/usr/bin/python

def funk():
 global x

 print 'x ist', x
 x = 2
 print 'Globales x ist jetzt', x

x = 50
funk()
print 'Der Wert von x ist', x
```

Ausgabe

```
$ python funk_global.py
x ist 50
Globales x ist jetzt 2
Der Wert von x ist 2
```

So funktioniert es

Die Anweisung `global` wird benutzt, um `x` als globale Variable zu erklären. Wenn wir also `x` innerhalb der Funktion einen Wert zuweisen, wird sich diese Veränderung widerspiegeln, wenn wir den Wert von `x` im Hauptblock benutzen.

Sie können in derselben `global`-Anweisung mehr als eine globale Variable angeben. Zum Beispiel: `global x, y, z`.

Voreingestellte Argumentwerte

Bei einigen Funktionen möchten Sie vielleicht die Parameter ganz oder teilweise *optional* machen und voreingestellte Werte verwenden, wenn der Benutzer für solche Parameter keine Werte bereitstellt. Dies kann man mit voreingestellten Argumentwerten erreichen. Sie können voreingestellte Argumentwerte für Parameter angeben, indem Sie in der Funktionsdefinition nach dem Parameternamen den Zuweisungsoperator (=) und danach den voreingestellten Wert schreiben.

Beachten Sie, dass der voreingestellte Wert für das Argument eine Konstante sein sollte. Genauer gesagt sollte der voreingestellte Wert unveränderbar sein - wir werden in den nächsten Kapiteln näher erklären, was das heißt. Halten Sie dies zuerst einmal so in Erinnerung.

Verwendung voreingestellter Argumentwerte

Beispiel 7.5. Verwendung voreingestellter Argumentwerte (funk_vorein.py [code/funk_vorein.py])

```
#!/usr/bin/python

def sag(nachricht, wiederholungen = 1):
 print nachricht * wiederholungen

sag('Hallo')
sag('Welt', 5)
```

Ausgabe

```
$ python funk_vorein.py
Hallo
WeltWeltWeltWeltWelt
```

So funktioniert es

Die Funktion `sag` wird verwendet, um einen String so oft wie gewünscht auszugeben. Wenn wir keinen Wert übergeben, wird der String wie voreingestellt nur einmal ausgegeben. Wir erreichen dies, indem wir für den Parameter `wiederholungen` den voreingestellten Argumentwert `1` angeben.

Beim ersten Aufruf von `sag` übergeben wir nur den String, so dass dieser einmal ausgegeben wird. Beim zweiten Aufruf von `sag` übergeben wir sowohl den String an als auch ein Argument `5`, das angibt, dass *sag* den String 5 mal ausgeben soll.

Wichtig

Nur die Parameter am Ende der Parameterliste können voreingestellte Werte besitzen, d.h. es darf kein Parameter mit einem voreingestellten Argumentwert vor einem Parameter ohne voreingestellten Argumentwert nach der Reihenfolge der Parameter vorkommen, wie sie in der Parameterliste der Funktion angegeben sind.

Der Grund hierfür ist, dass die Werte den Parametern durch Ihre Position zugewiesen werden. Zum Beispiel ist `def funk(a, b=5)` eine gültige Funktionsdefinition, wohingegen `def funk(a=5, b)` *nicht gültig* ist.

Schlüsselwort-Argumente

Wenn Sie Funktionen mit vielen Parametern haben, aber nur einige von ihnen angeben wollen, dann können Sie solchen Parametern Werte übergeben, indem Sie sie benennen. Man spricht hier auch von *Schlüsselwort-Argumenten*. Wir benutzen hier den Namen (ein Schlüsselwort) anstatt (wie bisher) die Position, um die Argumente der Funktion zu spezifizieren.

Dies hat zwei *Vorteile*. Erstens: Die Benutzung der Funktion wird einfacher, da wir uns keine Gedanken über die Reihenfolge der Argumente machen müssen. Zweites: Wir brauchen nur denjenigen Parametern Werte zu übergeben, die wir wirklich benötigen, vorausgesetzt, die anderen Parameter haben für uns akzeptable voreingestellte Argumentwerte.

Schlüsselwort-Argumente benutzen

Beispiel 7.6. Schlüsselwort-Argumente benutzen (funk_schluessel.py [code/funk_schluessel.py])

```
#!/usr/bin/python

def funk(a, b=5, c=10):
 print 'a ist', a, 'und b ist', b, 'und c ist', c

funk(3, 7)
funk(25, c=24)
funk(c=50, a=100)
```

Ausgabe

```
$ python funk_schluessel.py
a ist 3 und b ist 7 und c ist 10
a ist 25 und b ist 5 und c ist 24
a ist 100 und b ist 5 und c ist 50
```

So funktioniert es

Die Funktion `funk` hat einen Parameter ohne voreingestellten Argumentwert, gefolgt von zwei Parametern mit voreingestellten Argumentwerten.

Wenn wir die Funktion zum ersten Mal aufrufen, `funk(3, 7)`, bekommt der Parameter `a` den Wert `3`, der Parameter `b` den Wert `5` und `c` den voreingestellten Wert `10` zugewiesen.

Beim zweiten Aufruf, `funk(25, c=24)`, bekommt die Variable `a` den Wert 25 aufgrund der Position des Argumentes. Danach bekommt der Parameter `c` den Wert `24` aufgrund seiner Benennung, d.h. als Schlüsselwort-Argument. Die Variable `b` erhält den voreingestellten Wert `5`.

Beim dritten Aufruf der Funktion, `funk(c=50, a=100)`, verwenden wir ausschließlich Schlüsselwort-Argumente, um die Werte anzugeben. Beachten Sie, dass wir den Wert des Parameters `c` vor dem des Parameters `a` angeben, obwohl `a` in der Funktionsdefinition vor `c` angegeben ist.

Die return-Anweisung

Die `return`-Anweisung wird benutzt, um aus einer Funktion *zurückzukehren*, d.h. die Funktion an dieser Stelle wieder zu verlassen. Bei Bedarf können wir die Funktion an dieser Stelle auch einen *Wert zurückgeben* lassen.

Verwendung der return-Anweisung

Beispiel 7.7. Verwendung der return-Anweisung (funk_return.py [code/funk_return.py])

```
#!/usr/bin/python

def maximum(x, y):
 if x > y:
  return x
 else:
  return y

print maximum(2, 3)
```

Ausgabe

```
$ python funk_return.py
3
```

So funktioniert es

Die Funktion `maximum` gibt als Wert das Maximum ihrer Parameter zurück. In diesem Fall sind dies Zahlen, die der Funktion als Argumente übergeben werden. In der Funktion wird eine einfache `if..else`-Anweisung benutzt, um den größeren Wert zu finden, und dieser Wert wird dann mit `return` zurückgegeben.

Beachten Sie, dass eine `return`-Anweisung ohne Wert gleichbedeutend ist mit `return None`. `None` ist ein besonderer Datentyp in Python, der einfach für *Nichts* steht. Man benutzt ihn zum Beispiel, um anzuzeigen, dass eine Variable keinen speziellen Wert hat, wenn sie den Wert `None` besitzt.

Jede Funktion hat am Ende implizit die Anweisung `return None`, wenn Sie nicht stattdessen Ihre eigene `return`-Anweisung geschrieben haben. Sie können das ausprobieren, indem Sie `print eineFunktion()` ausführen, wobei `eineFunktion` eine Funktion ohne `return`-Anweisung sein soll, etwa:

```
def eineFunktion():
 pass
```

Die `pass`-Anweisung wird in Python benutzt, um einen leeren Anweisungsblock anzuzeigen.

DocStrings

In Python gibt es eine sehr elegante Möglichkeit, Funktionen mit so genannten *Dokumentations-Strings* oder kurz *DocStrings* zu erläutern. DocStrings sind ein wichtiges Werkzeug, von dem Sie Gebrauch machen sollten, da sie dazu beitragen, das Programm zu dokumentieren und es besser verständlich zu machen. Bemerkenswert daran ist z.B., dass wir den DocString etwa einer Funktion sogar während des Programmablaufs abfragen können!

Verwendung von DocStrings

Beispiel 7.8. Verwendung von DocStrings (funk_doc.py [code/funk_doc.py])

```
#!/usr/bin/python

def printMax(x, y):
 """Gib das Maximum von zwei Zahlen aus.

 Die beiden Werte muessen ganze Zahlen sein."""
 x = int(x) # Umwandlung in ganze Zahlen, falls moeglich
 y = int(y)

 if x > y:
  print x, 'ist das Maximum'
 else:
  print y, 'ist das Maximum'

printMax(3, 5)
print printMax.__doc__
```

Ausgabe

```
$ python funk_doc.py
5 ist das Maximum
Gib das Maximum von zwei Zahlen aus.

 Die beiden Werte muessen ganze Zahlen sein.
```

So funktioniert es

Ein String in der ersten logischen Zeile einer Funktion ist der *DocString* für diese Funktion. Beachten Sie, dass man DocStrings auch für Module und Klassen verwenden kann, die wir in den jeweiligen Kapiteln noch kennen lernen werden.

Als Konvention benutzt man für DocStrings einen mehrzeiligen String, wobei die erste Zeile mit einem Großbuchstaben beginnt und mit einem Punkt endet. Danach bleibt die zweite Zeile leer. Dann kommt ab der dritten Zeile eine ausführlichere Erläuterung. Es wird *ausdrücklich empfohlen*, dieser Konvention in allen DocStrings für alle nichttrivialen Funktionen zu folgen. Bei sehr einfachen Funktionen verwendet man einen einzeiligen String, aber dennoch die dreifachen Anführungszeichen. Weitere Einzelheiten dieser Konvention finden Sie in der Python-Dokumentation unter dem Stichwort *PEP 257*.

Wir erhalten den DocString der Funktion `printMax` über das mit der Funktion verbundene Attribut `__doc__` (beachten Sie die doppelten Unterstriche). Behalten Sie im Kopf, dass in Python *alles* als ein Objekt behandelt wird, also auch Funktionen. Wir werden im Kapitel über Klassen mehr über Objekte lernen.

Wenn Sie die Hilfefunktion `help()` in Python benutzt haben, haben Sie bereits den Gebrauch von DocStrings gesehen! Sie macht nämlich nichts anderes, als das `__doc__`-Attribut der Funktion auszuwerten und es in benutzerfreundlicher Weise auszugeben. Sie können das an der obigen Funktion ausprobieren - fügen Sie dem Programm einfach die Zeile `help(printMax)` hinzu. Denken Sie daran, dass Sie die Hilfefunktion mit der Taste **q** beenden.

Automatisierte Werkzeuge können so die Dokumentation Ihres Programms abrufen. Deshalb möchte ich Ihnen *ausdrücklich empfehlen*, allen nichttrivialen Funktionen DocStrings hinzuzufügen. Der

Befehl **pydoc**, der mit zur Python-Distribution gehört, arbeitet ähnlich wie `help()` auf der Basis von DocStrings.

Zusammenfassung

Wir haben viele Aspekte von Funktionen kennen gelernt, aber beachten Sie, dass wir noch nicht alle ihrer Aspekte behandelt haben. Jedoch haben wir die meisten Dinge behandelt, die Sie hinsichtlich Funktionen in Python im Alltag wissen müssen.

Als Nächstes werden wir lernen, wie man Python-Module sowohl verwendet als auch erzeugt.

Kapitel 8. Module

Einführung

Sie haben gesehen, wie Sie Code in Ihren Programm wieder verwenden können, indem Sie Funktionen nur einmal definieren. Was machen Sie aber, wenn Sie eine ganze Anzahl von Funktionen in anderen Programmen benutzen wollen? Wie Sie vielleicht vermutet haben, sind Module die Lösung dafür. Ein Modul ist im Grunde nur eine Datei, die alle von Ihnen definierten Funktionen und Variablen beinhaltet. Um ein Modul in anderen Programmen wiederverwenden zu können, **muss** der Dateiname eines Moduls die Endung `.py` haben.

Ein Modul kann von einem anderen Programm *importiert* werden, um von dessen Funktionalität Gebrauch zu machen. Genauso können wir auch die Python-Standardbibliothek verwenden. Zunächst werden wir sehen, wie wir die Module der Standardbibliothek verwenden können.

Benutzung des sys-Moduls

Beispiel 8.1. Benutzung des sys-Moduls (beispiel_sys.py [code/beispiel_sys.py])

```
#!/usr/bin/python

import sys

print 'Die Kommandozeilenparameter sind:'
for i in sys.argv:
 print i

print '\n\nDer PYTHONPATH ist', sys.path, '\n'
```

Ausgabe

```
$ python beispiel_sys.py Wir sind Argumente
Die Kommandozeilenparameter sind:
beispiel_sys.py
Wir
sind
Argumente


Der PYTHONPATH ist ['/home/swaroop/byte/code', '/usr/lib/python23.zip',
'/usr/lib/python2.3', '/usr/lib/python2.3/plat-linux2',
'/usr/lib/python2.3/lib-tk', '/usr/lib/python2.3/lib-dynload',
'/usr/lib/python2.3/site-packages', '/usr/lib/python2.3/site-packages/gtk-2.0']
```

So funktioniert es

Zuerst *importieren* wir das Modul `sys` mittels der `import`-Anweisung. Im Wesentlich teilen wir Python dadurch mit, dass wir dieses Modul in unserem Programm verwenden wollen. Das Modul `sys` enthält Funktionalität, die mit dem Python-Interpreter selbst und dessen Umgebung zu tun hat.

Wenn Python die Anweisung `import sys` ausführt, sucht es nach dem Modul `sys.py` in einem der Verzeichnisse, die in seiner Variablen `sys.path` aufgelistet werden. Wird die Datei gefunden, so werden die Anweisungen im Hauptblock dieses Moduls ausgeführt. Das Modul wird für Sie dann *verfügbar* gemacht. Beachten Sie, dass die Initialisierung nur beim *ersten* Mal durchgeführt wird, wenn wir ein Modul importieren. Übrigens steht 'sys' als Abkürzung für 'System'.

Der Zugriff auf die Variable `argv` im Modul `sys` geschieht mittels der Punktnotation `sys.argv`. Ein Vorteil dieser Vorgehensweise besteht darin, dass der Name nicht mit einer anderen Variablen des Namens `argv` innerhalb Ihres Programms kollidiert. Auch weist er eindeutig darauf hin, dass er Teil des Moduls `sys` ist.

Die Variable `sys.argv` ist eine *Liste* von Strings (Listen werden in späteren Abschnitten genauer erklärt). Diese spezielle Liste `sys.argv` besteht aus den *Kommandozeilenparametern*, d.h. den zusätzlichen Argumenten, die Ihrem Programm über die Kommandozeile übergegeben wurden.

Wenn Sie eine Entwicklungsumgebung verwenden, um diese Programme zu schreiben und auszuführen, suchen in den Menüs nach einer Möglichkeit, dem Programm Kommandozeilenparameter zu übergeben.

Wenn wir `python beispiel_sys.py Wir sind Argumente` ausführen, starten wir das Modul `beispiel_sys.py` mit dem **python**-Befehl, und alles andere danach sind Argumente, die dem Programm übergeben werden. Python speichert sie für uns in der Variablen `sys.argv`.

Denken Sie daran, dass der Name des laufenden Skripts immer der erste Parameter in der Liste `sys.argv` ist. In diesem Fall haben wir also `'beispiel_sys.py'` als `sys.argv[0]`, `'Wir'` als `sys.argv[1]`, `'sind'` als `sys.argv[2]` und `'Argumente'` als `sys.argv[3]`. Beachten Sie, dass Python mit 0 anfängt zu zählen, und nicht mit 1.

Die Variable `sys.path` enthält die Liste der Namen der Verzeichnisse, aus denen Module importiert werden. Beachten Sie, dass der erste String in `sys.path` leer ist - dieser Leerestring zeigt an, dass das aktuelle Verzeichnis auch Teil von `sys.path` ist, das ansonsten mit der Umgebungsvariablen `PYTHONPATH` übereinstimmt. Das bedeutet, dass Sie Module, die sich im gleichen Verzeichnis befinden, direkt importieren können. Andernfalls müssen Sie Ihr Modul in eines der Verzeichnisse platzieren, die in `sys.path` aufgelistet sind.

Byte-kompilierte .pyc-Dateien

Ein Modul zu importieren, ist hinsichtlich verbrauchter Rechenzeit eine relativ teure Angelegenheit, weswegen Python einige Tricks anwendet, um dies zu beschleunigen. Einer davon ist, *byte-kompilierte* Dateien mit der Endung `.pyc` zu erzeugen, die im Zusammenhang mit der Zwischenform stehen, in die Python ein Programm vor der Ausführung verwandelt (siehe die Einführung zur Arbeitsweise von Python [2]). Die `.pyc`-Datei ist nützlich, wenn Sie das Modul das nächste Mal von einem anderen Programm aus importieren - dies wird viel schneller durchgeführt, da ein Teil des Vorgangs, der zum Import eines Moduls nötig ist, schon erledigt ist. Die byte-kompilierten Dateien sind auch plattformunabhängig. Jetzt wissen Sie also, wozu diese `.pyc`-Dateien eigentlich da sind.

Die Anweisung from..import

Wenn Sie einfach nur die `argv` Variable in Ihr Programm importieren wollen (um zu vermeiden, jedes Mal `sys.` eingeben zu müssen), können Sie die Anweisung `from sys import argv` verwenden. Wenn Sie alle vom Modul `sys` verwendeten Namen importieren wollen, können Sie dafür die Anwei-

sung `from sys import *` verwenden. Das funktioniert bei allen Modulen. Im Allgemeinen ist es besser, die Anweisung `from..import` zu vermeiden, und stattdessen die `import`-Anweisung zu verwenden. Ihr Programm wird dadurch um einiges lesbarer, und Sie vermeiden dadurch mögliche Namenskonflikte.

Der __name__ eines Moduls

Jedes Modul hat einen Namen, und auf diesen Namen können Anweisungen in einem Modul zugreifen. Das ist in einer bestimmten Situation besonders nützlich: Wie vorher schon erwähnt, wird der Hauptblock eines Moduls ausgeführt, wenn es zum ersten Mal importiert wird. Was machen wir aber, wenn wir wollen, dass der Block nur ausgeführt wird, wenn das Modul als eigenständiges Programm selbst gestartet wurde, aber nicht, wenn es von einem anderen Modul importiert wurde? Dies können wir erreichen, indem wir das Attribut `__name__` des Moduls auswerten.

Verwendung von __name__

Beispiel 8.2. Verwendung von __name__ (beispiel_name.py [code/beispiel_name.py])

```
#!/usr/bin/python

if __name__ == '__main__':
 print 'Dieses Programm laeuft selbst'
else:
 print 'Ich werde von einem anderen Modul importiert'
```

Ausgabe

```
$ python beispiel_name.py
Dieses Programm laeuft selbst

$ python
>>> import beispiel_name
Ich werde von einem anderen Modul importiert
>>>
```

So funktioniert es

Jedes Python Modul hat einen im Attribut `__name__` definierten Namen, und wenn dieser Name `'__main__'` lautet, dann bedeutet dies, dass das Modul als eigenständiges Programm vom Benutzer ausgeführt wurde, und wir können dementsprechende Aktionen durchführen.

Eigene Module erstellen

Es ist einfach, seine eigenen Module zu erstellen, Sie haben es schon die ganze Zeit getan! Jedes Python-Programm ist auch ein Modul. Sie müssen nur sicherstellen, dass es die Dateiendung `.py` hat. Das folgende Beispiel soll dies verdeutlichen:

Eigene Module erstellen

Beispiel 8.3. Wie man ein eigenes Modul erstellt (meinmodul.py [code/meinmodul.py])

```
#!/usr/bin/python

def saghallo():
 print 'Hallo, hier spricht meinmodul.'

version = '0.1'

# Ende von meinmodul.py
```

Hiermit haben wir ein sehr einfaches Beispiel-*Modul* erstellt. Wie Sie sehen können, gibt es daran nichts Besonderes gegenüber einem gewöhnlichen Python-Programm. Als nächstes werden wir sehen, wie wir dieses Modul in unseren anderen Python-Programmen verwenden können.

Erinnern Sie sich, dass das Modul in das gleiche Verzeichnis platziert werden sollte wie das Programm, von dem es importiert wird, oder dass es in einem der Verzeichnisse liegen sollte, die in `sys.path` aufgeführt sind.

Beispiel: meinmodul_demo.py [code/meinmodul_demo.py]

```
#!/usr/bin/python

import meinmodul

meinmodul.saghallo()
print 'Version', meinmodul.version
```

Ausgabe

```
$ python meinmodul_demo.py
Hallo, hier spricht meinmodul.
Version 0.1
```

So funktioniert es

Beachten Sie, dass wir beide Male die Punktnotation verwenden, um auf die Mitglieder des Moduls zuzugreifen. In Python verwendet man immer wieder die gleichen Schreibweisen, die einen besonderen 'pythonischen' Stil bilden, so dass wir nicht immer neue Weisen lernen müssen, Dinge zu tun.

from..import

Hier ist eine Version, in der die `from..import`-Syntax verwendet wird.

Beispiel: meinmodul_demo2.py [code/meinmodul_demo2.py]

```
#!/usr/bin/python

from meinmodul import saghallo, version
# Andere Moeglichkeit:
# from meinmodul import *

saghallo()
print 'Version', version
```

Die Ausgabe von `meinmodul_demo2.py` ist dieselbe wie die von `meinmodul_demo.py`.

Die Funktion dir()

Sie können die eingebaute Funktion `dir()` benutzen, um die Bezeichner aufzulisten, die von einem Modul definiert werden. Die Bezeichner sind die Funktionen, Klassen und Variablen, die in dem jeweiligen Modul definiert werden.

Wenn Sie der `dir`-Funktion einen Modulnamen übergeben, dann gibt sie die Liste der Namen zurück, die in diesem Modul definiert werden. Wenn kein Parameter übergegeben wird, gibt sie die Liste der Namen des aktuellen Moduls zurück.

Die dir-Funktion verwenden

Beispiel 8.4. Die dir-Funktion verwenden

```
$ python
>>> import sys
>>> dir(sys) # Liste aller Attribute des sys-Moduls anzeigen
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'exc_clear', 'exc_info', 'exc_type',
'excepthook', 'exec_prefix', 'executable', 'exit', 'getcheckinterval',
'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding',
'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
'meta_path','modules', 'path', 'path_hooks', 'path_importer_cache',
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
'version', 'version_info', 'warnoptions']
>>> dir() # Liste aller Attribute des aktuellen Moduls anzeigen
['__builtins__', '__doc__', '__name__', 'sys']
>>>
>>> a = 5 # erzeuge eine neue Variable 'a'
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # loesche/entferne einen Namen
>>>
>>> dir()
```

```
['__builtins__', '__doc__', '__name__', 'sys']
>>>
```

So funktioniert es

Zuerst sehen wir die Anwendung von `dir` auf das importierte `sys`-Modul. Wir können die riesige Liste von Attributen sehen, die darin enthalten sind.

Als Nächstes verwenden wir die `dir`-Funktion, ohne ihr einen Parameter zu übergeben: Die Voreinstellung ist, dass sie die Attribute des aktuellen Moduls zurückliefert. Beachten Sie, dass die importierten Module ebenfalls Teil der Liste sind.

Um zu sehen, wie die `dir`-Funktion arbeitet, definieren wir eine neue Variable `a`, und weisen ihr einen Wert zu. Danach überprüfen wir, was sich an der Ausgabe von `dir` geändert hat. Wir stellen fest, dass es in der Liste ein zusätzliches Element des gleichen Namens `a` gibt. Wir entfernen die Variable bzw. das Attribut des Moduls mittels der Anweisung `del`. Als Resultat der Änderung sehen wir wieder eine entsprechende Ausgabe der `dir`-Funktion.

Eine Anmerkung zu `del`: Diese Anweisung wird benutzt, um eine Variable bzw. einen Namen zu *entfernen* (`del` steht kurz für *delete*). Nachdem die Anweisung ausgeführt wurde (in diesem Fall `del a`), können Sie nicht mehr auf die Variable `a` zugreifen - es ist, als hätte es sie nie gegeben.

Zusammenfassung

Module sind nützlich, weil sie Dienste und Funktionalität zur Verfügung stellen, die Sie in anderen Programmen wiederverwenden können. Die Standardbibliothek, die mit Python ausgeliefert wird, ist ein Beispiel für einen solchen Satz von Modulen. Wir haben gesehen, wie wir solche Module verwenden können, und auch, wie wir eigene Module erstellen können.

Als Nächstes werden wir einige interessante Konzepte kennen lernen, die Datenstrukturen genannt werden.

Kapitel 9. Datenstrukturen

Einführung

Datenstrukturen sind im Wesentlichen einfach das - *Strukturen*, die einige *Daten* zusammenhalten können. Mit anderen Worten werden sie benutzt, um eine Ansammlung von verwandten Daten zu speichern.

In Python gibt es drei eingebaute Datenstrukturen - Listen, Tupel und Dictionaries (Wörterbücher). Wir werden sehen, wie sich jede dieser drei Datenstrukturen verwenden lässt und wie sie unser Leben vereinfachen.

Listen

Eine `Liste` ist eine Datenstruktur, die eine angeordnete Sammlung von Objekten enthält, d.h. in einer Liste kann man eine *Sequenz* (eine Abfolge) von Objekten abspeichern. Sie können sich das wie eine Einkaufsliste vorstellen, die eine Liste von zu kaufenden Dingen darstellt, außer dass Sie auf ihrer Einkaufsliste die Dinge wahrscheinlich in getrennten Zeilen stehen haben, wohingegen in Python zwischen den Objekten Kommas gesetzt werden müssen.

Eine Liste von Objekten muss durch eckige Klammern (`[` und `]`) eingeschlossen sein, damit Python sie als Liste erkennt. Sobald Sie eine Liste erzeugt haben, können Sie ihr Objekte hinzufügen, welche aus ihr entfernen oder nach bestimmten Objekten in ihr suchen. Da wir Objekte hinzufügen und entfernen können, nennen wir eine Liste einen *veränderlichen* Datentyp.

Kurzeinführung in Objekte und Klassen

Bisher habe ich zwar die Diskussion von Objekten und Klassen prinzipiell aufgeschoben, doch brauchen wir jetzt eine kurze Erläuterung dieser Konzepte, damit Sie Listen besser verstehen können. Wir werden dieses Thema noch detaillierter in einem eigenen Kapitel untersuchen.

Eine Liste ist ein Beispiel für die Verwendung von Objekten und Klassen. Wenn Sie eine Variable `i` verwenden und ihr einen Wert, z.B. die Ganzzahl `5` zuweisen, können Sie sich das als die Erzeugung eines **Objekts** (einer Instanz) `i` der **Klasse** (des Typs) `int` vorstellen. Sie können sich auch den Hilfetext `help(int)` anzeigen lassen, um dies besser zu verstehen.

Eine Klasse kann **Methoden** haben, d.h. Funktionen, die zur Verwendung im Zusammenhang mit genau dieser Klasse definiert werden. Sie können diese Funktionen einer Klasse nur verwenden, wenn Sie ein Objekt der Klasse haben. Zum Beispiel stellt Python die Methode `append` der Klasse `list` zur Verfügung. Diese Methode erlaubt es Ihnen, ein Objekt am Ende einer Liste anzufügen. So wird zum Beispiel `meineliste.append("ein Ding")` den String `"ein Ding"` am Ende der Liste `meineliste` anfügen. Beachten Sie die Verwendung der Punktnotation, mit der Sie die Methode eines Objekts aufrufen können.

Außerdem kann eine Klasse **Felder** haben, bei denen es sich um nichts weiter als Variablen handelt, die nur im Zusammenhang mit der Klasse verwendet werden können. Sie können diese Variablen/Namen nur verwenden, wenn Sie ein Objekt der Klasse haben. Auf Felder wird ebenfalls mit der Punktnotation zugegriffen, z.B. `meineliste.feld`.

Benutzung von Listen

Beispiel 9.1. Benutzung von Listen (listen.py [code/listen.py])

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

```
# Dies ist meine Einkaufsliste
einkaufsliste = ['Aepfel', 'Mangos', 'Karotten', 'Bananen']

print 'Ich habe', len(einkaufsliste), 'Dinge einzukaufen.'

print 'Diese Dinge sind:', # Beachten Sie das Komma am Zeilenende
for ding in einkaufsliste:
 print ding,

print '\nIch muss auch Reis einkaufen.'
einkaufsliste.append('Reis')
print 'Meine Einkaufsliste ist jetzt:'
print einkaufsliste

print 'Jetzt werde ich meine Einkaufsliste sortieren.'
einkaufsliste.sort()
print 'Die sortierte Einkaufsliste ist:'
print einkaufsliste

print 'Zuerst werde ich', einkaufsliste[0], 'kaufen.'
altesding = einkaufsliste[0]
del einkaufsliste[0]
print 'Ich habe', altesding, 'gekauft.'
print 'Meine Einkaufsliste ist jetzt:'
print einkaufsliste
```

Ausgabe

```
$ python listen.py
Ich habe 4 Dinge einzukaufen.
Diese Dinge sind: Aepfel Mangos Karotten Bananen
Ich muss auch Reis einkaufen.
Meine Einkaufsliste ist jetzt:
['Aepfel', 'Mangos', 'Karotten', 'Bananen', 'Reis']
Jetzt werde ich meine Einkaufsliste sortieren.
Die sortierte Einkaufsliste ist:
['Aepfel', 'Bananen', 'Karotten', 'Mangos', 'Reis']
Zuerst werde ich Aepfel kaufen.
Ich habe Aepfel gekauft.
Meine Einkaufsliste ist jetzt:
['Bananen', 'Karotten', 'Mangos', 'Reis']
```

So funktioniert es

Die Variable `einkaufsliste` ist eine Einkaufsliste für jemanden, der auf den Markt geht. In `einkaufsliste` speichern wir nur die Namen der Dinge als Strings ab; aber denken Sie daran, dass Sie einer Liste *alle Arten von Objekten* hinzufügen können, also auch Zahlen und sogar andere Listen.

Wir haben auch die `for..in`-Schleife verwendet, um die Elemente in der Liste zu durchlaufen. Inzwischen werden Sie gemerkt haben, dass eine Liste auch eine Sequenz ist. Die Besonderheiten von Sequenzen werden wir in einem späteren Abschnitt besprechen.

Beachten Sie, dass wir ein *Komma* am Ende der `print`-Anweisung verwendet haben, um die automatische Ausgabe eines Zeilenumbruchs nach der `print`-Anweisung zu unterdrücken. Das ist eine etwas unschöne Weise, dies zu erreichen, aber sie ist einfach und erreicht genau das, was wir wollen.

Als Nächstes fügen wir unter Benutzung der Methode `append` der Liste ein Objekt hinzu, wie wir es bereits besprochen haben. Dann überprüfen wir, ob das Objekt der Liste wirklich hinzugefügt wurde, indem wir den Inhalt der Liste einfach der `print`-Anweisung übergeben, die uns die Liste in einer übersichtlichen Weise ausgibt.

Danach sortieren wir die Liste mit der zur Liste gehörenden Methode `sort`. Beachten Sie, dass diese Methode die Liste selbst verändert und nicht etwa eine veränderte Liste zurückliefert - hierin unterscheidet sie sich im Verhalten von Strings. Das ist gemeint, wenn wir sagen, eine Liste sei *veränderlich*, Strings dagegen *unveränderlich*.

Nachdem wir etwas auf dem Markt gekauft haben, wollen wir es aus der Liste entfernen. Das erreichen wir durch die Anweisung `del`. Wir geben hierbei an, welches Objekt wir aus der Liste entfernen wollen, und die `del`-Anweisung löscht es für uns aus der Liste. Da wir das erste Objekt aus der Liste entfernen wollen, verwenden wir `del einkaufsliste[0]` (erinnern Sie sich daran, dass Python bei 0 zu zählen beginnt).

Wenn Sie alle Methoden wissen möchten, die für Listen-Objekte definiert sind, dann erhalten Sie die Details mit `help(list)`.

Tupel

Tupel sind genau wie Listen, außer dass sie **unveränderlich** wie Strings sind, d.h. man kann Tupel nicht abändern. Tupel werden definiert, indem man durch Komma getrennte Objekte innerhalb eines normalen Klammerpaars angibt. Tupel werden normalerweise immer dann gebraucht, wenn eine Anweisung oder eine benutzerdefinierte Funktion mit Sicherheit davon ausgehen kann, dass die verwendete Sammlung von Werten, d.h. das Wertetupel, sich nicht verändern wird.

Benutzung von Tupeln

Beispiel 9.2. Benutzung von Tupeln (tupel.py [code/tupel.py])

```
#!/usr/bin/python

zoo = ('Wolf', 'Elefant', 'Pinguin')
print 'Die Zahl der Tiere im Zoo ist', len(zoo)

neuer_zoo = ('Affe', 'Delfin', zoo)
print 'Die Zahl der Tiere im neuen Zoo ist', len(neuer_zoo)
print 'Alle Tiere im neuen Zoo sind', neuer_zoo
print 'Die aus dem alten Zoo uebernommenen Tiere sind', neuer_zoo[2]
print 'Das letzte aus dem alten Zoo uebernommene Tier ist ein', neuer_zoo[2][2]
```

Ausgabe

```
$ python tupel.py
Die Zahl der Tiere im Zoo ist 3
Die Zahl der Tiere im neuen Zoo ist 3
Alle Tiere im neuen Zoo sind ('Affe', 'Delfin', ('Wolf', 'Elefant', 'Pinguin'))
Die aus dem alten Zoo uebernommenen Tiere sind ('Wolf', 'Elefant', 'Pinguin')
```

```
Das letzte aus dem alten Zoo uebernommene Tier ist ein Pinguin
```

So funktioniert es

Die Variable `zoo` bezeichnet ein Tupel von Objekten. Wir erkennen, dass die Funktion `len` eingesetzt werden kann, um die Länge des Tupels zu ermitteln. Dies zeigt außerdem, dass auch ein Tupel eine Sequenz ist.

Wir transferieren diese Tiere nun in einen neuen Zoo, weil der alte Zoo geschlossen wird. Daher erhält das Tupel `neuer_zoo` einige Tiere, die bereits dort vorhanden sind, zusammen mit den Tieren, die vom alten Zoo übernommen worden sind. Um zur Realität zurückzukehren, beachten Sie, dass ein Tupel innerhalb eines Tupels nicht seine Identität verliert.

Wir können auf die Objekte in einem Tupel zugreifen, indem wir die Position des Objekts innerhalb eines Paars eckiger Klammern angeben, genau wie wir es bei den Listen getan haben. Dies wird der *Indizierungsoperator* genannt. Wir greifen auf das dritte Objekt in `neuer_zoo` mittels `neuer_zoo[2]` zu. Dies ist ziemlich einfach, wenn man einmal die Schreibweise verstanden hat.

Tupel mit keinem oder nur einem Objekt. Ein leeres Tupel wird durch ein leeres Klammerpaar gebildet, etwa `meinleerestupel = ()`. Ein Tupel mit nur einem einzigen Objekt ist jedoch nicht so einfach. Sie müssen es durch ein auf das erste (und einzige) Objekt folgendes Komma kennzeichnen, damit Python zwischen einem Tupel und einem Klammerpaar unterscheiden kann, das einfach nur ein Objekt in einem Ausdruck umschließt, d.h. Sie müssen `einzelnesetwas = (2 , )` schreiben, wenn Sie ein Tupel meinen, das nur die Zahl `2` enthält.

Anmerkung für Perl-Programmierer

Eine Liste innerhalb einer Liste verliert nicht ihre Identität, d.h. Listen werden nicht automatisch "plattgebügelt" wie in Perl. Das Gleiche gilt für ein Tupel innerhalb eines Tupels, oder ein Tupel innerhalb einer Liste, oder eine Liste innerhalb eine Tupels usw. Soweit es Python betrifft, sind dies nur Objekte, die mittels eines anderen Objekts gespeichert werden, das ist alles.

Tupel und die print-Anweisung

Eine der häufigsten Anwendungen von Tupeln ist bei der print-Anweisung. Hier ist ein Beispiel:

Beispiel 9.3. Ausgabe mittels Tupeln (print_tupel.py [code/print_tupel.py])

```
#!/usr/bin/python

alter = 22
name = 'Swaroop'

print '%s ist %d Jahre alt.' % (name, alter)
print 'Warum spielt %s mit diesem Python?' % name
```

Ausgabe

```
$ python print_tuple.py
Swaroop ist 22 Jahre alt.
Warum spielt Swaroop mit diesem Python?
```

So funktioniert es

Der `print`-Anweisung kann man einen String mit speziellen Angaben übergeben, der von einem %-Zeichen und einem Tupel von Objekten gefolgt wird, die auf die Angaben in dem String passen. Diese Angaben werden dazu benutzt, die Ausgabe in einer bestimmten Weise zu formatieren. Sie können zum Beispiel aus einem `%s` für Strings und `%d` für Ganzzahlen bestehen. Das Tupel muss Objekte haben, die diesen Angaben in der gleichen Reihenfolge entsprechen.

Beachten Sie die erste Anwendung, wo wir zuerst `%s` benutzen, was der Variablen `name` entspricht, die das erste Objekt in dem Tupel ist, und als zweite Angabe `%d`, was `alter` entspricht, dem zweiten Objekt im Tupel.

Python wandelt hier jedes Objekt des Tupels in einen String um und setzt diese anstelle der Formatierungsangaben. Daher wird `%s` durch den Wert der Variablen `name` ersetzt und so weiter.

Dieser Gebrauch der `print`-Anweisung macht es äußerst einfach, Ausgaben auf den Bildschirm zu schreiben, und vermeidet eine Menge von String-Manipulationen, mit denen man das Gleiche erreichen könnte. Man kann so auch vermeiden, Kommas zu benutzen, wie wir es bisher gemacht haben.

Meistens kann man die Angabe `%s` verwenden, und sich Python um den Rest kümmern lassen. Das funktioniert sogar für Zahlen. Sie können aber auch die korrekten Angaben machen, um damit eine weitere Überprüfung auf Korrektheit in Ihr Programm einzubauen.

In der zweiten `print`-Anweisung benutzen wir eine einzelne Formatierungsangabe, die von einem %-Zeichen und einem einzelnen Objekt gefolgt wird - hier gibt es kein Klammerpaar. Dies funktioniert aber nur in dem Fall, dass nur eine einzelne Formatierungsangabe im String vorhanden ist.

Dictionaries

Ein Dictionary (Wörterbuch) ist wie ein Adressbuch, in dem man die Adressen oder Kontaktangaben zu einer Person finden kann, wenn man ihren Namen kennt, d.h. wir ordnen **Schlüssel** (Namen) **Werte** (Kontaktdaten) zu. Beachten Sie, dass der Schlüssel eindeutig sein muss, genau wie man auch nicht die korrekte Information herausfinden kann, wenn man zwei Personen des exakt gleichen Namens in seinem Adressbuch hat.

Beachten Sie, dass man nur unveränderliche Objekte (wie Strings) als Schlüssel eines Dictionaries verwenden kann, aber dass man sowohl unveränderliche als auch veränderliche Objekte als Werte des Dictionaries benutzen darf. Das bedeutet im Wesentlichen, dass man nur einfache Objekte als Schlüssel verwenden sollte.

Paare von Schlüsseln und Werten werden in einem Dictionary mittels der Schreibweise `d = { schluessel1 : wert1, schluessel2 : wert2 }` angegeben. Beachten Sie, dass die Schlüssel/Wert-Paare mit Doppelpunkte getrennt werden und die Paare selbst durch Kommas voneinander getrennt werden, und alles insgesamt in ein paar geschweifter Klammern eingeschlossen wird.

Denken Sie daran, dass Schlüssel/Wert-Paare in einem Dictionary in keiner Weise sortiert sind. Wenn Sie eine bestimmte Reihenfolge wünschen, dann müssen Sie diese vor Gebrauch durch Sortierung der Schlüssel selbst herstellen.

Die Dictionaries, die Sie benutzen werden, sind Instanzen/Objekte der Klasse `dict`.

Benutzung von Dictionaries

Beispiel 9.4. Benutzung von Dictionaries (dict.py [code/dict.py])

```
#!/usr/bin/python

# 'ab' steht kurz fuer 'A'dress'b'uch

ab = {  'Swaroop'   : 'swaroopch@byteofpython.info',
  'Larry'     : 'larry@wall.org',
  'Matsumoto' : 'matz@ruby-lang.org',
  'Spammer'   : 'spammer@hotmail.com'
 }

print "Swaroops Adresse ist %s" % ab['Swaroop']

# Ein Schluessel/Wert-Paar hinzufuegen
ab['Guido'] = 'guido@python.org'

# Ein Schluessel/Wert-Paar loeschen
del ab['Spammer']

print '\nEs gibt %d Kontakte im Adressbuch\n' % len(ab)

for name, adresse in ab.items():
 print '%s hat die Adresse %s' % (name, adresse)

if 'Guido' in ab: # oder: ab.has_key('Guido')
 print "\nGuidos Adresse ist %s" % ab['Guido']
```

Ausgabe

```
$ python dict.py
Swaroops Adresse ist swaroopch@byteofpython.info

Es gibt 4 Kontakte im Adressbuch

Swaroop hat die Adresse swaroopch@byteofpython.info
Matsumoto hat die Adresse matz@ruby-lang.org
Larry hat die Adresse larry@wall.org
Guido hat die Adresse guido@python.org

Guidos Adresse ist guido@python.org
```

So funktioniert es

Wir haben unter Benutzung der bereits erläuterten Schreibweise ein Dictionary `ab` erzeugt. Wir greifen dann auf die Schlüssel/Wert-Paare zu, indem wir den Schlüssel mittels des Indizierungsoperators angeben, den wir im Zusammenhang mit Listen und Tupeln erörtert haben. Beachten Sie, dass die Schreibweise für Dictionaries damit ebenfalls sehr einfach ist.

Wir können neue Schlüssel/Wert-Paare hinzufügen, indem wir einfach den Indizierungsoperator verwenden, um auf einen Schlüssel zuzugreifen und den Wert zuzuweisen, so wie wir es für 'Guido' im obigen Fall getan haben.

Wir können Schlüssel/Wert-Paare löschen, indem wir unseren alten Freund benutzen - die `del`-Anweisung. Wir geben einfach das Dictionary und den Indizierungsoperator für den zu entfernenden

Schlüssel an und übergeben es an die `del`-Anweisung. Man braucht für diese Operation nicht den Wert zu kennen, der diesem Schlüssel entspricht.

Als Nächstes greifen wir auf alle Schlüssel/Wert-Paare des Dictionaries zu, indem wir die `items`-Methode des Dictionaries verwenden, die eine Liste von Tupeln zurückgibt, bei der jedes Tupel ein Paar von Objekten enthält - den Schlüssel, gefolgt von seinem zugehörigen Wert. Wir ermitteln dieses Paar und weisen es den Variablen `name` bzw. `adresse` zu, und zwar mittels der `for..in`-Schleife für jedes vorhandene Schlüssel/Wert-Paar im Dictionary, und geben diese Werte dann im for-Block aus.

Wir können den `in`-Operator oder auch die Methode `has_key` der Klasse `dict` verwenden, um zu überprüfen, ob ein bestimmter Schlüssel im Dictionary einen Wert zugeordnet hat. Sie können sich die Dokumentation für die vollständige Liste von Methoden der Klasse `dict` mittels `help(dict)` anzeigen lassen.

Schlüsselwort-Argumente und Dictionaries. Nebenbei bemerkt haben Sie bereits Dictionaries verwendet, wenn Sie in Ihren Funktionen Schlüsselwort-Argumente benutzt haben! Denken Sie einmal darüber nach - jedes Schlüssel/Wert-Paar wurde von Ihnen in der Parameterliste der Funktionsdefinition angegeben, und wenn Sie auf Variablen innerhalb der Funktion zugreifen, ist dies nur ein Schlüsselzugriff auf ein Dictionary (das in der Sprache der Compilerbauer die *Symbol-Tabelle* genannt wird).

Sequenzen

Listen, Tupel und Strings sind Beispiele für Sequenzen, aber was sind Sequenzen und was ist an ihnen so besonders? Zwei Haupteigenschaften einer Sequenz sind der **Indizierungsoperator**, der es uns ermöglicht, direkt auf ein bestimmtes Objekt in einer Sequenz zuzugreifen, und der **Teilbereichsoperator**, der es uns erlaubt, eine Teilsequenz der gesamten Sequenz zu erhalten.

Benutzung von Sequenzen

Beispiel 9.5. Benutzung von Sequenzen (seq.py [code/seq.py])

```
#!/usr/bin/python

einkaufsliste = ['Aepfel', 'Mangos', 'Karotten', 'Bananen']

# Indizierungs-Operation
print 'Position 0 ist', einkaufsliste[0]
print 'Position 1 ist', einkaufsliste[1]
print 'Position 2 ist', einkaufsliste[2]
print 'Position 3 ist', einkaufsliste[3]
print 'Position -1 ist', einkaufsliste[-1]
print 'Position -2 ist', einkaufsliste[-2]

# Teilbereichs-Operation auf einer Liste
print 'Position 1 bis 3 ist', einkaufsliste[1:3]
print 'Position 2 bis Ende ist', einkaufsliste[2:]
print 'Position 1 bis -1 ist', einkaufsliste[1:-1]
print 'Position Anfang bis ist', einkaufsliste[:]

# Teilbereichs-Operation auf einem String
name = 'swaroop'
print 'Zeichen 1 bis 3 ist', name[1:3]
print 'Zeichen 2 bis Ende ist', name[2:]
```

```
print 'Zeichen 1 bis -1 ist', name[1:-1]
print 'Zeichen Anfang bis Ende', name[:]
```

Ausgabe

```
$ python seq.py
Position 0 ist Aepfel
Position 1 ist Mangos
Position 2 ist Karotten
Position 3 ist Bananen
Position -1 ist Bananen
Position -2 ist Karotten
Position 1 bis 3 ist ['Mangos', 'Karotten']
Position 2 bis Ende ist ['Karotten', 'Bananen']
Position 1 bis -1 ist ['Mangos', 'Karotten']
Position Anfang bis ist ['Aepfel', 'Mangos', 'Karotten', 'Bananen']
Zeichen 1 bis 3 ist wa
Zeichen 2 bis Ende ist aroop
Zeichen 1 bis -1 ist waroo
Zeichen Anfang bis Ende swaroop
```

So funktioniert es

Zuerst sehen wir, wie der Indizierungsoperator verwendet wird, um einzelne Elemente einer Sequenz zu ermitteln. Immer wenn man wie oben bei einer Sequenz eine Zahl in eckigen Klammern angibt, holt Python Ihnen das Element, das dieser Position in der Liste entspricht. Denken Sie daran, dass Python von 0 anfängt zu zählen. Daher holt `einkaufsliste[0]` das erste Element und `einkaufsliste[3]` das vierte Element der Sequenz `einkaufsliste`.

Der Index kann auch eine negative Zahl sein. In diesem Fall wird die Position vom Ende der Sequenz her gezählt. Daher bezieht sich `einkaufsliste[-1]` auf das letzte Element der Sequenz und `einkaufsliste[-2]` auf das vorletzte Element der Sequenz.

Der Teilbereichsoperator wird verwendet, indem man den Namen der Sequenz gefolgt von einem optionalen Zahlenpaar angibt, das durch einen Doppelpunkt innerhalb von eckigen Klammern getrennt wird. Beachten Sie, dass dies sehr ähnlich dem Indizierungsoperator ist, den Sie bisher verwendet haben. Denken Sie daran, dass die Zahlen optional sind, der Doppelpunkt aber nicht.

Die erste Zahl (vor dem Doppelpunkt) in der Teilbereichs-Operation bezieht sich auf die Position, wo der Teilbereich anfängt, und die zweite Zahl (nach dem Doppelpunkt) gibt an, wo der Teilbereich aufhört. Wenn die erste Zahl nicht angegeben wird, dann fängt Python am Anfang der Sequenz an. Wenn die zweite Zahl weggelassen wird, dann hört Python am Ende der Sequenz auf. Beachten Sie, dass die zurückgegebene Teilsequenz *an* der Startposition *anfängt* und direkt *vor* der Endposition *aufhört*, d.h. die Startposition ist in der Teilsequenz enthalten, aber die Endposition gehört nicht mehr dazu.

Daher gibt `einkaufsliste[1:3]` einen Teilbereich der Sequenz zurück, der bei der Position 1 beginnt, die Position 2 enthält, aber vor der Position 3 endet, weswegen ein *Teilbereich* mit nur zwei Elementen zurückgegeben wird. Entsprechend liefert `einkaufsliste[:]` eine Kopie der gesamten Liste zurück.

Man kann auch beim Teilbereichsoperator negative Positionen verwenden. Negative Zahlen werden für Positionen vom Ende der Sequenz aus verwendet. Zum Beispiel gibt `einkaufsliste[:-1]`

eine Teilsequenz zurück, die das letzte Element der Sequenz ausschließt, aber alle anderen Elemente der Sequenz enthält.

Experimentieren Sie mit verschiedenen Kombinationen solcher Teilbereichsangaben, indem Sie den Python-Interpreter interaktiv benutzen, d.h. die Interpreter-Eingabeaufforderung, damit Sie die Ergebnisse direkt sehen können. Das großartige an Sequenzen ist, dass man auf Tupel, Listen und Strings alle auf die gleiche Weise zugreifen kann.

Referenzen

Wenn Sie ein Objekt anlegen und es einer Variablen zuweisen, dann *referenziert* die Variable nur das Objekt, und stellt nicht das Objekt selber dar! Das heißt, der Variablenname zeigt nur auf den Teil des Speichers Ihres Computers, an dem das Objekt abgespeichert ist. Dies wird als **Bindung** des Namens an das Objekt bezeichnet.

Im Allgemeinen brauchen Sie sich darüber keine Gedanken zu machen, aber es gibt aufgrund der Referenzierung einen subtilen Effekt, dessen Sie sich bewusst sein sollten. Dies wird durch das folgende Beispiel demonstriert.

Objekte und Referenzen

Beispiel 9.6. Objekte und Referenzen (referenz.py [code/referenz.py])

```
#!/usr/bin/python

print 'Einfache Zuweisung'
einkaufsliste = ['Aepfel', 'Mangos', 'Karotten', 'Bananen']
meineliste = einkaufsliste
# meineliste ist nur ein anderer Name, der auf das gleiche Objekt zeigt!

# Ich habe den ersten Posten gekauft und entferne ihn daher von der Liste
del einkaufsliste[0]

print 'einkaufsliste ist', einkaufsliste
print 'meineliste ist', meineliste
# Beachten Sie, dass sowohl einkaufsliste als auch meineliste
# die gleiche Liste ohne die 'Aepfel' ausgeben, was bestaetigt,
# dass sie auf das gleiche Objekt zeigen

print 'Kopie mittels Teilbereichsoperation'
meineliste = einkaufsliste[:] # auf diese Weise wird die gesamte Liste kopiert
del meineliste[0] # entferne das erste Element

print 'einkaufsliste ist', einkaufsliste
print 'meineliste ist', meineliste
# Beachten Sie, dass die beiden Listen nun unterschiedlich sind
```

Output

```
$ python referenz.py
Einfache Zuweisung
```

```
einkaufsliste ist ['Mangos', 'Karotten', 'Bananen']
meineliste ist ['Mangos', 'Karotten', 'Bananen']
Kopie mittels Teilbereichsoperation
einkaufsliste ist ['Mangos', 'Karotten', 'Bananen']
meineliste ist ['Karotten', 'Bananen']
```

So funktioniert es

Die Erklärung findet sich größtenteils schon in den Kommentaren. Denken Sie daran, dass Sie den Teilbereichsoperator verwenden müssen, wenn Sie eine Kopie einer Liste oder derartiger Sequenzen oder komplexer Objekte (nicht einfacher *Objekte* wie Ganzzahlen) erstellen wollen. Wenn Sie nur den Variablennamen einem anderen Namen zuweisen, dann *referenzieren* beide Namen das gleiche Objekt, was zu allen möglichen Schwierigkeiten führen kann, wenn Sie dies nicht beachten.

Anmerkung für Perl-Programmierer

Denken Sie daran, dass Zuweisungsanweisungen für Listen **nicht** eine Kopie erzeugen. Sie müssen den Teilbereichsoperator verwenden, um eine Kopie der Sequenz zu erzeugen.

Mehr über Strings

Wir haben Strings bereits früher ausführlich besprochen. Was kann es noch mehr geben, das man darüber wissen muss? Nun, wussten Sie, dass Strings auch Objekte sind und Methoden haben, die alle möglichen Dinge tun können, von der Überprüfung auf einen Teilstring bis hin zum Entfernen von Leerzeichen am Anfang oder Ende!

Die Strings, die Sie in Ihren Programmen benutzen, sind allesamt Objekte der Klasse `str`. Einige nützliche Methoden dieser Klasse werden im nächsten Beispiel demonstriert. Eine vollständige Liste dieser Methoden erhalten Sie mit `help(str)`.

String-Methoden

Beispiel 9.7. String-Methoden (str_methoden.py [code/str_methoden.py])

```
#!/usr/bin/python

name = 'Swaroop' # Dies ist ein String-Objekt

if name.startswith('Swa'):
 print 'Ja, der String beginnt mit "Swa".'

if 'a' in name:
 print 'Ja, er enthält den String "a".'

if name.find('war') != -1:
 print 'Ja, er enthält den String "war".'

trennzeichen = '_*_'
meineliste = ['Brasilien', 'Russland', 'Indien', 'China']
print trennzeichen.join(meineliste)
```

Ausgabe

```
$ python str_methoden.py
Ja, der String beginnt mit "Swa".
Ja, er enthält den String "a".
Ja, er enthält den String "war".
Brasilien_*_Russland_*_Indien_*_China
```

So funktioniert es

Wir sehen hier eine Anzahl von String-Methoden in Aktion. Die Methode `startswith` wird benutzt, um herauszufinden, ob der String mit einem vorgegebenen String beginnt. Der `in`-Operator wird benutzt, um zu überprüfen, ob ein vorgegebener String ein Teil des Strings ist.

Die Methode `find` wird benutzt, um die Position eines vorgegebenen Strings im String zu finden. Sie liefert eine -1 zurück, wenn sie den Teilstring nicht erfolgreich finden konnte. Die Klasse `str` hat auch eine praktische Methode namens `join`, mit der man die Elemente einer Sequenz zusammenfügen kann, wobei der String als eine Kette von Trennzeichen fungiert, die zwischen jedes Element der Sequenz eingefügt wird, um so einen größeren daraus zusammengebauten String zu erzeugen.

Zusammenfassung

Wir haben die verschiedenen eingebauten Datenstrukturen von Python ausführlich untersucht. Diese Datenstrukturen sind unabdingbar, wenn man Programme mit einer vernünftigen Größe schreiben möchte.

Nachdem wir uns nun mit den Grundlagen von Python vertraut gemacht haben, werden wir als Nächstes sehen, wie wir ein richtiges Python-Programm entwerfen und schreiben können.

Kapitel 10. Problemlösung - So schreibt man ein Python-Skript

Wir haben bereits verschiedene Teile der Sprache Python kennen gelernt und wollen nun sehen, wie all diese Teile zusammenspielen, indem wir ein Prográmmchen entwerfen und schreiben, das etwas Nützliches *tut*.

Das Problem

Das Problem lautet: *'Ich möchte ein Programm, das eine Sicherungskopie all meiner wichtigen Dateien erstellt.'*

Dies ist zwar ein einfaches Problem, aber dennoch reicht die Information noch nicht aus, um sofort eine Lösung anbieten zu können. Es ist ein wenig weitere **Analyse** nötig. Zum Beispiel: Wie geben wir an, welche Dateien gesichert werden sollen? Wo wird die Datensicherung gespeichert? Wie werden die Dateien in der Sicherungskopie gespeichert?

Nachdem wir das Problem gründlich analysiert haben, **entwerfen** wir unser Programm. Wir machen eine Liste von Dingen, wie unser Programm funktionieren sollte. In diesem Fall habe ich die folgende Liste erstellt, in der ich festhalte, wie es nach *meiner* Meinung funktionieren sollte. Wenn Sie den Entwurf erstellen, kann er bei Ihnen anders aussehen - jeder Mensch hat seine eigene Weise, Dinge zu tun, das ist völlig in Ordnung so.

1. Die Dateien und Verzeichnisse, die gesichert werden sollen, werden als Liste angegeben.
2. Die Datensicherung muss in einem Hauptverzeichnis für Sicherungen gespeichert werden.
3. Die Dateien werden in einer ZIP-Datei komprimiert gesichert.
4. Der Name des ZIP-Archivs setzt sich aus dem aktuellen Datum und der Uhrzeit zusammen.
5. Wir benutzen den gewöhnlichen **zip**-Befehl, der normalerweise in jeder Linux/Unix-Distribution vorhanden ist. Windows-Benutzer können PKZIP oder das Info-ZIP-Programm verwenden. Beachten Sie, dass Sie jedes beliebige Programm für die Archivierung verwenden können, solange es eine Kommandozeilenschnittstelle zur Verfügung stellt, über die es Parameter von unserem Skript entgegennehmen kann.

Die Lösung

Nachdem der Entwurf unseres Programms nun feststeht, können wir den Code schreiben, der eine **Implementierung** unserer Lösung darstellt.

Die erste Version

Beispiel 10.1. Sicherungsskript - erste Version (sicherung_ver1.py [code/sicherung_ver1.py])

```
#!/usr/bin/python

import os
import time

# 1. Die Dateien und Verzeichnisse, die gesichert werden sollen,
```

```
# werden in der folgenden Liste angegeben:
quellen = ['/home/swaroop/byte', '/home/swaroop/bin']
# Unter Windows muessen Sie die Pfade auf diese Weise angeben:
# quellen = ['C:\\Dokumente', 'D:\\Arbeit']

# 2. Die Sicherung muss in das folgende Hauptverzeichnis fuer
# Sicherungen gespeichert werden:
ziel_verzeichnis = '/mnt/e/sicherung/'
# Denken Sie daran, dies an Ihre Gegebenheiten anzupassen.

# 3. Die Dateien werden in einer ZIP-Datei gesichert.

# 4. Der Name der ZIP-Datei setzt sich aus dem aktuellen Datum
# und der Uhrzeit wie folgt zusammen:
ziel = ziel_verzeichnis + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. Wir benutzen den Befehl zip (unter Unix/Linux), um die Dateien
# zu einem ZIP-Archiv zu komprimieren:
zip_befehl = 'zip -qr %s %s' % (ziel, ' '.join(quellen))
# Windows-Benutzer koennen z.B. PKZIP oder Info-ZIP in das
# Windows-Systemverzeichnis kopieren, damit dies funktioniert.

# Sicherung starten
if os.system(zip_befehl) == 0:
 print 'Erfolgreiche Sicherung nach', ziel
else:
 print 'Sicherung fehlgeschlagen!'
```

Ausgabe

```
$ python sicherung_ver1.py
Erfolgreiche Sicherung nach /mnt/e/backup/20041208073244.zip
```

Wir sind nun in der **Test**-Phase, in der wir ausprobieren, ob unser Programm ordentlich funktioniert. Wenn es sich nicht so verhält, wie erwartet, dann müssen wir unser Programm **debuggen**, d.h. die *Bugs* (Fehler) aus dem Programm entfernen.

So funktioniert es

Sie werden bemerkt haben, wie wir unseren *Entwurf* Schritt für Schritt in Programm-*Code* umgewandelt haben.

Wir benutzen die Module `os` und `time`, weswegen wir sie am Anfang importieren. Danach geben wir in einer Liste namens `Quellen` die Quelldateien und -verzeichnisse an, die gesichert werden sollen. Das Zielverzeichnis, in das wir die Sicherungsdateien speichern, wird in der Variable `ziel_verzeichnis` angegeben. Der Name des ZIP-Archivs, das wir erzeugen werden, setzt sich aus dem aktuellen Datum und der Uhrzeit zusammen, die wir mit der Funktion `time.strftime()` ermitteln. Dem Namen wird noch die Dateiendung `.zip` angehängt und das Zielverzeichnis vorangestellt. Dieser vollständige Dateiname mit Pfad wird in der Variable `ziel` gespeichert.

Der Funktion `time.strftime()` muss eine Formatspezifikation übergeben werden, wie wir sie im obigen Programm benutzt haben. Das `%Y` wird hierbei durch das Jahr ersetzt. Das `%m` wird durch den

Monat als Dezimalzahl zwischen `01` und `12` ersetzt, und so weiter. Die vollständige Liste aller solcher Format-Spezifikationen kann im [Python-Referenz-Handbuch] nachgelesen werden, das zusammen mit Python ausgeliefert wird. Beachten Sie, dass dies der Spezifikation ähnelt (aber nicht das gleiche ist), wie wir sie im `print`-Befehl benutzt haben (mit dem `%`-Zeichen, gefolgt von einem Tupel).

Wir erzeugen den Namen der Ziel-ZIP-Datei, indem wir den Summierungs-Operator (das Plus-Zeichen) benutzen, der die einzelnen Strings miteinander *verkettet*, d.h. er verbindet zwei Strings und liefert dies als neuen String zurück. Danach erzeugen wir einen String `zip_befehl`, der den Befehl enthält, den wir ausführen wollen. Sie können ausprobieren, ob dieser Befehl funktioniert, indem Sie ihn auf der Kommandozeilen-Ebene ausprobieren (Linux-Terminal oder DOS-Eingabeaufforderung).

Dem von uns verwendeten Befehl **zip** geben wir noch einige Optionen und Parameter mit. Die Option `-q` wird benutzt, um anzuzeigen, dass der zip-Befehl ohne Ausgaben arbeiten soll (d.h. ohne viel bei der Arbeit zu **q**uatschen). Die Option `-r` gibt an, dass der zip-Befehl **r**ekursiv arbeiten soll, d.h. er soll die Unterverzeichnisse und Dateien in den Unterverzeichnissen mit archivieren. Die beiden Optionen werden kombiniert in der Kurzform `-qr` angegeben. Den Optionen folgt der Name des zu erzeugenden ZIP-Archivs und die Liste der Dateien und Verzeichnisse, die in dem Archiv gesichert werden sollen. Wir konvertieren die Liste `quellen` in einen String, indem wir die `join`-Methode für Strings benutzen, die wir bereits früher kennen gelernt haben.

Danach *starten* wir endlich den Befehl, indem wir die Funktion `os.system` benutzen, die einen Befehl auf Betriebssystemebene des *Systems* ausführt, d.h. in einer Kommandozeilen-Umgebung - wobei eine `0` zurückgegeben wird, wenn der Befehl erfolgreich ausgeführt wurde, oder einen Fehlercode ungleich `0` im Fall, dass ein Fehler aufgetreten ist.

Je nachdem, wie die Ausführung des Befehls ausgegangen ist, geben wir eine entsprechende Meldung auf dem Bildschirm aus, dass die Sicherung fehlgeschlagen ist oder erfolgreich war, und das war's, wir haben damit ein Skript erzeugt, das eine Sicherungskopie unserer wichtigen Dateien erzeugt!

Anmerkung für Windows-Benutzer

Sie können in der Liste `quellen` und der Variable `ziel_verzeichnis` beliebige Datei- bzw. Verzeichnisnamen angeben, aber Sie müssen unter Windows ein wenig Acht geben. Das Problem ist, dass Windows den umgekehrten Schrägstrich (`\`) als das Trennzeichen für Verzeichnisse benutzt, Python jedoch den umgekehrten Schrägstrich für Maskierungscodes benutzt!

Daher müssen Sie einen umgekehrten Schrägstrich selber durch einen Maskierungscode oder durch die Benutzung von rohen Zeichenketten darstellen. Zum Beispiel können Sie `'C:\\Dokumente'` oder `r'C:\Dokumente'` schreiben, **nicht** aber `'C:\Dokumente'` - Sie verwenden hier einen unbekannten Maskierungscode `\D`!

Nachdem wir nun ein funktionierendes Sicherungsskript haben, können wir es jederzeit benutzen, um eine Sicherungskopie der Dateien zu erstellen. Für Linux/Unix-Benutzer empfiehlt es sich, die früher besprochene Methode für ausführbare Dateien zu verwenden, damit das Sicherungsskript jederzeit und überall ausgeführt werden kann. Dies wird die **Produktions**- oder **Einsatz**-Phase der Software genannt.

Das obige Programm funktioniert zufrieden stellend, aber (meistens) funktionieren erste Versionen von Programmen nicht genau so, wie man es erwartet. Zum Beispiel könnten Probleme entstehen, wenn das Programm nicht sauber entworfen wurde, oder wenn man einen Fehler beim Eintippen des Programmcodes gemacht hat, usw. Dementsprechend muss man dann wieder in die Entwurfsphase zurückgehen oder sein Programm debuggen.

Die zweite Version

Die erste Version unseres Skripts funktioniert. Wir können aber noch einige Verfeinerungen daran vornehmen, so dass es für den täglichen Einsatz besser tauglich wird. Man nennt dies die **Wartungs**-Phase der Software.

Eine der Verfeinerungen, die ich nützlich fand, ist ein besserer Mechanismus zur Benennung der Sicherungsdateien - indem die *Uhrzeit* als der Name der Datei verwendet wird, die in einem Verzeichnis gespeichert wird, dessen Name aus dem aktuellen *Datum* gebildet wird, und das sich im Hauptverzeichnis für die Sicherungen befindet. Ein Vorteil davon ist, dass die Sicherungen in einer hierarchischen Weise gespeichert werden und daher einfacher zu verwalten sind. Ein weiterer Vorteil ist, dass die Länge der Dateinamen auf diese Weise viel kürzer wird. Noch ein weiterer Vorteil ist, dass verschiedene Verzeichnisse dabei helfen, auf einfache Weise zu überprüfen, ob man an jedem Tag eine Sicherung gemacht hat, denn das Verzeichnis wird nur angelegt, wenn am jeweiligen Tag eine Sicherung vorgenommen wurde.

Beispiel 10.2. Sicherungsskript - zweite Version (sicherung_ver2.py [code/sicherung_ver2.py])

```
#!/usr/bin/python

import os
import time

# 1. Die Dateien und Verzeichnisse, die gesichert werden sollen,
# werden in der folgenden Liste angegeben:
quellen = ['/home/swaroop/byte', '/home/swaroop/bin']
# Unter Windows muessen Sie die Pfade auf diese Weise angeben:
# quellen = ['C:\\Dokumente', 'D:\\Arbeit']

# 2. Die Sicherung muss in das folgende Hauptverzeichnis fuer
# Sicherungen gespeichert werden:
ziel_verzeichnis = '/mnt/e/sicherung/'
# Denken Sie daran, dies an Ihre Gegebenheiten anzupassen.

# 3. Die Dateien werden in einer ZIP-Datei gesichert.

# 4. Das Tagesdatum ist der Name des Unterverzeichnisses:
heute = ziel_verzeichnis + time.strftime('%Y%m%d')
# Die aktuelle Uhrzeit ist der Name des ZIP-Archivs:
jetzt = time.strftime('%H%M%S')

# Erzeuge das Unterverzeichnis, wenn noch nicht vorhanden:
if not os.path.exists(heute):
 os.mkdir(heute) # erzeuge das Verzeichnis
 print 'Verzeichnis', heute, 'erfolgreich angelegt'

# Der Name der ZIP-Datei:
ziel = heute + os.sep + jetzt + '.zip'

# 5. Wir benutzen den Befehl zip (unter Unix/Linux), um die Dateien
# zu einem ZIP-Archiv zu komprimieren:
zip_befehl = 'zip -qr %s %s' % (ziel, ' '.join(quellen))
# Windows-Benutzer koennen z.B. PKZIP oder Info-ZIP in das
# Windows-Systemverzeichnis kopieren, damit dies funktioniert.

# Sicherung starten
if os.system(zip_befehl) == 0:
 print 'Erfolgreiche Sicherung nach', ziel
else:
 print 'Sicherung fehlgeschlagen!'
```

Ausgabe

```
$ python sicherung_ver2.py
Verzeichnis /mnt/e/backup/20041208 erfolgreich angelegt
Erfolgreiche Sicherung nach /mnt/e/backup/20041208/080020.zip

$ python sicherung_ver2.py
Erfolgreiche Sicherung nach /mnt/e/backup/20041208/080428.zip
```

So funktioniert es

Das Programm bleibt im Wesentlichen das gleiche. Die Änderungen bestehen darin, dass wir prüfen, of es bereits ein Verzeichnis mit dem aktuellen Tagesdatum als Namen innerhalb des Hauptverzeichnisses für Sicherungen gibt, wofür wir die Funktion `os.exists` einsetzen. Wenn es nicht existiert, legen wir das Verzeichnis mit der Funktion `os.mkdir` an.

Beachten Sie die Benutzung der Variablen `os.sep` - sie stellt das Trennzeichen für Verzeichnisse des jeweils benutzten Betriebssystems dar, d.h. sie wird unter Unix oder Linux `'/'` sein, unter Windows wird sie `'\\'` sein, und `':'` unter MacOS. Indem man `os.sep` verwendet, anstatt diese Zeichen direkt hinzuschreiben, wird das Programm portabel und funktioniert plattformunabhängig unter all diesen Betriebssystemen.

Die dritte Version

Die zweite Version funktioniert hervorragend, wenn man viele Sicherungskopien macht, aber wenn man so viele Sicherheitskopien macht, dann wird es schwer zu unterscheiden, wofür diese Sicherungskopien gemacht worden sind. Zum Beispiel könnte ich einige größere Änderungen an einem Programm oder einer Präsentation gemacht haben, und möchte dann, dass diese Änderungen mit dem Namen des ZIP-Archivs verknüpft werden. Dies kann einfach dadurch erreicht werden, dass man dem Namen des ZIP-Archivs einen benutzerdefinierten Kommentar anhängt.

Beispiel 10.3. Sicherungsskript - dritte Version (funktioniert nicht!) (sicherung_ver3.py [code/sicherung_ver3.py])

```
#!/usr/bin/python

import os
import time

# 1. Die Dateien und Verzeichnisse, die gesichert werden sollen,
# werden in der folgenden Liste angegeben:
quellen = ['/home/swaroop/byte', '/home/swaroop/bin']
# Unter Windows muessen Sie die Pfade auf diese Weise angeben:
# quellen = ['C:\\Dokumente', 'D:\\Arbeit']

# 2. Die Sicherung muss in das folgende Hauptverzeichnis fuer
# Sicherungen gespeichert werden:
ziel_verzeichnis = '/mnt/e/sicherung/'
# Denken Sie daran, dies an Ihre Gegebenheiten anzupassen.
```

```
# 3. Die Dateien werden in einer ZIP-Datei gesichert.

# 4. Das Tagesdatum ist der Name des Unterverzeichnisses:
heute = ziel_verzeichnis + time.strftime('%Y%m%d')
# Die aktuelle Uhrzeit ist der Name des ZIP-Archivs:
jetzt = time.strftime('%H%M%S')

# Eine Anmerkung als Benutzereingabe entgegennehmen,
# die fuer den Namen der ZIP-Datei verwendet wird:
anmerkung = raw_input('Geben Sie eine Anmerkung ein --> ')
if len(anmerkung) == 0: # pruefe, ob eine Anmerkung eingegeben wurde
 ziel = heute + os.sep + jetzt + '.zip'
else:
 ziel = heute + os.sep + jetzt + '_' +
  anmerkung.replace(' ', '_') + '.zip'

# Erzeuge das Unterverzeichnis, wenn noch nicht vorhanden:
if not os.path.exists(heute):
 os.mkdir(heute) # erzeuge das Verzeichnis
 print 'Verzeichnis', heute, 'erfolgreich angelegt'

# 5. Wir benutzen den Befehl zip (unter Unix/Linux), um die Dateien
# zu einem ZIP-Archiv zu komprimieren:
zip_befehl = 'zip -qr %s %s' % (ziel, ' '.join(quellen))
# Windows-Benutzer koennen z.B. PKZIP oder Info-ZIP in das
# Windows-Systemverzeichnis kopieren, damit dies funktioniert.

# Sicherung starten
if os.system(zip_befehl) == 0:
 print 'Erfolgreiche Sicherung nach', ziel
else:
 print 'Sicherung fehlgeschlagen!'
```

Ausgabe

```
$ python sicherung_ver3.py
  File "sicherung_ver3.py", line 30
    ziel = heute + os.sep + jetzt + '_' +
                                          ^
SyntaxError: invalid syntax
```

Warum dies nicht funktioniert

Dieses Programm funktioniert nicht! Python sagt, dass es einen Syntax-Fehler hat, was bedeutet, dass das Skript an einer Stelle nicht die Struktur aufweist, die Python hier erwartet. Wenn wir auf die Fehlermeldung von Python Acht geben, gibt sie uns auch die genaue Stelle an, wo der Fehler entdeckt wurde. Wir fangen also an, das Programm ab dieser Zeile zu *debuggen*.

Bei genauerer Untersuchung erkennen wir, dass die eine logische Zeile in zwei physikalische Zeilen aufgespalten worden ist, aber dass wir nicht angegeben haben, dass diese zwei physikalischen Zeilen zusammengehören. Für Python fehlt dem Summierungs-Operator (+) am Ende der logischen Zeile

daher ein Operand, weswegen es an dieser Stelle abbricht. Erinnern Sie sich daran, dass wir angeben können, dass die logische Zeile in der folgenden physikalischen Zeile fortgeführt wird, indem wir einen umgekehrten Schrägstrich am Ende der physikalischen Zeile benutzen. Wir machen daher diese Verbesserung an unserem Programm. Eine solche Fehlerkorrektur wird auch als **Bug-Fixing** bezeichnet.

Die vierte Version

Beispiel 10.4. Sicherungsskript - vierte Version (sicherung_ver4.py [code/sicherung_ver4.py])

```
#!/usr/bin/python

import os
import time

# 1. Die Dateien und Verzeichnisse, die gesichert werden sollen,
# werden in der folgenden Liste angegeben:
quellen = ['/home/swaroop/byte', '/home/swaroop/bin']
# Unter Windows muessen Sie die Pfade auf diese Weise angeben:
# quellen = ['C:\\Dokumente', 'D:\\Arbeit']

# 2. Die Sicherung muss in das folgende Hauptverzeichnis fuer
# Sicherungen gespeichert werden:
ziel_verzeichnis = '/mnt/e/sicherung/'
# Denken Sie daran, dies an Ihre Gegebenheiten anzupassen.

# 3. Die Dateien werden in einer ZIP-Datei gesichert.

# 4. Das Tagesdatum ist der Name des Unterverzeichnisses:
heute = ziel_verzeichnis + time.strftime('%Y%m%d')
# Die aktuelle Uhrzeit ist der Name des ZIP-Archivs:
jetzt = time.strftime('%H%M%S')

# Eine Anmerkung als Benutzereingabe entgegennehmen,
# die fuer den Namen der ZIP-Datei verwendet wird:
anmerkung = raw_input('Geben Sie eine Anmerkung ein --> ')
if len(anmerkung) == 0: # pruefe, ob eine Anmerkung eingegeben wurde
 ziel = heute + os.sep + jetzt + '.zip'
else:
 ziel = heute + os.sep + jetzt + '_' + \
  anmerkung.replace(' ', '_') + '.zip'
 # Beachten Sie den umgekehrten Schraegstrich!

# Erzeuge das Unterverzeichnis, wenn noch nicht vorhanden:
if not os.path.exists(heute):
 os.mkdir(heute) # erzeuge das Verzeichnis
 print 'Verzeichnis', heute, 'erfolgreich angelegt'

# 5. Wir benutzen den Befehl zip (unter Unix/Linux), um die Dateien
# zu einem ZIP-Archiv zu komprimieren:
zip_befehl = 'zip -qr %s %s' % (ziel, ' '.join(quellen))
# Windows-Benutzer koennen z.B. PKZIP oder Info-ZIP in das
# Windows-Systemverzeichnis kopieren, damit dies funktioniert.

# Sicherung starten
```

```
if os.system(zip_befehl) == 0:
 print 'Erfolgreiche Sicherung nach', ziel
else:
 print 'Sicherung fehlgeschlagen!'
```

Ausgabe

```
$ python sicherung_ver4.py
Geben Sie eine Anmerkung ein --> neue beispiele ergaenzt
Erfolgreiche Sicherung nach /mnt/e/backup/20041208/082156_neue_beispiele_ergaenz

$ python sicherung_ver4.py
Geben Sie eine Anmerkung ein -->
Erfolgreiche Sicherung nach /mnt/e/backup/20041208/082316.zip
```

So funktioniert es

So funktioniert das Programm jetzt! Wir wollen die eigentlichen Änderungen durchgehen, die wir in Version 3 gemacht haben. Wir nehmen eine Anmerkung des Benutzers mit der Funktion `raw_input` als Eingabe entgegen und prüfen, ob der Benutzer wirklich irgendetwas eingegeben hat, indem wir die Länge der Eingabe mit der Funktion `len` ermitteln. Wenn der Benutzer aus irgendeinem Grund (zum Beispiel, weil es nur eine Routine-Sicherung ist, und keine besonderen Änderungen vorgenommen wurden) nur einfach die **Enter**-Taste gedrückt hat, dann verfahren wir genauso wie vorher.

Wenn jedoch eine Anmerkung eingegeben wurde, dann wird diese an den Namen des ZIP-Archivs angehängt, direkt vor der Dateiendung `.zip`. Beachten Sie, dass wir Leerzeichen in der Anmerkung durch Unterstriche ersetzen - aus dem Grund, dass solche Dateinamen leichter handhabbar sind.

Weitere Verfeinerungen

Die vierte Version ist für die meisten Benutzer ein zufrieden stellend arbeitendes Skript, aber es gibt stets Möglichkeiten zur weiteren Verbesserung. Zum Beispiel könnte man einen Grad der *Ausführlichkeit* der Programmausgaben einbauen, den man mit einer Option `-v` angeben könnte, und der das Programm dazu bringt, geschwätziger in der Ausgabe zu sein.

Eine weitere mögliche Verbesserung würde es erlauben, dem Skript zusätzliche Dateien und Verzeichnisse auf der Kommandozeile zu übergeben. Wir könnten diese Übergabeparameter der Liste `sys.argv` entnehmen, und sie der Liste `quellen` hinzufügen, indem wir die `extend`-Methode benutzen, die von der Klasse `list` bereitgestellt wird.

Eine Verfeinerung, die ich bevorzuge, ist die Verwendung des **tar**-Befehls anstelle des **zip**-Befehls. Ein Vorteil hierbei ist, dass die Sicherung viel schneller durchgeführt wird, und die Sicherungsdatei außerdem viel kleiner wird, wenn man den **tar**-Befehl zusammen mit **gzip** verwendet. Wenn ich dieses Archiv unter Windows benötige, dann werden `.tar.gz`-Dateien auch problemlos von WinZip entpackt. Der **tar**-Befehl ist standardmäßig auf den meisten Linux/Unix-Systemen vorhanden. Windows-Benutzer können ihn sich ebenfalls aus dem Internet [http://gnuwin32.sourceforge.net/packages/tar.htm] holen und auf ihrem Windows-System installieren.

Die entsprechende Befehlszeile sähe dann zum Beispiel so aus:

```
tar = 'tar -cvzf %s %s -X ausnahmen.txt' % (ziel, ' '.join(quellen))
```

Die Optionen werden im Folgenden erläutert.

- `-c` (für **c**reate) zeigt an, dass eine Archivdatei erzeugt werden soll.
- `-v` (für **v**erbose) bedeutet, dass die Ausgabe des Programms geschwätziger sein sollte.
- `-z` bedeutet, dass g**z**ip als Filter benutzt werden soll.
- `-f` (für **f**orce) bedeutet, dass beim Anlegen der Archivdatei eine eventuell bereits vorhandene Datei gleichen Namens überschrieben werden soll.
- `-X` (für e**x**clude) gibt eine Ausnahmedatei an, d.h. eine Textdatei, die eine Liste von Dateinamen enthält, die von der Sicherung ausgeschlossen werden sollen. Zum Beispiel könnte man in diese Datei den Eintrag `*~` schreiben, wenn man nicht möchte, dass Dateien, deren Name mit dem Zeichen ~ endet (gewöhnlicherweise sind das temporär angelegte Dateien), mit in die Sicherung aufgenommen werden sollen.

Wichtig

Die bevorzugte Weise, solche Archive anzulegen, ist unter Verwendung des Moduls `zipfile` bzw. `tarfile`- Sie gehören zur Python-Standard-Bibliothek und stehen Ihnen bereits zum Gebrauch zur Verfügung. Durch den Einsatz dieser Bibliotheken kann man auch die Benutzung von `os.system` vermeiden, was grundsätzlich ratsam ist, weil man bei Benutzung dieser Funktion sehr schnell kostspielige Fehler machen kann.

Ich habe dennoch aus didaktischen Gründen `os.system` im Sicherungskript verwendet, damit das Beispiel einfach genug ist, dass es jeder verstehen kann, und real genug, um nützlich zu sein.

Der Softwareentwicklungsprozess

Wir sind nun durch die verschiedenen **Phasen** gegangen, die beim Schreiben einer Software durchlaufen werden. Diese Phasen können wie folgt zusammengefasst werden:

1. Was (Analyse)
2. Wie (Entwurf)
3. Ausführung (Implementierung)
4. Test (Testen und Debuggen)
5. Anwendung (Produktion oder Einsatz)
6. Wartung (Verfeinerung)

Wichtig

Eine empfohlene Weise, Programme zu schreiben ist die Vorgehensweise, der wir beim Erstellen des Sicherungsskripts verfolgt haben - führen Sie die Analyse durch und machen Sie einen Entwurf. Beginnen Sie die Implementierung mit einer einfachen ersten Version. Testen und debuggen Sie das Programm. Benutzen Sie das Programm, um sicherzustellen, dass alles wie erwartet funktioniert. Fügen Sie nun weitere gewünschte Funktionalität hinzu und durchlaufen Sie den Zyklus Implementierung-Test-Anwendung so lange wie es nötig ist. Denken Sie daran, dass man '**Software wachsen lässt, und nicht fertig baut**'.

Zusammenfassung

Wir haben gesehen, wie wir unsere eigenen Python-Programme/Skripte erstellen, und die verschiedenen Phasen kennen gelernt, die beim Schreiben solcher Programme durchlaufen werden. Sie mögen es hilfreich finden, Ihre eigenen Programme auf die gleiche Weise zu erstellen, wie wir es in diesem Kapitel beispielhaft durchgeführt haben, so dass sie sowohl mit Python als auch mit Problemlösung allgemein vertraut werden.

Als Nächstes werden wir die objektorientierte Programmierweise besprechen.

Kapitel 11. Objektorientierte Programmierung

Einführung

In all unseren bisherigen Programmen haben wir unser Programm um Funktionen oder Anweisungsblöcke herum entwickelt, in denen Daten manipuliert werden. Dies wird die *prozedurorientierte* Programmierweise genannt. Es gibt eine andere Weise, sein Programm zu organisieren, die darin besteht, Daten und Funktionalität zu kombinieren und sie in etwas zusammenzupacken, das man ein Objekt nennt. Dies wird das *objektorientierte* Paradigma der Programmierung genannt. Meistens kann man mit prozedurorientierter Programmierung auskommen, aber wenn man große Programme schreiben will oder eine Lösung haben möchte, die dafür besser geeignet ist, dann sollte man objektorientierte Programmiertechniken verwenden.

Klassen und Objekte sind die beiden Hauptaspekte der objektorientierten Programmierung. Eine **Klasse** erzeugt einen neuen *Datentyp*, wobei **Objekte** die *Instanzen* der Klasse sind. Man kann dies damit vergleichen, dass man Variablen vom Datentyp `int` haben kann, was dann mit anderen Worten bedeutet, dass Variablen, die Ganzzahlen speichern, Variablen sind, die Instanzen (Objekte) der Klasse `int` sind.

Anmerkung für C/C++/Java/C#-Programmierer

Beachten Sie, dass in Python sogar Ganzzahlen als Objekte (der Klasse `int`) behandelt werden. Dies ist anders als C++ und Java (vor Version 1.5), wo Ganzzahlen primitive eingebaute Datentypen sind. Siehe `help(int)` für weitere Einzelheiten zu dieser Klasse.

C#- und Java-1.5-Programmierer wird dieses Konzept bekannt vorkommen, weil es dem *Boxing/Unboxing*-Mechanismus ähnlich ist.

Objekte können Daten in gewöhnlichen Variablen speichern, die zu dem Objekt *gehören*. Variablen, die zu einem Objekt oder einer Klasse gehören, werden **Felder** genannt. Objekte können auch Funktionalität aufweisen, die durch Funktionen ausgelöst werden, die zu der Klasse *gehören*. Solche Funktionen werden **Methoden** der Klasse genannt. Die Sprechweise ist wichtig, weil sie uns hilft, zwischen Funktionen und Variablen zu unterscheiden, die separat für sich stehen, und solchen, die zu einer Klasse oder einem Objekt gehören. Insgesamt werden die Felder und Methoden als die **Attribute** der Klasse bezeichnet.

Von Feldern gibt es zwei Typen - sie können entweder zu jeder Instanz (jedem Objekt) der Klasse gehören, oder sie können zur Klasse selbst gehören. Dementsprechend werden sie im ersten Fall **Instanzvariablen** und im zweiten Fall **Klassenvariablen** genannt.

Eine Klasse wird mit dem Schlüsselwort `class` erzeugt. Die Felder und Methoden der Klasse werden in einem eingerückten Block aufgelistet.

Der Selbstbezug

Klassenmethoden haben nur eine Besonderheit gegenüber gewöhnlichen Funktionen - sie müssen einen zusätzlichen ersten Namen haben, der am Beginn der Parameterliste hinzugefügt wird, dabei übergibt man diesem Parameter jedoch **nicht** einen Wert, wenn man die Methode aufruft, sondern Python sorgt dafür. Diese spezielle Variable repräsentiert das Objekt selbst, und es hat sich eingebürgert, ihr den Namen `self` zu geben.

Man könnte diesem Parameter zwar irgendeinen Namen geben, aber es wird dennoch *ausdrücklich empfohlen*, den Namen `self` zu benutzen - jeder andere Name ist an dieser Stelle eindeutig ver-

pönt. Es hat viele Vorzüge, einen Standardnamen zu benutzen - jeder Leser Ihres Programms wird ihn sofort erkennen, und spezielle IDEs (Integrierte Entwicklungsumgebungen) können einen unterstützen, wenn man die Bezeichnung `self` verwendet.

Anmerkung für C++/Java/C#-Programmierer

Das `self` in Python entspricht dem `self`-Zeiger in C++ und dem `this` in Java und C#.

Sie fragen sich wahrscheinlich, auf welche Weise Python den Wert für `self` übergibt und warum Sie dafür keinen Wert zu übergeben brauchen. Ein Beispiel wird dies klar machen. Angenommen, Sie haben eine Klasse, die `MeineKlasse` heißt und eine Instanz dieser Klasse, die `MeinObjekt` heißt. Wenn Sie eine Methode dieses Objekts als `MeinObjekt.methode(param1, param2)` aufrufen, dann wird dies von Python automatisch in `MeineKlasse.methode(MeinObjekt, param1, param2)` umgewandelt - und genau dafür ist dieses besondere `self` da.

Das bedeutet auch, dass man selbst bei einer Methode, die keine Parameter entgegennimmt, dennoch diese Methode mit einem `self`-Parameter definieren muss.

Klassen

Die einfachste mögliche Klasse zeigt das folgende Beispiel.

Erzeugen einer Klasse

Beispiel 11.1. Erzeugen einer Klasse (einfachsteklasse.py [code/einfachsteklasse.py])

```
#!/usr/bin/python

class Person:
 pass # Ein leerer Block

p = Person()
print p
```

Ausgabe

```
$ python einfachsteklasse.py
<__main__.Person instance at 0xf6fcb18c>
```

So funktioniert es

Wir erzeugen eine neue Klasse, indem wir die Anweisung `class` benutzen, gefolgt von dem Namen der Klasse. Darauf folgt ein eingerückter Block von Anweisungen, die den Rumpf der Klasse bilden. In diesem Fall haben wir einen leeren Block, was durch die Anweisung `pass` angezeigt wird.

Als Nächstes erzeugen wir ein Objekt (eine Instanz) dieser Klasse, indem wir den Namen der Klasse benutzen, gefolgt von einem Klammerpaar. (Wir werden im nächsten Abschnitt mehr über diese so

genannte Instanziierung lernen). Zur Sicherheit überprüfen wir den Typ der Variable, indem wir sie einfach mit `print` ausgeben. Dies zeigt uns, dass es sich um eine Instanz der Klasse `Person` im Modul `__main__` handelt.

Beachten Sie, dass die Adresse, an der Ihr Objekt im Hauptspeicher Ihres Computers gespeichert ist, ebenfalls ausgegeben wird. Diese Adresse wird auf Ihrem Computer einen anderen Wert haben, weil Python die Objekte überall speichern kann, wo es Platz dafür gibt.

Objektmethoden

Wir haben bereits besprochen, dass Klassen/Objekte Methoden haben können, die bis auf den zusätzlichen Übergabeparameter `self` gewöhnliche Funktionen sind. Wir werden hierfür nur ein Beispiel sehen.

Benutzung von Objektmethoden

Beispiel 11.2. Benutzung von Objektmethoden (methode.py [code/methode.py])

```
#!/usr/bin/python

class Person:
 def sagHallo(self):
  print 'Hallo, wie geht es Ihnen?'

p = Person()
p.sagHallo()

# Dieses kurze Beispiel kann auch als
# Person().sagHallo() geschrieben werden.
```

Ausgabe

```
$ python methode.py
Hallo, wie geht es Ihnen?
```

So funktioniert es

Hier sehen wir, wie die Sache mit dem `self` funktioniert. Beachten Sie, dass die Methode `sagHallo` keinen Parameter entgegennimmt, aber dennoch das `self` in der Funktionsdefinition hat.

Die __init__-Methode

Es gibt viele Methodennamen, die in Pythonklassen eine besondere Bedeutung haben. Wir werden nun die Bedeutung der Methode `__init__` sehen.

Die Methode `__init__` wird aufgerufen, sobald ein Objekt einer Klasse instanziiert wird. Die Methode kann dafür benutzt werden, ihr Objekt auf irgendeine Weise zu *initialisieren*. Beachten Sie die doppelten Unterstriche sowohl am Anfang als auch am Ende des Namens.

Benutzung der __init__-Method

Beispiel 11.3. Benutzung der __init__-Method (klasse_init.py [code/klasse_init.py])

```
#!/usr/bin/python

class Person:
 def __init__(self, name):
  self.name = name
 def sagHallo(self):
  print 'Hallo, mein Name ist', self.name

p = Person('Swaroop')
p.sagHallo()

# Dieses kurze Beispiel kann auch als
# Person('Swaroop').sagHallo() geschrieben werden.
```

Ausgabe

```
$ python klasse_init.py
Hallo, mein Name ist Swaroop
```

So funktioniert es

Hier definieren wir die Methode `__init__` so, dass sie einen Parameter `name` entgegennimmt (zusammen mit dem üblichen `self`). Wir erzeugen hier einfach ein neues Feld, das ebenfalls `name` heißt. Beachten Sie, dass dies zwei unterschiedliche Variablen sind, obwohl sie den gleichen Namen haben. Die Schreibweise mit dem Punkt ermöglicht es uns, zwischen den beiden zu unterscheiden.

Beachten Sie vor allem, dass wir die Methode `__init__` nicht explizit aufrufen, sondern die Argumente in Klammern nach dem Klassennamen übergeben, wenn wir eine neue Instanz der Klasse erzeugen. Das ist die besondere Bedeutung dieser Methode.

Nun können wir das Feld `self.name` in unseren Methoden benutzen, wie es anhand der Methode `sagHallo` demonstriert wird.

Anmerkung für C++/Java/C#-Programmierer

Die Methode `__init__` entspricht einem *Konstruktor* in C++, C# oder Java.

Klassen- und Objektvariablen

Wir haben bereits den Teil der Klassen und Objekte besprochen, der für ihre Funktionalität sorgt; nun werden wir den Teil betrachten, der die Daten enthält. Eigentlich handelt es sich um nichts anderes als gewöhnliche Variablen, die an die **Namensräume** der Klassen und Objekte *gebunden* sind, d.h. die Namen sind nur innerhalb des Kontextes der Klassen und Objekte gültig.

Es gibt zwei Arten von *Feldern* - Klassenvariablen und Objektvariablen, die danach klassifiziert werden, ob die Klasse oder das Objekt die jeweiligen Variablen *besitzt*.

Klassenvariablen werden gemeinsam benutzt, in dem Sinne, dass auf sie von allen Objekten (Instanzen) der Klasse zugegriffen wird. Es gibt nur eine Kopie einer Klassenvariable, und wenn irgendein Objekt eine Änderung an einer Klassenvariable vornimmt, dann spiegelt sich diese Änderung sofort auch in allen anderen Instanzen der Klasse wieder.

Objektvariablen gehören den einzelnen Objekten (Instanzen) der Klasse individuell. In diesem Fall hat jedes Objekt seine eigene Kopie des Feldes, d.h. sie werden nicht gemeinsam benutzt und sind auf keine Weise mit dem Feld des gleichen Namens in einer anderen Instanz der selben Klasse verknüpft. An einem Beispiel werden wir das leicht verstehen.

Benutzung von Klassen- und Objektvariablen

Beispiel 11.4. Benutzung von Klassen- und Objektvariablen (objvar.py [code/objvar.py])

```
#!/usr/bin/python

class Person:
 '''Stellt eine Person dar.'''
 bevoelkerung = 0

 def __init__(self, name):
  '''Initialisiert die Daten der Person.'''
  self.name = name
  print '(Initialisiere %s)' % self.name

  # Wenn diese Person erzeugt wird,
  # traegt er/sie zur Bevoelkerung bei
  Person.bevoelkerung += 1

 def __del__(self):
  '''Ich sterbe.'''
  print '%s verabschiedet sich.' % self.name

  Person.bevoelkerung -= 1

  if Person.bevoelkerung == 0:
   print 'Ich bin der letzte.'
  else:
   print 'Es gibt noch %d Leute.' % Person.bevoelkerung

 def sagHallo(self):
  '''Begruessung durch die Person.

  Das ist wirklich alles, was hier geschieht.'''
  print 'Hallo, mein Name ist %s.' % self.name

 def wieViele(self):
  '''Gibt die aktuelle Bevoelkerungszahl aus.'''
  if Person.bevoelkerung == 1:
   print 'Ich bin ganz allein hier.'
  else:
   print 'Es gibt hier %d Leute.' % Person.bevoelkerung
```

```
swaroop = Person('Swaroop')
swaroop.sagHallo()
swaroop.wieViele()

kalam = Person('Abdul Kalam')
kalam.sagHallo()
kalam.wieViele()

swaroop.sagHallo()
swaroop.wieViele()
```

Ausgabe

```
$ python objvar.py
(Initialisiere Swaroop)
Hallo, mein Name ist Swaroop.
Ich bin ganz allein hier.
(Initialisiere Abdul Kalam)
Hallo, mein Name ist Abdul Kalam.
Es gibt hier 2 Leute.
Hallo, mein Name ist Swaroop.
Es gibt hier 2 Leute.
Abdul Kalam verabschiedet sich.
Es gibt noch 1 Leute.
Swaroop verabschiedet sich.
Ich bin der letzte.
```

So funktioniert es

Dies ist ein langes Beispiel, aber es hilft uns, das Wesen von Klassen- und Objektvariablen zu demonstrieren. Die Variable `bevoelkerung` gehört hier zur Klasse `Person` und ist daher eine Klassenvariable. Die Variable `name` gehört dagegen zum jeweiligen Objekt (das durch `self` zugewiesen wird) und ist daher eine Objektvariable.

Wir nehmen daher auf die Klassenvariable `bevoelkerung` Bezug, indem wir schreiben `Person.bevoelkerung` und nicht `self.bevoelkerung`. Beachten Sie, dass eine Objektvariable mit dem gleichen Namen wie eine Klassenvariable diese Klassenvariable versteckt! Wir nehmen auf die Objektvariable `name` Bezug, indem wir die Schreibweise `self.name` in den Methoden dieses Objekts benutzen. Merken Sie sich diesen einfachen Unterschied zwischen Klassen- und Objektvariablen.

Beachten Sie, dass die `__init__`-Methode benutzt wird, um die Instanz von `Person` mit einem Namen zu initialisieren. In dieser Methode erhöhen wir den Zähler `bevoelkerung` um 1, weil wir eine weitere Person hinzugefügt haben. Beachten Sie auch, dass der Wert von `self.name` vom jeweiligen Objekt abhängt, was das Wesen von Objektvariablen ausmacht.

Denken Sie daran, dass Sie auf die Variablen und Methoden des gleichen Objekts **nur** mit Hilfe der Variablen `self` Bezug nehmen können. Dies wird als *Attribut-Referenzierung* bezeichnet.

Wir sehen in diesem Programm auch die Benutzung von **Dokumentations-Strings** sowohl für Klassen als auch für Methoden. Wir können auf den Dokumentations-String der Klasse zur Laufzeit

mit `Person.__doc__` zugreifen, und mit `Person.sagHallo.__doc__` auf den Dokumentations-String der Methode.

Ganz ähnlich zur Methode `__init__` gibt es eine andere spezielle Methode `__del__`, die aufgerufen wird, wenn ein Objekt aufhört zu leben, d.h. wenn es nicht mehr gebraucht wird und der von ihm belegte Speicher an das System zur Wiederverwendung zurückgegeben wird. In dieser Methode erniedrigen wir einfach den Zähler `Person.bevoelkerung` um 1.

Die `__del__`-Methode wird automatisch aufgerufen, wenn das Objekt nicht mehr gebraucht wird, und es gibt keine Garantie dafür, *wann* dies der Fall ist. Wenn Sie dies explizit tun wollen, müssen Sie einfach die `del`-Anweisung benutzen, die wir bereits aus früheren Beispielen kennen.

Anmerkung für C++/Java/C#-Programmierer

In Python sind alle Mitglieder, d.h. alle Attribute einschließlich der Felder *public*, und alle Methoden sind *virtual*.

Eine Ausnahme: Wenn Sie Felder verwenden, deren Namen mit *einem doppelten Unterstrich beginnt*, so wie `__privatvar`, dann modifiziert Python den Namen so, dass daraus praktisch eine private Variable wird.

Daraus hat sich die Konvention ergeben, dass Variablen, die nur innerhalb der Klasse oder des Objekts benutzt werden, mit einem Unterstrich beginnen sollten, während alle anderen Namen öffentlich sind und von anderen Klassen/Objekten benutzt werden können. Denken Sie daran, dass dies nur eine Konvention ist, und nicht von Python erzwungen wird (außer bei Variablen, deren Name mit einem doppelten Unterstrich beginnt).

Beachten Sie auch, dass die `__del__`-Methode dem Konzept eines Destruktors entspricht.

Vererbung

Eine der Hauptvorzüge objektorientierter Programmierung ist die Möglichkeit der **Wiederverwendung** von Programmcode, und eine Weise, mit der dies erreicht wird, ist durch den Mechanismus der *Vererbung*. Vererbung kann man sich am besten als eine mit Klassen realisierte Beziehung zwischen *Datentypen und Unterdatentypen* vorstellen.

Angenommen, Sie wollen ein Programm schreiben, das die Dozenten und Studenten an einer Hochschule verwaltet. Diese beiden Personengruppen haben einige gemeinsame Merkmale wie Name, Alter und Adresse. Sie haben auch einige spezielle Merkmale wie Gehalt, Vorlesungen und Beurlaubungen für Dozenten und Prüfungsnoten und Studiengebühren für Studenten.

Sie können zwei unabhängige Klassen für die beiden Typen und Prozesse anlegen, aber dies würde bedeuten, dass Sie, wenn Sie ein neues gemeinsames Merkmal hinzufügen wollen, es bei jeder der beiden voneinander unabhängigen Klassen tun hinzufügen müssen. Das wird sehr schnell unhandlich.

Eine bessere Lösung besteht darin, eine gemeinsame Klasse namens `SchulMitglied` anzulegen, und dann die Klasssen für Dozenten und Studenten von dieser Klasse *erben* zu lassen, d.h. sie werden zu Unterdatentypen dieses Datentyps (dieser Klasse) gemacht, und wir können diesen Unterdatentypen dann weitere besondere Merkmale hinzufügen.

Dieser Ansatz hat viele Vorteile. Wenn wir der Klasse `SchulMitglied` Funktionalität hinzufügen oder diese ändern, dann spiegelt sich dies automatisch in den Unterdatentypen wieder. Zum Beispiel können Sie ein neues Feld für die Nummer einer Identitätskarte sowohl für Dozenten als auch für Studenten hinzufügen, indem Sie dies einfach in der Klasse `SchulMitglied` ergänzen. Änderungen an den Unterdatentypen haben jedoch keine Auswirkung auf andere Unterdatentypen. Ein weiterer Vorteil besteht darin, dass man auf ein Objekt der Klasse Dozent oder Student auch als ein Objekt der Klasse `SchulMitglied` Bezug nehmen kann, was in einigen Situationen nützlich sein könnte, z. B. wenn man die Anzahl der Mitglieder der Hochschule zählen möchte. Diese Möglichkeit, einen

Unterdatentyp in jeder Situation, wo ein Elterndatentyp erwartet wird, ersetzen zu können, d.h. ein Objekt als eine Instanz der Elternklasse behandeln zu können, wird **Polymorphismus** genannt.

Beachten Sie auch, dass wir den Programmcode der Elternklasse *wiederverwenden*, dass wir ihn also nicht in den verschiedenen Klassen wiederholen müssen, wie wir es hätten tun müssen, wenn wir unabhängige Klassen benutzt hätten.

Die Klasse `SchulMitglied` wird in dieser Konstellation auch als die *Basisklasse* oder die *Superklasse* bezeichnet. Die Klassen `Dozent` und `Student` werden die *abgeleiteten Klassen* oder *Subklassen* genannt.

Hier ist nun das Beispiel als Programm.

Verwendung von Vererbung

Beispiel 11.5. Verwendung von Vererbung (vererbung.py [code/vererbung.py])

```
#!/usr/bin/python

class SchulMitglied:
 '''Repraesentiert ein beliebiges Mitglied der Hochschule.'''
 def __init__(self, name, alter):
  self.name = name
  self.alter = alter
  print '(SchulMitglied %s initialisiert)' % self.name

 def auskunft(self):
  '''Gib Auskunft ueber das Mitglied.'''
  print 'Name: "%s" Alter: "%s"' % (self.name, self.alter),

class Dozent(SchulMitglied):
 '''Repraesentiert einen Dozenten der Hochschule.'''
 def __init__(self, name, alter, gehalt):
  SchulMitglied.__init__(self, name, alter)
  self.gehalt = gehalt
  print '(Dozent %s initialisiert)' % self.name

 def auskunft(self):
  SchulMitglied.auskunft(self)
  print 'Gehalt: "%d Euro"' % self.gehalt

class Student(SchulMitglied):
 '''Repraesentiert einen Studenten der Hochschule.'''
 def __init__(self, name, alter, note):
  SchulMitglied.__init__(self, name, alter)
  self.note = note
  print '(Student %s initialisiert)' % self.name

 def auskunft(self):
  SchulMitglied.auskunft(self)
  print 'Letzte Pruefungsnote: "%1.1f"' % self.note

d = Dozent('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 22, 1.7)

print # gib eine Leerzeile aus
```

```
mitglieder = [d, s]
for mitglied in mitglieder:
 mitglied.auskunft() # geht bei Dozenten und Studenten
```

Ausgabe

```
$ python vererbung.py
(SchulMitglied Mrs. Shrividya initialisiert)
(Dozent Mrs. Shrividya initialisiert)
(SchulMitglied Swaroop initialisiert)
(Student Swaroop initialisiert)

Name: "Mrs. Shrividya" Alter: "40" Gehalt: "30000 Euro"
Name: "Swaroop" Alter: "22" Letzte Pruefungsnote: "1.7"
```

So funktioniert es

Um Vererbung zu benutzen, geben wir die Namen der Basisklassen in einem Tupel an, das in der Klassendefinition dem Namen der Klasse folgt. Danach ist zu beachten, dass die `__init__`-Methode der Basisklasse unter Benutzung der `self`-Variable explizit aufgerufen wird, damit der Anteil des Objekts, der von der Basisklasse bereitgestellt wird, initialisiert werden kann. Es ist sehr wichtig, sich dies zu merken - Python ruft nicht automatisch den Konstruktor der Basisklasse auf; Sie müssen ihn selber explizit aufrufen.

Wir sehen hier auch, dass wir Methoden der Basisklasse aufrufen können, indem wir den Namen der Basisklasse mit einem Punkt voranstellen, und dann die Variable `self` zusammen mit den anderen Parametern der Methode übergeben.

Beachten Sie, dass wir Instanzen von `Dozent` und `Student` einfach wie Instanzen von `SchulMitglied` behandeln, wenn wir die `auskunft`-Methode der Klasse `SchulMitglied` benutzen.

Beachten Sie auch, dass die `auskunft`-Methode der Subklassen aufgerufen wird, und nicht die `auskunft`-Methode der Superklasse `SchulMitglied`. Man kann dies so verstehen, dass Python *immer* zuerst versucht, Methoden der jeweiligen Klasse zu finden, die in diesem Fall vorhanden sind. Wenn die Methode nicht gefunden wird, dann fängt Python an, die Methoden der zugehörigen Basisklassen eine nach der anderen durchzugehen, in der Reihenfolge, wie sie in dem Tupel in der Klassendefinition angegeben ist.

Eine Anmerkung noch zur Sprechweise - wenn mehr als eine Klasse in dem Tupel der Basisklassen angegeben ist, von denen die Klasse erbt, dann spricht man von *Mehrfachvererbung*.

Zusammenfassung

Wir haben nun die verschiedenen Aspekte von Klassen und Objekten erforscht und die damit verbundenen Sprechweisen kennen gelernt. Wir haben auch die Vorteile und möglichen Stolperfallen bei der objektorientierten Programmierung gesehen. Python ist hochgradig objektorientiert, und wenn wir diese Konzepte genau verstehen, wird uns dies langfristig sehr viel helfen.

Als Nächstes lernen wir, mit Ein/Ausgaben umzugehen und in Python auf Dateien zuzugreifen.

Kapitel 12. Ein/Ausgabe

Sie werden häufig die Anforderung haben, dass Ihr Programm mit dem Benutzer (der Sie auch selber sein können) interagieren soll. Das Programm soll etwa eine Eingabe vom Benutzer entgegennehmen und dann irgendwelche Ergebnisse ausgeben. Wir können dies mit `raw_input` bzw. `print`-Anweisungen erreichen. Zur Ausgabe können wir auch die verschiedenen Methoden der Klasse `str` (String) verwenden. Zum Beispiel können Sie die `rjust`-Methode einsetzen, um einen String zu erhalten, der bezüglich einer angegeben Breite rechts ausgerichtet ist. Für weitere Einzelheiten, siehe `help(str)`.

Ein andere übliche Art von Ein/Ausgabe besteht aus dem Umgang mit Dateien. Die Möglichkeit, Dateien zu erzeugen, zu lesen und zu schreiben ist für viele Programme wesentlich, und wir werden diesen Aspekt in diesem Kapitel untersuchen.

Dateien

Sie können Dateien zum Lesen und Schreiben öffnen, indem Sie ein Objekt der Klasse `file` erzeugen und dessen Methoden `read`, `readline` oder `write` verwenden. Die Möglichkeit, aus der Datei zu lesen oder in die Datei zu schreiben hängt von dem Modus ab, den Sie beim Öffnen der Datei angegeben haben. Wenn Sie schließlich die Dateioperationen abschließen wollen, rufen Sie die Methode `close` auf, um Python mitzuteilen, dass Sie mit dieser Datei fertig sind.

Der Gebrauch von `file`

Beispiel 12.1. Verwendung von Dateien (beispiel_file.py [code/beispiel_file.py])

```
#!/usr/bin/python

gedicht = '''\
Programmieren mit Elan
und die Arbeit wird getan,
willst du Spass haben daran:
 Nimm Python!
'''

f = file('gedicht.txt', 'w') # "w" = Schreiben
f.write(gedicht) # schreibe den Text in die Datei
f.close() # schliesse die Datei

f = file('gedicht.txt') # kein Modus bedeutet "r" = Lesen
while True:
 line = f.readline()
 if len(line) == 0: # eine leere Zeile bedeutet Dateiende (EOF)
  break
 print line, # das Komma dient zur Unterdrueckung des Zeilenvorschubs
f.close() # schliesse die Datei
```

Ausgabe

```
$ python beispiel_file.py
Programmieren mit Elan
```

```
und die Arbeit wird getan,
willst du Spass haben daran:
        Nimm Python!
```

So funktioniert es

Zuerst erzeugen wir eine Instanz der Klasse `file`, indem wir den Namen der Datei und den Modus angeben, in dem die Datei geöffnet werden soll. Der Modus kann Lesemodus sein (`'r'` für *read*), Schreibmodus (`'w'` für *write*) oder Anfügemodus (`'a'` für *append*). Es gibt tatsächlich noch viele weitere Modi, und mit `help(file)` erhalten Sie nähere Einzelheiten dazu.

Zuerst öffnen wir die Datei im Schreibmodus und benutzen die Methode `write` der Klasse `file`, um in die Datei zu schreiben, danach schließen wir die Datei wieder mit der Methode `close`.

Als Nächstes öffnen wir die gleiche Datei zum Lesen. Wenn wir keinen Modus angeben, dann wird standardmäßig der Lesemodus benutzt. Wir lesen jede Zeile der Datei in einer Schleife ein, wobei wir die Methode `readline` benutzen. Diese Methode gibt jeweils eine vollständige Zeile zurück, einschließlich der Zeilenvorschubkennung am Zeilenende. Wenn daher ein *leerer* String zurückgeliefert wird, dann zeigt uns dies an, dass das Ende der Datei erreicht wurde, und wir beenden die Schleife.

Beachten Sie, dass wir an die `print`-Anweisung ein Komma angehängt haben, um den automatischen Zeilenvorschub zu unterdrücken, der normalerweise bei jeder `print`-Anweisung hinzugefügt wird. Die aus der Datei eingelesene Zeile endet bereits mit einer Zeilenvorschubkennung, ein weiterer ist unnötig. Am Ende schließen wir die Datei wieder mit der Methode `close`.

Nun betrachten wir den Inhalt der Datei `gedicht.txt`, um sicherzustellen, dass das Programm wirklich einwandfrei funktioniert hat.

Eingemachtes

Jetzt geht's ans 'Eingemachte'. Python stellt ein Standard-Modul namens `pickle` ('einmachen') zu Verfügung, mit dessen Hilfe man **jedes** Python-Objekt in eine Datei speichern kann und es dann später wieder intakt zurück erhält. Man nennt dies die *persistente* Speicherung eines Objekts. Sie können Sich dies auch als das 'Einfrieren' und wieder 'Auftauen' eines Objekts vorstellen.

Es gibt ein anderes Modul namens `cPickle`, das ganz genauso wie das `pickle`-Modul funktioniert, außer dass es in der Sprache C geschrieben und daher (bis zu 1000 mal) schneller ist. Sie können sich aussuchen, welches der beiden Module Sie verwenden wollen, auch wenn wir hier als Beispiel das Modul `cPickle` verwenden. Denken Sie aber daran, dass wir der Einfachheit halber immer nur vom `pickle`-Modul sprechen.

Einfrieren und wieder Auftauen

Beispiel 12.2. Einfrieren und wieder Auftauen (einmachen.py [code/einmachen.py])

```
#!/usr/bin/python

import cPickle as p
#import pickle as p

einkaufsdatei = 'einkaufsliste.data' # Datei, in der wir das Objekt speichern
```

```
einkaufsliste = ['Aepfel', 'Mangos', 'Karotten']

# Schreibe in die Datei
f = file(einkaufsdatei, 'w')
p.dump(einkaufsliste, f) # speichere das Objekt in der Datei
f.close()

del einkaufsliste # loesche die einkaufsliste

# Lies die Einkaufsliste aus der Datei wieder ein
f = file(einkaufsdatei)
gespeicherteliste = p.load(f)
print gespeicherteliste
```

Ausgabe

```
$ python einmachen.py
['Aepfel', 'Mangos', 'Karotten']
```

So funktioniert es

Beachten Sie als erstes, dass wir die `import..as`-Syntax verwendet haben. Dies ist praktisch, denn wir können einen kürzeren Namen für das Modul verwenden. Es ermöglicht uns in diesem Fall sogar, ein anderes Modul (`cPickle` oder `pickle`) zu verwenden, indem wir einfach nur eine Zeile ändern! Im Rest des Programms beziehen wir uns auf dieses Modul als `p`.

Um ein Objekt in einer Datei zu speichern, öffnen wir zunächst ein `file`-Objekt im Schreibmodus und speichern das Objekt in der offenen Datei, indem wir die Funktion `dump` des `pickle`-Moduls benutzen. Dieser Vorgang wird als das *Einmachen* (*pickling*) oder *Einfrieren* bezeichnet.

Als Nächstes holen wir das Objekt mit der Funktion `load` des `pickle`-Moduls wieder zurück. Dieser Vorgang wird als *Wiederauftauen* (oder *unpickling*) bezeichnet.

Zusammenfassung

Wir haben verschiedene Arten von Ein/Ausgabe besprochen, sowie den Umgang mit Dateien und den Gebrauch des `pickle`-Moduls.

Als Nächstes werden wir das Konzept der so genannten Ausnahmen erforschen.

Kapitel 13. Ausnahmen

Ausnahmen (*exceptions*) treten auf, wenn bestimmte *Ausnahmesituationen* in Ihrem Programm eintreten. Was zum Beispiel passiert, wenn Sie eine Datei einlesen wollen, und die Datei gar nicht existiert? Oder wenn Sie die Datei aus Versehen gelöscht haben, während das Programm lief? Solche Situationen werden mittels **Ausnahmen** behandelt.

Was passiert, wenn es in Ihrem Programm ungültige Anweisungen gibt? In einem solchen Fall reagiert Python, indem es eine Ausnahme **auslöst**, und Ihnen mitteilt, dass es auf einen **Fehler** gestoßen ist.

Fehler

Nehmen Sie eine einfache `print`-Anweisung als Beispiel. Was passiert, wenn wir uns vertippt haben, und `Print` statt `print` geschrieben haben? Beachten Sie die fehlerhafte Großschreibung. In diesem Fall wird von Python ein Syntaxfehler *ausgelöst*.

```
>>> Print 'Hallo Welt'
    File "<stdin>", line 1
      Print 'Hallo Welt'
                         ^
SyntaxError: invalid syntax

>>> print 'Hallo Welt'
Hallo Welt
```

Beachten Sie, dass eine Ausnahme namens `SyntaxError` ausgelöst wird, und dass außerdem die Stelle, an der dieser Fehler auftrat, ausgegeben wird. Dies geschieht durch eine eingebaute *Fehlerbehandlung* für solche Ausnahmen, die durch Syntaxfehler ausgelöst wurden.

try..except

Wir wollen *versuchen*, eine Eingabe vom Benutzer entgegenzunehmen. Drücken Sie **Strg**+**d** (oder **Strg**+**z** unter Windows), und achten Sie darauf, was passiert.

```
>>> s = raw_input('Geben Sie etwas ein --> ')
Enter something --> Traceback (most recent call last):
  File "<stdin>", line 1, in ?
EOFError
```

Python löst als Fehler eine Ausnahme namens `EOFError` aus, was grundsätzlich bedeutet, dass das *Ende einer Datei* (EOF) erreicht wurde (bewirkt durch die Eingabe **Strg**+**d** oder **Strg**+**z**), wo dies nicht erwartet wurde.

Wir sehen nun, wie man solche Fehler behandelt.

Ausnahmebehandlung

Wir können Ausnahmen mit Hilfe der Anweisung `try..except` behandeln. Wir schreiben dazu unsere normalen Anweisungen in den Block, der auf `try` folgt, und schreiben unsere Anweisungen zur Fehlerbehandlung in den Block, der auf `except` folgt.

Beispiel 13.1. Ausnahmebehandlung (try_except.py [code/try_except.py])

```
#!/usr/bin/python

import sys

try:
 s = raw_input('Geben Sie etwas ein --> ')
except EOFError:
 print '\nWarum haben Sie die Eingabe abgebrochen?'
 sys.exit() # beendet das Programm
except:
 print '\nIrgendein Fehler hat eine Ausnahme ausgeloest.'
 # an dieser Stelle beenden wir das Programm nicht

print 'Fertig'
```

Ausgabe

```
$ python try_except.py
Geben Sie etwas ein -->
Warum haben Sie die Eingabe abgebrochen?

$ python try_except.py
Geben Sie etwas ein --> Python ist eine Ausnahmeerscheinung!
Fertig
```

So funktioniert es

Wir stecken alle Anweisungen, die einen Fehler auslösen könnten, in den `try`-Block und behandeln dann alle Fehler und Ausnahmen in dem Block der `except`-Klausel. Die `except`-Klausel kann einzeln angegebene Fehler oder Ausnahmen behandeln, oder auch eine in Klammern angegebene Liste von Fehlern/Ausnahmen. Wenn keine Namen von Fehlern oder Ausnahmen angegeben werden, dann behandelt sie *alle* Fehler und Ausnahmen. Zu jeder `try`-Klausel muss mindestens eine `except`-Klausel gehören (oder stattdessen eine `finally`-Klausel, aber darüber sprechen wir später).

Wenn irgendein Fehler oder eine Ausnahme nicht behandelt wird, dann wird die Standard-Ausnahmebehandlung von Python aufgerufen, die einfach die Ausführung des Programms beendet und eine Fehlermeldung ausgibt. Wir haben dies bereits mehrfach gesehen.

Sie können auch eine `else`-Klausel mit einem `try..except`-Block kombinieren. Die `else`-Klausel wird dann ausgeführt, wenn keine Ausnahme auftritt.

Wir können auch auf das Ausnahmeobjekt zugreifen, um dadurch weitere Informationen über die Ausnahme zu erhalten, die aufgetreten ist. Dies wird im nächsten Beispiel gezeigt.

Auslösen von Ausnahmen

Sie können Ausnahmen mit der Anweisung `raise` auch selber *auslösen*. Sie müssen zusätzlich den Namen des Fehlers bzw. der Ausnahme angeben, und können außerdem weitere Informationen angeben, die bei der Auslösung dieser Ausnahme an die Ausnahmebehandlung weitergegeben werden sol-

len. Der Fehler oder die Ausnahme sollte eine Klasse sein, die direkt oder indirekt von der Klasse `Exception` abgeleitet ist.

So löst man Ausnahmen aus

Beispiel 13.2. So löst man Ausnahmen aus (ausnahmen.py [code/ausnahmen.py])

```
#!/usr/bin/python

class KurzeEingabeAusnahme(Exception):
 '''Eine benutzerdefinierte Ausnahmeklasse.'''
 def __init__(self, laenge, mindestens):
  Exception.__init__(self)
  self.laenge = laenge
  self.mindestens = mindestens

try:
 s = raw_input('Geben Sie etwas ein --> ')
 if len(s) < 3:
  raise KurzeEingabeAusnahme(len(s), 3)
 # Hier kann man ganz normal mit der Arbeit fortfahren
except EOFError:
 print '\nWarum haben Sie die Eingabe abgebrochen?'
except KurzeEingabeAusnahme, x:
 print 'KurzeEingabeAusnahme: Eingabe hatte die Laenge %d,' \
  ' gefordert war mindestens %d.' % (x.laenge, x.mindestens)
else:
 print 'Es wurde keine Ausnahme ausgeloest.'
```

Ausgabe

```
$ python ausnahmen.py
Geben Sie etwas ein -->
Warum haben Sie die Eingabe abgebrochen?

$ python ausnahmen.py
Geben Sie etwas ein --> ab
KurzeEingabeAusnahme: Eingabe hatte die Laenge 2, gefordert war mindestens 3.

$ python ausnahmen.py
Geben Sie etwas ein --> abc
Es wurde keine Ausnahme ausgeloest.
```

So funktioniert es

Wir erzeugen hier unseren eigenen Ausnahmetyp, obwohl wir auch jede andere vordefinierte Ausnahme zu Demonstrationszwecken hätten benutzen können. Dieser neue Ausnahmetyp ist die Klasse `KurzeEingabeAusnahme`. Sie hat zwei Felder - `laenge`, die Länge der Eingabe, und `mindestens`, die Mindestlänge für die Eingabe, die das Programm erwartet.

In der `except`-Klausel geben wir sowohl die Fehlerklasse an, als auch die Variable, die das entsprechende Fehler/Ausnahmeobjekt erhalten soll. Dies entspricht den Parametern und Argumenten in einem Funktionsaufruf. Innerhalb dieser speziellen `except`-Klausel benutzen wir die Felder `laenge` und `mindestens` des Ausnahmeobjekts, um eine entsprechende Fehlermeldung für den Benutzer auszugeben.

try..finally

Wie würden Sie vorgehen, wenn Sie eine Datei lesen, und diese Datei danach schließen wollen, unabhängig davon, ob eine Ausnahme ausgelöst wurde oder nicht? Dies kann mit Hilfe eines `finally`-Blocks erreicht werden. Beachten Sie jedoch, dass Sie *keine* `except`-Klausel zusammen mit einem `finally`-Block für den gleichen zugehörigen `try`-Block benutzen dürfen. Wenn Sie beide zusammen benutzen wollen, dann müssen Sie einen von beiden Blöcken in den anderen verschachteln.

Gebrauch von finally

Beispiel 13.3. Gebrauch von finally (finally.py [code/finally.py])

```
#!/usr/bin/python

import time

try:
 f = file('gedicht.txt')
 while True: # unsere uebliche Weise, Dateien zu lesen
  zeile = f.readline()
  if len(zeile) == 0:
   break
  time.sleep(2)
  print zeile,
finally:
 f.close()
 print 'Raeume auf... Datei geschlossen.'
```

Ausgabe

```
$ python finally.py
Programmieren mit Elan
und die Arbeit wird getan,
Raeume auf... Datei geschlossen.
Traceback (most recent call last):
  File "finally.py", line 12, in ?
    time.sleep(2)
KeyboardInterrupt
```

So funktioniert es

Wir lesen die Datei wieder auf die übliche Weise ein, aber ich habe absichtlich eine Pause von 2 Sekunden eingebaut, bevor jede Zeile ausgegeben wird, indem ich die Methode `time.sleep` benutzt

habe. Der einzige Grund, warum ich das getan habe, ist, damit das Programm langsam abläuft (Python ist von Natur aus sehr schnell). Während das Programm noch läuft, drücken Sie **Strg**+**c**, um das Programm zu unterbrechen und zu beenden.

Beachten Sie, dass eine `KeyboardInterrupt`-Ausnahme ausgelöst wird und das Programm abbricht, jedoch vor dem Programmabbruch noch die `finally`-Klausel ausführt und die Datei schließt.

Zusammenfassung

Wir haben den Gebrauch von `try..except` und `try..finally` erörtert. Wir haben gesehen, wie wir unsere eigenen Ausnahmetypen erzeugen können, und wie wir auch selber Ausnahmen auslösen können.

Als Nächstes untersuchen wir die Python-Standard-Bibliothek.

Kapitel 14. Die Standardbibliothek von Python

Einführung

Die Python-Standardbibliothek ist in jeder Python-Installation verfügbar. Sie enthält eine riesige Anzahl äußerst nützlicher Module. Es ist sehr wichtig, dass Sie sich mit der Python-Standardbibliothek vertraut machen, denn die meisten Ihrer Probleme können leichter und schneller gelöst werden, wenn Sie mit dieser Bibliothek von Modulen vertraut sind.

Wir werden einige der häufig gebrauchten Module in dieser Bibliothek untersuchen. Sie können die vollständigen Einzelheiten zu allen Modulen der Python-Standardbibliothek in der (leider nur in Englisch verfügbaren) 'Python Library Reference' nachlesen, die ebenfalls ein Bestandteil Ihrer Python-Installation ist.

Das sys-Modul

Das `sys`-Modul enthält systemspezifische Funktionalität. Wir haben bereits gesehen, dass die Liste `sys.argv` die Kommandozeilenparameter enthält.

Kommandozeilenparameter

Beispiel 14.1. Der Gebrauch von sys.argv (cat.py [code/cat.py])

```
#!/usr/bin/python

import sys

def liesdatei(dateiname):
 '''Gib eine Datei auf der Standardausgabe aus.'''
 f = file(dateiname)
 while True:
  zeile = f.readline()
  if len(zeile) == 0:
   break
  print zeile, # beachten Sie das Komma
 f.close()

# das Skript beginnt hier
if len(sys.argv) < 2:
 print 'Es wurden keine Parameter uebergeben.'
 sys.exit()

if sys.argv[1].startswith('--'):
 option = sys.argv[1][2:]
 # hole sys.argv[1], aber ohne die ersten beiden Zeichen
 if option == 'version':
  print 'Version 1.2'
 elif option == 'hilfe':
  print '''\
Dieses Programm gibt Dateien auf der Standardausgabe aus.
Es kann eine beliebige Anzahl von Dateien angegeben werden.
```

```
Als Optionen koennen angegeben werden:
  --version : Gibt die Versionsnummer aus
  --hilfe   : Gibt diese Hilfe aus'''
 else:
  print 'Unbekannte Option.'
 sys.exit()
else:
 for dateiname in sys.argv[1:]:
  liesdatei(dateiname)
```

Ausgabe

```
$ python cat.py
Es wurden keine Parameter uebergeben.

$ python cat.py --hilfe
Dieses Programm gibt Dateien auf der Standardausgabe aus.
Es kann eine beliebige Anzahl von Dateien angegeben werden.
Als Optionen koennen angegeben werden:
  --version : Gibt die Versionsnummer aus
  --hilfe   : Gibt diese Hilfe aus

$ python cat.py --version
Version 1.2

$ python cat.py --unsinn
Unbekannte Option.

$ python cat.py gedicht.txt
Programmieren mit Elan
und die Arbeit wird getan,
willst du Spass haben daran:
        Nimm Python!
```

So funktioniert es

Dieses Programm versucht, den **cat**-Befehl zu imitieren, der Linux/Unix-Benutzern wohlbekannt ist (er entspricht **type** unter DOS/Windows). Sie geben einfach den Namen einiger Textdateien an, und der Befehl gibt sie auf der Standardausgabe aus.

Wenn ein Python-Programm gestartet ist, d.h. nicht im interaktiven Modus, dann gibt es stets mindestens einen Eintrag in der Liste `sys.argv`, nämlich den Namen des gerade laufenden Programms, der als `sys.argv[0]` verfügbar ist, da Python von 0 anfängt zu zählen. Weitere eventuell vorhandene Kommandozeilenparameter folgen auf diesen Eintrag.

Um das Programm benutzerfreundlich zu gestalten, haben wir bestimmte Optionen vorgesehen, die der Benutzer angeben kann, um mehr über das Programm zu lernen. Wir benutzen den ersten Parameter, um zu prüfen, ob unserem Programm irgendwelche Optionen übergeben worden sind. Wenn die Option `--version` benutzt wird, dann wird die Versionsnummer des Programms ausgegeben. Auf ähnliche Weise geben wir ein wenig Erklärung über das Programm aus, wenn die Option `--help` angegeben wird. Wir benutzen die Funktion `sys.exit`, um das laufende Programm zu beenden. Wie immer erhalten Sie mit `help(sys.exit)` weitere Einzelheiten hierzu.

Wenn keine Optionen angegeben wurden und dem Programm Dateinamen übergeben werden, dann gibt es einfach jede Zeile jeder Datei aus, eine nach der anderen, in der Reihenfolge, wie sie auf der Kommandozeile angegeben wurde.

Nebenbei bemerkt ist der Name **cat** eine Abkürzung für *concatenate*, was 'verketten' bedeutet und genau das bezeichnet, was dieses Programm tut - es kann eine Datei ausgeben, oder zwei oder mehr Dateien in der Ausgabe einander anfügen und so miteinander verketten.

Weiteres aus sys

Der String `sys.version` liefert Informationen über die Python-Version, die Sie installiert haben. Mit dem Tupel `sys.version_info` kann man auf einfache Weise Fallunterscheidungen für verschiedene Python-Versionen durchführen.

```
[swaroop@localhost code]$ python
>>> import sys
>>> sys.version
'2.3.4 (#1, Oct 26 2004, 16:42:40) \n[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)]
>>> sys.version_info
(2, 3, 4, 'final', 0)
```

Für erfahrene Programmierer sind weitere interessante Dinge im `sys`-Modul zu finden, wie `sys.stdin`, `sys.stdout` und `sys.stderr`, die den Datenströmen Standardeingabe, Standardausgabe bzw. Standardfehlerausgabe Ihres Programms entsprechen.

Das os-Modul

Das Modul `os` stellt allgemeine Betriebssystemfunktionalität zur Verfügung (`os` steht hierbei für **O**perating **S**ystem). Dieses Modul ist besonders wichtig, wenn Sie Ihr Programm plattformunabhängig gestalten wollen, d.h. es ermöglicht es, das Programm so zu schreiben, dass es sowohl unter Linux als auch unter Windows oder MacOS problemlos und ohne erforderliche Änderungen abläuft.

Im Folgenden sind einige der nützlicheren Teile des `os`-Moduls aufgelistet. Die meisten davon sind selbsterklärend.

- Der String `os.name` gibt an, auf welcher Plattform das Programm läuft, etwa `'nt'` für Windows oder `'posix'` für Linux/Unix.
- Die Funktion `os.getcwd()` liefert das aktuelle Arbeitsverzeichnis, d.h. den Pfad des Verzeichnisses, unter dem das aktuelle Python-Skript läuft.
- Die Funktionen `os.getenv()` und `os.putenv()` werden benutzt, um Umgebungsvariablen auszulesen bzw. zu setzen.
- Die Funktion `os.listdir()` gibt die Namen aller Dateien und Verzeichnisse in dem angegeben Verzeichnis zurück.
- Die Funktion `os.remove()` wird benutzt, um eine Datei zu löschen.
- Die Funktion `os.system()` wird benutzt, um einen Befehl auf der Kommandozeilenebene des Betriebssystems ablaufen zu lassen.
- Der String `os.linesep` ist die Zeilenendekennung, die auf der jeweiligen Plattform benutzt wird. Zum Beispiel verwendet Windows `'\r\n'`, Linux verwendet `'\n'` und Mac verwendet `'\r'`.

- Die Funktion `os.path.split()` gibt den Verzeichnisnamen und den Dateinamen zurück, die zu einem bestimmten Pfad gehören.

```
>>> os.path.split('/home/swaroop/byte/code/gedicht.txt')
('/home/swaroop/byte/code', 'gedicht.txt')
```

- Die Funktionen `os.path.isfile()` und `os.path.isdir()` überprüfen, ob der angegebene Pfad auf eine Datei bzw. ein Verzeichnis verweist. Entsprechend wird die Funktion `os.path.exists()` benutzt, um zu überprüfen, ob der angegebene Pfad wirklich existiert.

Sie können die Python-Standard-Dokumentation zu Rate ziehen, wenn Sie weitere Einzelheiten zu diesen Funktionen und Variablen erfahren möchten. Sie können außerdem `help(sys)` usw. verwenden.

Zusammenfassung

Wir haben ein wenig der Funktionalität der Module `sys` und `os` der Python-Standardbibliothek kennen gelernt. Sie sollten die Python-Standard-Dokumentation zu Rate ziehen, um mehr über diese und auch über andere Module zu herauszufinden.

Als Nächstes werden wir verschiedene Aspekte von Python behandeln, die unsere Python-Rundreise *vollständiger* machen.

Kapitel 15. Noch mehr Python

Wir haben hiermit den größten Teil der verschiedenen Aspekte von Python angesprochen, die Sie benutzen werden. In diesem Kapitel werden wir einige weitere Aspekte besprechen, die Ihre Python-Kenntnisse noch etwas *vollständiger* machen werden.

Besondere Methoden

Es gibt einige besondere Methoden, die in Klassen eine spezielle Bedeutung haben, so wie die Methoden `__init__` oder `__del__`, deren Bedeutung wir bereits kennen gelernt haben.

Allgemein werden spezielle Methoden verwendet, um eine bestimmte Funktionalität zu imitieren. Wenn Sie zum Beispiel die Indizierungs-Operation `x[key]` für Ihre Klasse benutzen wollen (als wenn Sie eine Liste oder ein Tupel wäre), dann implementieren Sie einfach die Methode `__getitem__()`, und haben dieses Problem damit gelöst. Wenn Sie darüber nachdenken, ist es genau das, was Python selber bei der Klasse `list` macht. Man spricht bei dieser Vorgehensweise auch vom *Überladen von Operatoren*.

Einige nützliche speziellen Methoden sind in der folgenden Tabelle aufgelistet. Wenn Sie alle speziellen Methoden wissen wollen, dann finden Sie diese große Liste im Python-Referenz-Handbuch.

Tabelle 15.1. Einige spezielle Methoden

Name	Erklärung
__init__(self, ...)	Diese Methode wird aufgerufen, gerade bevor das neu erzeugte Objekt zum Gebrauch zurückgegeben wird.
__del__(self)	Wird aufgerufen gerade bevor das Objekt zerstört wird. ,
__str__(self)	Wird aufgerufen, wenn man das Objekt mit der `print`-Anweisung ausgibt oder wenn `str()` benutzt wird.
__lt__(self, other)	Wird aufgerufen, wenn der Operator < für 'kleiner als' (*less than*) benutzt wird. Entsprechend gibt es spezielle Methoden für alle möglichen Operatoren (+, >, usw.).
__getitem__(self, key)	Wird aufgerufen, wenn der Indizierungs-Operator `x[key]` verwendet wird.
__len__(self)	Wird aufgerufen, wenn die eingebaute `len()`-Funktion auf das Sequenz-Objekt angewendet wird.

Einzelanweisungsblöcke

Inzwischen sollten Sie gut verstanden haben, dass jeder Anweisungsblock vom Rest des Programms durch seine eigene Einrückungstiefe abgesetzt wird. Dies trifft zwar meistens zu, ist aber nicht 100% genau. Wenn Ihr Anweisungsblock nur aus einer einzigen Anweisung besteht, dann können Sie ihn in der gleichen Zeile von z.B. einer Bedingungs- oder Schleifenanweisung angeben. Das folgende Beispiel sollte dies deutlich machen:

```
>>> schalter = True
```

```
>>> if schalter: print 'Ja'
...
Ja
```

Wie man sieht, wird die einzelne Anweisung nicht in einen separaten Block, sondern direkt hinter die Bedingungsanweisung in die gleiche Zeile geschrieben. Man kann dies zwar verwenden, um sein Programm *kleiner* zu machen, aber ich empfehle dennoch **stark**, dass Sie diese abkürzende Schreibweise nicht benutzen, außer zur Fehlerprüfung usw. Ein Hauptgrund dafür ist, dass es sehr viel einfacher ist, eine weitere Anweisung zu ergänzen, wenn man die Anweisungen sauber separat eingerückt hinschreibt.

Beachten Sie auch, dass der Python-Interpreter Ihnen bei der Eingabe von Anweisungen hilft, wenn er im interaktiven Modus verwendet wird, indem er die Eingabeaufforderung je nach Situation ändert. Im obigen Fall wechselt die Eingabeaufforderung zu `...`, nachdem Sie das Schlüsselwort `if` eingegeben haben, um anzuzeigen, dass die Anweisung noch nicht vollständig ist. Nachdem wir die Anweisung auf diese Weise vervollständigt haben, schließen wir die Eingabe mit **Enter** an, um zu bestätigen, dass die Anweisung vollständig ist. Danach führt Python die gesamte Anweisung aus und zeigt wieder die Eingabeaufforderung für die nächste Eingabe an.

Listenkomprehension

Listenkomprehension (Listenbeschreibung) ist ein Fachbegriff für eine sehr nützliche Funktionalität von Python, die es erlaubt, aus einer existierenden Liste eine neue Liste abzuleiten. Nehmen Sie zum Beispiel an, Sie haben eine Liste von Zahlen und wollen eine entsprechende Liste erzeugen, die aus den mit 2 multiplizierten Zahlen besteht, aber nur, wenn die jeweilige Zahl selbst kleiner als 2 ist. Eine Listenkomprehension ist in solchen Fällen ideal.

Gebrauch von Listenkomprehension

Beispiel 15.1. Gebrauch von Listenkomprehension (listenkomprehension.py [code/listenkomprehension.py])

```
#!/usr/bin/python

listeeins = [2, 3, 4]
listezwei = [2*i for i in listeeins if i > 2]
print listezwei
```

Ausgabe

```
$ python listenkomprehension.py
[6, 8]
```

So funktioniert es

Wir leiten hier von der vorhandenen Liste `listeeins` eine neue Liste `listezwei` ab, indem wir die Manipulationsvorschrift angeben (`2*i`), die an der Liste vorgenommen werden soll, wenn eine

bestimmte Bedingung (`if i > 2`) erfüllt ist. Beachten Sie, dass die ursprüngliche Liste davon unberührt bleibt. Oftmals, wenn wir Schleifen verwenden, um alle Elemente einer Liste zu durchlaufen, können wir das Gleiche auf präzisere, kompaktere und deutlichere Weise durch eine Listenkomprehension ausdrücken.

Übergabe von Tupeln und Dictionaries in Funktionen

Es gibt eine besondere Methode, um einer Funktion Parameter als Tupel oder Dictionary zu übergeben, nämlich indem man den Parametern * bzw. ** voranstellt. Dies ist nützlich, wenn eine Funktion eine beliebige Zahl von Parametern entgegennehmen soll.

Beispiel: potenzsumme.py [code/potenzsumme.py]

```
>>> def potenzsumme(potenz, *parameter):
...     '''Gibt die Summe der angegebenen Potenz aller Parameter zurueck.'''
...     summe = 0
...     for i in parameter:
...             summe += pow(i, potenz)
...     return summe
...
>>> potenzsumme(2, 3, 4) # 3^2 + 4^2
25

>>> potenzsumme(2, 10) # 10^2
100
```

Aufgrund des vorangestellten * bei der Variable `parameter`, werden alle zusätzlichen Parameter, die der Funktion übergeben werden, als ein Tupel in `parameter` gespeichert. Wenn stattdessen ein ** vorangestellt worden wäre, dann wären die zusätzlichen Parameter als Schlüssel/Wert-Paare eines Dictionaries betrachtet worden.

Der lambda-Operator

Eine `lambda`-Anweisung wird verwendet, wenn man neue Funktions-Objekte erzeugen und zur Laufzeit zurückgeben möchte.

Gebrauch des lambda-Operators

Beispiel 15.2. Gebrauch des lambda-Operators (lambda.py [code/lambda.py])

```
#!/usr/bin/python

def erzeuge_wiederholer(n):
 return lambda s: s * n

verdoppler = erzeuge_wiederholer(2)

print verdoppler('wort')
print verdoppler(5)
```

Ausgabe

```
$ python lambda.py
wortwort
10
```

So funktioniert es

Wir benutzen hier eine Funktion `erzeuge_wiederholer`, um ein neues Funktions-Objekt zur Laufzeit zu erzeugen und es zurückzugeben. Um dieses Funktions-Objekt zu erzeugen, wird eine `lambda`-Anweisung verwendet. Der `lambda`-Operator nimmt dabei einen Parameter entgegen, gefolgt von einem einzelnen Ausdruck, der dann zum Rumpf der Funktion wird und von der neuen Funktion ausgewertet und zurückgegeben wird. Beachten Sie, dass innerhalb eines `lambda`-Operators keine Anweisung verwendet werden darf, nicht einmal eine `print`-Anweisung, sondern dass hier nur Ausdrücke erlaubt sind.

Die exec- und eval-Anweisungen

Die `exec`-Anweisung wird benutzt, um Python-Anweisungen auszuführen, die in einem String oder in einer Datei gespeichert sind. Zum Beispiel können wir zur Laufzeit einen String erzeugen, der Python-Code enthält, und diese Befehle dann mittels der `exec`-Anweisung ausführen. Unten sehen Sie ein einfaches Beispiel hierfür.

```
>>> exec 'print "Hallo Welt"'
Hallo Welt
```

Die `eval`-Anweisung wird verwendet, um gültige Python-Ausdrücke auszuwerten, die in einem String gespeichert sind. Unten sehen Sie wieder ein einfaches Beispiel.

```
>>> eval('2*3')
6
```

Die assert-Anweisung

Die `assert`-Anweisung wird benutzt, um sicherzustellen, dass eine bestimmte Voraussetzung, die Sie machen, wirklich erfüllt ist. Zum Beispiel könnten Sie von der Voraussetzung ausgehen, dass sich in einer bestimmten Liste, die Sie benutzen, mindestens ein Element befindet. Wenn Sie überprüfen wollen, ob dies wirklich stimmt und andernfalls einen Fehler ausgeben wollen, dann ist eine `assert`-Anweisung hierfür ideal. Wenn die Überprüfung mittels der `assert`-Anweisung fehlschlägt, dann wird ein `AssertionError` ausgelöst.

```
>>> meineliste = ['element']
>>> assert len(meineliste) >= 1
>>> meineliste.pop()
'element'
>>> assert len(meineliste) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
```

Die repr-Function

Die `repr`-Funktion wird benutzt um eine 'kanonische' Darstellung eines Objekts als String zu erhalten. Das Gleiche kann man mit umgekehrten einfachen Anführungszeichen erreichen (also nicht dem normalen einfachen Anführungszeichen, das mit dem Apostroph identisch ist, und das man für normale Strings in Python verwendet, sondern das schräg rückwärts gerichtete Anführungszeichen, das dem französischen Accent grave entspricht). Beachten Sie, dass meistens `eval(repr(objekt)) == objekt` gilt.

```
>>> liste = []
>>> liste.append('element')
>>> `liste`
"['element']"
>>> repr(liste)
"['element']"
```

Die Funktion `repr` bzw. die umgekehrten Anführungsstriche werden grundsätzlich dafür benutzt, eine Darstellung eines Objekts in druckbarer reiner Textform zu erhalten. Sie können bestimmen, was Ihre Objekte mit `repr` zurückgeben, indem Sie die für die zugehörige Klasse die Methode `__repr__` definieren.

Zusammenfassung

Wir haben in diesem Kapitel noch mehr Funktionalität von Python behandelt, aber Sie können sich sicher sein, dass die gesamte Funktionalität damit noch nicht abgehandelt ist. Aber wir haben inzwischen die meisten Dinge behandelt, die Sie jemals in der Praxis benutzen werden. Es sollte für Sie als Ausgangspunkt ausreichen, um alle möglichen Programme zu schreiben.

Im nächsten Kapitel sprechen wir darüber, was es in Python alles noch weiter zu entdecken gibt.

Kapitel 16. Was kommt als Nächstes?

Wenn Sie dieses Buch bis hierher gründlich gelesen haben, und sich darin geübt haben, viele Programme zu schreiben, dann müssten Sie sich inzwischen mit Python angefreundet haben und damit vertraut geworden sein. Sie haben vermutlich einige Python-Programme geschrieben, um verschiedene Dinge auszuprobieren, und um Ihre Python-Kenntnisse in der Praxis zu üben. Wenn Sie dies noch nicht getan haben, dann sollten Sie es noch tun. Jetzt stellt sich die Frage: "Was kommt als Nächstes?"

Ich möchte vorschlagen, dass Sie sich einmal an dieses Problem heran wagen: Schreiben Sie Ihr eigenes kommandozeilenbasiertes *Adressbuch*-Programm, mit dem Sie Ihre Kontaktadressen wie Freunde, Familie, Kollegen, und die zugehörigen Informationen wie E-Mail-Adresse und/oder Telefonnummer hinzufügen, verändern, löschen oder durchsuchen können. Die Daten müssen für den späteren Zugriff gespeichert werden.

Dies ist einigermaßen einfach, wenn Sie darüber vor dem Hintergrund der Dinge nachdenken, die wir bisher schon besprochen haben. Wenn Sie dennoch Hinweise wünschen, wie Sie dabei vorgehen können, dann ist hier ein Tipp für Sie.

Tipp (Sie sollten ihn eigentlich nicht lesen). Erzeugen Sie eine Klasse, um die Informationen der Person zu speichern. Benutzen Sie ein Dictionary, um die Personen-Objekte mit ihrem jeweiligen Namen als Schlüssel zu speichern. Benutzen Sie das `pickle`-Modul, um die Objekte persistent auf Ihrer Festplatte zu speichern. Benutzen Sie die eingebauten Dictionary-Methoden, um Personen hinzuzufügen, zu löschen und zu ändern.

Wenn Sie dies einmal geschafft haben, können Sie behaupten, ein Python-Programmierer zu sein. Danach sollten Sie mir sofort eine Mail schicken, um sich für dieses großartige Buch zu bedanken ;-) Dieser Schritt ist optional, aber empfohlen.

Hier sind einige weitere Möglichkeiten für Ihre weitere Python-Entdeckungsreise:

Software mit grafischer Oberfläche

GUI-Bibliotheken für Python - Sie brauchen diese, um Ihre eigenen grafischen Benutzeroberflächen (GUIs) zu programmieren. Sie können sich Ihr eigenes IrfanView oder Kuickshow oder irgendetwas in der Art schreiben, indem Sie GUI-Bibliotheken mit ihrer jeweiligen Python-Anbindung benutzen. Eine solche Anbindung ermöglicht es Ihnen, Programme in Python zu schreiben, in denen Sie auf diese Bibliotheken zurückgreifen, die ihrerseits in C oder C++ oder anderen Sprachen geschrieben sind.

Es gibt eine große Auswahl an GUI-Bibliotheken für Python:

- **PyQt.** Dies ist die Python-Anbindung für das Qt-Toolkit, das die Grundlage ist, auf der KDE aufgebaut wurde. Qt ist äußerst einfach zu benutzen und sehr mächtig, besonders aufgrund des Qt-Designers und der hervorragenden Qt-Dokumentation. Sie können es unter Linux kostenlos benutzen, aber Sie müssen es bezahlen, wenn Sie es unter Windows benutzen wollen. PyQt ist frei, wenn Sie freie Software (gemäß der GPL) unter Linux/Unix erstellen, und muss bezahlt werden, wenn Sie proprietäre Software erstellen wollen. Eine gute (englische) Informationsquelle zu PyQt ist "GUI Programming with Python: Qt Edition" [http://www.opendocs.org/pyqt/]. Weitere Einzelheiten finden Sie auf der offiziellen PyQt-Website [http://www.riverbankcomputing.co.uk/pyqt/].

- **PyGTK.** Dies ist die Python-Anbindung für das GTK+-Toolkit, das die Grundlage ist, auf der GNOME aufgebaut wurde. GTK+ hat im Gebrauch viele Eigenheiten; wenn Sie aber einmal damit vertraut sind, können Sie schnell GUI-Anwendungen erstellen. Das Werkzeug Glade zum entwerfen der grafischen Schnittstelle ist hierbei unverzichtbar. Die Dokumentation müsste noch verbessert werden. GTK+ läuft gut unter Linux, aber die Portierung auf Windows ist unvollständig. Sie können mit GTK+ sowohl freie als auch proprietäre Software entwickeln. Weitere Einzelheiten finden Sie auf der offiziellen PyGTK-Website [http://www.pygtk.org].

- **wxPython.** Dies ist die Python-Anbindung für das wxWidgets-Toolkit. Mit wxPython ist eine gewisse Lernkurve verbunden. Es ist jedoch sehr portabel und läuft unter Linux, Windows, Mac und

sogar auf Embedded-Systemen. Für wxPython sind viele IDEs verfügbar, darunter DrPython [http://drpython.sourceforge.net], SPE (Stani's Python Editor) [http://spe.pycs.net], das auch einen GUI-Designer beinhaltet, sowie der GUI-Builder wxGlade [http://wxglade.sourceforge.net]. Sie können mit wxPython sowohl freie als auch proprietäre Software entwickeln. Weitere Einzelheiten finden Sie auf der offiziellen wxPython-Website [http://www.wxpython.org].

- **TkInter.** Dies ist das älteste existierende GUI-Toolkit für Python. Wenn Sie IDLE benutzt haben, dann haben Sie bereits gesehen, wie ein TkInter-Programm aussehen kann. Die Dokumentation für TkInter in der PythonWare [http://www.pythonware.com/library/tkinter/introduction/]-Bibliothek ist umfangreich. TkInter ist portierbar und läuft sowohl unter Linux/Unix als auch unter Windows. Ein wichtiger Vorteil von TkInter ist, dass es Bestandteil der Standard-Distribution von Python ist.

- Eine weitere Auswahl von grafischen Oberflächen für Python finden Sie auf der Wiki-Seite über GUI-Programmierung [http://www.python.org/cgi-bin/moinmoin/GuiProgramming] auf Python.org.

Zusammenfassung GUI-Tools

Leider gibt es kein Standard-GUI-Tool für Python. Ich schlage vor, dass Sie eines der obigen Toolkits auswählen, je nach Ihrer Situation. Der erste Faktor, den Sie dabei bedenken müssen, ist, ob Sie bereit sind, für das GUI-Tool Geld auszugeben. Der zweite Faktor ist, ob Sie Ihr Programm für Linux oder für Windows oder für beides entwickeln wollen. Der dritte Faktor ist, ob Sie als Linux-Benutzer eher eine Präferenz für KDE oder für GNOME haben.

Zukünftige Kapitel

Ich erwäge, für dieses Buch ein oder zwei Kapitel über GUI-Programmierung zu schreiben. Ich werde wahrscheinlich hierbei wxPython als Toolkit auswählen. Wenn Sie Ihre Ansicht über dieses Thema äußern möchten, dann können Sie dies auf der Byte-of-Python-Mailingliste [http://lists.ibiblio.org/mailman/listinfo/byte-of-python] tun, wo Leser mit mir diskutieren, welche Verbesserungen an dem Buch vorgenommen werden können.

Entdecken Sie mehr

- Die **Python-Standardbibliothek** ist sehr umfangreich. In den meisten Fällen hält diese Bibliothek wahrscheinlich schon das bereit, wonach Sie vielleicht suchen. Für diesen Ansatz von Python, dem Programmierer möglichst viel Funktionalität von Haus aus bereitzustellen, wird oft der Slogan "Batterien im Lieferumfang" (*batteries included*) verwendet. Ich empfehle Ihnen auch sehr, dass Sie sich die Standard-Dokumentation von Python [http://docs.python.org] anschauen, bevor Sie anfangen, größere Programme zu schreiben. Auf der Website Python.org finden Sie übrigens auch Verweise auf Artikel und Anleitungen zu Python in deutscher Sprache [http://www.python.org/doc/NonEnglish.html#german].

- Python.org [http://www.python.org] - die offizielle Website der Programmiersprache Python. Hier finden Sie die jeweils aktuelle Version von Python. Es gibt hier auch verschiedene Mailinglisten, in denen aktiv über verschiedene Aspekte von Python diskutiert wird.

- **comp.lang.python** ist das Usenet-Forum, in dem über diese Sprache diskutiert wird. Sie können Ihre Zweifel und Anfragen in diesem Forum äußern. Sie können auf das Forum über Google Groups [http://groups.google.com/groups?hl=de&ie=UTF-8&group=comp.lang.python] online zugreifen, oder die Mailingliste [http://mail.python.org/mailman/listinfo/python-list] abbonieren, die in dieses Forum gespiegelt wird.

- Das Python-Kochbuch [http://aspn.activestate.com/ASPN/Python/Cookbook/] ist eine äußerst wertvolle Sammlung von Rezepten oder Tipps, wie man bestimmte Problemarten mit Python lösen kann. Es ist eine der Websites, die man als Python-Benutzer unbedingt gelesen haben muss.

- Charming Python [http://gnosis.cx/publish/tech_index_cp.html] ist eine exzellente Serie von Artikeln rund um Python von David Mertz.

- Dive Into Python [http://www.diveintopython.org/] ist ein sehr gutes Buch für erfahrene Python-Programmierer. Wenn Sie das Buch, das Sie gerade lesen, gründlich durchgelesen haben, empfehle ich Ihnen, als Nächstes "Dive Into Python" zu lesen. Es behandelt eine Reihe von Themen, darunter auch die Verarbeitung von XML, Test von Programmeinheiten (*Unit Testing*), sowie funktionale Programmierung.

- Jython [http://www.jython.org] ist eine Implementierung des Python-Interpreters in der Sprache Java. Das bedeutet, dass Sie Programme in Python schreiben können, und dabei auch auf die Java-Bibliotheken zugreifen können! Jython ist eine stabile und ausgereifte Software. Wenn Sie auch ein Java-Programmierer sind, empfehle ich Ihnen sehr, Jython einmal auszuprobieren.

- IronPython [http://www.ironpython.com] ist eine Implementierung des Python-Interpreters in der Sprache C# und läuft auf den .NET/Mono/DotGNU-Plattformen. Das bedeutet, dass Sie Programme in Python schreiben und dabei auf die .NET-Bibliotheken und andere Bibliotheken zugreifen können, die von diesen drei Plattformen zur Verfügung gestellt werden! IronPython ist noch Prä-Alpha-Software und bisher nur zum Experimentieren geeignet. Jim Hugunin, der IronPython geschrieben hat, ist inzwischen ein Microsoft-Mitarbeiter und wird zukünftig an einer vollständigen Version von IronPython arbeiten.

- Lython [http://www.caddr.com/code/lython/] ist ein Lisp-Frontend für die Sprache Python. Es ähnelt Common Lisp und kompiliert direkt in Python-Bytecode, was bedeutet, dass es mit normalem Python-Code zusammen arbeitet.

- Es gibt eine Menge weiteres Material über und für Python im Internet. Einige der interessanten Webseiten sind Daily Python-URL [http://www.pythonware.com/daily/], wo man über die letzten Trends rund um Python auf dem Laufenden gehalten wird, Vaults of Parnassus [http://www.vex.net/parnassus/], ONLamp.com Python DevCenter [http://www.onlamp.com/python/], dirtSimple.org [http://dirtsimple.org/], Python Notes [http://pythonnotes.blogspot.com/], und es gibt noch sehr viele weitere interessante Python-Webseiten zu entdecken.

Zusammenfassung

Wir sind nun am Ende des Buchs angelangt, aber, wie man sagt, *dies ist der Anfang vom Ende*! Sie sind nun ein begeisterter und eifriger Python-Anwender, und brennen bestimmt darauf, viele Probleme mit Python zu lösen. Sie können damit anfangen, Ihren Computer zu automatisieren, alle möglichen früher unvorstellbaren Dinge zu tun, oder Ihre eigenen Spiele zu schreiben und noch viel mehr. Also, worauf warten Sie noch?

Anhang A. Freie/Libre und Open-Source Software (FLOSS)

FLOSS basiert auf der Idee einer Gemeinschaft, die ihrerseits auf der Idee von gegenseitigem Austausch basiert, insbesondere dem Austausch von Wissen. FLOSS ist frei für die Nutzung, Änderung und Weiterverteilung.

Wenn Sie dieses Buch bereits gelesen haben, dann sind Sie auch schon mit FLOSS vertraut, denn Sie haben die ganze Zeit **Python** benutzt!

Wenn Sie mehr über FLOSS erfahren wollen, können Sie die folgende Liste erkunden. Ich habe einige große FLOSS-Projekte aufgelistet, und auch einige FLOSS-Projekte, die plattformunabhängig sind (d.h. unter Linux, Windows usw. laufen), so dass Sie diese Software ausprobieren können, ohne sofort zu Linux wechseln zu müssen, *obwohl Sie dies schließlich tun werden ;-)*

- **Linux.** Dies ist ein FLOSS-Betriebssystem, das langsam von der ganzen Welt begeistert angenommen wird! Das Projekt wurde von Linus Torvalds in seiner Studentenzeit begonnen. Inzwischen steht es in Konkurrenz mit Microsoft Windows. Der aktuelle Kernel 2.6 stellt einen größeren Durchbruch hinsichtlich Geschwindigkeit, Stabilität und Skalierbarkeit dar. [Linux Kernel [http://www.kernel.org]]

- **Knoppix.** Dies ist eine Linux-Distribution die allein von der CD startet! Es ist keine Installation notwendig - Sie können einfach Ihren Computer neu starten, die CD ins Laufwerk stecken, und damit anfangen, eine vollwertige Linux-Distribution zu benutzen! Sie können all die verschiedenen FLOSS-Projekte benutzen, die mit einer Standard-Linux-Distribution ausgeliefert werden, etwa Python-Programme laufen lassen, C-Programme kompilieren, Filme anschauen usw. Danach können Sie den Computer wieder neu starten, die CD entfernen und Ihr bestehendes Betriebssystem benutzen, als wenn nichts geschehen wäre. [Knoppix [http://www.knopper.net]]

- **Fedora.** Diese Distribution ist ein von einer Fangemeinde betriebenes Projekt, das von Red Hat gesponsert wird und eine der populärsten Linux-Distributionen darstellt. Sie enthält den Linux-Kernel, die Benutzeroberflächen KDE, GNOME und XFCE, und die Unmenge an verfügbarer FLOSS, und all dies in einer einfach zu benutzenden und einfach zu installierenden Weise.

  Wenn Sie ein völliger Linux-Neuling sind, dann würde ich Ihnen empfehlen, **Mandrake Linux** auszuprobieren. Das neulich freigegebene Mandrake 10.1 ist einfach fantastisch.

- **OpenOffice.org.** Dies ist eine hervorragende Office-Suite, die auf der StarOffice-Software von Sun Microsystems basiert. Zu OpenOffice gehören als Komponenten unter anderem eine Textverarbeitung, ein Präsentationsprogramm, eine Tabellenkalkulation und ein Zeichenprogramm. Es kann sogar MS Word und MS PowerPoint-Dateien problemlos öffnen. Es läuft auf fast allen Plattformen. Die nächste Version OpenOffice 2.0 bietet weitere einschneidende Verbesserungen. [OpenOffice [http://www.openoffice.org]]

- **Mozilla Firefox.** Dies ist **der** Web-Browser der nächsten Generation, von dem vorhergesagt wird, dass er in ein paar Jahren den Internet Explorer schlagen wird (nur hinsichtlich des Marktanteils ;-). Er ist extrem schnell und hat besonders wegen seiner durchdachten und beeindruckenden Möglichkeiten großen Anklang gefunden. Das Konzept der Extensions (Erweiterungen) ermöglicht es, alle Arten von Funktionalität hinzuzufügen.

  Sein Begleitprodukt Thunderbird ist ein hervorragender E-Mail-Client, der das Lesen von E-Mails zum Kinderspiel macht. [Mozilla Firefox [http://www.mozilla.org/products/firefox], Mozilla Thunderbird [http://www.mozilla.org/products/thunderbird]]

- **Mono.** Dies ist eine Open-Source-Implementation der .NET-Plattform von Microsoft. Sie erlaubt es, .NET-Anwendungen unter Linux, Windows, FreeBSD, Mac OS und vielen weiteren Plattformen zu erzeugen und laufen zu lassen. Mono implementiert die ECMA-Standards für die CLI (*Common*

Language Infrastructure) und C#, die Microsoft, Intel und HP zur Standardisierung eingereicht haben und die inzwischen offene Standards geworden sind. Dies ist ein Schritt in Richtung ISO-Standardisierung dieser Dinge.

Zur Zeit gibt es einen volltändigen **mcs** (Mono C# compiler) (der selber in C# geschrieben ist!), eine ASP.NET-Implementation mit allem, was dazu gehört, viele ADO.NET-Provider für Datenbanken und viele weitere Komponenten, die ständig verbessert und ergänzt werden. [Mono [http://www.mono-project.com], ECMA [http://www.ecma-international.org], Microsoft .NET [http://www.microsoft.com/net]]

- **Apache Webserver.** Dies ist der populäre Open-Source-Webserver. In der Tat ist er **der** populärste Webserver auf dem Planeten! Auf ihm laufen 60% aller Webseiten irgendwo auf der Welt. Das stimmt wirklich - Apache betreibt mehr Websites als die gesamte Konkurrenz (einschließlich Microsoft IIS) zusammen. [Apache [http://www.apache.org]]

- **MySQL.** Dies ist ein äußerst populärer Open-Source-Datenbankserver. Er ist besonders für seine extreme Geschwindigkeit bekannt. In den neusten Versionen werden ihm weitere Fähigkeiten verliehen. [MySQL [http://www.mysql.com]]

- **MPlayer.** Dies ist ein Video-Player, der alles abspielen kann, von DivX über MP3 und Ogg bis hin zu VCDs und DVDs... Wer sagt, dass Open-Source keinen Spaß macht? ;-) [MPlayer [http://www.mplayerhq.hu]]

- **Movix.** Dies ist eine Linux-Distribution, die auf Knoppix basiert und direkt von CD läuft, doch ist sie speziell dafür entworfen worden, Filme abzuspielen! Sie können Movix-CDs erzeugen, die einfach bootbare CDs sind, und wenn Sie Ihren Computer neu starten und die CD einlegen, dann startet der Film ganz von selber! Sie brauchen nicht einmal eine Festplatte, um einen Film mit Movix anzuschauen. [Movix [http://movix.sourceforge.net]]

Diese Liste war nur dafür gedacht, Ihnen in Kürze eine Vorstellung davon zu geben, was alles mit FLOSS möglich ist - aber es gibt noch viel mehr hervorragende FLOSS-Projekte in der Welt, etwa die Sprache Perl, die Sprache PHP, das Web-Content-Management-System Drupal, den PostgreSQL-Datenbankserver, das Autorennspiel TORCS, die IDE KDevelop, die IDE Anjuta, den Movie-Player Xine, den Editor VIM, den Editor Quanta+, den Audio-Player XMMS, das Bildverarbeitungsprogramm GIMP, ... diese Liste könnte endlos fortgesetzt werden.

Besuchen Sie die folgenden Websites, um weitere Informationen über FLOSS zu erhalten:

- SourceForge [http://www.sourceforge.net]

- FreshMeat [http://www.freshmeat.net]

- KDE [http://www.kde.org]

- GNOME [http://www.gnome.org]

Um die letzen Neuigkeiten in der FLOSS-Welt zu erfahren, sollten Sie auf folgenden Websites nachschauen:

- OSNews [http://www.osnews.com]

- LinuxToday [http://www.linuxtoday.com]

- NewsForge [http://www.newsforge.com]

- SwaroopCH's Blog [http://www.swaroopch.info]

Beginnen Sie also damit, die unermessliche, freie und offene Welt von FLOSS zu erforschen!

Anhang B. Zum Buch und seinem Autor

Schlusswort

Fast alle Software, die ich zur Erstellung dieses Buchs benutzt habe, ist *freie Open-Source-Software*. Für den ersten Entwurf des Buchs hatte ich die Linux-Distribution Red Hat 9.0 als Ausgangsbasis meiner Softwareinstallation benutzt, und für diesen sechsten Entwurf benutze ich nun Fedora Core 3, das ebenfalls auf Red Hat basiert.

Zunächst hatte ich KWord benutzt, um das Buch zu schreiben (wie in der Geschichtsstunde im Vorwort erläutert). Später habe ich zu DocBook XML mit Kate als Editor gewechselt, fand es aber zu mühsam. Daher habe ich zu OpenOffice gewechselt, das einfach hervorragend war, weil es einem Kontrolle über die Formatierung gab und die Generierung von PDF ermöglichte. Die Generierung von HTML war jedoch mangelhaft. Schließlich entdeckte ich XEmacs und habe das Buch (noch einmal) von Anfang an in DocBook XML neu geschrieben, nachdem ich entschieden hatte, dass dieses Format die langfristige Lösung war. In diesem sechsten Entwurf habe ich entschieden, Quanta+ als Editor zu verwenden.

Es werden die Standard-XSL-Stylesheets eingesetzt, wie sie Fedora Core 3 beiliegen. Es werden auch die Standard-Zeichensätze benutzt. Ich habe aber eine CSS-Datei geschrieben, um den HTML-Seiten etwas Farbe und Stil hinzuzufügen. Ich habe auch ein simples lexikalisches Analyseprogramm geschrieben, natürlich in Python, das in den Programmlistings automatisch Syntax-Hervorhebungen hinzufügt.

Über den Autor

C. H. Swaroop liebt seine Arbeit als ein Softwareentwickler für Yahoo! in der indischen Stadt Bangalore. Zu seinen technischen Interessengebieten gehören FLOSS (Free/Libre Open Source Software) wie Linux, DotGNU, Qt und MySQL, großartige Programmiersprachen wie Python und C#, in der Freizeit Dinge wie dieses Buch und alle mögliche Software zu schreiben, und auch sein Blog zu schreiben. Zu seinen weiteren Interessen gehören Kaffee, Romane von Robert Ludlum, Trekking und Politik.

Wenn Sie immer noch mehr über diesen Typ herausfinden möchten, schauen Sie sich sein Blog unter www.swaroopch.info [http://www.swaroopch.info] an.

Anhang C. Versionsgeschichte

Versionsgeschichte des englischen Originals

Versionsgeschichte
Version 1.20 13.01.2005
Mit Quanta+ unter FC3 komplett neu geschrieben, mit vielen Korrekturen und Aktualisierungen. Viele neue Beispiele. DocBook-Setup von Grund auf neu geschrieben.
Version 1.15 28.03.2004
Kleinere Überarbeitungen
Version 1.12 16.03.2004
Ergänzungen und Korrekturen.
Version 1.10 09.03.2004
Weitere Tippfehler korrigiert, dank vieler begeisterter und hilfreicher Leser.
Version 1.00 08.03.2004
Nach einer riesigen Menge von Rückmeldungen und Vorschlägen von Lesern habe ich den Inhalt erheblich überarbeitet und dabei Tippfehler korrigiert.
Version 0.99 22.02.2004
Ein neues Kapitel über Module hinzugefügt. Einzelheiten über eine variable Anzahl von Argumenten in Funktionen ergänzt.
Version 0.98 16.02.2004
Ein Python-Skript und ein CSS-Stylesheet geschrieben, um die XHTML-Ausgabe zu verbessern, samt einem groben, aber funktionalen lexikalischen Analyseprogramm zur automatischen VIM-ähnlichen Syntax-Hervorhebung in den Programmlistings.
Version 0.97 13.02.2004
Ein weiterer neu geschriebener Entwurf, (wieder) in DocBook-XML. Das Buch ist jetzt viel besser - es ist verständlicher und lesbarer.
Version 0.93 25.01.2004
Ausführungen zu IDLE und weitere Windows-spezifische Dinge hinzugefügt.
Version 0.92 05.01.2004
Änderungen an einigen Beispielen.
Version 0.91 30.12.2003
Tippfehler korrigiert. Viele Themen improvisiert.
Version 0.90 18.12.2003
Zwei weitere Kapitel hinzugefügt. OpenOffice-Format mit Berichtigungen.
Version 0.60 21.11.2003
Völlig neu geschrieben und erweitert.
Version 0.20 20.11.2003
Einige Tippfehler und inhaltliche Fehler korrigiert.
Version 0.15 20.11.2003
Nach DocBook-XML konvertiert.
Version 0.10 14.11.2003
Erster Entwurf mit KWord.

Ergänzungen in der deutschen Version

Für die deutsche Übersetzung wurden die folgenden Änderungen vorgenommen, die über die reine Übersetzung des englischen Originals in der Version 1.20 hinausgehen:

- Nennung der Übersetzer und Hinweis auf BerliOS bei den Metainformation.
- Hinweis auf den Status der Übersetzung und auf das Übersetzungs-Projekt im Vorwort.
- Anmerkung für deutsche Anwender in Kapitel 3.
- Sehr wenige kleinere inhaltliche Ergänzungen und Berichtigungen in anderen Kapiteln.

Zeitstempel

Dieses Dokument wurde am 20.10.2013 um 14:42 Uhr erzeugt.