Xpert.press

Mark Pilgrim
Florian Wollenschein

Python 3
Intensivkurs

Springer

Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Mark Pilgrim

Python 3 – Intensivkurs

Projekte erfolgreich realisieren

Übersetzt aus dem Amerikanischen von
Florian Wollenschein

Springer

Mark Pilgrim
101 Forestcrest Court
Apex NC 27502
USA
mark@diveintomark.org

Übersetzer
Florian Wollenschein
Wertheimer Straße 21
97828 Marktheidenfeld
Deutschland
fw@florianwollenschein.de

Übersetzung aus dem Amerikanischen mit freundlicher Genehmigung des Autors. Titel der amerikanischen Originalausgabe: Dive into Python 3, Apress 2009.

ISBN 978-3-642-04376-5 e-ISBN 978-3-642-04377-2
DOI 10.1007/978-3-642-04377-2
Springer Heidelberg Dordrecht London New York

Xpert.press ISSN 1439-5428

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Einbandentwurf: KünkelLopka GmbH, Heidelberg

Gedruckt auf säurefreiem Papier

Springer ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.com)

Vorwort

Geschichte einer Übersetzung

Es begann im April 2009. Ich hatte gerade nichts Wichtiges zu tun, da kam mir die Idee, *Dive Into Python* (das englische Original der älteren Version dieses Buches) ins Deutsche zu übersetzen. Erfahrung im Übersetzen und in der Programmierung hatte ich über die letzten Jahre gesammelt. Probleme sollte es also nicht geben. Doch: vielleicht eins. Ich wusste nicht, ob *DIP,* wie ich es mittlerweile nenne, bereits ins Deutsche übersetzt war (immerhin hatte es schon über fünf Jahre auf dem Buckel). Ein Besuch der Homepage brachte aber schnell Klarheit und es stellte sich heraus, dass es bisher keine deutsche Übersetzung gab. Sodann beschloss ich frohen Mutes die Idee in die Tat umzusetzen und schrieb Mark Pilgrim eine E-Mail mit einem Hinweis darauf. Einen Tag (es können auch zwei gewesen sein) später antwortete er auch schon mit der Mitteilung, dass bereits *Dive Into Python 3* in Arbeit sei und ich doch vielleicht lieber das übersetzen solle, da *DIP* veraltet sei und in nächster Zeit auch nicht aktualisiert werde. Gut, dachte ich mir, dann eben *DIP 3*! Je aktueller, desto besser.

Ich begann sofort mit der Übersetzung.

Am 18. Mai 2009 stöberte ich ein wenig im Internet und stieß eher zufällig auf das Impressum der amerikanischen Ausgabe von *DIP*, die bei *Apress* erschienen war. *Springer-Verlag, Heidelberg, Germany* fiel mir sofort ins Auge. Offensichtlich gehörte Apress also zum deutschen Springer-Verlag. Mir kam ein Gedanke, der sich im Nachhinein als nicht so abwegig herausstellte, wie ich anfangs vermutet hatte. Eine E-Mail an Springer musste her. Ich surfte also zu *springer.com* und klickte mich zum Kontaktformular. Ich fragte, ob sie Interesse daran hätten, *Dive Into Python 3* auf Deutsch herauszubringen.

Die Chance überhaupt eine – nicht automatisch generierte – Antwort zu erhalten, schätzte ich damals auf etwa 25 Prozent. Warum gerade 25 Prozent? Sagen wir vielleicht lieber, es erschien mir recht unwahrscheinlich. Springer ist immerhin der zweitgrößte Wissenschaftsverlag der Welt. Warum sollte gerade dieser riesige Verlag sich für die Idee eines jungen Übersetzers erwärmen? Ich sollte eines Besseren belehrt werden. Dorothea Glaunsinger, die Assistentin des für den Informatik-Bereich zuständigen Lektors Hermann Engesser, antwortete bereits nach kurzer Zeit

und teilte mir mit, dass es in der Tat eine sehr interessante Idee sei. Wenn möglich sollte ich ein Probekapitel senden, damit sie sich ein Urteil über meine Übersetzung bilden könnten. Hermann Engesser melde sich dann bei mir.

Ich schickte meine zwei, bereits fertigen, Kapitel per E-Mail an Frau Glaunsinger und war nun aufgeregter denn je. Werden sie die Übersetzung positiv aufnehmen?

(Nun, Sie wissen bereits, wie die Geschichte ausgegangen ist, da Sie dieses Buch nun in Händen halten, doch *ich* wusste es damals nicht.)

Nachdem Frau Glaunsinger mir bereits kurze Zeit später geschrieben hatte, dass die Probekapitel sehr gut seien, teilte mir auch Herr Engesser später sein Urteil mit. *Sehr gut*, meinte auch er. Hätte ich Champagner gehabt – würde ich überhaupt Champagner trinken – hätte ich mir nun eine Flasche aufgemacht. Sie müssen wissen, dass dies mein erster Versuch war, bei einem *richtigen* Verlag zu veröffentlichen. Wenn man dann ein so positives Feedback erhält, ist das wie ein Sechser im Lotto … oder … nein, sagen wir lieber wie ein Fünfer mit Zusatzzahl. Der Sechser kommt erst noch.

Nach weiteren E-Mails kam das erste Telefonat mit Hermann Engesser, meinem Lektor in Spe (und das meine ich in der wörtlichen Übersetzung „in der Hoffnung"). Es wurden hauptsächlich rechtliche Fragen besprochen und ich sollte Mark Pilgrim mitteilen, dass er doch bitte seine Erlaubnis zur Veröffentlichung an Springer sendet. Abschließend fragte ich, wie hoch er die Wahrscheinlichkeit einer Veröffentlichung einschätze. 80:20. Wow! Ich versuchte cool zu bleiben, was mir aber – auch bei unserem zweiten Telefonat – nicht sehr gut gelang. Doch auch Hermann Engesser klang immer so begeistert, dass er mich damit ansteckte.

Nachdem auch die Marketingleute das Projekt abgesegnet hatten, rief mich Herr Engesser am 20. Juli, also fast genau zwei Monate nach meiner ersten Kontaktaufnahme, erneut an. Er werde mir einen Vertrag zuschicken. Diesen Vertrag erhielt ich in dreifacher Ausfertigung. Als ich alle drei Exemplare unterschrieben hatte, schickte ich sie weiter an Mark Pilgrim nach North Carolina (die Deutsche Post berechnete dafür sechs Euro, die es mir aber mehr als wert waren). Parallel dazu erschien auf springer.com bereits „unser" Buch. Der Grafiker (oder die Grafikerin), der/die den Umschlag gestaltet hat, hat wirklich ganze Arbeit geleistet. Sehen Sie doch einfach selbst nach. Die Schlange in den Händen des Männleins ist doch ein toller Verweis auf Python.

Leider war die deutsche (oder die amerikanische) Post nicht in der Lage den Vertrag auszuliefern. Die Gründe dafür werde ich wohl niemals erfahren. Als er auch nach vier Wochen noch nicht bei Mark Pilgrim angekommen war, schickte Frau Glaunsinger den Vertrag per E-Mail an ihn. Kurze Zeit später erhielt ich zwei unterschriebene Exemplare zurück und sendete eines davon unverzüglich nach Heidelberg. Damit war nun also auch rechtlich alles in trockenen Tüchern.

Unterdessen übersetzte ich den restlichen Text und führte schließlich die Formatierung des Manuskripts aus. Die letze Seite stellte ich an einem Samstagmorgen fertig.

Ich kann nur sagen, dass die Arbeit an diesem Buch eine unglaubliche Erfahrung für mich war. Ein halbes Jahr Arbeit meinerseits steckt zwischen den beiden Buchdeckeln. Ein halbes Jahr Nachtschichten, Wochenendarbeit und ab und zu ein klein wenig Ärger mit meinem PC. Ich hoffe, dass sich diese Arbeit gelohnt hat und Sie

einiges aus diesem Buch lernen und in der Praxis anwenden werden. Sie finden in diesem Werk in sehr kompakter Form alle wichtigen Elemente, die Python 3 zu bieten hat.

Über Feedback freue ich mich sehr; besuchen Sie einfach *python3-intensivkurs.de* oder *florianwollenschein.de* und setzen Sie sich mit mir in Verbindung. Wenn Sie Mark Pilgrim etwas zukommen lassen möchten, surfen Sie zu *diveintomark.org*.

Unter *python3-intensivkurs.de* finden Sie außerdem den in diesem Buch verwendeten Beispielcode zum Herunterladen.

Nun wünsche ich Ihnen viel Spaß und Erfolg bei der Arbeit mit diesem Buch.

November 2009 Florian Wollenschein

Danksagung

Danksagung von Mark Pilgrim

Ich danke meiner Frau, die mit ihrer nie enden wollenden Unterstützung und Ermutigung dazu beigetragen hat, dass dieses Buch nicht nur ein weiterer Punkt auf meiner To-Do-Liste ist.

Mein Dank gilt Raymond Hettinger, dessen Alphametik-Löser die Grundlage des achten Kapitels bildet.

Ebenso danke ich Jesse Noller dafür, dass sie mir während der PyCon 2009 so viele Dinge erklärt hat, die ich somit jedem anderen erklären konnte.

Schließlich danke ich den vielen Menschen, die mir während des Schreibens ihr Feedback zukommen ließen, insbesondere gilt mein Dank Giulio Piancastelli, Florian Wollenschein und all den netten Menschen von python.reddit.com.

Danksagung des Übersetzers

Mein Dank gilt zu allererst Mark Pilgrim, der diese Übersetzung mit seinem Buch und seiner unglaublichen Unterstützung während der ganzen Monate überhaupt erst möglich gemacht hat. Thank you very much, Mark.

Außerdem danke ich Hermann Engesser, Dorothea Glaunsinger und Gabriele Fischer vom Springer-Verlag. Alle drei haben das Projekt voller Begeisterung betreut und mir immer und immer wieder Fragen beantwortet.

Es waren harte (und dennoch schöne) Wochen dieses Buch zu übersetzen und so gilt mein Dank auch und ganz besonders Daniela, die mir Tag für Tag unter die Arme gegriffen hat.

Meine Eltern Jürgen und Cornelia, meine Schwester Sarah und meine Oma Margarete darf ich nicht außen vor lassen … Hey, es ist endlich gedruckt!!!

Inhaltsverzeichnis

Kapitel 1
Python installieren

1.1 Los geht's

Willkommen bei Python 3. Lassen Sie uns loslegen. Im Verlauf dieses Kapitels werden Sie die für Sie passende Version von Python 3 installieren.

1.2 Welche Python-Version ist die Richtige für Sie?

Zuerst müssen Sie Python installieren, oder etwa nicht?

Benutzen Sie einen Account auf einem gehosteten Server, so könnte es sein, dass Ihr Provider Python 3 bereits installiert hat. Verwenden Sie zu Hause Linux, dann haben Sie Python 3 vielleicht auch schon. Die allermeisten bekannten GNU/Linux-Distributionen haben Python 2 standardmäßig installiert; eine kleinere, aber steigende Zahl von Distributionen hat zusätzlich auch Python 3 an Bord. (Wie Sie in diesem Kapitel noch erfahren werden, können Sie mehr als eine Python-Version auf Ihrem Computer installiert haben.) Mac OS X besitzt eine Kommandozeilen-Version von Python 2, doch zum Zeitpunkt da ich dies schreibe ist Python 3 nicht enthalten. Microsoft Windows hat von Haus aus gar kein Python dabei. Doch das ist kein Grund zu verzweifeln! Die Installation von Python läuft unter allen Betriebssystemen sehr einfach ab.

Die einfachste Möglichkeit zu überprüfen, ob Python 3 auf Ihrem Linux- oder Mac-OS-X-System bereits installiert ist, besteht darin, eine Kommandozeile zu öffnen. Unter Linux sehen Sie dazu einfach unter Ihrem `Anwendungen`-Menü nach einem Programm namens `Terminal` oder `Konsole`. (Vielleicht finden Sie es auch in einem Untermenü mit dem Namen `Zubehör` oder `System`.) Unter Mac OS X gibt es dazu die Anwendung `Terminal.app` im Ordner `/Programme/Dienstprogramme/`.

Haben Sie die Kommandozeile geöffnet, geben Sie einfach `python3` ein (nur Kleinbuchstaben, keine Leerzeichen) und sehen Sie, was passiert. Auf meinem Linux-System zu Hause ist Python 3 bereits installiert und der Befehl öffnet die interaktive Shell von Python.

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_1, © Springer-Verlag Berlin Heidelberg 2010

```
mark@atlantis:~$ python3
Python 3.0.1+ (r301:69556, Apr 15 2009, 17:25:52)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more infor-
mation.
>>>
```

(Geben Sie `exit()` ein und drücken Sie `EINGABE`, um die interaktive Shell zu verlassen.)

Mein Webhoster verwendet ebenfalls Linux und bietet Kommandozeilenzugriff; Python 3 ist aber auf meinem Server nicht installiert. (Buh!)

```
mark@manganese:~$ python3
bash: python3: command not found
```

Nun zurück zur Frage, mit der ich diesen Abschnitt begonnen habe: „Welche Python-Version ist die Richtige für Sie?“ Die Antwort: jede, die auf Ihrem Computer läuft.

1.3 Installation unter Microsoft Windows

Windows ist heute in zwei Architekturen erhältlich: 32-Bit und 64-Bit. Natürlich gibt es viele verschiedene Windows-Versionen – XP, Vista, Windows 7 – doch Python läuft unter all diesen Versionen. Es ist wichtiger, zwischen 32-Bit und 64-Bit zu unterscheiden. Wissen Sie nicht, welche Architektur Ihr Windows verwendet, ist es wahrscheinlich 32-Bit.

Laden Sie unter `python.org/download/` die zu Ihrer Architektur passende Version des Windows-Installers herunter. Sie haben dort etwa die folgenden Wahlmöglichkeiten:

- **Python 3.1 Windows installer** (Windows binary – does not include source)
- **Python 3.1 Windows AMD64 installer** (Windows AMD64 binary – does not include source); („does not include source“ bedeutet, dass der Quelltext nicht enthalten ist; Anm. d. Übers.)

Ich verzichte hier darauf, direkte Downloadlinks anzugeben, da ständig kleinere Aktualisierungen hinzukommen und ich nicht dafür verantwortlich sein möchte, dass Sie wichtige Updates verpassen. Sie sollten immer die aktuellste Python 3.x-Version installieren, es sei denn, Sie haben einen Grund dies nicht zu tun.

Ist der Download abgeschlossen, doppelklicken Sie auf die `.msi`-Datei. Windows öffnet daraufhin eine Sicherheitswarnung, da Sie versuchen ausführbaren Code zu starten. Das offizielle Python-Installationsprogramm ist digital signiert von der *Python Software Foundation*, einem Non-Profit-Unternehmen, das die Entwicklung von Python überwacht. Vertrauen Sie nur dem Original!

Klicken Sie auf die `Ausführen`-Schaltfläche, um das Installationsprogramm zu starten.

Zuerst müssen Sie wählen, ob Sie Python 3 für alle Benutzer oder nur für sich installieren möchten. Standardmäßig ist „Für alle Benutzer installieren“ (*Install for all users*) ausgewählt; dies ist auch die beste Wahl, sofern Sie keinen guten Grund haben, eine andere zu treffen. (Ein möglicher Grund, warum Sie „Nur für mich installieren“ (*Install just for me*) wählen würden, ist dass Sie Python auf dem Computer Ihres Unternehmens installieren möchten, aber Ihr Windows-Benutzerkonto keine Administratorrechte besitzt. Doch warum sollten Sie Python ohne die Erlaubnis Ihres Windows-Administrators installieren? Machen Sie mir hier keinen Ärger!)

Klicken Sie auf „Weiter“ (*Next*), um die von Ihnen gewählte Installationsart zu akzeptieren.

Nun fordert das Installationsprogramm Sie auf, ein Zielverzeichnis zu wählen. Das Standardverzeichnis für alle Versionen von Python 3.1.x ist `C:\Python31\`; dieses Verzeichnis sollte nur geändert werden, wenn es nötig ist. Haben Sie einen eigenen Laufwerksbuchstaben auf dem Sie Ihre Anwendungen installieren, können Sie diesen mithilfe der integrierten Schaltflächen suchen oder einfach den Pfadnamen in der unteren Box eingeben. Sie müssen Python nicht auf `C:` installieren, sondern können jedes beliebige Laufwerk und jeden beliebigen Ordner verwenden.

Klicken Sie auf „Weiter“ (*Next*), um Ihre Wahl des Zielverzeichnisses zu akzeptieren.

Das nächste Fenster sieht verwirrend aus, ist es aber eigentlich nicht. Wie bei vielen Installationen können Sie auch hier einzelne Komponenten von Python 3 abwählen. Ist Ihr Festplattenspeicher sehr begrenzt, können Sie bestimmte Teile von der Installation ausschließen.

- **Register Extensions** erlaubt es Ihnen, Python-Skripte (`.py`-Dateien) durch einen Doppelklick zu starten. Empfohlen, aber nicht notwendig. (Diese Option benötigt keinerlei Festplattenspeicher; es macht also wenig Sinn, sie abzuwählen.)
- **Tcl/Tk** ist die Grafikbibliothek die von der Python-Shell, die Sie im Verlauf des Buches benutzen werden, verwendet wird. Ich empfehle dringend, diese Option selektiert zu lassen.
- **Documentation** installiert eine Hilfe-Datei, welche die meisten der Informationen auf `docs.python.org` beinhaltet. Empfohlen, wenn Sie ein Analogmodem verwenden, oder nur begrenzt Zugang zum Internet haben.
- **Utility Scripts** enthält das Skript `2to3.py`, worüber Sie später in diesem Buch mehr erfahren werden. Wird benötigt, wenn Sie lernen möchten, wie man vorhandenen Code von Python 2 zu Python 3 portieren kann. Wenn Sie keinen vorhandenen Python-2-Code haben, können Sie diese Option abwählen.
- **Test Suite** ist eine Skriptsammlung, die verwendet wird, um den Python-Interpreter selbst zu überprüfen. Ich werde sie weder in diesem Buch verwenden, noch habe ich sie jemals während der Programmierung mit Python benutzt. Optional.

Sind Sie unsicher, wie viel freien Festplattenspeicher Sie haben, können Sie auf die Schaltfläche „Disk Usage“ klicken. Das Installationsprogramm listet daraufhin

alle Laufwerke auf, berechnet den verfügbaren Speicherplatz jedes Laufwerks und zeigt, wie viel Platz nach der Installation übrig bleiben würde.

Klicken Sie auf OK, um zum Fenster „Customize Python“ zurückzukehren.

Entscheiden Sie sich, eine Option abzuwählen, klicken Sie auf die Schaltfläche mit einer abgebildeten Festplatte und wählen Sie „Entire feature will be unavailable“. Wählen Sie z. B. die Test Suite ab, so sparen Sie gigantische 7.908 KB Speicherplatz.

Klicken Sie auf „Weiter“ (*Next*), um Ihre Auswahl zu akzeptieren.

Das Installationsprogramm kopiert nun alle notwendigen Dateien in das von Ihnen gewählte Zielverzeichnis.

Klicken Sie auf „Beenden“ (*Finish*), um die Installation abzuschließen und das Installationsprogramm zu schließen.

In Ihrem Startmenü sollte sich jetzt ein Eintrag mit der Bezeichnung `Python 3.1` finden. Gehen Sie mit dem Mauszeiger darüber, finden Sie darin ein Programm mit dem Namen `IDLE`. Wählen Sie dieses Programm aus, um die interaktive Python-Shell zu starten.

1.4 Installation unter Mac OS X

Jeder moderne Macintosh-Computer verwendet den Intel-Chip (wie die meisten Windows PCs). Ältere Macs benutzten dagegen PowerPC-Chips. Sie müssen den Unterschied zwischen diesen Chips nicht kennen, da es nur ein Mac-Python-Installationsprogramm für alle Macs gibt.

Rufen Sie `python.org/download/` auf und laden Sie das Mac-Installationsprogramm herunter. Die Bezeichnung lautet etwa **Python 3.1 Mac Installer Disk Image**, wobei die Versionsnummer abweichen kann. Stellen Sie jedoch sicher, dass Sie Version 3.x herunterladen und nicht 2.x.

Ihr Browser sollte das Speicherabbild automatisch einhängen und den Inhalt in einem Finder-Fenster anzeigen. (Geschieht dies nicht, müssen Sie das Speicherabbild in Ihrem Download-Ordner suchen und zum Einhängen darauf doppelklicken. Die Bezeichnung lautet etwa `python-3.1.dmg`.) Das Speicherabbild enthält einige Textdateien (`Build.txt`, `License.txt`, `ReadMe.txt`) und das eigentliche Installationspaket `Python.mpkg`.

Doppelklicken Sie auf das `Python.mpkg`-Installationspaket, um das Mac-Python-Installationsprogramm zu starten.

Das erste Fenster enthält eine kurze Beschreibung von Python selbst und verweist Sie auf die `ReadMe.txt`-Datei (die Sie nicht gelesen haben, richtig?), um weitere Details zu erfahren.

Klicken Sie auf „Fortfahren“ (*Continue*).

Das nächste Fenster enthält nun einige wichtige Informationen: Python setzt Mac OS X 10.3 oder höher voraus. Verwenden Sie immer noch Mac OS X 10.2, dann sollten Sie wirklich auf eine neuere Version umsteigen. Apple stellt für Ihr Betriebssystem nicht länger Sicherheitsaktualisierungen bereit, und Ihr Computer

könnte gefährdet sein, wenn Sie online gehen. Außerdem können Sie Python 3 nicht verwenden.

Klicken Sie auf „Fortfahren“ (*Continue*), um zum nächsten Schritt zu gelangen.

Pythons Installationsprogramm zeigt – wie alle guten Installationsprogramme – den Softwarelizenzvertrag an. Python ist Open Source und seine Lizenz von der *Open Source Initiative* anerkannt. Python hatte in seiner Geschichte einige Eigentümer und Förderer, von denen jeder seine Spuren in der Softwarelizenz hinterlassen hat. Doch das Endresultat ist dieses: Python ist Open Source und Sie können es auf jeder beliebigen Plattform, zu jedem beliebigen Zweck, ohne Gebühr oder Verpflichtungen nutzen.

Klicken Sie abermals auf „Fortfahren“ (*Continue*).

Aufgrund einer Eigenart des Apple Installations-Frameworks müssen Sie der Softwarelizenz „zustimmen”, wenn Sie die Installation abschließen möchten. Da Python Open Source ist, „stimmen Sie zu“, dass die Lizenz Ihnen zusätzliche Rechte gewährt, statt sie Ihnen wegzunehmen.

Klicken Sie zum Fortfahren auf „Annehmen“ (*Agree*).

Das nächste Fenster erlaubt es Ihnen, den Installationsort zu verändern. Sie müssen Python auf Ihrem Bootlaufwerk installieren, was das Installationsprogramm aufgrund von Begrenzungen aber nicht forciert.

Aus diesem Fenster heraus können Sie außerdem Ihre Installation anpassen und bestimmte Funktionen ausschließen. Klicken Sie auf *Customize*, wenn Sie dies tun möchten. Andernfalls klicken Sie auf *Install*.

Haben Sie eine angepasste Installation gewählt, stellt Ihnen das Installationsprogramm die folgende Liste von Funktionen zur Verfügung:

- **Python Framework** Dies ist das Herzstück von Python und sowohl ausgewählt, als auch deaktiviert, da es installiert werden muss.
- **GUI Applications** beinhaltet `IDLE`, die grafische Python-Shell, die Sie im Verlauf dieses Buches verwenden werden. Ich empfehle dringend, diese Option ausgewählt zu lassen.
- **UNIX command-line tools** beinhaltet die Kommandozeilenanwendung `python3`. Auch hier empfehle ich dringend, die Option ausgewählt zu lassen.
- **Python Documentation** beinhaltet die meisten der Informationen von `docs.python.org`. Empfohlen, wenn Sie ein Analogmodem verwenden, oder nur begrenzt Zugang zum Internet haben.
- **Shell profile updater** überwacht, ob Ihr Shell-Profil (das in `Terminal.app` Verwendung findet) aktualisiert werden muss, um sicherzustellen, dass diese Version von Python sich im Suchpfad Ihrer Shell befindet. Sie müssen dies wahrscheinlich nicht ändern.
- **Fix system Python** sollte nicht geändert werden. (Es teilt Ihrem Mac mit, dass er Python 3 als Standard für alle Python-Skripts verwenden soll, auch für die von Apple ins System integrierte. Dies wäre sehr schlecht, da die meisten dieser Skripts in Python 2 geschrieben sind und unter Python 3 nicht richtig laufen würden.)

Klicken Sie auf *Install*, um fortzufahren.

Da das Installationsprogramm systemweite Frameworks und ausführbare Programme in `/usr/local/bin/` installiert, fragt es nach Ihrem Administratorpasswort. Es besteht keine Möglichkeit, Mac Python ohne Administratorrechte zu installieren.

Klicken Sie auf OK, um die Installation zu starten.

Es wird ein Fortschrittsbalken angezeigt, während die von Ihnen gewählten Funktionen installiert werden.

Ist alles gut gelaufen, wird Ihnen ein großes, grünes Häkchen angezeigt, das Ihnen die erfolgreiche Installation bestätigt.

Klicken Sie auf *Close*, um das Installationsprogramm zu beenden.

Haben Sie den Installationsort nicht verändert, so finden Sie die gerade installierten Dateien im Ordner `Python 3.1` Ihres `/Anwendungen`-Ordners. Am wichtigsten ist hier `IDLE`, die grafische Python-Shell.

Doppelklicken Sie auf `IDLE`, um die Python-Shell zu starten.

Die Python-Shell ist das Programm, mit dem Sie während des Erkundens von Python die meiste Zeit verbringen werden. Die Beispiele in diesem Buch gehen davon aus, dass Sie wissen, wie Sie die Python-Shell starten.

1.5 Installation unter Ubuntu Linux

Moderne Linux-Distributionen werden durch sogenannte Repositorys mit einem enormen Umfang an vorkompilierten Anwendungen unterstützt, die einfach installiert werden können. Unter Ubuntu Linux besteht die einfachste Möglichkeit zur Installation von Python 3 darin, die Anwendung `Hinzufügen/Entfernen` in Ihrem `Anwendungen`-Menü zu verwenden.

Nach dem Start der `Hinzufügen/Entfernen`-Anwendung wird Ihnen eine Liste vorselektierter Anwendungen in unterschiedlichen Kategorien angezeigt. Einige sind schon installiert; die meisten jedoch nicht. Da das Repository über 10.000 Anwendungen enthält, existieren verschiedene Filter, die Sie anwenden können, um nur kleine Teile des Repositorys zu sehen. Als Standardfilter ist „Canonical-maintained applications“ ausgewählt, welcher eine kleine Untermenge aller Anwendungen die offiziell von Canonical – dem Unternehmen, das Ubuntu Linux entwickelt und pflegt – unterstützt werden, anzeigt.

Da Python 3 nicht von Canonical gepflegt wird, müssen Sie zuerst einmal das Filtermenü aufklappen und „All Open Source applications“ auswählen.

Haben Sie den Filter so erweitert, dass er alle Open-Source-Anwendungen anzeigt, benutzen Sie die Suche direkt neben dem Filtermenü, um nach Python 3 zu suchen.

Die Liste beschränkt sich nun nur noch auf die Anwendungen, die Python 3 beinhalten. Sie werden zwei Pakete auswählen. Das erste Paket ist `Python (v3.0)`. Dieses enthält den Python-Interpreter selbst.

Das zweite gewünschte Paket finden Sie direkt darüber: `IDLE (using Python-3.0)`. Dies ist eine grafische Shell für Python, die Sie im Verlauf dieses Buches nutzen werden.

Haben Sie diese beiden Pakete ausgewählt, klicken Sie auf *Änderungen anwenden*, um fortzufahren.

Der Paketmanager bittet Sie nun, zu bestätigen, dass Sie `IDLE (using Python-3.0)` und `Python (v3.0)` installieren möchten.

Klicken Sie zum Fortfahren auf *Anwenden*.

Der Paketmanager zeigt Ihnen einen Fortschrittsbalken an, während er die notwendigen Pakete von Canonicals Internet-Repository herunterlädt.

Sind die Pakete fertig heruntergeladen, beginnt der Paketmanager automatisch mit der Installation.

Ist alles korrekt verlaufen, bestätigt Ihnen der Paketmanager, dass beide Pakete erfolgreich installiert wurden. Hier können Sie mit einem Doppelklick auf `IDLE` die Python-Shell starten, oder den Paketmanager verlassen, indem Sie auf *Schließen* klicken.

Sie können die Python-Shell immer starten, indem Sie in Ihrem `Anwendungen`-Menü im Untermenü `Entwicklung` `IDLE` auswählen.

In der Python-Shell werden Sie beim Erkunden von Python die meiste Zeit verbringen. Die in diesem Buch aufgeführten Beispiele setzen voraus, dass Sie wissen, wie man die Python-Shell startet.

1.6 Installation auf anderen Plattformen

Python 3 ist für einige verschiedene Plattformen verfügbar. Im Besonderen für nahezu jede Linux-, BSD- und Solaris-basierte Distribution. RedHat Linux verwendet beispielsweise den Paketmanager `yum`; FreeBSD hat seine Ports und Paketsammlungen; Solaris benutzt `pkgadd` und Konsorten. Eine kurze Websuche nach `Python 3 + Ihr Betriebssystem` wird Ihnen zeigen, ob Python 3 dafür verfügbar ist und wenn ja, wie man es installiert.

1.7 Verwenden der Python-Shell

Die Python-Shell ist das Programm, in dem Sie die Syntax von Python erkunden, interaktive Hilfe zu Befehlen erhalten und kurze Programme debuggen können. Die grafische Python-Shell (`IDLE` genannt) verfügt außerdem über einen Texteditor, der die farbige Hervorhebung der Python-Syntax unterstützt und sich in die Python-Shell integriert. Haben Sie noch keinen bevorzugten Texteditor, sollten Sie `IDLE` ausprobieren.

Eins nach dem anderen. Die Python-Shell selbst ist eine unglaubliche interaktive Spielwiese. Im Verlauf dieses Buches werden Ihnen Beispiele wie dieses begegnen:

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

Die drei spitzen Klammern, >>>, stellen die Eingabeaufforderung der Python-Shell dar. Geben Sie diese nicht ein. Sie dienen nur dazu, Ihnen zu zeigen, dass Sie nach der Eingabeaufforderung etwas eingeben sollen.

`1 + 1` ist der Teil, den Sie eingeben müssen. Sie können jeden beliebigen gültigen Python-Ausdruck oder -Befehl in der Python-Shell eingeben. Scheuen Sie sich nicht; sie wird nicht beißen! Das Schlimmste was passieren kann, ist dass Sie eine Fehlermeldung erhalten. Befehle werden sofort ausgeführt (sobald Sie `EINGABE` drücken); Ausdrücke werden sofort ausgewertet und die Python-Shell zeigt das Ergebnis an.

`2` ist das Ergebnis des ausgewerteten Ausdrucks. Zufällig ist `1 + 1` ein gültiger Python-Ausdruck. Das Ergebnis ist natürlich `2`.

Lassen Sie uns noch etwas versuchen.

```
>>> print('Hello world!')
Hello world!
```

Ziemlich einfach, nicht wahr? Doch Sie können in der Python-Shell sehr viel mehr machen. Sollten Sie jemals stecken bleiben – Sie können sich an einen Befehl oder an die passenden Argumente einer bestimmten Funktion nicht mehr erinnern – können Sie innerhalb der Python-Shell interaktive Hilfe erhalten. Geben Sie dazu einfach `help` ein und drücken Sie `EINGABE`.

```
>>> help
Type help() for interactive help, or help(object) for help about
object.
```

Die Python-Shell bietet zwei Formen der Hilfe an. Die Hilfe zu einem einzelnen Objekt gibt einfach die Dokumentation aus und führt Sie zurück zur Eingabeaufforderung. Sie können aber auch in den Hilfemodus wechseln, was dazu führt, dass nun keine Python-Ausdrücke mehr ausgewertet werden, sondern Ihnen bei Eingabe eines Keywords oder Befehls alles angezeigt wird, was die Hilfe darüber weiß.

Um in den interaktiven Hilfemodus zu gelangen, geben Sie `help()` ein und drücken Sie `EINGABE`.

```
>>> help()
Welcome to Python 3.0!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

Beachten Sie, dass sich die Eingabeaufforderung von >>> in help> ändert. Dies erinnert Sie daran, dass Sie sich im interaktiven Hilfemodus befinden. Nun können Sie jedes beliebige Keyword, jeden beliebigen Befehl, Modulnamen, Funktionsnamen – so ziemlich alles, was Python kennt – eingeben und die Dokumentation dazu lesen.

```
help> print                                                    ①
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a  file-like  object  (stream);  defaults  to  the  current
sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.

help> PapayaWhip                                               ②
no Python documentation found for 'PapayaWhip'

help> quit                                                     ③

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>                                                            ④
```

① Um die Dokumentation zur `print()`-Funktion zu erhalten, geben Sie `print` ein und drücken Sie `EINGABE`. Der interaktive Hilfemodus zeigt etwas an, das einer *man page* ähnelt: den Funktionsnamen, eine kurze Zusammenfassung, die Funktionsargumente und ihre Standardwerte usw. Sollte Ihnen die Dokumentation schleierhaft erscheinen, machen Sie sich bitte keine Sorgen. Im Verlauf der nächsten Kapitel werden Sie mehr darüber erfahren.

② Der interaktive Hilfemodus kennt natürlich nicht alles. Geben Sie etwas ein, das weder ein Python-Befehl, noch ein Python-Modul, noch eine Python-Funktion, oder ein integriertes Keyword ist, zuckt der interaktive Hilfemodus nur mit seinen virtuellen Achseln.

③ Um den interaktiven Hilfemodus zu beenden, geben Sie `quit` ein und drücken Sie `EINGABE`.

④ Die Eingabeaufforderung ändert sich wieder zu >>>, um Ihnen anzuzeigen, dass Sie den interaktiven Hilfemodus verlassen haben und sich wieder in der Python-Shell befinden.

`IDLE`, die grafische Python-Shell, enthält auch einen Python-sensitiven Texteditor. Im nächsten Kapitel erfahren Sie, wie Sie diesen verwenden.

1.8 Python-Editoren und -IDEs

`IDLE` ist nicht das Maß aller Dinge, wenn es darum geht, Programme in Python zu schreiben. `IDLE` ist hilfreich zum Erlernen der Sprache selbst, doch viele Entwickler bevorzugen andere Texteditoren oder Integrierte Entwicklungsumgebungen (engl. *Integrated Development Environment*, kurz *IDE*; Anm. d. Übers.). Ich werde diese hier nicht behandeln, doch die Python-Community pflegt eine Liste von Python-sensitiven Editoren, die eine große Bandbreite an unterstützten Plattformen und Softwarelizenzen abdecken.

Eine IDE die Python 3 unterstützt ist PyDev, ein Plug-in für Eclipse, das Eclipse in eine ausgewachsene Python-IDE verwandelt. Sowohl Eclipse, als auch PyDev sind plattformübergreifend und Open Source.

Auf kommerzieller Seite hat Komodo IDE von ActiveState die Nase vorn. Man muss zwar pro Benutzer eine Lizenz kaufen, aber es gibt einen Studentenrabatt und eine kostenlose zeitlimitierte Demoversion.

Ich programmiere nun seit neun Jahren mit Python und ich bearbeite meine Programme mit GNU Emacs und debugge sie in der Python-Shell auf der Kommandozeile. Es gibt kein richtig oder falsch bei der Auswahl der Werkzeuge. Finden Sie Ihren eigenen Weg!

Kapitel 2
Ihr erstes Python-Programm

2.1 Los geht's

Bücher über das Programmieren beginnen meist mit einer Reihe langweiliger Kapitel über die Grundlagen, bis schlussendlich ein sinnvolles Programm herauskommt. Lassen Sie uns all das überspringen. Nachfolgend finden Sie ein komplettes, funktionierendes Python-Programm. Sie werden es vielleicht nicht sofort verstehen, doch machen Sie sich darüber keine Sorgen, denn wir werden es Zeile für Zeile auseinandernehmen. Sehen Sie es sich erst einmal an. Vielleicht verstehen Sie ja doch etwas.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
    if size < 0:
        raise ValueError('number must be non-negative')

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)
```

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_2, © Springer-Verlag Berlin Heidelberg 2010

```
    raise ValueError('number too large')


if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

Wenn wir dieses Programm nun über die Kommandozeile starten, sieht das unter Windows etwa so aus:

```
c:\home\diveintopython3> c:\python30\python.exe humansize.py
1.0 TB
931.3 GiB
```

Unter Mac OS X oder Linux sollte es wie folgt aussehen:

```
you@localhost:~$ python3 humansize.py
1.0 TB
931.3 GiB
```

Was ist da gerade passiert? Sie haben Ihr erstes Python-Programm ausgeführt. Sie haben den Python-Interpreter auf der Kommandozeile aufgerufen und ihm den Namen des Skripts übergeben, das Sie ausführen wollten. Dieses Skript definiert die Funktion `approximate_size()`, die eine genaue Dateigröße in Bytes übernimmt und eine „schönere“ (aber ungenauere) Größe berechnet.

Am Ende dieses Skripts sehen Sie zwei aufeinanderfolgende Aufrufe von `print(approximate_size(arguments))`. Dies sind Funktionsaufrufe; zunächst wird die `approximate_size()`-Funktion mit einigen übergebenen Argumenten aufgerufen, dann wird der Rückgabewert an die `print()`-Funktion übergeben. Die `print()`-Funktion ist eine integrierte Funktion. Sie können sie jederzeit, überall verwenden. (Es gibt sehr viele integrierte Funktionen, aber auch sehr viele Funktionen, die sich in Modulen befinden. Geduld, mein Freund.)

Warum gibt das Skript auf der Kommandozeile immer dasselbe aus? Dazu kommen wir noch. Erst einmal sehen wir uns die `approximate_size()`-Funktion an.

2.2 Funktionen deklarieren

Python nutzt, ebenso wie die meisten anderen Programmiersprachen, Funktionen. Es existieren jedoch keine separaten Header-Dateien wie in C++, oder `interface`/`implementation`-Abschnitte wie in Pascal. Wenn Sie eine Funktion benötigen, deklarieren Sie sie einfach wie folgt:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

Mit `def` wird die Funktionsdeklaration begonnen; darauf folgt der Name der Funktion und die Argumente eingeschlossen in Klammern. Mehrere Argumente werden durch Kommas getrennt.

Beachten Sie auch, dass die Funktion keinen Datentyp für einen Rückgabewert definiert. In Python geben Funktionen den Datentyp ihres Rückgabewertes nicht an; es wird nicht einmal angegeben, ob sie überhaupt einen Wert zurückgeben. Tatsächlich gibt jedoch jede Funktion in Python einen Wert zurück; sollte die Funktion eine `return`-Anweisung enthalten, wird der dort angegebene Wert zurückgegeben, ansonsten liefert sie `None` zurück, den Nullwert von Python.

☞ In manchen Sprachen beginnen Funktionen (die einen Wert zurückgeben) mit `function`; Subroutinen (die keinen Wert zurückgeben) beginnen mit `sub`. In Python gibt es keine Subroutinen. Alles ist eine Funktion, alle Funktionen geben einen Wert zurück (auch wenn es `None` ist) und alle Funktionen beginnen mit `def`.

Die Funktion `approximate_size` nimmt zwei Argumente entgegen – `size` und `a_kilobyte_is_1024_bytes` – doch keines der Argumente definiert einen Datentyp. (Wie Sie vielleicht an der `=True`-Syntax erkennen, handelt es sich bei dem zweiten Argument um einen booleschen Wert. Datentypen werden in Python niemals explizit angegeben. Python findet automatisch heraus, welchen Typ eine Variable hat und überwacht dies intern.)

☞ In Java, C++ und anderen statisch-typisierten Sprachen müssen Sie einen Datentyp für die Funktion, den Rückgabewert und jedes Argument angeben. In Python werden niemals explizit Datentypen für irgendetwas angegeben. Python überwacht den Datentyp intern, basierend auf dem Wert, den Sie zuweisen.

2.2.1 Pythons Datentypen im Vergleich mit denen anderer Sprachen

Ein pfiffiger Programmierer hat mir folgenden Vergleich von Python mit anderen Sprachen zugesendet.

Statisch typisierte Sprache Eine Programmiersprache, bei der Datentypen zur Kompilierzeit feststehen. Die Mehrheit der statisch typisierten Sprachen sorgen dafür, indem Sie den Programmierer zwingen, allen Variablen vor der Verwendung einen Datentyp zuzuweisen. Java und C sind statisch typisierte Programmiersprachen.

Dynamisch typisierte Sprache Eine Programmiersprache, bei der der Datentyp zur Laufzeit ermittelt wird; das Gegenteil von statisch typisiert. VBScript und Python sind dynamisch typisierte Sprachen, die den Datentyp einer Variablen bei der ersten Zuweisung ermitteln.

Stark typisierte Sprache Eine Programmiersprache, bei der Datentypen immer erzwungen werden. Java und Python sind stark typisiert. Sie können zum Beispiel eine Ganzzahl ohne explizite Konvertierung nicht wie einen String verwenden.

Schwach typisierte Sprache Eine Programmiersprache, bei der Datentypen ignoriert werden können; das Gegenteil von stark typisiert. VBScript ist schwach typisiert, da man zum Beispiel den String `'12'` und die Ganzzahl `3` zum String `'123'`

verketten und diesen wiederum als Ganzzahl `123` verwenden kann. Das alles funktioniert ohne explizite Umwandlung.

2.3 Lesbaren Code schreiben

Ich werde Sie nun nicht mit einer Ansprache darüber langweilen, wie wichtig es ist, seinen Code zu dokumentieren. Denken Sie jedoch daran, dass Code nur einmal geschrieben, aber sehr oft gelesen wird. Der wichtigste Leser Ihres Codes sind Sie selbst, sechs Monate nachdem Sie ihn geschrieben haben (d. h., nachdem Sie alles vergessen haben, aber irgendetwas ausbessern müssen). Mit Python ist es einfach, lesbaren Code zu schreiben, also nutzen Sie diesen Vorteil. In sechs Monaten werden Sie mir danken.

2.3.1 Docstrings

Sie können eine Python-Funktion dokumentieren, indem Sie ihr einen `docstring` (kurz für „Documentation String“) zuweisen. Im vorliegenden Programm hat die Funktion `approximate_size` einen `docstring`:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''
```

Dreifache Hochkommas kennzeichnen einen Blockkommentar. Alles, was zwischen den ersten und letzten Hochkommata steht, ist Teil eines einzelnen Strings. Dazu gehören auch Zeilenvorschübe, führende Leerzeichen und andere Zeichen. Sie können sie überall benutzen, doch sie werden meist zur Definition eines `docstring` verwendet.

Alles innerhalb der dreifachen Hochkommas gehört zum `docstring` der Funktion, der dokumentiert, was die Funktion tut. Ein `docstring` muss, sofern er existiert, innerhalb der Funktion als erstes definiert werden (d. h. in der Zeile, die auf die Deklaration der Funktion folgt). Es ist nicht zwingend erforderlich, dass Sie Ihrer Funktion einen `docstring` hinzufügen, doch Sie sollten es immer tun.

2.4 Der import-Suchpfad

Bevor wir weitermachen möchte ich kurz den Suchpfad der Bibliothek erwähnen. Wenn Sie versuchen ein Modul zu importieren, sucht Python dieses an verschiedenen Orten. Genaugenommen durchforstet Python alle Ordner, die unter `sys.path` angegeben sind. `sys.path` ist eine einfache Liste; Sie können sich diese ansehen oder mit den bekannten Methoden verändern. (Im nächsten Kapitel werden Sie mehr über Listen erfahren.)

```
>>> import sys                                                    ①
>>> sys.path                                                      ②
['',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
>>> sys                                                           ③
<module 'sys' (built-in)>
>>> sys.path.insert(0, '/home/mark/diveintopython3/examples')     ④
>>> sys.path                                                      ⑤
['/home/mark/diveintopython3/examples',
 '',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
```

① Durch das Importieren des Moduls `sys` werden uns all seine Funktionen und Attribute zur Verfügung gestellt.

② `sys.path` ist eine Liste von Ordnernamen, die den aktuellen Suchpfad bilden. (Bei Ihnen wird der Suchpfad anders aussehen, je nach Ihrem Betriebssystem, Ihrer Python-Version und dem ursprünglichen Installationsort.) Python sucht in diesen Ordnern (in dieser Reihenfolge) nach einer `.py`-Datei, deren Name dem angegebenen entspricht.

③ Nun ja, ich war nicht ganz aufrichtig; in Wahrheit ist es etwas komplizierter, weil nicht alle Module als `.py`-Dateien gespeichert sind. Manche der Module, so wie auch das `sys`-Modul, sind „integrierte Module“ (*built-in modules*); diese sind in Python selbst eingebaut. Integrierte Module verhalten sich genauso wie gewöhnliche Module, doch ihr Python-Quellcode ist nicht verfügbar, da sie gar nicht in Python geschrieben sind! (Das Modul `sys` ist in C geschrieben.)

④ Sie können Pythons Suchpfad zur Laufzeit einen neuen Ordner hinzufügen, indem Sie den Ordnernamen an `sys.path` anhängen. Dadurch sucht Python auch

in diesem Ordner, wenn Sie versuchen ein Modul zu importieren. Dies bleibt bestehen, solange Python läuft.

⑤ Durch die Verwendung von `sys.path.insert(0, neuer_pfad)` haben Sie ein neues Verzeichnis als erstes Element der `sys.path`-Liste, und damit am Anfang des Suchpfads, eingefügt. Das ist sehr nützlich. Sollte es zu Namenskonflikten kommen (Python könnte beispielsweise Version 2 einer bestimmten Bibliothek an Bord haben, Sie möchten aber Version 3 nutzen), wird so sichergestellt, dass Ihre Module gefunden und anstelle der mit Python gelieferten Module verwendet werden.

2.5 Alles ist ein Objekt

Eine Funktion ist, wie alles in Python, ein Objekt.

```
>>> import humansize                                    ①
>>> print(humansize.approximate_size(4096, True))       ②
4.0 KiB
>>> print(humansize.approximate_size.__doc__)           ③
Convert a file size to human-readable form.

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string
```

① Mit der ersten Zeile importieren wir das Programm `humansize` als Modul – ein Codestück, das man interaktiv, oder innerhalb eines größeren Python-Programms nutzen kann. Haben Sie einmal ein Modul importiert, so können Sie seine öffentlichen Funktionen, Klassen oder Attribute verwenden. Module können auf diese Weise Zugang zur Funktionalität anderer Module erhalten, und auch die interaktive Shell von Python kann dies tun. Dies ist ein wichtiges Konzept, das Ihnen überall in diesem Buch begegnen wird.

② Wollen Sie Funktionen aus einem importierten Modul verwenden, so müssen Sie den Namen des Moduls angeben. Sie können also nicht einfach `approximate_size` schreiben; `humansize.approximate_size` ist hier korrekt. Haben Sie jemals Klassen in Java benutzt, sollte Ihnen das bekannt vorkommen.

③ Anstatt, wie man es erwarten würde, die Funktion aufzurufen, fragen wir hier nach einem der Attribute, `__doc__`.

2.5.1 Was ist ein Objekt?

In Python ist alles ein Objekt; und fast alles hat Attribute und Methoden. Alle Funktionen besitzen das integrierte Attribut `__doc__`, welches den im Quellcode der

Funktion definierten `docstring` zurückgibt. Das Modul `sys` ist ein Objekt mit dem Attribut `path`. Und so weiter.

Doch dies beantwortet immer noch nicht die grundlegendere Frage: Was ist ein Objekt? Verschiedene Programmiersprachen definieren „Objekt“ auf verschiedene Art und Weise. Bei manchen bedeutet es, dass alle Objekte Attribute und Methoden haben müssen; bei anderen bedeutet es, dass von allen Objekten Klassen abgeleitet werden können. In Python ist die Definition freier; manche Objekte haben weder Attribute noch Methoden, und nicht von allen Objekten können Klassen abgeleitet werden. Doch alles ist ein Objekt in dem Sinne, dass es einer Variablen zugewiesen oder einer Funktion als Argument übergeben werden kann.

Das ist so wichtig, dass ich es für den Fall, dass Sie es übersehen haben wiederhole: Alles in Python ist ein Objekt. Strings sind Objekte. Listen sind Objekte. Funktionen sind Objekte. Selbst Module sind Objekte.

2.6 Code einrücken

Funktionen in Python besitzen kein explizites `begin` oder `end`. Ebenso werden Sie geschweifte Klammern als Kennzeichnung des Beginns oder Endes einer Funktion vergeblich suchen. Das einzige Begrenzungszeichen ist der Doppelpunkt (`:`) und die Einrückung des Codes selbst.

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):  ①
    if size < 0:                                            ②
        raise ValueError('number must be non-negative')     ③
                                                            ④
    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:                       ⑤
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('number too large')
```

① Codeblöcke werden durch ihre Einrückung definiert. Mit „Codeblock“ meine ich Funktionen, `if`-Anweisungen, `for`-Schleifen, `while`-Schleifen und so weiter. Ein Block beginnt bei der ersten und endet mit der letzten eingerückten Zeile. Es gibt keine Klammern oder Schlüsselwörter, um Codeblöcke zu definieren. Das bedeutet, dass Leerraum bedeutsam ist und gleichbleibend sein muss. In diesem Beispiel ist der Code der Funktion vier Zeichen eingerückt. Es müssen jedoch nicht vier Zeichen sein, die Einrückung muss nur gleichbleibend sein. Die erste Zeile, die nicht eingerückt ist markiert das Ende der Funktion.

② In Python folgt auf eine `if`-Anweisung ein Codeblock. Erweist sich der `if`-Ausdruck als wahr, so wird der eingerückte Block ausgeführt, andernfalls wird zum

`else`-Block gesprungen (sofern ein solcher vorhanden ist). (Beachten Sie das Fehlen von Klammern um den Ausdruck.)

③ Diese Zeile befindet sich innerhalb des `if`-Codeblocks. Die `raise`-Anweisung löst eine Ausnahme aus (vom Typ `ValueError`); dies jedoch nur wenn `size < 0` ist.

④ Das ist noch nicht das Ende der Funktion. Vollständig leere Zeilen zählen nicht. Die Funktion geht in der nächsten Zeile weiter.

⑤ Auch die `for`-Schleife kennzeichnet den Beginn eines Codeblocks. Codeblöcke können aus mehreren Zeilen bestehen, wenn diese alle gleich weit eingerückt sind. Diese `for`-Schleife beinhaltet drei Codezeilen. Es gibt keine andere besondere Syntax für mehrzeilige Codeblöcke. Rücken Sie sie einfach ein und machen Sie sich keine Gedanken!

Nach anfänglichen Beschwerden und einigen abfälligen Bemerkungen zu den Ähnlichkeiten mit Fortran werden Sie die Vorteile dieser Methode erkennen. Einer dieser Vorteile ist, dass alle Python-Programme ähnlich aussehen, da das Einrücken des Codes von elementarer Bedeutung und nicht nur ein Stilmittel ist. Dies macht es einfacher, den Code anderer zu lesen und zu verstehen.

☞ Python verwendet Zeilenumbrüche zum Trennen von Anweisungen und einen Doppelpunkt und Einrückung zum Trennen von Codeblöcken. C++ und Java nutzen Semikolons zum Trennen von Anweisungen und geschweifte Klammern zum Trennen von Codeblöcken.

2.7 Ausnahmen

Ausnahmen sind in Python überall vorhanden. Nahezu jedes Modul der Python-Standardbibliothek nutzt sie und auch Python selbst löst sie in vielen verschiedenen Fällen aus. Ausnahmen werden Ihnen in diesem Buch immer wieder begegnen.

Was ist eine Ausnahme? Im Normalfall ist eine Ausnahme ein Fehler, der Ihnen anzeigt, dass etwas schief gegangen ist. (Nicht alle Ausnahmen sind Fehler, doch das ist für den Moment ohne Bedeutung.) Einige Programmiersprachen fordern Sie dazu auf, Fehlercodes zu überprüfen. Python fordert Sie dazu auf, Ausnahmen zu behandeln.

Tritt in der Python-Shell ein Fehler auf, werden genauere Angaben über die Ausnahme und Informationen zur Ursache der Ausnahme ausgegeben. Dies nennt man *unbehandelte* Ausnahme. Als die Ausnahme ausgelöst wurde, war kein Code vorhanden, der sie eingefroren und verarbeitet hätte, also werden lediglich Informationen zum Debuggen ausgegeben und das war's. In der Shell ist das keine große Sache, doch passiert dies während Ihr eigentliches Programm ausgeführt wird, stürzt es ohne Ausnahmebehandlung ab. Vielleicht möchten Sie das, vielleicht aber auch nicht.

☞ Anders als bei Java, geben Funktionen in Python nicht an, welche Ausnahmen sie auslösen könnten. Es liegt an Ihnen, zu bestimmen, welche Ausnahmen Sie abfangen müssen.

Eine Ausnahme muss aber nicht unweigerlich zum Programmabsturz führen. Ausnahmen können *behandelt* werden. Manchmal wird eine Ausnahme aufgrund eines Bugs im Code ausgelöst (man möchte z. B. auf eine nicht vorhandene Variable zugreifen), doch manchmal können Sie eine Ausnahme auch vorhersehen. Möchten Sie eine Datei öffnen, könnte es sein, dass sie nicht existiert. Wollen Sie ein Modul importieren, ist es vielleicht nicht installiert. Haben Sie vor, eine Verbindung zu einer Datenbank aufzubauen, ist diese unter Umständen nicht erreichbar oder Sie besitzen nicht die passenden Zugangsdaten. Wenn Sie wissen, dass eine Codezeile eine Ausnahme auslösen könnte, sollten Sie sie mithilfe eines `try...except`-Blocks behandeln.

☞ Python verwendet `try...except`-Blöcke zum Behandeln und die Anweisung `raise` zum Auslösen von Ausnahmen. Java und C++ verwenden `try...catch`-Blöcke zum Behandeln und die Anweisung `throw` zum Auslösen von Ausnahmen.

Die `approximate_size()`-Funktion löst in zwei unterschiedlichen Fällen Ausnahmen aus: wenn die angegebene Größe (`size`) zu groß für die Funktion oder kleiner als `0` ist.

```
if size < 0:
    raise ValueError('number must be non-negative')
```

Die Syntax zum Auslösen einer Ausnahme ist sehr überschaubar. Verwenden Sie die `raise`-Anweisung, gefolgt vom Namen der Ausnahme und einem optionalen für Menschen lesbaren String (dient zum Debugging). Die Syntax erinnert ein wenig an den Aufruf einer Funktion. (In Wirklichkeit sind Ausnahmen als Klassen definiert. Die `raise`-Anweisung erstellt eine Instanz der Klasse `ValueError` und übergibt den String an die Initialisierungsmethode. Doch dazu später mehr.)

☞ Sie müssen eine Ausnahme nicht innerhalb der auslösenden Funktion behandeln. Behandelt eine Funktion die Ausnahme nicht, wird sie an die aufrufende Funktion übergeben usw. Wird die Ausnahme überhaupt nicht behandelt, stürzt Ihr Programm ab und Python gibt Traceback-Informationen aus, die Ihnen beim Debugging helfen. Vielleicht möchten Sie es so; das hängt ganz davon ab, wozu Ihr Programm dient.

2.7.1 Importfehler abfangen

Eine von Pythons integrierten Ausnahmen ist `ImportError`. Sie wird ausgelöst, wenn Sie versuchen ein Modul zu importieren, dies aber fehlschlägt. Es gibt eine Reihe von Gründen, warum das passieren kann; der einfachste ist der, dass das Modul nicht im Import-Suchpfad vorhanden ist. Diesen Suchpfad können Sie nutzen, um optionale Funktionen zu Ihrem Programm hinzuzufügen. Die `chardet`-Bibliothek stellt beispielsweise eine automatische Zeichencodierungserkennung zur Verfügung. Vielleicht soll Ihr Programm diese Bibliothek verwenden, wenn sie existiert, aber genauso gut laufen, wenn der Benutzer sie nicht installiert hat. Dies können Sie mithilfe eines `try...except`-Blocks erreichen.

```
try:
  import chardet
except ImportError:
  chardet = None
```

Später können Sie dann unter Verwendung einer einfachen `if`-Anweisung überprüfen, ob das `chardet`-Modul vorhanden ist:

```
if chardet:
  # tue etwas
else:
  # fahre dennoch fort
```

Die `ImportError`-Ausnahme wird oft verwendet, wenn zwei Module eine ähnliche API implementieren, eine aber der anderen vorgezogen werden soll (vielleicht ist sie schneller oder benötigt weniger Speicher). Sie können dann versuchen ein Modul zu importieren und – sollte dieser Versuch fehlschlagen – auf das andere Modul ausweichen. Im Kapitel zu XML geht es z. B. um zwei Module, die eine ähnliche API namens `ElementTree` implementieren. Das erste Modul, `lxml`, ist ein Modul eines Drittanbieters, das Sie selbst herunterladen und installieren müssen. Das zweite Modul, `xml.etree.ElementTree`, ist zwar langsamer, dafür aber Teil der Standardbibliothek von Python 3.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

Am Ende dieses `try...except`-Blocks haben Sie eines der beiden Module importiert und ihm den Namen `etree` gegeben. Da die Module eine ähnliche API implementieren, müssen Sie im restlichen Code nicht mehr überprüfen, welches Modul importiert wurde. Und da das importierte Modul immer `etree` heißt, müssen Sie Ihren Code nicht mit `if`-Anweisungen zum Auswählen unterschiedlich benannter Module verschandeln.

2.8 Ungebundene Variablen

Sehen wir uns noch einmal diese Zeile der `approximate_size()`-Funktion an:

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

Wir haben die Variable `multiple` niemals deklariert, sondern ihr einfach einen Wert zugewiesen. Das ist gut so, denn Python ermöglicht es. Was Python jedoch nicht ermöglicht, ist das Verweisen auf eine Variable, der niemals ein Wert zugewiesen wurde. Versuchen Sie es dennoch, wird eine `NameError`-Ausnahme ausgelöst.

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

Eines Tages werden Sie Python dafür danken.

2.9 Groß- und Kleinschreibung bei Namen

In Python wird immer zwischen Groß- und Kleinschreibung unterschieden. So bei: Variablennamen, Funktionsnamen, Klassennamen, Modulnamen, Ausnahmenamen. Können Sie ihm etwas zuweisen, es abrufen, aufrufen, erstellen, importieren oder auslösen, unterschiedet es zwischen Groß- und Kleinschreibung.

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

Und so weiter.

2.10 Skripte ausführen

Python-Module sind Objekte und besitzen eine Reihe nützlicher Attribute. Dies können Sie nutzen, um Ihre Module zu testen während Sie sie schreiben. Dazu müssen Sie einfach einen speziellen Codeblock einbauen, der ausgeführt wird, wenn Sie die Python-Datei auf der Kommandozeile ausführen. Sehen Sie sich die letzten Zeilen von `humansize.py` an:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

☞ Wie C verwendet auch Python == für Vergleiche und = für Zuweisungen. Anders als in C ist es in Python nicht möglich, Variablen unabsichtlich Werte zuzuweisen, obwohl sie eigentlich verglichen werden sollten.

Was ist nun also das besondere an dieser `if`-Anweisung? Nun ja, Module sind Objekte und alle Module haben ein integriertes Attribut namens `__name__`. Der `__name__` des Moduls hängt von der Verwendung des Moduls ab. Importieren Sie das Modul, dann ist `__name__` der Dateiname des Moduls (ohne Pfadangabe und Dateierweiterung).

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

Es ist aber auch möglich, das Modul als eigenständiges Programm zu starten. In einem solchen Fall enthält `__name__` den Wert `__main__`. Python überprüft diese `if`-Anweisung, befindet den Ausdruck für wahr und wird den `if`-Codeblock ausführen. Im vorliegenden Fall werden so zwei Werte angezeigt.

```
c:\home\diveintopython3> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

Kapitel 3
Native Datentypen

3.1 Los geht's

Schieben Sie Ihr erstes Python-Programm für einen Moment beiseite, während wir über Datentypen reden. In Python hat jede Variable einen Datentyp; Sie müssen ihn jedoch nicht explizit angeben. Python erkennt und überwacht den Datentyp intern, basierend auf dem Wert der ursprünglichen Zuweisung.

Python hat viele native Datentypen. Hier sind die Wichtigsten:

Boolesche Werte sind entweder `True` oder `False`.
Zahlen können Ganzzahlen (`1`, `2`, …), Fließkommazahlen (`1.1`; `1.2`; …), Brüche (`1/2`; `2/3`; ...) oder sogar komplexe Zahlen (`i`, die Quadratwurzel aus `-1`) sein.
Strings sind Folgen von Unicode-Zeichen, so z. B. ein HTML-Dokument.
Bytes und **Bytearrays**, z. B. ein JPEG-Bild.
Listen sind sortierte Folgen.
Tupel sind sortierte, unveränderliche Folgen
Sets sind unsortierte Behälter für Werte.
Dictionarys sind unsortierte Behälter für Schlüssel-Wert-Paare.

Natürlich gibt es weit mehr Typen als diese acht. Alles in Python ist ein Objekt, daher gibt es Typen wie Modul, Funktion, Klasse, Methode, Datei und sogar kompilierter Code. Einige von ihnen haben Sie bereits gesehen: Module haben Namen, Funktionen haben `docstrings` usw.

Strings und Bytes sind so wichtig – und so kompliziert – dass sie ein eigenes Kapitel erhalten haben. Sehen wir uns zunächst einmal die anderen etwas genauer an.

3.2 Boolesche Werte

Boolesche Werte sind entweder wahr (true) oder falsch (false). Python besitzt zwei Konstanten, `True` und `False`, um boolesche Werte direkt zuzuweisen. Ausdrücke können auch boolesche Werte zurückgeben. In bestimmten Situationen (so z. B. bei `if`-Anweisungen) erwartet Python, dass ein Ausdruck einen booleschen Wert

DOI 10.1007/978-3-642-04377-2_3,

zurückliefert. Diese Situationen werden boolesche Kontexte (*boolean contexts*) genannt. Sie können nahezu jeden beliebigen Ausdruck in einem booleschen Kontext verwenden; Python versucht dann, dessen Wahrheitswert zu ermitteln. Verschiedene Datentypen besitzen verschiedene Regeln darüber, welche Werte wahr und welche falsch sind. (Dies wird mehr Sinn ergeben, wenn wir später in diesem Kapitel einige konkrete Beispiele betrachten.)

Sehen Sie sich diesen Ausschnitt von `humansize.py` an:

```
if size < 0:
    raise ValueError('number must be non-negative')
```

`size` ist eine Ganzzahl, `0` ist eine Ganzzahl und < ist ein numerischer Operator. Das Ergebnis des Ausdrucks `size < 0` ist immer ein boolescher Wert. Dies können Sie selbst in der Python-Shell überprüfen.

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
>>> size = -1
>>> size < 0
True
```

Aufgrund einiger Überbleibsel aus Python 2 können boolesche Werte als Zahlen behandelt werden. `True` ist `1`; `False` ist `0`.

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Nein, nein, nein! Tun Sie das nicht. Vergessen Sie es am besten sofort wieder.

3.3 Zahlen

Zahlen sind der absolute Wahnsinn. Es gibt eine immense Auswahl. Python unterstützt sowohl Ganz-, als auch Fließkommazahlen. Es gibt keine Typendeklaration, um sie zu unterscheiden; Python unterscheidet sie nach dem Vorhandensein oder Nichtvorhandensein eines Dezimalpunkts.

```
>>> type(1)                ①
<class 'int'>
>>> isinstance(1, int)     ②
True
>>> 1 + 1                  ③
2
>>> 1 + 1.0                ④
2.0
>>> type(2.0)
<class 'float'>
```

① Die `type()`-Funktion können Sie nutzen, um den Typ jedes Werts und jeder Variablen zu überprüfen. Wie Sie wahrscheinlich erwartet haben, ist `1` ein `int`.

② Sie können auch die Funktion `isinstance()` nutzen, um zu prüfen, ob ein Wert oder eine Variable von einem bestimmten Datentyp ist.

③ Addieren Sie einen `int`-Wert zu einem `int`-Wert, kommt dabei ein `int`-Wert heraus.

④ Addiert man einen `int`-Wert zu einem `float`-Wert, so erhält man einen `float`-Wert. Python wandelt den `int`-Wert in einen `float`-Wert um, um die Addition auszuführen. Als Ergebnis wird dann ein `float`-Wert zurückgegeben.

3.3.1 `int`- in `float`-Werte umwandeln und anders herum

Wie Sie gerade gesehen haben, wandeln manche Operatoren (wie die Addition) bei Bedarf Ganzzahlen in Fließkommazahlen um. Sie können sie aber außerdem auch selbst umwandeln.

```
>>> float(2)               ①
2.0
>>> int(2.0)               ②
2
>>> int(2.5)               ③
2
>>> int(-2.5)              ④
-2
>>> 1.12345678901234567890  ⑤
1.1234567890123457
>>> type(1000000000000000)  ⑥
<class 'int'>
```

① Durch Aufrufen der `float()`-Funktion können Sie einen `int`-Wert explizit in einen `float`-Wert umwandeln.

② Sie können ebenfalls einen `float`-Wert in einen `int`-Wert umwandeln. Dazu rufen Sie einfach die `int()`-Funktion auf.

③ Die `int()`-Funktion rundet den Wert nicht, sondern schneidet die Nachkommastellen einfach ab.

④ Die `int()`-Funktion schneidet die Nachkommastellen negativer Zahlen in Richtung `0` ab (aus `-5,8` wird so `-5`). Es ist eine wirkliche Abschneid-, keine Rundungsfunktion.

⑤ Fließkommazahlen sind auf 15 Nachkommastellen genau.

⑥ Ganzzahlen können beliebig groß sein.

3.3.2 Einfache Rechenoperationen

Mit Zahlen können Sie alle möglichen Dinge tun.

```
>>> 11 / 2      ①
5.5
>>> 11 // 2     ②
5
>>> -11 // 2    ③
-6
>>> 11.0 // 2   ④
5.0
>>> 11 ** 2     ⑤
121
>>> 11 % 2      ⑥
1
```

① Der `/`-Operator führt Fließkommadivisionen aus. Er gibt einen `float`-Wert zurück, auch wenn Dividend und Divisor `int`-Werte sein sollten.

② Der `//`-Operator führt eine spezielle Ganzzahldivision aus. Ist das Ergebnis positiv, können Sie es sich so vorstellen, als wäre das Ergebnis auf `0` Nachkommastellen abgeschnitten (nicht gerundet); seien Sie damit aber vorsichtig.

③ Führen Sie eine Ganzzahldivision mit negativen Zahlen aus, so rundet der `//`-Operator „auf“ zur nächsten Ganzzahl. Mathematisch gesehen rundet er „ab“, da `-6` weniger ist als `-5`. Es könnte Sie jedoch verwirren, wenn Sie erwarten, dass das Ergebnis `-5` ist.

④ Der `//`-Operator gibt nicht immer eine Ganzzahl zurück. Ist entweder der Dividend oder der Divisor ein `float`, so wird zwar dennoch zur nächsten Ganzzahl gerundet, doch der tatsächliche Rückgabewert ist ein `float`.

⑤ Der `**`-Operator bedeutet „hoch“. 11^2 ist `121`.

⑥ Der `%`-Operator gibt den Rest einer Ganzzahldivision zurück. `11` geteilt durch `2` ergibt `5` mit einem Rest von `1`. Der `%`-Operator liefert hier also `1`.

3.3.3 Brüche

Python ist nicht auf Ganz- und Fließkommazahlen beschränkt. Es kann all die verrückten Dinge tun, die Sie im Mathematikunterricht gelernt und sofort wieder vergessen haben.

```
>>> import fractions              ①
>>> x = fractions.Fraction(1, 3)  ②
>>> x
Fraction(1, 3)
>>> x * 2                         ③
Fraction(2, 3)
>>> fractions.Fraction(6, 4)      ④
Fraction(3, 2)
>>> fractions.Fraction(0, 0)      ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fractions.py", line 96, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(0, 0)
```

① Um Brüche zu nutzen, müssen Sie das Modul `fractions` importieren.

② Erstellen Sie ein `Fraction`-Objekt und übergeben Sie den Zähler und den Nenner, um einen Bruch zu erzeugen.

③ Sie können alle gewöhnlichen mathematischen Operationen auf Brüche anwenden. Jede Operation gibt ein neues `Fraction`-Objekt zurück. `2 * (1/3) = (2/3)`

④ Das `Fraction`-Objekt kürzt Brüche automatisch. `(6/4) = (3/2)`

⑤ Python erstellt keine Brüche mit `0` im Nenner.

3.3.4 Trigonometrie

Auch grundlegende Trigonometrie ist in Python möglich.

```
>>> import math
>>> math.pi                 ①
3.1415926535897931
>>> math.sin(math.pi / 2)   ②
1.0
>>> math.tan(math.pi / 4)   ③
0.99999999999999989
```

① Das Modul `math` beinhaltet eine Konstante für π, das Verhältnis des Umfangs eines Kreises zu seinem Durchmesser.

② Außerdem beinhaltet das `math`-Modul alle grundlegenden trigonometrischen Funktionen, wie `sin()`, `cos()`, `tan()` und Abwandlungen wie `asin()`.

③ Beachten Sie aber, dass Python keine unendliche Genauigkeit besitzt. `tan(π/4)` beispielsweise sollte `1.0` ergeben, nicht `0.99999999999999989`.

3.3.5 Zahlen in einem booleschen Kontext

Sie können Zahlen auch in einem booleschen Kontext verwenden, so zum Beispiel in einer `if`-Anweisung. Null bedeutet `False`, Werte ungleich null sind `True`.

```
>>> def is_it_true(anything):                ①
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(1)                            ②
yes, it's true
>>> is_it_true(-1)
yes, it's true
>>> is_it_true(0)
no, it's false
>>> is_it_true(0.1)                          ③
yes, it's true
>>> is_it_true(0.0)
no, it's false
>>> import fractions
>>> is_it_true(fractions.Fraction(1, 2))     ④
yes, it's true
>>> is_it_true(fractions.Fraction(0, 1))
no, it's false
```

① Wussten Sie, dass Sie in der interaktiven Shell von Python Funktionen definieren können? Drücken Sie einfach am Ende jeder Zeile die Eingabetaste. Um die Funktion abzuschließen, drücken Sie die Eingabetaste in einer leeren Zeile.

② In einem booleschen Kontext ergeben Werte ungleich `0` `True`; `0` ist `False`.

③ Fließkommazahlen ungleich `0.0` sind `True`; `0.0` ist `False`. Seien Sie damit vorsichtig! Wenn nur der kleinste Rundungsfehler auftritt (was, wie Sie schon gesehen haben, nicht unmöglich ist), wird Python `0.0000000000001` statt `0` auswerten und `True` zurückgeben.

④ Auch Brüche können in einem booleschen Kontext verwendet werden. `Fraction(o,n)` ist für alle Werte `n` `False`. Alle anderen Brüche sind `True`.

3.4 Listen

Listen sind die Arbeitspferde unter Pythons Datentypen. Wenn ich „Liste“ sage, denken Sie vielleicht „ein Array, dessen Größe ich im Voraus angeben muss und das nur Werte desselben Typs enthalten kann usw.“ Denken Sie das nicht. Listen sind viel cooler als das.

☞ Eine Liste in Python ist wie ein Array in Perl 5. In Perl 5 beginnen Variablen die ein Array enthalten immer mit einem `@`-Zeichen; in Python können Sie Variablen beliebige Namen geben und Python kümmert sich um den Rest.

☞ Eine Liste in Python ist viel mehr als ein Array in Java (Sie können es jedoch als solches benutzen, wenn Ihnen dies genügt). Sie ähnelt eher der Klasse `ArrayList`, die beliebige Objekte enthalten und dynamisch neue Elemente aufnehmen kann.

3.4.1 Erstellen einer Liste

Das Erstellen einer Liste ist einfach: Benutzen Sie eckige Klammern, um eine Liste kommagetrennter Werte einzuschließen.

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example']  ①
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[0]                                        ②
'a'
>>> a_list[4]                                        ③
'example'
>>> a_list[-1]                                       ④
'example'
>>> a_list[-3]                                       ⑤
'mpilgrim'
```

① Zuerst definieren wir eine Liste mit fünf Elementen. Beachten Sie, dass sie ihre ursprüngliche Reihenfolge behalten. Dies ist kein Fehler. Eine Liste ist eine geordnete Menge von Elementen.

② Eine Liste kann wie ein Array verwendet werden. Das erste Element jeder nicht-leeren Liste ist immer `a_list[0]`.

③ Das letzte Element dieser Liste ist `a_list[4]`, da Listen immer mit dem Index `0` beginnen.

④ Ein negativer Index erlaubt den Zugriff vom Ende der Liste aus. Das letzte Element einer gefüllten Liste ist immer `a_list[-1]`.

⑤ Finden Sie den negativen Index verwirrend, stellen Sie ihn sich wie folgt vor: `a_list[-n] == a_list[len(a_list) - n]`. In dieser Liste gilt also: `a_list[-3] == a_list[5 - 3] == a_list[2]`.

3.4.2 Slicing einer Liste

Haben Sie eine Liste definiert, so können Sie jeden Teil der Liste als neue Liste erhalten. Dies wird als *Slicing* bezeichnet.

```
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[1:3]            ①
['b', 'mpilgrim']
>>> a_list[1:-1]           ②
['b', 'mpilgrim', 'z']
>>> a_list[0:3]            ③
['a', 'b', 'mpilgrim']
>>> a_list[:3]             ④
['a', 'b', 'mpilgrim']
>>> a_list[3:]             ⑤
['z', 'example']
>>> a_list[:]              ⑥
['a', 'b', 'mpilgrim', 'z', 'example']
```

① Sie können einen Teil einer Liste, „Slice“ genannt, erhalten, indem Sie zwei Indizes angeben. Der Rückgabewert ist eine neue Liste, die alle Elemente vom ersten Slice-Index (in unserem Fall `a_list[1]`) bis zum zweiten Slice-Index (in unserem Fall `a_list[3]`) enthält. Der zweite Slice-Index ist jedoch nicht Teil der Liste.

② Slicing funktioniert auch dann, wenn ein oder beide Indizes negativ sind. Sollte es Ihnen helfen, können Sie sich dies so vorstellen: Liest man die Liste von links nach rechts, so gibt der erste Slice-Index das Element an, das Sie möchten. Der zweite Slice-Index gibt das erste Element an, das Sie nicht möchten. Der Rückgabewert ist alles dazwischen.

③ Listen beginnen mit dem Index `0`. `a_list[0:3]` gibt also die ersten drei Elemente der Liste zurück. Begonnen wird hier bei `a_list[0]`; das letzte Element ist `a_list[2]`.

④ Ist der Slice-Index `0`, so können Sie ihn einfach weglassen. Python nimmt dann automatisch `0` als Index. `a_list[:3]` ist also dasselbe wie `a_list[0:3]`, da als Startindex `0` angenommen wird.

⑤ Entsprechend können Sie den zweiten Index auslassen, wenn dieser der Länge der Liste entspricht. `a_list[3:]` ist dasselbe wie `a_list[3:5]`, da diese Liste fünf Elemente enthält. Hier enthält Python eine wunderbare Symmetrie. In dieser Liste gibt `a_list[:3]` die ersten drei Elemente zurück und `a_list[3:]` die letzten zwei Elemente. `a_list[:n]` gibt also immer die ersten `n` Elemente zurück und `a_list[n:]` den Rest, ganz egal wie lang die Liste sein mag.

⑥ Werden beide Slice-Indizes weggelassen, so sind in der neuen Liste alle Elemente enthalten. `a_list[:]` erstellt eine vollständige Kopie der ursprünglichen Liste.

3.4.3 Elemente zu einer Liste hinzufügen

Es gibt vier Möglichkeiten, einer Liste Elemente hinzuzufügen.

```
>>> a_list = ['a']
>>> a_list = a_list + [2.0, 3]    ①
>>> a_list                        ②
['a', 2.0, 3]
>>> a_list.append(True)           ③
>>> a_list
['a', 2.0, 3, True]
>>> a_list.extend(['four', 'Ω'])  ④
>>> a_list
['a', 2.0, 3, True, 'four', 'Ω']
>>> a_list.insert(0, 'Ω')         ⑤
>>> a_list
['Ω', 'a', 2.0, 3, True, 'four', 'Ω']
```

① Der +-Operator verkettet Listen. Eine Liste kann eine unbegrenzte Anzahl an Elementen enthalten; es existiert keine Größenbeschränkung (außer der verfügbare Speicher). Spielt freier Arbeitsspeicher aber eine Rolle, sollten Sie bedenken, dass die Listenverkettung eine zweite Liste im Speicher erstellt. Im vorliegenden Fall wird diese neue Liste sofort der bereits vorhandenen Variable `a_list` zugewiesen. Diese Codezeile besteht also eigentlich aus zwei Schritten – Verkettung und Zuweisung – die (temporär) sehr viel Speicher beanspruchen können, wenn Sie mit großen Listen arbeiten.

② Die Elemente der Liste können einen beliebigen Datentyp aufweisen; sie müssen nicht alle vom gleichen Typ sein. Hier haben wir eine Liste, die einen String, eine Fließkommazahl und eine Ganzzahl enthält.

③ Die `append()`-Methode fügt ein einzelnes Element am Ende der Liste hinzu. (Jetzt haben wir vier verschiedene Datentypen in der Liste!)

④ Listen sind als Klassen implementiert. Das „Erstellen" einer Liste ist in Wirklichkeit das Instanziieren einer Klasse. Als solche besitzt eine Liste Methoden, die man verwenden kann, um mit der Liste zu arbeiten. Die `extend()`-Methode nimmt ein Argument entgegen, eine Liste, und hängt alle Elemente dieser Liste an die ursprüngliche Liste an.

⑤ Die `insert()`-Methode fügt ein einzelnes Element in die Liste ein. Das erste Argument ist der Index des ersten Elements in der Liste, das in seiner Position verschoben wird. Listenelemente müssen nicht einzigartig ein; nun existieren beispielsweise zwei Elemente mit dem Wert `'Ω'`, `a_list[0]` und `a_list[6]`.

Lassen Sie uns einen näheren Blick auf den Unterschied zwischen `append()` und `extend()` werfen.

```
>>> a_list = ['a', 'b', 'c']
>>> a_list.extend(['d', 'e', 'f'])  ①
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(a_list)                     ②
6
>>> a_list[-1]
'f'
>>> a_list.append(['g', 'h', 'i'])  ③
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
>>> len(a_list)                     ④
7
>>> a_list[-1]
['g', 'h', 'i']
```

① Die `extend()`-Methode nimmt ein Argument entgegen. Dieses Argument ist immer eine Liste und jedes Element dieser Liste wird zu `a_list` hinzugefügt.

② Beginnen Sie mit einer Liste aus drei Elementen und erweitern (*extend*) diese um eine Liste mit drei weiteren Elementen, so erhalten Sie eine Liste aus sechs Elementen.

③ `append()` hingegen übernimmt unbegrenzt viele Argumente, welche von jeglichem Datentyp sein können. Hier rufen wir die `append()`-Methode mit einem Argument auf, einer Liste aus drei Elementen.

④ Beginnen Sie mit einer Liste, die sechs Elemente enthält, und hängen eine Liste daran an (*append*), so erhalten Sie ... eine Liste aus sieben Elementen. Warum sieben? Das letzte Element (das Sie gerade angehängt haben) ist selbst eine Liste. Listen können jede Art von Daten enthalten, auch andere Listen. Das könnte genau das sein, was Sie möchten, oder auch nicht. Doch es ist das, wonach Sie gefragt haben.

3.4.4 Innerhalb einer Liste nach Werten suchen

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list.count('new')       ①
2
>>> 'new' in a_list           ②
True
>>> 'c' in a_list
False
>>> a_list.index('mpilgrim')  ③
3
>>> a_list.index('new')       ④
2
>>> a_list.index('c')         ⑤
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
```

① Wie Sie vielleicht schon erwartet haben, gibt die `count()`-Methode die Anzahl der Vorkommen eines bestimmten Werts in der Liste zurück.

② Verwenden Sie den `in`-Operator, um lediglich zu prüfen, ob ein bestimmter Wert überhaupt in der Liste enthalten ist. Dieser arbeitet etwas schneller als die `count()`-Methode. Er gibt `True` zurück, wenn der Wert in der Liste vorhanden ist, oder `False`, wenn er sich nicht darin befindet. Er wird Ihnen nicht mitteilen, wo sich der Wert in der Liste befindet.

③ Wenn Sie genau wissen müssen, an welcher Stelle der Liste sich ein Wert befindet, rufen Sie die `index()`-Methode auf. In der Standardeinstellung durchsucht diese die gesamte Liste. Sie können jedoch auch ein zweites Argument übergeben, das als Startindex genutzt wird; ja, sogar ein drittes Argument ist möglich, um den Index anzugeben, bei dem die Suche enden soll.

④ Die `index()`-Methode sucht nach dem ersten Vorkommen eines Wertes in der Liste. In unserem Beispiel kommt `'new'` in der Liste zweimal vor: in `a_list[2]` und `a_list[4]`. Die `index()`-Methode gibt aber nur den Index des ersten Vorkommens zurück.

⑤ Wie Sie vielleicht nicht erwartet haben, löst das Nichtvorhandensein des gesuchten Wertes eine Ausnahme aus.

Hier unterscheidet sich Python von den meisten anderen Programmiersprachen, die in einem solchen Fall einen ungültigen Index (wie `-1`) zurückgeben. Auch wenn das im ersten Moment ärgerlich erscheinen mag, denke ich, dass Sie die Vorzüge dieser Methode schnell erkennen werden. Es bedeutet nämlich, dass Ihr Programm schon beim Ursprung des Problems abstürzt und nicht erst später, ohne dass man erkennt, woran es liegt.

3.4.5 Elemente aus einer Liste entfernen

Listen können automatisch erweitert und verkürzt werden. Die Erweiterung haben Sie bereits kennengelernt. Es gibt ebenso verschiedene Möglichkeiten, Elemente aus einer Liste zu entfernen.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list[1]
'b'
>>> del a_list[1]          ①
>>> a_list
['a', 'new', 'mpilgrim', 'new']
>>> a_list[1]              ②
'new'
```

① Die Anweisung `del` können Sie zum Entfernen eines bestimmten Elements aus der Liste verwenden.

② Der Zugriff auf Index `1` nachdem dieser gelöscht wurde, führt nicht zu einem Fehler. Die anderen Elemente verändern ihre Indizes und füllen diese Lücke.

Sie kennen den Index nicht? Kein Problem; dann verwenden Sie eben den Wert selbst.

```
>>> a_list.remove('new')  ①
>>> a_list
['a', 'mpilgrim', 'new']
>>> a_list.remove('new')  ②
>>> a_list
['a', 'mpilgrim']
>>> a_list.remove('new')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

① Sie können ein Element auch mithilfe der Methode `remove()` entfernen. Die `remove()`-Methode übernimmt einen Wert und entfernt das erste Vorkommen dieses Wertes aus der Liste. Wieder einmal füllen die restlichen Elemente die Lücke. Listen haben niemals Lücken.

② Sie können die `remove()`-Methode so oft aufrufen, wie Sie möchten. Sie wird jedoch eine Ausnahme auslösen, wenn Sie versuchen einen nicht vorhandenen Wert zu entfernen.

3.4.6 Elemente aus einer Liste entfernen: Bonusrunde

Eine weitere interessante Listenmethode ist `pop()`. Mithilfe der `pop()`-Methode kann man Elemente aus einer Liste entfernen; aber diesmal mit einem Pfiff.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim']
>>> a_list.pop()   ①
'mpilgrim'
>>> a_list
['a', 'b', 'new']
>>> a_list.pop(1)  ②
'b'
>>> a_list
['a', 'new']
>>> a_list.pop()
'new'
>>> a_list.pop()
'a'
>>> a_list.pop()   ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

① Wird die `pop()`-Methode ohne Argumente aufgerufen, entfernt sie das letzte Element der Liste und gibt den entfernten Wert zurück.

② Mit `pop()` können Sie beliebige Elemente aus der Liste entfernen. Übergeben Sie der Methode einfach einen Index. Daraufhin wird dieses Element entfernt, die „Lücke geschlossen“ und der entfernte Wert zurückgegeben.

③ Rufen Sie `pop()` mit einer leeren Liste auf, wird eine Ausnahme ausgelöst.

3.4.7 Listen in einem booleschen Kontext

Sie können eine Liste auch in einem booleschen Kontext verwenden, so zum Beispiel in einem `if`-Ausdruck.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true([])             ①
no, it's false
>>> is_it_true(['a'])          ②
yes, it's true
>>> is_it_true([False])        ③
yes, it's true
```

① In einem booleschen Kontext verwendet, ergibt eine leere Liste `False`.

② Jede Liste mit mindestens einem Element ergibt `True`.

③ Jede Liste mit mindestens einem Element ergibt `True`, egal welchen Wert das Element hat.

3.5 Tupel

Ein Tupel ist eine unveränderliche Liste. Ein Tupel kann nach der Erstellung nicht mehr verändert werden.

```
>>> a_tuple = ("a", "b", "mpilgrim", "z", "example")  ①
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple[0]                                        ②
'a'
>>> a_tuple[-1]                                       ③
'example'
>>> a_tuple[1:3]                                      ④
('b', 'mpilgrim')
```

① Ein Tupel wird genauso definiert wie eine Liste, abgesehen davon, dass die Menge der Elemente in normale Klammern eingeschlossen wird und nicht in eckige Klammern.

② Die Elemente eines Tupels besitzen eine definierte Reihenfolge, genau wie bei einer Liste. Die Indizes eines Tupels beginnen bei `0`, genau wie bei einer Liste, das erste Element eines gefüllten Tupels ist also immer `a_tuple[0]`.

③ Negative Indizes werden vom Ende des Tupels gezählt, genau wie bei einer Liste.

④ Auch Slicing funktioniert genau wie bei einer Liste. Slicen Sie eine Liste, so erhalten Sie eine neue Liste; slicen Sie ein Tupel, so erhalten Sie ein neues Tupel.

Der Hauptunterschied zwischen Tupeln und Listen besteht darin, dass Tupel nicht verändert werden können. Technisch gesehen sind Tupel also unveränderlich. Praktisch ist es so, dass sie gar keine Methoden besitzen, die eine Veränderung erlauben würden. Listen haben Methoden wie `append()`, `extend()`, `insert()`, `remove()` und `pop()`. Tupel haben solche Funktionen nicht. Sie können ein Tupel slicen (da dies ein neues Tupel erstellt) und Sie können prüfen, ob ein Tupel einen bestimmten Wert enthält (weil dadurch das Tupel nicht verändert wird) und … das war's.

```
# Fortsetzung des vorherigen Beispiels
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple.append("new")               ①
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> a_tuple.remove("z")                 ②
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> a_tuple.index("example")            ③
4
>>> "z" in a_tuple                      ④
True
```

① Sie können einem Tupel keine Elemente hinzufügen. Tupel besitzen keine `append()`- oder `extend()`-Methode.

② Sie können aus einem Tupel keine Elemente entfernen. Tupel besitzen keine `remove()`- oder `pop()`-Methode.

③ Sie können Elemente eines Tupels suchen, da das Tupel dadurch nicht verändert wird.

④ Sie können außerdem den `in`-Operator verwenden, um zu prüfen, ob in dem Tupel ein Element existiert.

Wozu sind Tupel nun also zu gebrauchen?

- Tupel sind schneller als Listen. Wollen Sie eine konstante Wertemenge definieren und diese lediglich durchlaufen, verwenden Sie ein Tupel statt einer Liste.
- Ihr Code wird sicherer, wenn Sie Daten, die nicht geändert werden müssen, „schreibgeschützt“ anlegen. Die Verwendung eines Tupels anstatt einer Liste lässt sich mit einer indirekten `assert`-Anweisung vergleichen, die anzeigt, dass die Daten konstant sind und eine Änderung dessen ausdrücklich erwünscht (und durch eine spezielle Funktion erreicht) sein muss.
- Manche Tupel können als Schlüssel eines Dictionarys verwendet werden, wie Sie später in diesem Kapitel noch sehen werden. (Listen können niemals als Dictionary-Schlüssel verwendet werden.)

☞ Tupel können in Listen umgewandelt werden und umgekehrt. Die integrierte Funktion `tuple()` übernimmt eine Liste und gibt ein Tupel mit denselben Elementen zurück. Die Funktion `list()` übernimmt ein Tupel und gibt eine Liste zurück. Das bedeutet, `tuple()` „friert eine Liste ein“ und `list()` „taut ein Tupel auf“.

3.5.1 Tupel in einem booleschen Kontext

Sie können Tupel in einem booleschen Kontext, wie einer `if`-Anweisung, verwenden.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(())              ①
no, it's false
>>> is_it_true(('a', 'b'))      ②
yes, it's true
>>> is_it_true((False,))        ③
yes, it's true
>>> type((False))               ④
<class 'bool'>
>>> type((False,))
<class 'tuple'>
```

① In einem booleschen Kontext ergibt ein leeres Tupel `False`.

② Ein Tupel mit mindestens einem Element ist `True`.

③ Ein Tupel mit mindestens einem Element ist `True`. Der Wert des Elements spielt keine Rolle. Doch was macht das Komma da?

④ Um ein Tupel mit nur einem Element zu erstellen, müssen Sie nach dem Wert ein Komma angeben. Ohne dieses Komma nimmt Python an, dass es sich lediglich um ein zusätzlich Paar Klammern handelt. Das ist zwar harmlos, erstellt aber kein Tupel.

3.5.2 Mehrere Werte auf einmal zuweisen

Hier ist eine coole Programmierabkürzung: In Python können Sie mithilfe eines Tupels mehrere Werte auf einmal zuweisen.

```
>>> v = ('a', 2, True)
>>> (x, y, z) = v        ①
>>> x
'a'
>>> y
2
>>> z
True
```

① `v` ist ein Tupel aus drei Elementen und `(x, y, z)` ein Tupel aus drei Variablen. Weisen Sie nun ein Tupel dem anderen zu, so wird jeder Wert von `v` der Reihe nach einer der Variablen zugewiesen.

Dies kann man für alle möglichen Dinge verwenden. Stellen Sie sich vor, Sie möchten einer Reihe Variablen Namen zuweisen. In Python können Sie die integrierte Funktion `range()` mit mehrfacher Variablenzuweisung verwenden, um fortlaufende Werte schnell zuzuweisen.

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7)  ①
>>> MONDAY                                                                        ②
0
>>> TUESDAY
1
>>> SUNDAY
6
```

① Die integrierte `range()`-Funktion erstellt eine Folge von Ganzzahlen. (Technisch gesehen gibt die `range()`-Funktion einen Iterator zurück, keine Liste oder ein Tupel, doch über den Unterschied erfahren Sie später mehr.) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` und `SUNDAY` sind die von Ihnen definierten Variablen. (Dieses Beispiel stammt aus dem Modul `calendar`, welches ein spaßiges, kleines Modul zum Ausgeben von Kalendern ist; ganz so wie das UNIX-Programm `cal`. Das `calendar`-Modul definiert für jeden Wochentag eine Ganzzahlkonstante.)

② Nun hat jede Variable ihren Wert: `MONDAY` ist `0`, `TUESDAY` ist `1` usw.

Sie können die mehrfache Variablenzuweisung auch zum Anlegen von Funktionen verwenden, die mehrere Werte zurückgeben sollen. Dazu geben Sie einfach ein Tupel all dieser Werte zurück. Die aufrufende Funktion kann dieses als ein Tupel behandeln, aber auch die Werte einzelnen Variablen zuweisen. Viele Python-Bibliotheken gehen so vor. So auch das Modul `os`, das Sie im nächsten Kapitel kennenlernen werden.

3.6 Sets

Ein Set (dt. Menge) ist ein ungeordneter Wertebehälter. Ein Set kann Werte jedes beliebigen Datentyps enthalten. Sind zwei Sets vorhanden, können Sie die bekannten Mengenoperationen ausführen, also Vereinigung, Durchschnitt und Differenz.

3.6.1 Ein Set erstellen

Ein Set zu erstellen ist einfach.

```
>>> a_set = {1}      ①
>>> a_set
{1}
>>> type(a_set)      ②
<class 'set'>
>>> a_set = {1, 2}   ③
>>> a_set
{1, 2}
```

① Um ein Set mit einem Wert zu erstellen, schließen Sie den Wert in geschweifte Klammern ein.

② Sets sind als Klassen implementiert, doch machen Sie sich darüber noch keine Gedanken.

③ Möchten Sie ein Set mit mehreren Werten erstellen, trennen Sie die Werte durch Kommas und schließen Sie alles in geschweifte Klammern ein.

Sie können ein Set auch aus einer Liste erstellen.

```
>>> a_list = ['a', 'b', 'mpilgrim', True, False, 42]
>>> a_set = set(a_list)                              ①
>>> a_set                                            ②
{'a', False, 'b', True, 'mpilgrim', 42}
>>> a_list                                           ③
['a', 'b', 'mpilgrim', True, False, 42]
```

① Zum Erstellen eines Sets aus einer Liste verwenden Sie die Funktion `set()`. (Schlaumeier, die wissen wie Sets implementiert sind, werden nun darauf hinweisen, dass hier nicht wirklich eine Funktion aufgerufen, sondern eine Klasse instanziiert wird. Ich verspreche Ihnen, dass Sie den Unterschied später in diesem Buch kennenlernen werden. Im Moment genügt es, wenn Sie wissen, dass `set()` wie eine Funktion arbeitet und ein Set zurückgibt.)

② Wie ich schon sagte, kann ein Set Werte jedes beliebigen Datentyps enthalten. Außerdem sind Sets, wie ich auch bereits anmerkte, ungeordnet. Dieses Set erinnert sich nicht an die ursprüngliche Reihenfolge der Werte in der Liste, die benutzt wurde, um dieses Set zu erstellen. Würden Sie Werte zu diesem Set hinzufügen, würde das Set sich nicht daran erinnern, in welcher Reihenfolge sie hinzugefügt wurden.

③ Die ursprüngliche Liste bleibt unverändert.

Sie besitzen noch gar keine Werte? Kein Problem. Sie können ein leeres Set erstellen.

```
>>> a_set = set()    ①
>>> a_set            ②
set()
>>> type(a_set)      ③
<class 'set'>
>>> len(a_set)       ④
```

```
0
>>> not_sure = {}     ⑤
>>> type(not_sure)
<class 'dict'>
```

① Zum Erstellen eines leeren Sets rufen Sie `set()` ohne Argumente auf.

② Die Ausgabe des leeren Sets sieht etwas seltsam aus. Haben Sie vielleicht `{ }` erwartet? Dies stünde für ein leeres Dictionary, nicht für ein leeres Set. Dictionarys werden Sie später in diesem Kapitel kennenlernen.

③ Auch wenn die Darstellung seltsam anmutet, ist dies ein Set…

④ … und dieses Set hat keine Elemente.

⑤ Aufgrund einer von Python 2 übernommenen Eigenart ist es Ihnen nicht möglich, ein leeres Set durch zwei geschweifte Klammern zu erstellen. Dadurch wird tatsächlich ein leeres Dictionary erzeugt, kein leeres Set.

3.6.2 Ein Set verändern

Es gibt zwei Möglichkeiten, Werte zu einem bestehenden Set hinzuzufügen: die `add()`-Methode und die `update()`-Methode.

```
>>> a_set = {1, 2}
>>> a_set.add(4)  ①
>>> a_set
{1, 2, 4}
>>> len(a_set)    ②
3
>>> a_set.add(1)  ③
>>> a_set
{1, 2, 4}
>>> len(a_set)    ④
3
```

① Die `add()`-Methode übernimmt ein Argument, egal von welchem Datentyp, und fügt den Wert zum Set hinzu.

② Dieses Set hat nun drei Elemente.

③ Sets sind Behälter für einmalige Werte. Versuchen Sie einen Wert hinzuzufügen, der bereits vorhanden ist, so geschieht nichts. Es wird keine Ausnahme ausgelöst, es passiert einfach nichts.

④ Dieses Set hat *immer noch* drei Elemente.

```
>>> a_set = {1, 2, 3}
>>> a_set
{1, 2, 3}
>>> a_set.update({2, 4, 6})                    ①
>>> a_set                                      ②
```

```
{1, 2, 3, 4, 6}
>>> a_set.update({3, 6, 9}, {1, 2, 3, 5, 8, 13})  ③
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 13}
>>> a_set.update([10, 20, 30])                    ④
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}
```

① Die `update()`-Methode übernimmt ein Argument, ein Set, und fügt all seine Elemente zum ursprünglichen Set hinzu. Das ist so, als würden Sie die `add()`-Methode mit jedem Element des Sets aufrufen.

② Doppelte Werte werden ignoriert, da Sets keine doppelten Werte enthalten können.

③ Sie können die `update()`-Methode mit jeder beliebigen Anzahl an Argumenten aufrufen. Rufen Sie sie mit zwei Sets auf, fügt die `update()`-Methode alle Elemente jedes Sets zum ursprünglichen Set hinzu (und lässt doppelte Werte aus).

④ Die `update()`-Methode kann Objekte verschiedener Datentypen übernehmen, so auch Listen. Rufen Sie die `update()`-Methode mit einer Liste als Argument auf, so werden alle Elemente der Liste zum ursprünglichen Set hinzugefügt.

3.6.3 Elemente aus einem Set entfernen

Es gibt drei Möglichkeiten, einzelne Werte aus einem Set zu entfernen. Die ersten beiden, `discard()` und `remove()`, haben einen feinen Unterschied.

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set
{1, 3, 36, 6, 10, 45, 15, 21, 28}
>>> a_set.discard(10)                          ①
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.discard(10)                          ②
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.remove(21)                           ③
>>> a_set
{1, 3, 36, 6, 45, 15, 28}
>>> a_set.remove(21)                           ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 21
```

① Die `discard()`-Methode übernimmt einen Wert als Argument und entfernt diesen Wert aus dem Set.

② Rufen Sie die `discard()`-Methode mit einem nicht vorhandenen Wert auf, so geschieht nichts. Kein Fehler. Es passiert einfach nichts.

③ Auch die `remove()`-Methode übernimmt einen Wert als Argument und entfernt diesen aus dem Set.

④ Das ist der Unterschied: Ist der Wert im Set nicht vorhanden, löst die `remove()`-Methode eine `KeyError`-Ausnahme aus.

Sets haben, wie Listen, eine `pop()`-Methode.

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set.pop()                              ①
1
>>> a_set.pop()
3
>>> a_set.pop()
36
>>> a_set
{6, 10, 45, 15, 21, 28}
>>> a_set.clear()                            ②
>>> a_set
set()
>>> a_set.pop()                              ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

① Die `pop()`-Methode entfernt einen einzelnen Wert aus einem Set und gibt diesen Wert zurück. Da Sets aber ungeordnet sind, gibt es keinen „letzten“ Wert des Sets und Sie können nicht kontrollieren, welcher Wert entfernt wird. Das ist reiner Zufall.

② Die `clear()`-Methode entfernt alle Werte eines Sets und hinterlässt somit ein leeres Set. Dies ist so, als führten Sie `a_set = set()` aus, was ein neues leeres Set erstellen und den vorherigen Wert von `a_set` überschreiben würde.

③ Bei dem Versuch einen Wert aus einem leeren Set zu „poppen“ wird eine `KeyError`-Ausnahme ausgelöst.

3.6.4 Einfache Mengenoperationen

Pythons Set-Datentyp unterstützt eine Reihe einfacher Mengenoperationen.

```
>>> a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
>>> 30 in a_set                                                    ①
True
>>> 31 in a_set
False
>>> b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
>>> a_set.union(b_set)                                             ②
{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}
>>> a_set.intersection(b_set)                                      ③
{9, 2, 12, 5, 21}
>>> a_set.difference(b_set)                                        ④
{195, 4, 76, 51, 30, 127}
>>> a_set.symmetric_difference(b_set)                              ⑤
{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
```

① Um zu prüfen, ob ein bestimmter Wert in einem Set vorhanden ist, verwenden Sie den `in`-Operator. Dieser funktioniert hier auf die gleiche Weise wie bei Listen.

② Die `union()`-Methode gibt ein neues Set zurück, das alle Elemente enthält, die entweder in dem einen oder dem anderen Set vorhanden sind.

③ Die `intersection()`-Methode gibt ein neues Set zurück, das alle Elemente enthält, die in beiden Sets vorhanden sind.

④ Die `difference()`-Methode gibt ein neues Set zurück, das alle Elemente enthält, die in `a_set` vorhanden sind, aber nicht in `b_set`.

⑤ Die `symmetric_difference()`-Methode gibt ein neues Set zurück, das alle Elemente enthält, die in genau einem der Sets vorhanden sind.

Drei dieser Methoden sind symmetrisch.

```
# Fortsetzung des vorherigen Beispiels
>>> b_set.symmetric_difference(a_set)                                          ①
{3, 1, 195, 4, 6, 8, 76, 15, 17, 18, 51, 30, 127}
>>> b_set.symmetric_difference(a_set) == a_set.symmetric_difference(b_set)     ②
True
>>> b_set.union(a_set) == a_set.union(b_set)                                   ③
True
>>> b_set.intersection(a_set) == a_set.intersection(b_set)                     ④
True
>>> b_set.difference(a_set) == a_set.difference(b_set)                         ⑤
False
```

① Die symmetrische Differenz von `a_set` aus `b_set` sieht anders aus, als die symmetrische Differenz von `b_set` aus `a_set`. Bedenken Sie aber, dass Sets ungeordnet sind. Alle Sets, die genau dieselben Werte beinhalten werden als gleich angesehen.

② Das ist genau das, was hier passiert. Lassen Sie sich nicht von der Darstellung der Sets durch die Python-Shell ins Bockshorn jagen. Sie enthalten dieselben Werte, also sind sie gleich.

③ Die Vereinigungsmenge zweier Sets ist ebenfalls symmetrisch.

④ Die Schnittmenge zweier Sets ist ebenfalls symmetrisch.

⑤ Die Differenzmenge zweier Sets ist nicht symmetrisch. Das leuchtet ein; es ist so, als würden Sie eine Zahl von einer anderen subtrahieren. Die Reihenfolge der Operanden ist wichtig.

Schlussendlich gibt es noch einige Fragen, die Sie den Sets stellen können.

```
>>> a_set = {1, 2, 3}
>>> b_set = {1, 2, 3, 4}
>>> a_set.issubset(b_set)     ①
True
>>> b_set.issuperset(a_set)   ②
True
>>> a_set.add(5)              ③
>>> a_set.issubset(b_set)
False
>>> b_set.issuperset(a_set)
False
```

① `a_set` ist eine Teilmenge von `b_set` – alle Elemente von `a_set` sind auch Elemente von `b_set`.

② Dies ist dieselbe Frage rückwärts: `b_set` ist eine Obermenge von `a_set`, da alle Elemente von `a_set` ebenfalls Elemente von `b_set` sind.

③ Sobald Sie einen Wert zu `a_set` hinzufügen, der nicht auch in `b_set` enthalten ist, ergeben beide Auswertungen `False`.

3.6.5 Sets in einem booleschen Kontext

Sie können Sets in einem booleschen Kontext, wie einer `if`-Anweisung, verwenden.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(set())          ①
no, it's false
>>> is_it_true({'a'})          ②
yes, it's true
>>> is_it_true({False})        ③
yes, it's true
```

① In einem booleschen Kontext ergibt ein leeres Set `False`.

② Jedes Set mit mindestens einem Element ergibt `True`.

③ Jedes Set mit mindestens einem Element ergibt `True`. Der Wert des Elements spielt keine Rolle.

3.7 Dictionarys

Ein Dictionary (dt. Wörterbuch) ist eine unsortierte Menge von Schlüssel-Wert-Paaren. Fügen Sie einen Schlüssel zu einem Dictionary hinzu, so müssen Sie auch einen Wert für diesen Schlüssel hinzufügen. (Diesen Wert können Sie später immer noch ändern.) Dictionarys erlauben es Ihnen, anhand des Schlüssels einen Wert zu erhalten; andersherum geht es jedoch nicht.

☞ Ein Dictionary in Python ist wie ein Hash in Perl 5. In Perl 5 werden Variablen, die ein Hash enthalten mit einem führenden `%` benannt. In Python können Variablen beliebige Namen haben, da der Datentyp intern verwaltet wird.

3.7.1 Erstellen eines Dictionarys

Die Erstellung eines Dictionary ist einfach. Die Syntax ähnelt der von Sets, doch statt Werte werden Schlüssel-Wert-Paare definiert. Haben Sie einmal ein Dictionary erstellt, so können Sie Werte anhand ihres Schlüssels finden.

```
>>> a_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'}  ①
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['server']                                                    ②
'db.diveintopython3.org'
>>> a_dict['database']                                                  ③
'mysql'
>>> a_dict['db.diveintopython3.org']                                    ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'db.diveintopython3.org'
```

① Zuerst erstellen wir ein neues Dictionary mit zwei Elementen und weisen dieses der Variable `a_dict` zu. Jedes der Elemente ist ein Schlüssel-Wert-Paar und die gesamte Menge der Elemente wird in geschweifte Klammern gesetzt.

② `'server'` ist ein Schlüssel, der den Wert `'db.diveintopython3.org'` enthält. Dieser Wert kann über `a_dict["server"]` angesprochen werden.

③ `'database'` ist ein Schlüssel, dessen Wert `'mysql'` über `a_dict["database"]` angesprochen werden kann.

④ Über die Schlüssel können Sie an die Werte gelangen, doch umgekehrt ist dies nicht möglich. Über die Werte kann man nicht die Schlüssel ansprechen. `a_dict["server"]` ist `'db.diveintopython3.org'`, doch `a_dict["db.diveintopython3.org"]` löst eine Ausnahme aus, da `'db.diveintopython3.org'` kein Schlüssel ist.

3.7.2 Ein Dictionary verändern

Dictionarys besitzen keine vordefinierte Größenbeschränkung. Sie können einem Dictionary zu jeder Zeit neue Schlüssel-Wert-Paare hinzufügen, oder den Wert eines bestehenden Schlüssels verändern. Lassen Sie uns das vorherige Beispiel fortführen:

```
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['database'] = 'blog'  ①
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'blog'}
>>> a_dict['user'] = 'mark'      ②
>>> a_dict                       ③
{'server': 'db.diveintopython3.org', 'user': 'mark', 'database': 'blog'}
>>> a_dict['user'] = 'dora'      ④
>>> a_dict
{'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
>>> a_dict['User'] = 'mark'      ⑤
>>> a_dict
{'User': 'mark', 'server': 'db.diveintopython3.org', 'user': 'dora',
'database': 'blog'}
```

① Es ist nicht möglich, in einem Dictionary zwei gleiche Schlüssel zu haben. Durch die Zuweisung eines Wertes zu einem bestehenden Schlüssel wird der alte Wert gelöscht.

② Sie können jederzeit neue Schlüssel-Wert-Paare hinzufügen. Diese Syntax ist identisch zu der, mit der man bestehende Werte verändert.

③ Das neue Element (Schlüssel: `'user'`, Wert: `'mark'`) scheint in der Mitte des Dictionary eingefügt worden zu sein. Tatsächlich ist es jedoch Zufall, dass die Elemente im ersten Beispiel geordnet erscheinen. Es ist genauso ein Zufall, wie der, dass die Elemente nun nicht mehr geordnet sind.

④ Durch Zuweisung eines neuen Wertes zu einem bestehenden Schlüssel wird der alte Wert durch den neuen ersetzt.

⑤ Ändert diese Anweisung den Wert von `user` wieder zu „mark“? Nein! Sehen Sie sich den Schlüssel genau an – das `U` in „`User`“ ist groß geschrieben. Dictionary-

Schlüssel beachten die Groß- und Kleinschreibung. Die Anweisung erzeugt hier ein neues Schlüssel-Wert-Paar, statt ein bestehendes zu überschreiben. Für Sie sieht es vielleicht sehr ähnlich aus, doch für Python ist es ein komplett anderes Paar.

3.7.3 Dictionarys mit gemischten Werten

Dictionarys können nicht nur mit Strings verwendet werden. Die Werte können von jedem Datentyp sein, also auch Ganzzahlen, boolesche Werte, beliebige Objekte oder sogar andere Dictionarys. Innerhalb eines Dictionary müssen die Werte ebenfalls nicht von einem Typ sein. Sie können sie nach Belieben mischen. Dictionary-Schlüssel sind etwas eingeschränkter, doch auch hier können Strings, Ganzzahlen und einige andere Typen verwendet werden. Auch die Schlüssel innerhalb eines Dictionary können von verschiedenen Datentypen sein.

Sie haben sogar in Ihrem ersten Python-Programm schon ein Dictionary mit anderen Datentypen gesehen.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

Lassen Sie uns diesen Code in der interaktiven Shell analysieren.

```
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
...             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>> len(SUFFIXES)      ①
2
>>> 1000 in SUFFIXES   ②
True
>>> SUFFIXES[1000]     ③
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> SUFFIXES[1024]     ④
['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
>>> SUFFIXES[1000][3]  ⑤
'TB'
```

① Genau wie bei Listen und Sets gibt die `len()`-Funktion auch hier die Anzahl der Elemente des Dictionary zurück.

② Und genau wie bei Listen und Sets können Sie auch hier mithilfe des `in`-Operators prüfen, ob sich ein bestimmter Schlüssel im Dictionary befindet.

③ `1000` ist ein Schlüssel im `SUFFIXES`-Dictionary; sein Wert ist eine Liste von acht Elementen (acht Strings, um genau zu sein).

④ `1024` ist ebenfalls ein Schlüssel im `SUFFIXES`-Dictionary; sein Wert ist auch eine Liste von acht Elementen.

⑤ Da `SUFFIXES[1000]` eine Liste ist, können Sie die Elemente der Liste mit ihrem Index ansprechen.

3.7.4 Dictionarys in einem booleschen Kontext

Sie können Dictionarys auch in einem booleschen Kontext, wie einer `if`-Anweisung, verwenden.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true({})             ①
no, it's false
>>> is_it_true({'a': 1})       ②
yes, it's true
```

① In einem booleschen Kontext ergibt ein leeres Dictionary `False`.
② Ein Dictionary mit mindestens einem Schlüssel-Wert-Paar ergibt `True`.

3.8 `None`

`None` ist in Python eine besondere Konstante. Es ist ein Nullwert. `None` ist nicht dasselbe wie `False`. `None` ist nicht `0`. `None` ist kein leerer String. Vergleicht man `None` mit irgendetwas anderem als `None`, so ergibt dies immer `False`.

`None` ist der einzige Nullwert. Es hat seinen eigenen Datentyp (`NoneType`). Sie können `None` jeder beliebigen Variable zuweisen, doch Sie können keine eigenständigen Objekte vom Typ `NoneType` erstellen. Alle Variablen mit dem Wert `None` entsprechen einander.

```
>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ''
False
>>> None == None
```

```
True
>>> x = None
>>> x == None
True
>>> y = None
>>> x == y
True
```

3.8.1 `None` in einem booleschen Kontext

In einem booleschen Kontext ergibt `None` immer `False` und `not None` immer `True`.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(None)
no, it's false
>>> is_it_true(not None)
yes, it's true
```

Kapitel 4
Comprehensions

4.1 Los geht's

In diesem Kapitel lernen Sie List Comprehensions, Dictionary Comprehensions und Set Comprehensions kennen. Dies sind drei miteinander verbundene Konzepte, die derselben sehr mächtigen Vorgehensweise folgen. Zuerst möchte ich jedoch einen kleinen Abstecher machen und Ihnen zwei Module vorstellen, die Ihnen bei der Navigation durch Ihr Dateisystem helfen werden.

4.2 Mit Dateien und Verzeichnissen arbeiten

Python 3 enthält ein Modul namens `os`, was für „operating system" – also „Betriebssystem" – steht. Das `os`-Modul beinhaltet unzählige Funktionen, mit denen man Informationen über lokale Verzeichnisse, Dateien, Prozesse und Umgebungsvariablen abrufen kann. In manchen Fällen lassen sich diese sogar manipulieren. Python tut sein Bestes beim Versuch eine einheitliche API für alle unterstützten Betriebssysteme anzubieten, um so dafür zu sorgen, dass Ihre Programme auf jedem beliebigen Computer mit möglichst wenig plattformspezifischem Code laufen.

4.2.1 Das aktuelle Arbeitsverzeichnis

Beginnen Sie gerade erst mit Python, verbringen Sie viel Zeit in der Python-Shell. Im Verlauf dieses Buches werden Ihnen Beispiele begegnen, die folgenden Ablauf haben:

1. Importiere eines der Module im Ordner `examples`
2. Rufe eine Funktion dieses Moduls auf
3. Erkläre das Ergebnis

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_4, © Springer-Verlag Berlin Heidelberg 2010

Kennen Sie das aktuelle Arbeitsverzeichnis nicht, wird Schritt 1 möglicherweise mit einem `ImportError` fehlschlagen. Warum? Python sucht das Modul im Import-Suchpfad, wird es jedoch nicht finden, da der Ordner `examples` keines der Verzeichnisse im Suchpfad ist. Um dies zu beheben, können Sie zweierlei tun:

1. Den Ordner `examples` zum Import-Suchpfad hinzufügen
2. Als aktuelles Arbeitsverzeichnis den Ordner `examples` auswählen

Das aktuelle Arbeitsverzeichnis ist eine unsichtbare Eigenschaft, die Python ständig im Speicher hält. Ein aktuelles Arbeitsverzeichnis existiert immer, egal ob Sie sich in der Python-Shell befinden, ein eigenes Python-Skript auf der Kommandozeile starten oder ein Python-CGI-Skript auf einem Webserver laufen lassen.

Das `os`-Modul enthält zwei Funktionen zum Umgang mit dem aktuellen Arbeitsverzeichnis.

```
>>> import os                                                    ①
>>> print(os.getcwd())                                          ②
C:\Python31
>>> os.chdir('/Users/pilgrim/diveintopython3/examples')         ③
>>> print(os.getcwd())                                          ④
C:\Users\pilgrim\diveintopython3\examples
```

① Das `os`-Modul ist in Python enthalten. Sie können es jederzeit, von überall importieren.

② Verwenden Sie die Funktion `os.getcwd()`, um das aktuelle Arbeitsverzeichnis zu erhalten. Wenn Sie die grafische Python-Shell starten, ist das aktuelle Arbeitsverzeichnis das Verzeichnis, in dem sich die ausführbare Datei der Python-Shell befindet. Unter Windows hängt dies davon ab, wo Sie Python installiert haben. Das Standardverzeichnis ist hier `c:\Python31`. Starten Sie die Python-Shell über die Kommandozeile, so ist das aktuelle Arbeitsverzeichnis das Verzeichnis, in dem Sie sich befanden, als Sie `python3` gestartet haben.

③ Mit der Funktion `os.chdir()` können Sie das aktuelle Arbeitsverzeichnis ändern.

④ Ich rufe die `os.chdir()`-Funktion mit einem Pfadnamen im Linux-Stil auf (normale Slashes, kein Laufwerkbuchstabe), auch wenn ich unter Windows arbeite. Dies ist ein Beispiel dafür, wie Python versucht, die Unterschiede zwischen den Betriebssystemen zu verschleiern.

4.2.2 Mit Dateinamen und Verzeichnisnamen arbeiten

Wenn wir schon über Verzeichnisse reden, möchte ich auf das Modul `os.path` hinweisen. `os.path` beinhaltet Funktionen zur Manipulation von Datei- und Verzeichnisnamen.

```
>>> import os
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples/', 'humansize.py'))①
/Users/pilgrim/diveintopython3/examples/humansize.py
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples', 'humansize.py')) ②
/Users/pilgrim/diveintopython3/examples\humansize.py
>>> print(os.path.expanduser('~'))                                              ③
c:\Users\pilgrim
>>>    print(os.path.join(os.path.expanduser('~'),    'diveintopython3',    'examples',
'humansize.py'))  ④
c:\Users\pilgrim\diveintopython3\examples\humansize.py
```

① Mit der Funktion `os.path.join()` lässt sich aus einem oder mehreren Teilpfad(en) ein Pfadname konstruieren. Im vorliegenden Fall werden einfach Strings verkettet.

② In diesem etwas weniger trivialen Fall fügt `join` einen zusätzlichen Backslash ein, bevor der Pfadname mit dem Dateinamen verbunden wird. Ich war überaus erfreut, als ich dies entdeckte, denn `addSlashIfNecessary()` ist eine dieser dummen, kleinen Funktionen, die ich immer schreiben muss, wenn ich meine Toolbox in einer neuen Sprache aufbaue. Schreiben Sie diese dumme, kleine Funktion in Python nicht; clevere Leute haben dies bereits für Sie erledigt.

③ Die Funktion `os.path.expanduser()` baut einen Pfadnamen auf, der ~ benutzt, um das Heimatverzeichnis des aktuellen Benutzers wiederzugeben. Das funktioniert auf jeder Plattform, auf der Benutzer ein Heimatverzeichnis haben, also auch unter Linux, Mac OS X und Windows.

④ Kombinieren Sie diese Verfahren, können Sie sehr einfach Pfadnamen für Verzeichnisse und Dateien im Heimatverzeichnis des Benutzers erstellen.

`os.path` enthält außerdem Funktionen zum Aufteilen ganzer Pfad-, Verzeichnis- und Dateinamen in ihre Bestandteile.

```
>>> pathname = '/Users/pilgrim/diveintopython3/examples/humansize.py'
>>> os.path.split(pathname)                                                    ①
('/Users/pilgrim/diveintopython3/examples', 'humansize.py')
>>> (dirname, filename) = os.path.split(pathname)                              ②
>>> dirname                                                                    ③
'/Users/pilgrim/diveintopython3/examples'
>>> filename                                                                   ④
'humansize.py'
>>> (shortname, extension) = os.path.splitext(filename)                        ⑤
>>> shortname
'humansize'
>>> extension
'.py'
```

① Die `split()`-Funktion teilt einen Pfadnamen und gibt ein Tupel aus Pfad und Dateiname zurück. Erinnern Sie sich an die Methode zur mehrfachen Variablenzuweisung, mit der Sie eine Funktion mehrere Werte zurückgeben lassen können? Die Funktion `os.path.split()` macht genau das.

② Sie weisen den Rückgabewert der `split()`-Funktion einem Tupel aus zwei Variablen zu. Jede Variable erhält den Wert des entsprechenden Elements des zurückgegebenen Tupels.

③ Die erste Variable, `dirname`, erhält den Wert des ersten Tupel-Elements, das von der Funktion `os.path.split()` zurückgegeben wurde, also den Dateipfad.

④ Die zweite Variable, `filename`, erhält den Wert des zweiten Tupel-Elements, das von der Funktion `os.path.split()` zurückgegeben wurde, also den Dateinamen.

⑤ `os.path` enthält auch die Funktion `os.path.splitext()`, mit deren Hilfe ein Dateiname geteilt und ein Tupel das den Dateinamen und die Dateierweiterung enthält zurückgegeben werden kann. Die gleiche Vorgehensweise können Sie verwenden, um Dateiname und Dateierweiterung verschiedenen Variablen zuzuweisen.

4.2.3 Verzeichnisse auflisten

Ebenfalls in Pythons Standardbibliothek enthalten ist das Modul `glob`. Dieses Modul stellt eine einfache Möglichkeit bereit, programmatisch an den Inhalt eines Verzeichnisses zu gelangen. Es werden dieselben Platzhalter verwendet, die Sie vielleicht schon von der Arbeit auf der Kommandozeile kennen.

```
>>> os.chdir('/Users/pilgrim/diveintopython3/')
>>> import glob
>>> glob.glob('examples/*.xml')    ①
['examples\\feed-broken.xml',
 'examples\\feed-ns0.xml',
 'examples\\feed.xml']
>>> os.chdir('examples/')          ②
>>> glob.glob('*test*.py')         ③
['alphameticstest.py',
 'pluraltest1.py',
 'pluraltest2.py',
 'pluraltest3.py',
 'pluraltest4.py',
 'pluraltest5.py',
 'pluraltest6.py',
 'romantest1.py',
 'romantest10.py',
 'romantest2.py',
 'romantest3.py',
 'romantest4.py',
 'romantest5.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

① Das `glob`-Modul nimmt einen Platzhalter entgegen und gibt die Pfade aller Dateien und Verzeichnisse zurück, die mit diesem Platzhalter übereinstimmen. In diesem Beispiel besteht der Platzhalter aus einem Verzeichnispfad mit angehängtem `*.xml`; so werden alle `.xml`-Dateien im Verzeichnis `examples` gesucht.

② Ändern Sie das aktuelle Arbeitsverzeichnis nun auf das Unterverzeichnis `examples`. Der Funktion `os.chdir()` können relative Pfadnamen übergeben werden.

③ Ihr Suchmuster kann auch mehrere Platzhalter enthalten. In diesem Beispiel werden alle Dateien im aktuellen Arbeitsverzeichnis gesucht, die mit der Erweiterung `.py` enden und irgendwo das Wort `test` im Dateinamen haben.

4.2.4 Metadaten von Dateien erhalten

Jedes moderne Dateisystem speichert Metadaten zu jeder Datei: Erstellungsdatum, Datum der letzten Bearbeitung, Dateigröße usw. Python stellt eine API zum Zugriff auf diese Metadaten zur Verfügung. Sie müssen die Datei nicht öffnen; Sie benötigen nur den Dateinamen.

```
>>> import os
>>> print(os.getcwd())                  ①
c:\Users\pilgrim\diveintopython3\examples
>>> metadata = os.stat('feed.xml')      ②
>>> metadata.st_mtime                   ③
1247520344.9537716
>>> import time                         ④
>>> time.localtime(metadata.st_mtime)   ⑤
time.struct_time(tm_year=2009, tm_mon=7, tm_mday=13, tm_hour=17,
  tm_min=25, tm_sec=44, tm_wday=0, tm_yday=194, tm_isdst=1)
```

① Das aktuelle Arbeitsverzeichnis ist der Ordner `examples`.

② `feed.xml` ist eine Datei im `examples`-Ordner. Die Funktion `os.stat()` gibt ein Objekt zurück, das verschiedene Arten von Metadaten der Datei enthält.

③ `st_mtime` ist die Bearbeitungszeit, die jedoch ein recht unnützes Format aufweist. (Genauer gesagt sind es die Sekunden, die seit der `EPOCH` vergangen sind, also seit dem 1. Januar 1970, 00:00 Uhr UTC. Ernsthaft.)

④ Das Modul `time` ist Bestandteil von Pythons Standardbibliothek. Es enthält Funktionen zur Konvertierung von verschiedenen Zeitdarstellungen, zum Formatieren von Zeitwerten als Strings und zur Handhabung von Zeitzonen.

⑤ Die Funktion `time.localtime()` konvertiert einen Zeitwert vom Format *Sekunden-seit-der-`EPOCH`* (von der Eigenschaft `st_time`, die von der Funktion `os.stat()` zurückgegeben wurde) in ein weitaus nützlicheres Format aus Jahr, Monat, Tag, Stunde, Minute, Sekunde usw. Diese Datei wurde zuletzt am 13. Juli 2009 um ca. 17:25 Uhr bearbeitet.

```
# Fortsetzung des vorherigen Beispiels
>>> metadata.st_size                              ①
3070
>>> import humansize
>>> humansize.approximate_size(metadata.st_size)  ②
'3.0 KiB'
```

① Die `os.stat()`-Funktion gibt außerdem die Größe einer Datei zurück, die in der Eigenschaft `st_size` gespeichert wird. Die Datei `feed.xml` hat eine Größe von `3070` Bytes.

② Sie können der Funktion `approximate_size()` die Eigenschaft `st_size` übergeben.

4.2.5 Absolute Pfadnamen erstellen

Im vorherigen Beispiel gab die `glob.glob()`-Funktion eine Liste relativer Pfadnamen zurück. Beim ersten Beispiel waren dies Pfadnamen wie `'examples\feed.xml'`, beim zweiten Beispiel noch kürzere Pfadnamen wie `'romantest1.py'`. Bewegen Sie sich innerhalb desselben aktuellen Arbeitsverzeichnisses, lassen sich mit diesen relativen Pfadnamen problemlos Dateien öffnen oder Metadaten abrufen. Möchten Sie aber einen absoluten Pfadnamen erstellen, d. h. einen Pfadnamen, der alle Verzeichnisse bis zum `root`-Verzeichnis oder Laufwerksbuchstaben beinhaltet, so müssen Sie die Funktion `os.path.realpath()` nutzen.

```
>>> import os
>>> print(os.getcwd())
c:\Users\pilgrim\diveintopython3\examples
>>> print(os.path.realpath('feed.xml'))
c:\Users\pilgrim\diveintopython3\examples\feed.xml
```

4.3 List Comprehensions

Eine sogenannte List Comprehension stellt eine einfache Möglichkeit dar, eine Liste unter Anwendung einer Funktion auf deren Elemente auf eine andere Liste abzubilden.

```
>>> a_list = [1, 9, 8, 4]
>>> [elem * 2 for elem in a_list]           ①
[2, 18, 16, 8]
>>> a_list                                  ②
[1, 9, 8, 4]
>>> a_list = [elem * 2 for elem in a_list]  ③
>>> a_list
[2, 18, 16, 8]
```

① Um dies zu verstehen, sehen sie es sich von rechts nach links an. `a_list` ist die Liste die Sie abbilden möchten. Der Python-Interpreter durchläuft `a_list` und weist den Wert jedes Elements temporär der Variable `elem` zu. Python wendet dann die Funktion `elem*2` an und hängt das Ergebnis an die zurückgegebene Liste an.

② Eine List Comprehension erstellt eine neue Liste; die ursprüngliche Liste wird nicht verändert.

③ Die Zuweisung des Ergebnisses einer List Comprehension zur abzubildenden Variable ist ungefährlich. Python erstellt die neue Liste im Speicher und weist der ursprünglichen Variable das Ergebnis erst zu, wenn die List Comprehension fertig ist.

Sie können jeden beliebigen Python-Ausdruck in einer List Comprehension nutzen, auch die Funktionen des `os`-Moduls zur Manipulation von Dateien und Verzeichnissen.

```
>>> import os, glob
>>> glob.glob('*.xml')                                    ①
['feed-broken.xml', 'feed-ns0.xml', 'feed.xml']
>>> [os.path.realpath(f) for f in glob.glob('*.xml')]  ②
['c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml']
```

① Dies gibt eine Liste aller `.xml`-Dateien im aktuellen Arbeitsverzeichnis zurück.

② Diese List Comprehension nimmt diese Liste von `.xml`-Dateien entgegen und wandelt sie in eine Liste von absoluten Pfadnamen um.

List Comprehensions können auch Elemente filtern, was dazu führt, dass das Ergebnis kleiner sein kann, als die ursprüngliche Liste.

```
>>> import os, glob
>>> [f for f in glob.glob('*.py') if os.stat(f).st_size > 6000]  ①
['pluraltest6.py',
 'romantest10.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

① Zum Filtern einer Liste können Sie am Ende der List Comprehension eine `if`-Anweisung einfügen. Der Ausdruck nach dem `if` wird für jedes Element der Liste ausgewertet. Ergibt der Ausdruck `True`, so wird das Element zur Ausgabe hinzugefügt. Diese List Comprehension sucht nach allen `.py`-Dateien im aktuellen Verzeichnis, während der `if`-Ausdruck diese Dateien filtert, indem für jede Datei ermittelt wird, ob sie größer als `6000` Bytes ist. Hier gibt es sechs solcher Dateien, also gibt die List Comprehension sechs Dateinamen zurück.

Alle bisherigen Beispiele zu List Comprehension haben recht einfache Ausdrücke enthalten – multipliziere eine Zahl mit einer Konstante, rufe eine einzelne Funktion auf, gebe das ursprüngliche Listenelement (nach der Filterung) zurück. Es gibt jedoch keine Begrenzung bei der Komplexität einer List Comprehension.

```
>>> import os, glob
>>> [(os.stat(f).st_size, os.path.realpath(f)) for f in
glob.glob('*.xml')]  ①
[(3074,'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml'),
 (3386, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml'),
 (3070, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml')]
>>> import humansize
>>> [(humansize.approximate_size(os.stat(f).st_size), f) for f in
glob.glob('*.xml')]  ②
[('3.0 KiB', 'feed-broken.xml'),
 ('3.3 KiB', 'feed-ns0.xml'),
 ('3.0 KiB', 'feed.xml')]
```

① Diese List Comprehension sucht alle `.xml`-Dateien im aktuellen Arbeitsverzeichnis, holt sich deren Dateigrößen (durch den Aufruf der `os.stat()`-Funktion) und erstellt ein Tupel aus der Dateigröße und dem absoluten Pfad jeder Datei (durch den Aufruf der `os.path.realpath()`-Funktion).

② Diese Comprehension baut auf der vorherigen auf und ruft mit den Dateigrößen jeder `.xml`-Datei die `approximate_size()`-Funktion auf.

4.4 Dictionary Comprehensions

Eine Dictionary Comprehension funktioniert wie eine List Comprehension, erstellt jedoch statt einer Liste ein Dictionary.

```
>>> import os, glob
>>> metadata = [(f, os.stat(f)) for f in glob.glob('*test*.py')]  ①
>>> metadata[0]                                                   ②
('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0, st_dev=0,
 st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
 st_mtime=1247520344, st_ctime=1247520344))
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*test*.py')}③
>>> type(metadata_dict)                                           ④
<class 'dict'>
>>> list(metadata_dict.keys())                                    ⑤
['romantest8.py', 'pluraltest1.py', 'pluraltest2.py', 'pluraltest5.py',
 'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
 'romantest9.py', 'pluraltest3.py', 'romantest1.py', 'romantest2.py',
 'romantest3.py', 'romantest5.py', 'romantest6.py', 'alphameticstest.py',
 'pluraltest4.py']
>>> metadata_dict['alphameticstest.py'].st_size                   ⑥
2509
```

① Dies ist keine Dictionary Comprehension, sondern eine List Comprehension. Sie sucht alle `.py`-Dateien mit einem `test` im Namen und erstellt dann ein Tupel aus dem Dateinamen und den Metadaten der Datei (durch den Aufruf der `os.stat()`-Funktion).

② Jedes Element der fertigen Liste ist ein Tupel.

③ Dies ist eine Dictionary Comprehension. Die Syntax ähnelt der einer List Comprehension, mit zwei Unterschieden. Erstens ist sie in geschweifte Klammern eingeschlossen, statt in eckige Klammern. Zweitens enthält sie statt *einem* Ausdruck für jedes Element *zwei* Ausdrücke, die durch einen Doppelpunkt getrennt sind. Der Ausdruck vor dem Doppelpunkt (hier `f`) ist der Schlüssel des Dictionary; der Ausdruck nach dem Doppelpunkt (hier `os.stat(f)`) ist der Wert.

④ Eine Dictionary Comprehension gibt ein Dictionary zurück.

⑤ Die Schlüssel dieses Dictionarys sind die Dateinamen, die von `glob.glob('*test*.py')` zurückgegeben wurden.

⑥ Der jedem Schlüssel zugeordnete Wert ist der Rückgabewert der `os.stat()`-Funktion. Das bedeutet: Wir können eine Datei anhand ihres Namens „auswählen“, um ihre Metadaten zu erhalten. Eine dieser Metadaten ist `st_size`, die Dateigröße. Die Datei `alphameticstest.py` ist `2509` Bytes groß.

Genau wie zu List Comprehensions können Sie auch eine `if`-Anweisung zu einer Dictionary Comprehension hinzufügen, um die Eingabefolge, basierend auf einem Ausdruck, der für jedes Element ausgewertet wird, zu filtern.

```
>>> import os, glob, humansize
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*')}       ①
>>> humansize_dict =
{os.path.splitext(f)[0]:humansize.approximate_size(meta.st_size) \
...                   for f, meta in metadata_dict.items() if me-
ta.st_size > 6000}                                               ②
>>> list(humansize_dict.keys())                                  ③
['romantest9', 'romantest8', 'romantest7', 'romantest6', 'romantest10',
'pluraltest6']
>>> humansize_dict['romantest9']                                 ④
'6.5 KiB'
```

① Diese Dictionary Comprehension erstellt eine Liste aller Dateien im aktuellen Arbeitsverzeichnis (`glog.glob('*')`), ruft die Metadaten jeder Datei ab (`os.stat(f)`) und erzeugt ein Dictionary, dessen Schlüssel Dateinamen und dessen Werte die Metadaten der Dateien sind.

② Diese Dictionary Comprehension baut auf der vorherigen auf, filtert die Dateien, so dass nur Dateien größer als `6000` Bytes berücksichtigt werden (`if meta.st_size > 6000`) und verwendet diese gefilterte Liste, um ein Dictionary zu erstellen, dessen Schlüssel die Dateinamen ohne Erweiterung (`os.path.splitext(f)[0]`) und dessen Werte die ungefähre Größe jeder Datei (`humansize.approximate_size(meta.st_size)`) sind.

③ Wie Sie bereits in einem vorherigen Beispiel gesehen haben, gibt es sechs solcher Dateien, daher befinden sich auch sechs Elemente in diesem Dictionary.

④ Der Wert jedes Schlüssels ist der von der `approximate_size()`-Funktion zurückgegebene String.

4.4.1 Andere tolle Sachen, die man mit Dictionary Comprehensions machen kann

Hier nun ein Trick, den man mit Dictionary-Schlüsseln durchführen kann und der vielleicht mal nützlich sein könnte: Schlüssel und Werte eines Dictionary vertauschen.

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
{1: 'a', 2: 'b', 3: 'c'}
```

4.5 Set Comprehensions

Auch Sets haben ihre eigene Syntax für Comprehensions. Sie ist der von Dictionary Comprehensions sehr ähnlich. Der einzige Unterschied besteht darin, dass Sets nur Werte anstatt Schlüssel:Wert-Paare besitzen.

```
>>> a_set = set(range(10))
>>> a_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {x ** 2 for x in a_set}           ①
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> {x for x in a_set if x % 2 == 0}  ②
{0, 8, 2, 4, 6}
>>> {2**x for x in range(10)}         ③
{32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```

① Set Comprehensions können ein Set als Eingabe übernehmen. Diese Set Comprehension berechnet die Quadrate der Zahlen von `0` bis `9`.

② Genau wie List Comprehensions und Dictionary Comprehensions können auch Set Comprehensions eine `if`-Anweisung zum Filtern jedes Elements enthalten.

③ Set Comprehensions müssen kein Set als Eingabe übernehmen: sie können jede Folge übernehmen.

Kapitel 5
Strings

5.1 Langweiliges Zeug, das Sie wissen müssen, bevor es losgeht

Wussten Sie, dass die Menschen von Bougainville (eine Insel im Pazifischen Ozean, Anm. d. Übers.) das kleinste Alphabet der Welt haben? Ihr sogenanntes Rotokas-Alphabet besteht aus gerade einmal zwölf Buchstaben: A, E, G, I, K, O, P, R, S, T, U und V. Andere Sprachen, wie zum Beispiel Chinesisch, Japanisch und Koreanisch haben Tausende von Buchstaben bzw. Schriftzeichen. Englisch, die Sprache von Python, hat 26 Buchstaben – 52, wenn Sie Groß- und Kleinbuchstaben getrennt zählen – und außerdem einige Sonderzeichen wie !@#$%&.

Wenn Menschen von *„Text“* reden, meinen Sie damit *„Zeichen und Symbole auf dem Computerbildschirm“*. Computer arbeiten jedoch nicht mit Zeichen und Symbolen, sondern mit Bits und Bytes. Jeder Text, den Sie jemals auf einem Monitor gesehen haben, ist tatsächlich in einer bestimmten Zeichencodierung gespeichert. Sie können sich diese Zeichencodierung vorstellen, wie die Bildschirmabbildung der im Speicher des Computers gesicherten Daten. Es gibt unzählige Zeichencodierungen, von denen einige an bestimmte Sprachen wie Russisch, Chinesisch oder Englisch angepasst sind. Andere dagegen können für verschiedene Sprachen genutzt werden.

In Wirklichkeit ist es etwas komplizierter. Viele Zeichen werden in mehreren Codierungen verwendet, doch jede Codierung kann eine andere Bytefolge nutzen, um die Zeichen im Speicher abzulegen. Sie können sich die Zeichencodierung eher wie eine Art Entschlüsselung vorstellen. Übergibt Ihnen jemand eine Bytefolge – eine Datei, eine Internetseite etc. – und Sie möchten den *„Text“* lesen, so müssen Sie wissen, welche Zeichencodierung verwendet wurde. Nur dann können Sie die Bytes entschlüsseln. Hat man Ihnen den falschen oder gar keinen Schlüssel zum Entschlüsseln gegeben, bleibt Ihnen nichts anderes übrig, als selbst zu versuchen, den Code zu knacken. Es ist ziemlich unwahrscheinlich, dass Ihnen das gelingt. Das Ergebnis wird dann unbefriedigend sein.

Ihnen sind sicher auch schon Internetseiten begegnet, in deren Text seltsame Fragezeichen dort auftauchten, wo eigentlich ein Apostroph oder Ähnliches stehen sollte. Dies bedeutet im Allgemeinen, dass der Ersteller der Seite die verwendete Zeichencodierung nicht korrekt angegeben hat. Der Browser kann die korrekten

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_5, © Springer-Verlag Berlin Heidelberg 2010

Zeichen dann nicht anzeigen. Bei englischsprachigen Texten ist das sicher ärgerlich, doch in anderen Sprachen kann das Ergebnis komplett unlesbar werden.

Für jede verbreitete Sprache der Welt existieren eigene Zeichencodierungen. Da jede Sprache anders ist und Speicher in der Vergangenheit sehr teuer war, ist jede Zeichencodierung für eine bestimmte Sprache optimiert. Jede Codierung verwendet jedoch dieselben Zahlen (0–255), um die jeweiligen Zeichen darzustellen. Sie kennen vielleicht die ASCII-Codierung, die die Zeichen der englischen Sprache enthält und von 0–127 reicht. (65 ist ein großes „A“, 97 ein kleines „a“ usw.) Die englische Sprache hat ein sehr simples Alphabet, sodass alle Zeichen in weniger als 128 Zahlen darstellbar sind. Für die unter Ihnen, die binär zählen können: Das sind 7 von 8 Bits in einem Byte.

Westeuropäische Sprachen wie Französisch, Spanisch und Deutsch besitzen mehr Buchstaben als die englische Sprache. Genauer gesagt gibt es in diesen Sprachen Buchstaben mit verschiedenen Aussprachezeichen, wie etwa das Zeichen ñ im Spanischen oder das ü im Deutschen. Die am häufigsten eingesetzte Codierung für diese Sprachen ist CP-1252. Da diese Codierung vor allem unter Microsoft Windows verwendet wird, bezeichnet man sie auch als „windows-1252“. Die CP-1252-Codierung enthält die ersten 127 Zeichen der ASCII-Codierung, erweitert diese jedoch um die Zeichen 128–255, um Zeichen wie n-mit-einer-Tilde-darüber (241) oder u-mit-zwei-Punkten-darüber (252) usw. darzustellen. Diese Codierung besteht dennoch nur aus einem Byte.

Es gibt allerdings auch Sprachen wie Chinesisch, Japanisch und Koreanisch. Diese besitzen so viele Schriftzeichen, dass man mehrere Bytes umfassende Zeichencodierungen benötigt, um sie alle darzustellen. Hier wird jeder „Buchstabe“ durch eine aus zwei Bytes bestehende Zahl von 0–65.535 repräsentiert. Leider ist es jedoch so, dass auch Codierungen die mehrere Bytes umfassen zum gleichen Problem führen wie Codierungen mit nur einem Byte. Auch sie nutzen dieselben Zahlen, um verschiedene Dinge darzustellen. Einzig die Spanne der Zahlen ist größer, da es viel mehr Zeichen gibt.

Das ging solange gut, wie die Welt noch nicht so vernetzt war. *„Text“* war etwas, dass man selbst tippte und ab und zu ausdruckte. Es gab nicht viel *plain text* (Text ohne jegliche softwareseitige Formatierung). Für Quellcode benutzte man ASCII, für alles andere eine Textverarbeitung, die ihre eigenen Formate zur Speicherung der Zeichencodierung und Formatierung hatte. Andere Leute lasen diese Dokumente mit dem gleichen Textverarbeitungsprogramm und alles funktionierte reibungslos. Mehr oder weniger.

Denken Sie nun an den Einzug globaler Netzwerke wie E-Mail oder das World Wide Web. Das ist eine Menge *plain text*, der rund um den Globus geschickt wird. Auf einem Computer geschrieben, übertragen durch einen Zweiten und schließlich angezeigt auf einem Dritten. Computer sehen in all dem nur Zahlen. Doch diese Zahlen können ganz unterschiedliche Dinge bedeuten. Oh, nein! Was kann man da tun? Nun, die Systeme mussten so entworfen werden, dass sie die Codierungsinformationen mit jedem Stückchen *plain text* mitschickten. Erinnern Sie sich daran, dass die Entschlüsselung dafür zuständig ist, dass aus den nur vom Computer lesbaren

Zahlen für den Menschen lesbare Buchstaben werden? Fehlt die Entschlüsselung, führt das zu unsinnigem Kauderwelsch.

Stellen Sie sich vor, Sie wollten verschiedene Texte am selben Ort speichern, wie z. B. in derselben Tabelle einer Datenbank, die all Ihre E-Mails enthält. Für jeden einzelnen Text müssen Sie die dazugehörige Zeichencodierung speichern, damit der Text korrekt angezeigt wird. Sie denken das sei viel Arbeit? Versuchen Sie Ihre E-Mail-Datenbank zu durchsuchen. Dazu müssen Sie innerhalb von wenigen Sekunden verschiedene Codierungen konvertieren. Das klingt nach einer Menge Spaß, oder?

Es besteht auch die Möglichkeit, dass mehrsprachige Dokumente dabei sind, die Zeichen verschiedener Sprachen enthalten. (Hinweis: Programme, die dies versucht haben, benutzten für gewöhnlich Escape-Codes, um zwischen sogenannten „Modes“ zu wechseln. Schwups, Sie verwenden nun den russischen koi8-r-Mode. 241 bedeutet hier Я. Schwups, jetzt sind wir im Mac-Greek-Mode. Hier bedeutet 241 ώ.) Natürlich wollen Sie auch *diese* Dokumente durchsuchen.

Sie sollten sich nun erst einmal ausweinen, denn alles, was Sie je über Strings zu wissen geglaubt haben, ist falsch. Außerdem existiert *plain text* nicht.

5.2 Unicode

Unicode wird gestartet. Unicode ist ein System, das entwickelt wurde, um jedes Zeichen jeder Sprache darzustellen. In Unicode wird jeder Buchstabe, jedes Zeichen und jedes Schriftzeichen durch eine vier Bytes große Zahl repräsentiert. Jede Zahl steht für genau ein Zeichen, das in mindestens einer Sprache Verwendung findet. (Es werden nicht alle Zahlen genutzt, doch über 65.535; zwei Bytes würden also nicht ausreichen.) In mehreren Sprachen vorzufindende Zeichen haben im Allgemeinen dieselbe Zahl, es sei denn, es besteht ein etymologischer Grund, dass es nicht so ist. Sei’s drum, es gibt genau eine Zahl pro Zeichen und genau ein Zeichen pro Zahl. Jede Zahl bedeutet immer dasselbe. Es gibt keine „Modes“, die man beachten müsste. `U+0041` ist immer ‚A‘, selbst wenn es in Ihrer Sprache gar kein ‚A‘ gibt.

Auf den ersten Blick scheint dies ein grandioses Konzept zu sein. Eine Codierung für alles. Mehrere Sprachen in einem Dokument. Nie mehr zwischen „Modes“ wechseln. Ihnen sollte aber direkt eine Frage in den Kopf kommen. Vier Bytes? Für jedes einzelne Zeichen?! Das klingt nach absoluter Verschwendung, vor allem bei Sprachen wie Englisch und Deutsch, die weniger als ein Byte (256 Zahlen) benötigen, um jedes mögliche Zeichen darzustellen. Tatsächlich ist es sogar im Hinblick auf schriftzeichenbasierte Sprachen (wie Chinesisch) Verschwendung, da diese auch mit zwei Bytes pro Zeichen auskommen.

Es gibt eine Unicode-Codierung, die vier Bytes pro Zeichen verwendet. Diese Variante heißt UTF-32, da 32 Bits vier Bytes sind. Die UTF-32-Codierung ist sehr einfach gehalten; jedes Unicode-Zeichen (eine Zahl aus vier Bytes) wird mit genau

dieser Zahl repräsentiert. Dies hat einige Vorteile, von denen der wichtigste der ist, dass Sie das *n*-te Zeichen eines Strings innerhalb einer konstanten Zeitspanne finden können, da das *n*-te Zeichen beim *4*n*-ten Byte beginnt. UTF-32 hat jedoch auch einige Nachteile. Der offensichtlichste ist wohl der, dass vier verdammte Bytes für jedes verdammte Zeichen benötigt werden.

Auch wenn es unglaublich viele Unicode-Zeichen gibt, ist es so, dass die Mehrzahl der Menschen niemals ein Zeichen oberhalb des 65.535. nutzen wird. Aus genau diesem Grund gibt es noch eine andere Unicode-Codierung, die als UTF-16 bezeichnet wird (16 Bits entsprechen zwei Bytes). Mit UTF-16 werden alle Zeichen von 0–65.535 mit zwei Bytes codiert. Durch einige Programmiertricks wird es Ihnen ermöglicht, auch die selten genutzten Unicode-Zeichen zu verwenden, die sich oberhalb des 65.535. Zeichens befinden. Der offensichtliche Vorteil: UTF-16 benötigt lediglich halb so viel Platz wie UTF-32, da jedes Zeichen nur noch zwei Bytes statt vier Bytes benötigt (außer denen, bei denen es anders ist). Sie können außerdem immer noch sehr einfach das *n*-te Zeichen eines Strings innerhalb einer konstanten Zeitspanne finden, unter der Annahme, dass der String keine der höherwertigen Zeichen enthält. Dies ist solange eine sinnvolle Annahme, bis das Gegenteil eintritt.

UTF-32 und UTF-16 haben jedoch auch nicht ganz offensichtliche Nachteile. Verschiedene Computersysteme speichern einzelne Bytes auf verschiedene Weise. Je nachdem, ob das System das *Big-Endian*-Format oder das *Little-Endian*-Format verwendet, könnte das Zeichen `U+4E2D` in UTF-16 entweder als `4E 2D` oder als `2D 4E` gespeichert werden. (Bei UTF-32 gibt es sogar noch mehr Möglichkeiten.) Bleiben Ihre Dokumente nur auf Ihrem eigenen Computer, spielt dies keine Rolle, da alle Anwendungen auf ein und demselben System dieselbe Byte-Reihenfolge benutzen. Sobald Sie aber Dokumente zwischen verschiedenen Systemen tauschen möchten, z. B. im World Wide Web, müssen Sie irgendwie kenntlich machen, wie die Bytes angeordnet sind. Das System, das Ihr Dokument erhält, hat andernfalls keine Möglichkeit, herauszufinden, ob die Zwei-Byte-Sequenz `4E 2D` `U+4E2D` oder `U+2D4E` bedeuten soll.

Um dieses Problem zu lösen, gibt es bei Unicode-Codierungen mit mehreren Bytes eine sogenannte „Byte Order Mark“ (Kennzeichnung der Byte-Reihenfolge). Diese besteht aus einem nicht-druckbaren Zeichen, das Sie am Anfang Ihres Dokuments einfügen können, um die Byte-Reihenfolge kenntlich zu machen. Die Byte Order Mark für UTF-16 lautet `U+FEFF`. Erhalten Sie ein Dokument, dass mit den Bytes `FF FE` beginnt, so wissen Sie, dass die Bytes in dieser Reihenfolge angeordnet sind; beginnt das Dokument mit `FE FF`, dann sind sie andersherum geordnet.

UTF-16 ist aber dennoch nicht perfekt, vor allem dann nicht, wenn Sie häufig mit ASCII-Zeichen zu tun haben. Selbst eine chinesische Internetseite enthält sehr viele ASCII-Zeichen – all die Elemente und Attribute, die die druckbaren chinesischen Zeichen umschließen. Das *n*-te Zeichen in einer kontanten Zeitspanne zu finden ist gut und schön, aber das Problem der höherwertigen Zeichen bleibt weiterhin bestehen. Sie können nicht sicher sein, dass jedes Zeichen genau zwei Bytes groß ist, also können Sie auch nicht wirklich das *n*-te Zeichen in einer kontanten

Zeitspanne finden, es sei denn, die verwenden einen separaten Index. Und es gibt einen Haufen ASCII-Text auf der Welt …

Viele Leute haben über diese Probleme nachgedacht und sind zu folgender Lösung gelangt: UTF-8 ist eine Unicode-Codierung mit variabler Länge. Das heißt, dass verschiedene Zeichen eine unterschiedliche Anzahl an Bytes benötigen. Für ASCII-Zeichen (A–Z etc.) verwendet UTF-8 nur ein Byte pro Zeichen. Tatsächlich nutzt es sogar die gleichen Bytes; die ersten 128 Zeichen (0–127) lassen sich in UTF-8 nicht von ASCII unterscheiden. Zeichen wie ñ und ö nutzen zwei Bytes. Chinesische Zeichen wie 中 benötigen drei Bytes. Die selten verwendeten höherwertigen Zeichen schließlich verwenden vier Bytes.

Nachteile: Da jedes Zeichen aus einer anderen Anzahl an Bytes bestehen kann, ist das Auffinden des *n*-ten Zeichens u. U. ein langwieriges Unterfangen, d. h., je länger der String ist, desto länger dauert es, ein bestimmtes Zeichen zu finden. Außerdem werden Bit-Verschiebungen verwendet, um Zeichen in Bytes zu codieren und Bytes in Zeichen zu decodieren.

Vorteile: sehr effizientes Codieren von ASCII-Zeichen. Für Zeichen wie ñ und ö ebenso gut geeignet wie UTF-16. Für chinesische Zeichen besser geeignet als UTF-32. Außerdem (hier müssen Sie mir einfach glauben, da ich nicht auf die Mathematik dahinter eingehen werde) gibt es keine Probleme mit der Byte-Reihenfolge. Ein mit UTF-8 codiertes Dokument verwendet auf jedem Computer dieselbe Byte-Reihenfolge.

5.3 Los geht's

In Python 3 bestehen alle Strings aus Unicode-Zeichen. Solche Dinge wie einen mit UTF-8 codierten String oder einen mit CP-1252 codierten String gibt es in Python nicht. „Verwendet dieser String UTF-8?" ist eine Frage, die Sie nicht stellen dürfen. UTF-8 ist eine Möglichkeit, Zeichen als eine Folge von Bytes zu codieren. Möchten Sie einen String in eine Byte-Folge umwandeln und dazu eine bestimmte Codierung verwenden, kann Ihnen Python 3 dabei behilflich sein. Möchten Sie eine Byte-Folge in einen String umwandeln, so kann Ihnen Python 3 auch dabei helfen. Bytes sind keine Zeichen, Bytes sind Bytes. Zeichen sind eine Abstraktion. Ein String ist eine Folge dieser Abstraktionen.

```
>>> s = '深入 Python'       ①
>>> len(s)                  ②
9
>>> s[0]                    ③
'深'
>>> s + ' 3'                ④
'深入 Python 3'
```

① Um einen String zu erstellen, schließen Sie ihn in Anführungszeichen ein. In Python können Strings sowohl in einfache (`'`), als auch in doppelte (`"`) Anführungszeichen eingeschlossen sein.

② Die integrierte `len()`-Funktion gibt die Länge eines Strings zurück, d. h. die Anzahl der Zeichen. Es ist die gleiche Funktion, die auch verwendet wird, um die Länge einer Liste herauszufinden. Ein String ist wie eine Liste von Zeichen.

③ Mithilfe der Indizes können Sie wie bei einer Liste die einzelnen Zeichen eines Strings ansprechen.

④ Sie können ebenso wie bei Listen Strings mithilfe des +-Operators verketten.

5.4 Strings formatieren

Lassen Sie uns einen weiteren Blick auf `humansize.py` werfen:

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],          ①
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.                           ②

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''                                                                      ③
    if size < 0:
        raise ValueError('number must be non-negative')                      ④

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)                        ⑤

    raise ValueError('number too large')
```

① `'KB'`, `'MB'`, `'GB'`, … all dies sind Strings.

② Docstrings von Funktionen sind Strings. Da dieser Docstring sich über mehrere Zeilen erstreckt, verwenden wir drei Anführungszeichen am Beginn und am Ende des Strings.

③ Diese drei Anführungszeichen beenden den Docstring.

④ Dies ist ein weiterer String, den wir der Ausnahme als lesbare Fehlernachricht übergeben.

⑤ Dies ist ein … Oh, was ist das?

In Python 3 können Werte in Strings formatiert werden. Auch wenn dies zu sehr komplexen Ausdrücken führen kann, besteht die einfachste Verwendung aus dem Einfügen eines Wertes in einen String durch Benutzung eines einzelnen Platzhalters.

```
>>> username = 'mark'
>>> password = 'PapayaWhip'                                    ①
>>> "{0}'s password is {1}".format(username, password)         ②
"mark's password is PapayaWhip"
```

① Nein, mein Passwort ist in Wirklichkeit nicht `PapayaWhip`.

② Hier gibt es sehr viel zu entdecken. Es wird eine Methode eines Stringliterals aufgerufen. Strings sind Objekte und Objekte haben Methoden. Das Ergebnis dieses Ausdrucks ist ein String. `{0}` und `{1}` sind Platzhalter, die durch die Argumente der `format()`-Methode ersetzt werden.

5.4.1 Zusammengesetzte Feldnamen

Das vorherige Beispiel zeigt den einfachsten Fall, bei dem die Platzhalter einfache Ganzzahlen sind. Ganzzahl-Platzhalter werden als Positionsindizes der Argumente der `format()`-Methode behandelt. Dies bedeutet, dass `{0}` durch das erste Argument (in diesem Fall `username`) und `{1}` durch das zweite Argument (hier `password`) ersetzt wird etc. Sie können so viele Positionsindizes verwenden wie Argumente vorhanden sind. Argumente können Sie so viele haben wie Sie möchten. Doch Platzhalter sind noch viel leistungsfähiger.

```
>>> import humansize
>>> si_suffixes = humansize.SUFFIXES[1000]      ①
>>> si_suffixes
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> '1000{0[0]} = 1{0[1]}'.format(si_suffixes)  ②
'1000KB = 1MB'
```

① Statt eine Funktion des `humansize`-Moduls aufzurufen, verwenden wir hier nur eine der im Modul definierten Datenstrukturen: die Liste der „SI“-Suffixe (Mehrfache von 1.000).

② Dies sieht komplizierter aus als es in Wirklichkeit ist. `{0}` würde normalerweise auf das erste Argument der `format()`-Methode, `si_suffixes`, verweisen. Doch `si_suffixes` ist eine Liste. `{0[0]}` verweist daher auf das erste Element der Liste, die der `format()`-Methode als erstes Argument übergeben wurde. Hier ergibt sich also `'KB'`. `{0[1]}` verweist auf das zweite Element derselben

Liste, hier `'MB'`. Alles, was außerhalb der geschweiften Klammern steht – `1000`, das Gleichheitszeichen und die Leerzeichen – verbleiben so, wie sie eingegeben wurden. Das Endresultat ist der String `'1000KB = 1MB'`.

Dieses Beispiel zeigt Folgendes: Formatmodifizierer können auf Elemente und Eigenschaften von Datenstrukturen (fast) durch die Verwendung von Python-Syntax zugreifen. Dies nennt man zusammengesetzte Feldnamen. Die folgenden zusammengesetzten Feldnamen „funktionieren einfach so“:

- Übergabe einer Liste und Zugriff auf ein Element dieser Liste über den Index (wie im vorherigen Beispiel)
- Übergabe eines Dictionary und Zugriff auf einen Wert des Dictionary über seinen Schlüssel
- Übergabe eines Moduls und Zugriff auf seine Variablen und Funktionen über deren Namen
- Übergabe einer Klasseninstanz und Zugriff auf ihre Eigenschaften und Methoden über deren Namen
- Jede beliebige Kombination der oben genannten Möglichkeiten

Um Ihre grauen Zellen anzuregen hier ein Beispiel, das alle der oben genannten Möglichkeiten verwendet:

```
>>> import humansize
>>> import sys
>>> '1MB = 1000{0.modules[humansize].SUFFIXES[1000][0]}'.format(sys)
'1MB = 1000KB'
```

Sehen Sie sich an, wie dies funktioniert:

- Das `sys`-Modul enthält Informationen über die derzeit laufende Instanz von Python. Da Sie dieses Modul gerade importiert haben, können Sie es als Argument der `format()`-Methode übergeben. Der Platzhalter `{0}` verweist somit auf das `sys`-Modul.
- `sys.modules` ist ein Dictionary, das alle Module enthält, die während der laufenden Python-Instanz importiert wurden. Die Schlüssel sind die Modulnamen als Strings; die Werte sind die Modulobjekte selbst. Der Platzhalter `{0.modules}` verweist daher auf das Dictionary aller importierten Module.
- `sys.modules['humansize']` ist das gerade von Ihnen importierte `humansize`-Modul. Der Platzhalter `{0.modules[humansize]}` verweist auf das `humansize`-Modul. Beachten Sie hier den kleinen Syntax-Unterschied. In echtem Python-Code sind die Schlüssel des `sys.modules`-Dictionarys Strings; um sie zu verwenden, müssen Sie Anführungszeichen benutzen (z. B. `'humansize'`). Innerhalb eines Platzhalters müssen Sie die Anführungszeichen weglassen (z. B. `humansize`).
- `sys.modules['humansize'].SUFFIXES` ist das Dictionary, das am Anfang des `humansize`-Moduls definiert wurde. Der Platzhalter `{0.modules[humansize].SUFFIXES}` verweist auf dieses Dictionary.

- `sys.modules['humansize'].SUFFIXES[1000]` ist eine Liste von SI-Suffixen: `['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']`. Der Platzhalter `{0.modules[humansize].SUFFIXES[1000]}` verweist also auf diese Liste.
- `sys.modules['humansize'].SUFFIXES[1000][0]` ist das erste Element der Liste von SI-Suffixen, hier also `'KB'`. Der Platzhalter `{0.modules[humansize].SUFFIXES[1000][0]}` wird daher durch den aus zwei Zeichen bestehenden String `KB` ersetzt.

5.4.2 Formatmodifizierer

Halt! Es gibt noch etwas zu sehen! Werfen wir einen weiteren Blick auf diese seltsame Codezeile in `humansize.py`:

```
if size < multiple:
    return '{0:.1f} {1}'.format(size, suffix)
```

`{1}` wird durch das zweite Argument der `format()`-Methode ersetzt, hier also `suffix`. Doch was bedeutet `{0:.1f}`? Es bedeutet zweierlei: `{0}`, was Ihnen bereits bekannt ist und `:.1f`, was Sie bisher noch nicht kennen. Der zweite Teil (ab dem Doppelpunkt, inklusive diesem) definiert den Formatmodifizierer, der angibt, wie die eingesetzte Variable formatiert werden soll.

Innerhalb eines Platzhalters markiert der Doppelpunkt (`:`) den Beginn des Formatmodifizierers. Der Formatmodifizierer `.1` bedeutet, dass „zum nächsten Zehntel gerundet“ werden soll (d. h., dass nur eine Nachkommastelle angezeigt wird). Der Formatmodifizierer `f` gibt an, dass die Zahl als „Festkommazahl“ dargestellt werden soll (im Gegensatz zur Exponentialschreibweise oder einer anderen Dezimaldarstellung). Wenn `size 698.24` und `suffix 'GB'` wäre, würde der formatierte String so aussehen: `'693.2 GB'`. `698.24` wird auf eine Nachkommastelle gerundet und dann das Suffix hinten angehängt.

```
>>> '{0:.1f} {1}'.format(698.24, 'GB')
'698.2 GB'
```

5.5 Andere häufig verwendete String-Methoden

Strings beherrschen neben dem Formatieren noch eine Reihe anderer nützlicher Tricks.

```
>>> s = '''Finished files are the re-  ①
... sult of years of scientif-
... ic study combined with the
... experience of years.'''
>>> s.splitlines()                    ②
['Finished files are the re-',
 'sult of years of scientif-',
 'ic study combined with the',
 'experience of years.']
>>> print(s.lower())                  ③
finished files are the re-
sult of years of scientif-
ic study combined with the
experience of years.
>>> s.lower().count('f')              ④
6
```

① In Pythons interaktiver Shell können Sie mehrzeilige Strings eingeben. Haben Sie einen mehrzeiligen String durch dreifache Anführungszeichen begonnen und drücken die Eingabetaste, so können Sie eine weitere Zeile eingeben. Geben Sie die dreifachen Anführungszeichen zum Beenden des Strings ein und drücken dann die Eingabetaste, wird der Befehl ausgeführt (im vorliegenden Fall wird damit der String `s` zugewiesen).

② Die Methode `splitlines()` nimmt einen mehrzeiligen String entgegen und gibt eine Liste von Strings zurück; jeder String in der Liste enthält je eine Zeile des ursprünglichen Strings. Die Zeilenvorschübe am Ende jeder Zeile werden dabei nicht berücksichtigt.

③ Die Methode `lower()` wandelt den gesamten String in Kleinbuchstaben um. (Die Methode `upper()` macht das Gegenteil und wandelt den String in Großbuchstaben um.)

④ Die Methode `count()` zählt alle Vorkommen eines Teilstrings innerhalb eines Strings. Ja, in diesem Satz ist wirklich sechsmal der Buchstabe „`f`“ zu finden!

Hier ein weiteres häufig vorzufindendes Beispiel. Nehmen wir an, Sie haben eine Liste von Schlüssel-Wert-Paaren der Form `key1=value1&key2=value2`. Nun möchten Sie dies in die Form eines Dictionary – `{key1: value1, key2: value2}` – bringen.

```
>>> query = 'user=pilgrim&database=master&password=PapayaWhip'
>>> a_list = query.split('&')                              ①
>>> a_list
['user=pilgrim', 'database=master', 'password=PapayaWhip']
>>> a_list_of_lists = [v.split('=', 1) for v in a_list]  ②
>>> a_list_of_lists
[['user', 'pilgrim'], ['database', 'master'], ['password', 'PapayaWhip']]
>>> a_dict = dict(a_list_of_lists)                         ③
>>> a_dict
{'password': 'PapayaWhip', 'user': 'pilgrim', 'database': 'master'}
```

① Die `split()`-Methode nimmt ein Argument entgegen – ein Trennzeichen. Die Methode teilt einen String dort, wo das Trennzeichen (hier `&`) auftritt und gibt eine Liste der Strings zurück, die dabei entstehen. Das Trennzeichen kann beliebig gewählt werden.

② Nun haben wir eine Liste von Strings; jeder dieser Strings besteht aus einem Schlüssel, gefolgt von einem Gleichheitszeichen, gefolgt von einem Wert. Diese Liste wollen wir jetzt durchlaufen und dabei jeden String dort, wo das Gleichheitszeichen auftritt, teilen. (Theoretisch könnte ein Wert auch ein Gleichheitszeichen enthalten. Würden wir einfach `'key=value=foo'.split('=')` benutzen, erhielten wir als Ergebnis die folgende Liste aus drei Elementen: `['key', 'value', 'foo']`.)

③ Python kann diese Liste von Listen in ein Dictionary verwandeln, indem Sie sie einfach der `dict()`-Methode übergeben.

5.5.1 Slicen eines Strings

Haben Sie einen String definiert, können Sie jeden Teil dieses Strings als neuen String erhalten. Dies wird als *Slicing* des Strings bezeichnet. Das Slicen von Strings funktioniert genau wie das Slicen von Listen, was einleuchtet, da Strings lediglich Zeichenfolgen sind.

```
>>> a_string = 'My alphabet starts where your alphabet ends.'
>>> a_string[3:11]           ①
'alphabet'
>>> a_string[3:-3]           ②
'alphabet starts where your alphabet en'
>>> a_string[0:2]            ③
'My'
>>> a_string[:18]            ④
'My alphabet starts'
>>> a_string[18:]            ⑤
' where your alphabet ends.'
```

① Sie erhalten einen Teil eines Strings, *Slice* genannt, indem Sie zwei Indizes angeben. Der Rückgabewert ist ein neuer String, der alle Zeichen des Strings enthält und beim ersten Slice-Index beginnt.

② Wie beim Slicen von Listen können Sie auch hier negative Indizes verwenden.

③ Strings beginnen mit dem Index `0`; `a_string[0:2]` gibt daher die ersten beiden Elemente des Strings zurück. `a_string[2]` gehört nicht mehr dazu.

④ Sie können den linken Slice-Index auslassen, wenn er `0` ist, da Python dann automatisch `0` verwendet. `a_string[:18]` ist also dasselbe wie `a_string[0:18]`.

⑤ Wenn der rechte Slice-Index der Länge des Strings entspricht, müssen Sie diesen ebenfalls nicht angeben. `a_string[18:]` ist dasselbe wie `a_string[18:44]`, da der String aus 44 Zeichen besteht. Die Symmetrie ist fantastisch: `a_string[:18]` gibt die ersten 18 Zeichen zurück, während `a_string[18:]` alles außer den ersten 18 Zeichen zurückgibt. `a_string[:n]` gibt immer die ersten `n` Zeichen zurück und `a_string[n:]` den Rest, egal wie lang der String auch sein mag.

5.6 Strings vs. Bytes

Bytes sind Bytes; Zeichen sind eine Abstraktion. Eine unveränderliche Folge von Unicode-Zeichen wird String genannt. Eine unveränderliche Folge von Zahlen zwischen `0` und `255` wird als `bytes`-Objekt bezeichnet.

```
>>> by = b'abcd\x65'  ①
>>> by
b'abcde'
>>> type(by)          ②
<class 'bytes'>
>>> len(by)           ③
5
>>> by += b'\xff'     ④
>>> by
b'abcde\xff'
>>> len(by)           ⑤
6
>>> by[0]             ⑥
97
>>> by[0] = 102       ⑦
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

① Um ein `bytes`-Objekt zu definieren, verwenden Sie die `b''`-„Byte-Literal"-Syntax. Jedes Byte innerhalb des Byte-Literals kann ein ASCII-Zeichen oder eine codierte Hexadezimalzahl von `\x00` bis `\xff` (`0–255`) sein.

② Der Datentyp eines `bytes`-Objekts ist `bytes`.

③ Verwenden Sie, wie bei Listen und Strings, die integrierte `len()`-Funktion, um die Länge eines `bytes`-Objekts zu ermitteln.

④ Sie können, wie bei Listen und Strings, auch `bytes`-Objekte mithilfe des `+`-Operators verketten. Das Ergebnis ist ein neues `bytes`-Objekt.

⑤ Das Verketten eines fünf Bytes großen `bytes`-Objekts mit einem ein Byte großen `bytes`-Objekt ergibt ein sechs Bytes umfassendes `bytes`-Objekt.

⑥ Genau wie bei Listen und Strings können Sie auch bei `bytes`-Objekten einzelne Bytes unter Verwendung von Indizes anwählen. Die Elemente eines Strings sind Strings; die Elemente eines `bytes`-Objekts sind Ganzzahlen. Genauer: Ganzzahlen zwischen `0` und `255`.

⑦ Ein `bytes`-Objekt ist unveränderlich; Sie können die Bytes nicht einzeln zuweisen. Um ein individuelles Byte zu ändern, müssen Sie entweder Slice-Methoden (die genauso funktionieren wie bei Strings) und Verkettungsoperatoren verwenden (die ebenfalls so funktionieren wie bei Strings), oder das `bytes`-Objekt in ein `bytearray`-Objekt umwandeln.

```
>>> by = b'abcd\x65'
>>> barr = bytearray(by)  ①
>>> barr
bytearray(b'abcde')
>>> len(barr)             ②
5
>>> barr[0] = 102         ③
>>> barr
bytearray(b'fbcde')
```

① Verwenden Sie die integrierte `bytearray()`-Funktion, um ein `bytes`-Objekt in ein veränderbares `bytearray`-Objekt umzuwandeln.

② Alle Methoden und Operationen die Sie mit einem `bytes`-Objekt ausführen können, lassen sich auch mit einem `bytearray`-Objekt verwenden.

③ Der wichtige Unterschied ist der, dass Sie einem `bytearray`-Objekt mithilfe der Indizes individuelle Bytes zuweisen können. Der zugewiesene Wert muss dabei eine Ganzzahl zwischen `0` und `255` sein.

Sie können niemals Bytes und Strings vermischen.

```
>>> by = b'd'
>>> s = 'abcde'
>>> by + s                        ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
>>> s.count(by)                   ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> s.count(by.decode('ascii'))   ③
1
```

① Bytes und Strings können nicht verkettet werden. Es sind zwei verschiedene Datentypen.

② Die Häufigkeit des Vorkommens eines Bytes in einem String kann nicht gezählt werden, da es in einem String keine Bytes gibt. Vielleicht ist hier Folgendes gemeint: „Zähle die Häufigkeit des Vorkommens des Strings, den man erhalten würde, wenn man die Bytefolge mithilfe einer bestimmten Zeichencodierung decodieren würde.“ Nun, dann muss man es auch so angeben. Python 3 wandelt implizit weder Bytes in Strings um, noch Strings in Bytes.

③ Es ist ein unglaublicher Zufall, dass diese Zeile bedeutet: „Zähle die Häufigkeit des Vorkommens des Strings, den man erhalten würde, wenn man die Bytefolge mithilfe dieser bestimmten Zeichencodierung decodieren würde.“

Hier nun der Zusammenhang zwischen Strings und Bytes: `bytes`-Objekte besitzen eine `decode()`-Methode, die eine Zeichencodierung übernimmt und einen String zurückgibt. Strings dagegen besitzen eine `encode()`-Methode, die eine Zeichencodierung übernimmt und ein `bytes`-Objekt zurückgibt. Im obigen Beispiel war die Decodierung recht einfach, es wurde eine Bytefolge in der ASCII-Codierung in einen String aus Zeichen umgewandelt. Genauso ist die Vorgehensweise jedoch mit jeder Codierung, welche die Zeichen des Strings unterstützt – selbst alte (nicht-Unicode) Codierungen können so verwendet werden.

```
>>> a_string = '深入 Python'                ①
>>> len(a_string)
9
>>> by = a_string.encode('utf-8')           ②
>>> by
b'\xe6\xb7\xb1\xe5\x85\xa5 Python'
>>> len(by)
13
>>> by = a_string.encode('gb18030')         ③
>>> by
b'\xc9\xee\xc8\xeb Python'
>>> len(by)
11
>>> by = a_string.encode('big5')            ④
>>> by
b'\xb2`\xa4J Python'
>>> len(by)
11
>>> roundtrip = by.decode('big5')           ⑤
>>> roundtrip
'深入 Python'
>>> a_string == roundtrip
True
```

① Das ist ein String. Er besteht aus neun Zeichen.

② Das ist ein `bytes`-Objekt. Es besteht aus 13 Bytes. Dieses `bytes`-Objekt enthält den in UTF-8 codierten String `a_string`.

③ Das ist ein `bytes`-Objekt. Es besteht aus 11 Bytes. Es enthält den in GB18030 codierten String `a_string`.

④ Das ist ein `bytes`-Objekt. Es besteht aus 11 Bytes. Es enthält eine völlig andere Byte-Folge, die entsteht, wenn man `a_string` in Big5 codiert.

⑤ Das ist ein String. Er besteht aus neun Zeichen. Er enthält die Zeichenfolge, die herauskommt, wenn man by mithilfe der *Big5*-Codierung decodiert. Dieser String ist identisch mit dem ursprünglichen String.

5.7 Nachbemerkung – Zeichencodierung von Python-Quelltext

Python 3 geht davon aus, dass Ihr Quelltext – also jede `.py`-Datei – in UTF-8 codiert ist.

☞ In Python 2 war die Standardcodierung für `.py`-Dateien ASCII. In Python 3 ist es UTF-8.

Möchten Sie in Ihrem Python-Code eine andere Codierung verwenden, so müssen Sie diese in der ersten Zeile jeder Datei angeben. Die folgende Deklaration gibt an, dass die `.py`-Datei die *windows-1252*-Codierung verwendet:

```
# -*- coding: windows-1252 -*-
```

Die Angabe zur Zeichencodierung kann auch erst in der zweiten Zeile erfolgen, wenn in der ersten Zeile die *Shebang*-Markierung steht.

```
#!/usr/bin/python3
# -*- coding: windows-1252 -*-
```

Kapitel 6
Reguläre Ausdrücke

6.1 Los geht's

Jede moderne Programmiersprache besitzt integrierte Funktionen zur Bearbeitung von Strings. In Python besitzen Strings Methoden zum Suchen und Ersetzen: `index()`, `find()`, `split()`, `count()`, `replace()` usw. Diese Methoden sind jedoch auf die simpelsten aller Fälle beschränkt. Die `index()`-Methode z. B. sucht nach einem einzelnen fest einprogrammierten Unterstring. Die Methode unterscheidet dabei immer zwischen Groß- und Kleinschreibung. Um einen String `s` ohne Beachtung der Groß-/Kleinschreibung zu suchen, müssen Sie `s.lower()` oder `s.upper()` aufrufen, um dafür zu sorgen, dass alle Buchstaben klein bzw. groß geschrieben sind. Denselben Beschränkungen unterliegen auch die `replace()`- und die `split()`-Methode.

Kann Ihr Ziel mithilfe dieser Stringmethoden erreicht werden, so sollten Sie sie nutzen. Sie sind schnell, einfach und leicht zu lesen, und ich könnte Ihnen einiges über schnellen, einfachen, leicht lesbaren Code erzählen. Wenn Sie jedoch innerhalb von `if`-Anweisungen sehr viele Stringfunktionen nutzen, um Spezialfälle zu behandeln, oder `split()` und `join()` verketten, um Ihre Strings zu zerstückeln, dann sollten Sie vielleicht doch besser reguläre Ausdrücke in Betracht ziehen.

Reguläre Ausdrücke sind eine mächtige und (weitgehend) standardisierte Möglichkeit, Text mithilfe komplexer Zeichenmuster zu durchsuchen, zu ersetzen und zu parsen. Auch wenn die Syntax der regulären Ausdrücke sehr unübersichtlich ist und nicht wie gewöhnlicher Code aussieht, kann das Ergebnis lesbarer sein, als eine handgemachte Lösung, die eine Menge Stringfunktionen einsetzt. Es besteht sogar die Möglichkeit, innerhalb regulärer Ausdrücke Kommentare zu nutzen. So können Sie Ihre regulären Ausdrücke mit einer erklärenden Dokumentation versehen.

☞ Haben Sie bereits in anderen Sprachen (wie Perl 5) reguläre Ausdrücke eingesetzt, so wird Ihnen die Python-Syntax sehr bekannt vorkommen.

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_6, © Springer-Verlag Berlin Heidelberg 2010

6.2 Fallbeispiel: Adresse

Zu diesen Beispielen inspirierte mich ein Alltagsproblem, das ich vor einigen Jahren während meiner Arbeit hatte. Ich musste Adressen bereinigen und vereinheitlichen, um sie von einem alten System in ein neueres zu übertragen. (Sehen Sie, ich erfinde dieses Zeug nicht einfach, es ist sogar nützlich.) Das folgende Beispiel zeigt, wie ich an das Problem herangegangen bin.

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')                ①
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')                ②
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.')  ③
'100 NORTH BROAD RD.'
>>> import re                               ④
>>> re.sub('ROAD$', 'RD.', s)               ⑤
'100 NORTH BROAD RD.'
```

① Mein Ziel ist es, die Adresse so zu vereinheitlichen, dass `'ROAD'` immer mit `'RD.'` abgekürzt wird. Auf den ersten Blick nahm ich an, dass ich dies recht einfach mit der Stringmethode `replace()` erreichen könnte. Alle Daten lagen bereits in Großschreibung vor, das sollte also nicht das Problem sein. Der Suchstring, `'ROAD'`, war außerdem eine Konstante. Und tatsächlich, in diesem einfachen Fall funktioniert `s.replace()`.

② Doch das Leben ist leider voller Gegenbeispiele und so fand ich sehr schnell dieses. Das Problem besteht hier darin, dass `'ROAD'` zweimal innerhalb der Adresse auftritt, einmal als Teil des Straßennamens `'BROAD'` und einmal als eigenes Wort. Die `replace()`-Methode ersetzt nun beide Vorkommen, während ich zusehe, wie meine Adresse zerstört wird.

③ Um das Problem mehrerer Vorkommen von `'ROAD'` innerhalb von Adressen zu lösen, könnten Sie auf etwas dieser Art zurückgreifen: Suche und ersetze `'ROAD'` nur dann, wenn es innerhalb der letzten vier Zeichen der Adresse (`s[-4:]`) vorkommt und kümmere dich nicht um den Rest des Strings (`s[:-4]`). Doch Sie sehen sicher bereits, dass dies jetzt schon ziemlich unflexibel wird. Das Muster ist z. B. abhängig von der Länge des Strings, den es zu ersetzen gilt. (Wollten Sie `'STREET'` durch `'ST.'` ersetzen, müssten Sie `s[:-6]` und `s[-6:].replace(…)` benutzen.) Würden Sie sich dies gerne in sechs Monaten noch einmal ansehen und Fehler beseitigen? Ich jedenfalls nicht.

④ Es ist an der Zeit, reguläre Ausdrücke einzusetzen. In Python befindet sich die gesamte Funktionalität der regulären Ausdrücke im Modul `re`.

⑤ Sehen Sie sich den ersten Parameter an: `'ROAD$'`. Dies ist ein einfacher regulärer Ausdruck, der `'ROAD'` nur dann entspricht, wenn es am Ende eines Strings

auftaucht. Das `$` bedeutet „Ende des Strings“. (Das Gegenstück dazu ist das Zeichen `^`, das „Anfang des Strings“ bedeutet.) Durch Verwendung der `re.sub()`-Funktion durchsuchen Sie den String `s` nach dem regulären Ausdruck `'ROAD$'` und ersetzen ihn mit `'RD.'`. Damit wird das `'ROAD'` am Ende des Strings `s` ersetzt, jedoch nicht das `'ROAD'`, das Teil des Wortes `'BROAD'` ist, denn dieses befindet sich ja in der Mitte von `s`.

Weiterhin mit der Bereinigung der Adressen beschäftigt, entdeckte ich, dass das vorherige Beispiel nicht gut genug war. Nicht alle Adressen enthalten eine Straßenbezeichnung. Einige Adressen enden einfach mit dem Straßennamen. Die meiste Zeit über klappte meine Methode, doch wenn der Straßenname `'BROAD'` wäre, würde der reguläre Ausdruck `'ROAD'` als Teil von `'BROAD'` am Ende des Strings finden. Das ist nicht das was ich wollte.

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)   ①
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)   ②
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)   ③
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s)  ④
'100 BROAD RD. APT 3'
```

① Eigentlich wollte ich, dass `'ROAD'` nur dann erkannt wird, wenn es am Ende des Strings steht und ein eigenes Wort ist (und nicht Teil eines längeren Wortes). Um dies als regulären Ausdruck darzustellen, verwendet man `\b`, was so viel bedeutet wie „genau hier muss sich eine Wortgrenze befinden“. In Python ist dies etwas kompliziert, da dem `\`-Zeichen innerhalb eines Strings zuerst die Sonderbedeutung genommen werden muss. Dieser Umstand wird manchmal als *Backslash-Plage* bezeichnet und ist einer der Gründe dafür, dass reguläre Ausdrücke in Perl einfacher zu verwenden sind als in Python. Andererseits vermischt Perl normale Syntax und reguläre Ausdrücke, was dazu führt, dass Bugs schwer zu finden sein können, da man nicht weiß, ob sich der Fehler in der Syntax oder im regulären Ausdruck befindet.

② Um die *Backslash-Plage* zu umgehen, können Sie einen sogenannten *Raw-String* verwenden, indem Sie vor den String den Buchstaben `r` setzen. Dies teilt Python mit, dass er bei diesem String nichts *escapen* (Sonderbedeutung des `\`-Zeichens) soll. `'\t'` ist das Zeichen für einen Tabulator, doch `r'\t'` ist ein Backslash gefolgt vom Buchstaben `t`. Ich empfehle Ihnen, immer *Raw-Strings* zu verwenden, wenn Sie mit regulären Ausdrücken arbeiten; andernfalls werden sie schnell zu verwirrend (und reguläre Ausdrücke sind von Haus aus schon verwirrend genug).

③ **Seufz** Leider fand ich recht bald weitere Fälle, die meiner Logik widersprachen. In einem der Fälle enthielt die Adresse zwar `'ROAD'` als ein eigenes Wort, doch es befand sich nicht am Ende, da nach der Straße noch eine Apartmentnummer folgte. Da sich `'ROAD'` aber eben nicht am Ende des Strings befindet, stimmt es nicht mit dem Suchmuster überein und wird nicht gefunden. `re.sub()` ersetzt also rein gar nichts und Sie erhalten den ursprünglichen String, was sie nicht wollen.

④ Um dieses Problem zu lösen entfernte ich das `$`-Zeichen und fügte ein weiteres `\b` hinzu. So bedeutet der reguläre Ausdruck nun „finde `'ROAD'`, wenn es ein ganzes eigenes Wort irgendwo innerhalb des Strings ist“, also entweder am Ende, am Anfang, oder irgendwo dazwischen.

6.3 Fallbeispiel: römische Zahlen

Sie haben sicher schon römische Zahlen gesehen, auch wenn Sie sie gar nicht bemerkt haben. Sie könnten sie beim Copyright-Vermerk alter Filme oder Fernsehserien gesehen haben („Copyright MCMXLVI“ anstatt „Copyright 1946“), oder bei Bibliotheken oder Universitäten („errichtet MDCCCLXXXVIII“ anstatt „errichtet 1888“). Sie könnten sie auch auf Grundrissen oder bei bibliografischen Angaben gesehen haben. Das System hinter dieser Zahldarstellung stammt aus dem antiken Römischen Reich (daher der Name).

Es gibt insgesamt sieben Zeichen, mit denen durch Wiederholung und Kombination Zahlen dargestellt werden können.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

Im Folgenden einige allgemeine Regeln zum Bilden römischer Zahlen:

- Die Zeichen werden zusammengezählt. I ist 1, II ist 2 und III ist 3. VI ist 6 („5 und 1“), VII ist 7 und VIII ist 8.
- Die Zehner-Zeichen (I, X, C und M) können bis zu drei Mal wiederholt werden. Bei 4 müssen Sie vom nächsthöheren Fünfer-Zeichen subtrahieren. Sie können 4 nicht als IIII darstellen, sondern nur als IV („1 weniger als 5“). Die Zahl 40 wird als XL geschrieben („10 weniger als 50“), 41 als XLI, 42 als XLII, 43 als XLIII und schließlich 44 als XLIV („10 weniger als 50, und dann 1 weniger als 5“).
- Genauso müssen Sie auch bei 9 verfahren. Hier müssen Sie vom nächsthöheren Zehner-Zeichen subtrahieren: 8 ist VIII, doch 9 ist IX („1 weniger als 10“), nicht

VIIII (da das Zeichen I nicht vier Mal wiederholt werden darf). Die Zahl 90 wird somit als XC dargestellt und 900 als CM.
- Die Fünfer-Zeichen dürfen niemals wiederholt werden. Die Zahl 10 wird immer als X dargestellt, niemals als VV. Die Zahl 100 ist immer C, niemals LL.
- Römische Zahlen werden von hoch nach niedrig geschrieben und von links nach rechts gelesen. Die Reihenfolge der Zeichen spielt daher eine sehr wichtige Rolle. DC ist 600; CD ist eine völlig andere Zahl (400, „100 weniger als 500“). CI ist 101; IC ist nicht einmal eine gültige römische Zahl (man kann 1 nicht direkt von 100 subtrahieren, sondern müsste XCIX schreiben, „10 weniger als 100, und dann 1 weniger als 10“).

6.3.1 Prüfen der Tausender

Was benötigt man, um herauszufinden, ob ein beliebiger String eine gültige römische Zahl ist? Lassen Sie uns eine Ziffer nach der anderen ansehen. Da römische Zahlen immer mit der höchsten Ziffer beginnen, sollten wir es auch so machen. Wir fangen also mit den Tausendern an. Für Zahlen ab 1.000 werden die Tausender durch eine Reihe von M-Zeichen dargestellt.

```
>>> import re
>>> pattern = '^M?M?M?$'        ①
>>> re.search(pattern, 'M')     ②
<_sre.SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')    ③
<_sre.SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')   ④
<_sre.SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM')  ⑤
>>> re.search(pattern, '')      ⑥
<_sre.SRE_Match object at 0106F4A8>
```

① Dieses Suchmuster besteht aus drei Teilen. Das `^`-Zeichen sorgt dafür, dass der folgende Ausdruck nur am Anfang eines Strings gefunden wird. Wäre dies nicht angegeben, so würde das Muster jedes `M` finden, egal an welcher Stelle es sich befindet. Das wollen Sie nicht. Sie wollen sicherstellen, dass das `M`-Zeichen nur gefunden wird, wenn es am Anfang eines Strings steht. `M?` sucht ein optionales `M`-Zeichen. Da dies dreimal wiederholt wird, findet das Muster 0 bis 3 `M`-Zeichen hintereinander. Das `$`-Zeichen steht für das Ende des Strings. In Kombination mit dem `^`-Zeichen am Anfang sorgt dies dafür, dass nur Strings erkannt werden, die einzig und allein aus `M`-Zeichen bestehen.

② Das Herzstück des `re`-Moduls ist die `search()`-Funktion, die einen regulären Ausdruck (`pattern`) und einen String (`'M'`) übernimmt und versucht, das

Muster innerhalb des Strings zu finden. Hat `search()` etwas gefunden, gibt die Funktion ein Objekt zurück, das verschiedene Methoden zur Beschreibung des Fundes besitzt; findet `search()` nichts, wird `None` zurückgegeben, der Null-Wert von Python. Alles was Sie momentan interessieren sollte ist, ob das Muster gefunden wurde. Das sehen Sie schon am Rückgabewert von `search().'M'` stimmt hier mit dem regulären Ausdruck überein, da das erste optionale `M`-Zeichen passt und sowohl das zweite als auch das dritte ignoriert werden.

③ `'MM'` wird gefunden, da das erste und das zweite optionale `M`-Zeichen passen und das dritte ignoriert wird.

④ `'MMM'` wird gefunden, da alle drei `M`-Zeichen passen.

⑤ `'MMMM'` wird nicht gefunden. Alle drei `M`-Zeichen sind vorhanden, doch dann besteht der reguläre Ausdruck auf dem Ende des Strings (wegen des `$`-Zeichens), doch der String endet noch nicht (aufgrund des vierten `M`-Zeichens). `search()` gibt also `None` zurück.

⑥ Interessanterweise stimmt auch ein leerer String mit dem Muster überein, da alle `M`-Zeichen optional sind.

6.3.2 Prüfen der Hunderter

Die Hunderter sind schwieriger zu handhaben als die Tausender, da es je nach dem Wert verschiedene Möglichkeiten der Darstellung gibt, die sich gegenseitig ausschließen.

- 100 = C
- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

Es gibt also vier mögliche Muster:

- CM
- CD
- Null bis drei C-Zeichen (null, wenn die Hunderterstelle 0 ist)
- D, gefolgt von null bis drei C-Zeichen

Die beiden letzten Muster können kombiniert werden.

- Ein optionales D, gefolgt von null bis drei C-Zeichen

Dieses Beispiel zeigt, wie man die Hunderterstelle einer römischen Zahl überprüft.

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$'  ①
>>> re.search(pattern, 'MCM')             ②
<_sre.SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')              ③
<_sre.SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')          ④
<_sre.SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')            ⑤
>>> re.search(pattern, '')                ⑥
<_sre.SRE_Match object at 01071D98>
```

① Dieses Muster beginnt genau wie das vorherige. Zuerst prüft es den Anfang des Strings (`^`), dann die Tausenderstelle (`M?M?M?`). Darauf folgt – in Klammern – der neue Teil, der drei sich gegenseitig ausschließende Muster angibt, welche wiederum durch vertikale Balken getrennt sind: `CM`, `CD` und `D?C?C?C?` (das ist ein optionales `D` gefolgt von null bis drei optionalen `C`-Zeichen). Der Parser für reguläre Ausdrücke sucht der Reihe nach jedes der Muster (von links nach rechts), benutzt das erste das passt und ignoriert den Rest.

② `'MCM'` wird gefunden, da das erste `M` passt, das zweite und dritte `M`-Zeichen ignoriert wird und das `CM` ebenfalls passt (die Muster `CD` und `D?C?C?C?` werden also gar nicht erst in die Prüfung mit einbezogen). `MCM` ist die römische Zahl für `1900`.

③ `'MD'` wird gefunden, weil das erste `M` passt, das zweite und dritte `M`-Zeichen ignoriert wird und das Muster `D?C?C?C?` zu `D` passt (jedes der `C`-Zeichen ist optional und wird ignoriert). `MD` ist die römische Zahl für `1500`.

④ `'MMMCCC'` wird gefunden, da alle drei `M`-Zeichen passen und das Muster `D?C?C?C?` zu `CCC` passt (das `D` ist optional und wird ignoriert). `MMMCCC` ist die römische Zahl für `3300`.

⑤ `'MCMC'` wird nicht gefunden. Das erste `M` passt, das zweite und dritte `M`-Zeichen wird ignoriert und das `CM` passt. Doch dann passt das `$`-Zeichen nicht, da wir uns noch nicht am Ende des Strings befinden (es ist immer noch ein `C`-Zeichen vorhanden). Das `C` passt nicht zum Muster `D?C?C?C?`, da sich die Muster gegenseitig ausschließen und das Muster `CM` bereits gefunden wurde.

⑥ Interessanterweise passt auch ein leerer String zu dem Muster, da alle `M`-Zeichen optional sind und ignoriert werden. Der leere String passt außerdem auch zum Muster `D?C?C?C?`, da auch hier alle Zeichen optional sind und ignoriert werden.

Puh! Haben Sie bemerkt, wie schnell reguläre Ausdrücke wirklich unangenehm werden können? Und Sie haben gerade mal die Tausender und Hunderter hinter sich gebracht. Wenn Sie aber all dem gefolgt sind, dann werden die Zehner und Einer ein Spaziergang, da sie genau dasselbe Muster verwenden. Lassen Sie uns aber eine andere Möglichkeit zur Darstellung des Musters ansehen.

6.4 Verwenden der `{n,m}`-Syntax

Im vorherigen Abschnitt hatten Sie es mit einem Muster zu tun, bei dem dasselbe Zeichen bis zu dreimal wiederholt werden konnte. Bei den regulären Ausdrücken gibt es noch eine andere Möglichkeit dies auszudrücken. Diese Methode finden einige Leute lesbarer. Sehen Sie sich zuerst die im vorherigen Beispiel angewandte Methode an.

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')     ①
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')    ②
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')   ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM')  ④
>>>
```

① Hier passen der Beginn des Strings und das erste optionale `M`, nicht jedoch das zweite und dritte `M` (das ist aber in Ordnung, da sie optional sind). Das Ende des Strings passt dann wieder.

② Hier passen der Beginn des Strings und das erste und das zweite optionale `M`, nicht aber das dritte `M` (das ist aber in Ordnung, da es optional ist). Das Ende des Strings passt dann wieder.

③ Hier passen der Beginn des Strings und alle drei optionalen `M`-Zeichen. Das Ende des Strings passt hier ebenfalls.

④ Hier passen der Beginn des Strings und alle drei optionalen `M`-Zeichen. Das Ende des Strings passt hier jedoch nicht (da immer noch ein `M` übrig ist). Das Muster wird also nicht gefunden und gibt `None` zurück.

```
>>> pattern = '^M{0,3}$'        ①
>>> re.search(pattern, 'M')     ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')    ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')   ④
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM')  ⑤
>>>
```

① Dieses Muster bedeutet: „Suche den Beginn des Strings, dann null bis drei M-Zeichen und dann das Ende des Strings.“ 0 und 3 können beliebige Zahlen sein; wollen Sie, dass mindestens ein M vorhanden ist, aber nicht mehr als drei, könnten Sie sagen M{1,3}.

② Hier passen der Beginn des Strings, eines der möglichen drei M-Zeichen und das Ende des Strings.

③ Hier passen der Beginn des Strings, zwei der möglichen drei M-Zeichen und das Ende des Strings.

④ Hier passen der Beginn des Strings, drei der möglichen drei M-Zeichen und das Ende des Strings.

⑤ Hier passen der Beginn des Strings und drei der möglichen drei M-Zeichen, doch nicht das Ende des Strings. Der reguläre Ausdruck erlaubt nur bis zu drei M-Zeichen vor dem Ende des Strings, doch hier sind es vier. Das Muster passt hier also nicht und gibt None zurück.

6.4.1 Prüfen der Zehner und Einer

Lassen Sie uns den regulären Ausdruck nun so erweitern, dass er auch die Zehner und Einer umfasst. Dieses Beispiel zeigt das Prüfen der Zehner.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')     ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')      ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')     ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')   ④
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX')  ⑤
>>>
```

① Hier passen der Beginn des Strings, das erste optionale M, außerdem CM, XL und schließlich das Ende des Strings. Denken Sie daran, dass (A|B|C) bedeutet „finde A *oder* B *oder* C“. Hier passt XL, also wird XC und L?X?X?X? ignoriert und zum Ende des Strings gegangen. MCML ist die römische Zahl für 1940.

② Hier passen der Beginn des Strings, das erste optionale M, CM und L?X?X?X?. Vom Muster L?X?X?X? passt hier das L; die optionalen X-Zeichen werden ignoriert. Dann wird zum Ende des Strings gegangen. MCML ist die römische Zahl für 1950.

③ Hier passen der Beginn des Strings sowie das erste optionale `M`, `CM`, das optionale `L` und das optionale `X`. Das zweite und dritte optionale `X` wird ignoriert. Das Ende des Strings passt ebenfalls. `MCMLX` ist die römische Zahl für `1960`.

④ Hier passen der Beginn des Strings, das erste optionale `M`, `CM`, das optionale `L`, alle drei optionalen `X`-Zeichen und das Ende des Strings. `MCMLXXX` ist die römische Zahl für `1980`.

⑤ Hier passen der Beginn des Strings, das erste optionale `M`, `CM`, das optionale `L` und alle drei optionalen `X`-Zeichen. Das Ende des Strings passt nicht, da es noch ein unbeachtetes `X` gibt. Das komplette Muster wird also nicht erkannt und gibt `None` zurück. `MCMLXXXX` ist keine gültige römische Zahl.

Der Ausdruck für die Einer folgt demselben Muster. Ich erspare Ihnen die Details und zeige Ihnen das Ergebnis.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

Wie sieht dies jetzt also unter Verwendung der `{n,m}`-Syntax aus? Dieses Beispiel zeigt die neue Syntax.

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')              ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI')          ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII')   ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I')                 ④
<_sre.SRE_Match object at 0x008EEB48>
```

① Hier passen der Beginn des Strings, eines von möglichen drei `M`-Zeichen und `D?C{0,3}`. Davon passt das optionale `D` und null der möglichen drei `C`-Zeichen. Außerdem wird `L?X{0,3}` erkannt, da das optionale `L` und null der möglichen drei `X`-Zeichen passen. Auch `V?I{0,3}` wird gefunden, da das optionale `V` und null von möglichen drei `I`-Zeichen passen. Das Ende des Strings passt ebenso. `MDLV` ist die römische Zahl für `1555`.

② Hier passen der Beginn des Strings, zwei von möglichen drei `M`-Zeichen und `D?C{0,3}` mit einem `D` und einem von möglichen drei `C`-Zeichen. Außerdem wird `L?X{0,3}` mit einem `L` und einem von möglichen drei `X`-Zeichen erkannt. Auch `V?I{0,3}` wird mit einem `V` und einem von möglichen drei `I`-Zeichen erkannt. Das Ende des Strings passt ebenso. `MMDCLXVI` ist die römische Zahl für `2666`.

③ Hier passen der Beginn des Strings, drei von möglichen drei `M`-Zeichen und `D?C{0,3}` mit einem `D` und drei von möglichen drei `C`-Zeichen. Außerdem wird `L?X{0,3}` mit einem `L` und drei von möglichen drei `X`-Zeichen erkannt. Auch `V?I{0,3}` wird mit einem `V` und drei von möglichen drei `I`-Zeichen erkannt. Das Ende des Strings passt ebenfalls. `MMMDCCCLXXXVIII` ist die römische Zahl

für `3888` und die längste mit römischen Zahlen ohne Erweiterung darstellbare Zahl.

④ Sehen Sie her. (Ich fühle mich wie ein Zauberer. „Seht her, Kinder! Ich werde ein Kaninchen aus meinem Hut zaubern.“) Hier passen der Beginn des Strings, null von drei `M`-Zeichen, `D?C{0,3}`, da das optionale `D` ignoriert wird und null von drei `C`-Zeichen erkannt werden, `L?X{0,3}`, da das optionale `L` ignoriert wird und null von drei `X`-Zeichen erkannt werden, `V?I{0,3}`, da das optionale `V` ignoriert wird und null von drei `I`-Zeichen erkannt werden. Auch das Ende des Strings passt hier. Whoa!

Wenn Sie all das verstanden haben, sind Sie besser dran als ich es war. Stellen Sie sich nun vor, Sie versuchten die regulären Ausdrücke eines anderen inmitten einer kritischen Funktion eines großen Programms zu verstehen. Oder stellen Sie sich bloß vor, dass Sie nach einigen Monaten Ihre eigenen regulären Ausdrücke ansehen. Ich habe es getan und kann Ihnen sagen, dass es kein schöner Anblick ist.

Lassen Sie uns nun eine alternative Syntax ansehen, die Ihnen behilflich sein kann, Ihre Ausdrücke wartungsfähig zu halten.

6.5 Ausführliche reguläre Ausdrücke

Bisher haben Sie es nur mit – wie ich sie nenne – „kompakten“ regulären Ausdrücken zu tun gehabt. Wie Sie gesehen haben, sind diese schwer zu lesen und selbst wenn Sie herausfinden, was ein Ausdruck tut, ist das keine Garantie dafür, dass Sie es auch nach sechs Monaten noch wissen. Was Sie also brauchen ist eine beigefügte Dokumentation.

Python erlaubt Ihnen genau das durch sogenannte *verbose regular expressions* (ausführliche reguläre Ausdrücke; Anm. d. Übers.). Ein ausführlicher regulärer Ausdruck unterscheidet sich von einem kompakten regulären Ausdruck auf zweierlei Arten:

- Whitespace wird ignoriert. Leerzeichen, Tabulatoren und Zeilenumbrüche werden nicht als Leerzeichen, Tabulatoren und Zeilenumbrüche erkannt. Sie werden überhaupt nicht erkannt. (Möchten Sie, dass ein Leerzeichen in einem ausführlichen regulären Ausdruck erkannt wird, müssen Sie es escapen, indem Sie einen Backslash davor setzen.)
- Kommentare werden ignoriert. Ein Kommentar innerhalb eines ausführlichen regulären Ausdrucks ist genau wie ein Kommentar im Python-Code: er beginnt mit einem #-Zeichen und geht bis zum Ende der Zeile. In diesem Fall liegt ein Kommentar in einem mehrzeiligen String vor statt im Quellcode, doch die Funktionsweise ist die gleiche.

Ein Beispiel wird dies verdeutlichen. Sehen wir uns den bisher genutzten kompakten regulären Ausdruck an und machen daraus einen ausführlichen regulären Ausdruck. Dieses Beispiel zeigt wie.

```
>>> pattern = '''
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    '''
>>> re.search(pattern, 'M', re.VERBOSE)                ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)        ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMDCCCLXXXVIII', re.VERBOSE)  ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')                            ④
```

① Das Wichtigste das Sie beim Verwenden von ausführlichen regulären Ausdrücken bedenken müssen, ist dass Sie ein zusätzliches Argument übergeben müssen: `re.VERBOSE` ist eine im `re`-Modul definierte Konstante, die anzeigt, dass das Muster als ausführlicher regulärer Ausdruck behandelt werden soll. Wie Sie sehen, besitzt dieses Muster sehr viel Whitespace (der komplett ignoriert wird) und einige Kommentare (die komplett ignoriert werden). Vernachlässigen Sie den Whitespace und die Kommentare, so ist dies exakt der gleiche reguläre Ausdruck wie Sie ihn im vorherigen Abschnitt gesehen haben. Er ist jedoch viel lesbarer.

② Hier wird der Beginn des Strings, eines von möglichen drei `M`-Zeichen, `CM`, `L` und drei von möglichen drei `X`-Zeichen, `IX` und schließlich das Ende des Strings erkannt.

③ Hier wird der Beginn des Strings, drei von möglichen drei `M`-Zeichen, `D` und drei von möglichen drei `C`-Zeichen, `L` und drei von möglichen drei `X`-Zeichen, `V` und drei von möglichen drei `I`-Zeichen und schließlich das Ende des Strings erkannt.

④ Hier wird nichts erkannt. Warum nicht? Es ist kein `re.VERBOSE`-Flag vorhanden. Die `re.search()`-Funktion behandelt das Muster wie einen kompakten regulären Ausdruck mit erheblichem Whitespace und Raute-Symbolen. Python kann nicht automatisch feststellen, ob ein regulärer Ausdruck ausführlich ist oder nicht. Stattdessen nimmt Python an, dass jeder reguläre Ausdruck kompakt ist, es sei denn Sie geben explizit an, dass er ausführlich ist.

6.6 Fallbeispiel: Telefonnummern gliedern

Bis jetzt haben Sie sich darauf konzentriert, ganze Muster zu erkennen. Entweder das Muster passt, oder es passt nicht. Doch reguläre Ausdrücke sind noch weit mächtiger. Passt ein regulärer Ausdruck, dann können Sie bestimmte Teile davon herauspicken. Sie können herausfinden, was wo erkannt wurde.

Dieses Beispiel stammt von einem weiteren wirklichen Problem dem ich mich während meiner früheren Arbeit gegenübersah. Das Problem: eine US-amerikanische Telefonnummer gliedern. Der Kunde wollte die Möglichkeit haben, die Nummer in jeder beliebigen Form einzugeben (in ein einzelnes Eingabefeld). In der Datenbank seines Unternehmens sollte die Nummer dann jedoch aufgeteilt in Ortsvorwahl, Amt, Teilnehmernummer und eine eventuelle Durchwahl gespeichert werden. Ich durchkämmte das Internet und habe viele Beispiele für reguläre Ausdrücke gefunden, die behaupteten genau dies zu können, doch keines davon war tolerant genug.

Hier sind die Telefonnummern, die ich erkennen musste:

- `800-555-1212`
- `800 555 1212`
- `800.555.1212`
- `(800) 555-1212`
- `1-800-555-1212`
- `800-555-1212-1234`
- `800-555-1212x1234`
- `800-555-1212 ext. 1234`
- `work 1-(800) 555.1212 #1234`

Ziemlich viele Möglichkeiten! In jedem der Fälle musste ich wissen, dass die Ortsvorwahl `800` lautete, das Amt `555` und der Rest der Telefonnummer `1212`. Bei den Nummern die eine Durchwahl hatten, musste ich außerdem wissen, dass diese `1234` war.

Lassen Sie uns eine Lösung zum Gliedern von Telefonnummern erarbeiten. Dieses Beispiel zeigt den ersten Schritt.

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$')  ①
>>> phonePattern.search('800-555-1212').groups()             ②
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')                 ③
>>> phonePattern.search('800-555-1212-1234').groups()        ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

① Lesen Sie reguläre Ausdrücke immer von links nach rechts. Dieser hier vergleicht den Beginn des Strings und `({\d{3})`. Was bedeutet `\d{3}`? Nun, `\d` bedeutet „jede beliebige Ziffer“ (`0` bis `9`). `{3}` heißt „vergleiche genau drei Ziffern“; es stellt eine Variante der eben gesehenen `{n,m}`-Syntax dar. Setzt man dies alles in Klammern heißt das „suche genau drei Ziffern und behalte Sie als Gruppe, nach der ich später fragen kann“. Suche dann einen Bindestrich. Dann erneut eine Gruppe von genau drei Ziffern. Dann wieder einen Bindestrich. Dann eine weitere Gruppe von genau vier Ziffern. Dann das Ende des Strings.

② Um Zugang zu den Gruppen zu erhalten, die der Parser für reguläre Ausdrücke sich gemerkt hat, verwenden Sie die `groups()`-Methode des Objekts, das die

`search()`-Methode zurückgibt. Die Methode gibt dann ein Tupel dieser Gruppen zurück. Im vorliegenden Fall haben Sie drei Gruppen definiert, eine mit drei Ziffern, eine weitere mit drei Ziffern und eine mit vier Ziffern.

③ Dieser reguläre Ausdruck ist nicht die endgültige Lösung, da er nicht mit einer Durchwahl am Ende einer Telefonnummer umgehen kann. Um dies zu erreichen, müssen Sie den regulären Ausdruck erweitern.

④ Hier sehen Sie, warum Sie im fertigen Code niemals die `search()`-Methode und die `groups()`-Methode „verketten" sollten. Gibt die `search()`-Methode keinen Treffer zurück, wird `None` geliefert. `None.groups()` aber verursacht eine eindeutige Ausnahme: `None` besitzt keine Methode namens `groups()`. (Natürlich ist dies weniger eindeutig, wenn die Ausnahme irgendwo versteckt in Ihrem Code auftritt. Ja, ich spreche aus Erfahrung.)

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$')  ①
>>> phonePattern.search('800-555-1212-1234').groups()              ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                       ③
>>>
>>> phonePattern.search('800-555-1212')                            ④
>>>
```

① Dieser reguläre Ausdruck ist beinahe mit dem vorherigen identisch. Genau wie vorhin wird auch hier der Beginn des Strings verglichen, dann eine zu merkende Gruppe von drei Ziffern, ein Bindestrich, wieder eine zu merkende Gruppe von drei Ziffern, ein weiterer Bindestrich und eine zu merkende Gruppe von vier Ziffern. Neu ist hier, dass hier ein weiterer Bindestrich und schließlich eine zu merkende Gruppe von einer oder mehr Ziffern gesucht werden. Dann erst das Ende des Strings.

② Die `groups()`-Methode gibt nun ein Tupel von vier Elementen zurück, da der reguläre Ausdruck jetzt vier Gruppen angibt, die der Parser sich merken soll.

③ Leider ist auch dieser reguläre Ausdruck keine endgültige Lösung. Er geht davon aus, dass die verschiedenen Teile der Telefonnummer durch Bindestriche getrennt sind. Was also passiert, wenn sie durch Leerzeichen, Kommas, oder Punkte getrennt werden. Sie benötigen eine allgemeinere Lösung, die die verschiedenen Arten der Trennung berücksichtigt.

④ Huch! Dieser reguläre Ausdruck tut nicht nur nicht alles was Sie wollen, er ist sogar ein Rückschritt, weil Sie nun keine Telefonnummern ohne eine Durchwahl gliedern können. Das ist absolut nicht das was Sie wollten. Wenn eine Durchwahl vorhanden ist, wollen Sie diese wissen. Wenn Sie jedoch nicht vorhanden ist, wollen Sie trotzdem die einzelnen Teile der Hauptnummer erhalten.

Das nächste Beispiel zeigt, wie man den regulären Ausdruck so gestaltet, dass er mit verschiedenen Trennzeichen zwischen den einzelnen Teilen der Telefonnummer umgehen kann.

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$')  ①
>>> phonePattern.search('800 555 1212 1234').groups()  ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()  ③
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234')              ④
>>>
>>> phonePattern.search('800-555-1212')                ⑤
>>>
```

① Halten Sie sich fest! Hier vergleichen Sie den Beginn des Strings, eine Gruppe von drei Ziffern und `\D+`. Was ist das nun wieder? Nun, `\D` entspricht jedem beliebigen Zeichen ausgenommen einer Ziffer. Das + bedeutet „`1` oder mehr“. `\D+` entspricht also einem oder mehr Zeichen, die keine Ziffern sind. Dies benutzen Sie anstelle eines Bindestrichs, um verschiedene Trennzeichen zu erkennen.

② Die Verwendung von `\D+` führt dazu, dass Sie nun Telefonnummern erkennen können, bei denen das Trennzeichen ein Leerzeichen ist.

③ Natürlich werden auch Bindestriche weiterhin erkannt.

④ Leider ist dies immer noch nicht die endgültige Lösung, weil davon ausgegangen wird, dass überhaupt ein Trennzeichen vorhanden ist. Was passiert nun aber, wenn die Telefonnummer ohne Leerzeichen oder Bindestriche eingegeben wird?

⑤ Huch! Dies hat immer noch nicht unser Problem mit der Durchwahl beseitigt. Sie stehen nun vor zwei Problemen, doch diese können Sie beide mit derselben Technik lösen.

Das folgende Beispiel zeigt den regulären Ausdruck, der auch Telefonnummern ohne Trennzeichen verarbeiten kann.

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  ①
>>> phonePattern.search('80055512121234').groups()      ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()  ③
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()        ④
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')           ⑤
>>>
```

① Die einzige Änderung die Sie gegenüber dem letzten Schritt durchgeführt haben besteht darin, dass alle +-Zeichen durch *-Zeichen ersetzt wurden. Statt zwischen den Teilen der Telefonnummer nach `\D+` zu suchen, suchen Sie nun nach `\D*`. Erinnern Sie sich, dass + „`1` oder mehr“ bedeutet? Nun, * bedeutet „`0` oder mehr“. Sie sollten jetzt also auch Telefonnummern gliedern können, die gar kein Trennzeichen beinhalten.

② Und siehe da, es funktioniert! Warum? Sie haben den Beginn des Strings verglichen, dann eine zu merkende Gruppe von drei Ziffern (`800`), null nicht-nu-

merische Zeichen, wieder eine zu merkende Gruppe von drei Ziffern (`555`), null nicht-numerische Zeichen, eine zu merkende Gruppe von vier Ziffern (`1212`), null nicht-numerische Zeichen, eine zu merkende Gruppe von einer variablen Anzahl an Ziffern (`1234`) und dann das Ende des Strings.

③ Auch andere Varianten funktionieren nun: Punkte anstelle von Bindestrichen, und sowohl ein Leerzeichen wie auch ein `x` vor der Durchwahl.

④ Endlich haben Sie unser Problem gelöst: eine Durchwahl ist nun wieder optional. Ist keine Durchwahl vorhanden gibt die `groups()`-Methode zwar weiterhin ein Tupel mit vier Elementen zurück, doch das vierte Element ist nur ein leerer String.

⑤ Ich hasse es der Überbringer schlechter Nachrichten zu sein, aber Sie sind noch nicht fertig. Was ist das Problem? Hier gibt es ein zusätzliches Zeichen vor der Ortsvorwahl. Der reguläre Ausdruck geht jedoch davon aus, dass die Ortsvorwahl das Erste ist womit der String beginnt. Kein Problem. Sie können dieselbe Technik der „`0` oder mehr nicht-numerischen Zeichen“ nutzen, um führende Zeichen vor der Ortsvorwahl zu ignorieren.

Das nachfolgende Beispiel zeigt, wie man führende Zeichen in Telefonnummern handhabt.

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  ①
>>> phonePattern.search('(800)5551212 ext. 1234').groups()                  ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                            ③
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')                      ④
>>>
```

① Dies ist dasselbe wie im vorherigen Beispiel, abgesehen davon, dass Sie jetzt vor der ersten zu merkenden Gruppe (die Ortsvorwahl) nach `\D*` suchen. Beachten Sie, dass der Parser sich diese nicht-numerischen Zeichen nicht merkt (sie sind nicht in Klammern eingeschlossen). Werden Sie gefunden, dann werden sie einfach ignoriert und der Parser merkt sich die Ortsvorwahl wann immer sie beginnt.

② Sie können nun die Telefonnummer auch mit der führenden linken Klammer gliedern. (Die rechte Klammer nach der Ortsvorwahl wurde schon vorher verarbeitet; sie wird als nicht-numerisches Trennzeichen behandelt, das von `\D*` nach der ersten zu merkenden Gruppe erkannt wird.)

③ Dies ist nur ein Funktionstest, um sicherzustellen, dass nichts kaputt gegangen ist, was vorher funktioniert hat. Da die führenden Zeichen optional sind, werden hier der Beginn des Strings, null nicht-numerische Zeichen, eine zu merkende Gruppe von drei Ziffern (`800`), ein nicht-numerisches Zeichen (der Bindestrich), eine zu merkende Gruppe von drei Ziffern (`555`), ein nicht-numerisches Zeichen (der Bindestrich), eine zu merkende Gruppe von vier Ziffern (`1212`), null nicht-numerische Zeichen, eine zu merkende Gruppe von null Ziffern und schließlich das Ende des Strings erkannt.

④ Hier sehen wir, warum ich mir angesichts regulärer Ausdrücke manchmal die Augen mit einem stumpfen Gegenstand ausstechen will. Warum wird diese Telefon-

nummer nicht erkannt? Ganz einfach: es befindet sich eine `1` vor der Ortsvorwahl, doch Sie sind davon ausgegangen, dass alle führenden Zeichen vor der Ortsvorwahl nicht-numerische Zeichen sind (`\D*`). Oh je!

Lassen Sie uns kurz zurückblicken. Bisher haben all unsere regulären Ausdrücke nach dem Beginn des Strings gesucht. Sie sehen nun aber, dass am Beginn des Strings eine unbestimmte Anzahl an Dingen stehen kann die Sie ignorieren wollen. Statt nun zu versuchen, all diese Dinge zu erkennen und zu überspringen, gehen wir es anders an: suchen Sie gar nicht erst den Beginn des Strings. Diese Vorgehensweise wird im nächsten Beispiel gezeigt.

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  ①
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()         ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                                 ③
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234')                               ④
('800', '555', '1212', '1234')
```

① Beachten Sie bei diesem regulären Ausdruck das Fehlen des ^-Zeichens. Sie suchen nicht länger den Beginn des Strings. Es gibt kein Gesetz, das Sie dazu zwingt, die komplette Eingabe mit Ihrem regulären Ausdruck zu erkennen. Die Engine für reguläre Ausdrücke wird die harte Arbeit erledigen und herausfinden, ab wo der Eingabestring passt. Von dort aus wird er dann weiter verarbeitet.

② Nun können Sie jede beliebige Telefonnummer die führende Zeichen oder eine führende Ziffer enthält und außerdem jedwedes Trennzeichen in beliebiger Anzahl beinhaltet zergliedern.

③ Funktionstest. Es funktioniert immer noch.

④ Auch dies funktioniert noch.

Haben Sie bemerkt, wie schnell reguläre Ausdrücke außer Kontrolle geraten können? Sehen Sie sich noch einmal kurz alle vorherigen aufeinander aufbauenden Ausdrücke an. Können Sie den Unterschied zwischen einem und dem nächsten benennen?

Während Sie die endgültige Lösung noch verstehen (und es ist die endgültige Lösung; wenn Sie einen Fall entdecken, bei dem es nicht funktioniert, will ich es nicht wissen), sollten wir sie nun als ausführlichen regulären Ausdruck schreiben, bevor Sie vergessen, warum Sie die Entscheidungen getroffen haben, die Sie getroffen haben.

```
>>> phonePattern = re.compile(r'''
                # don't match beginning of string, number can start anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
    \D*         # optional separator is any number of non-digits
    (\d{3})     # trunk is 3 digits (e.g. '555')
    \D*         # optional separator
    (\d{4})     # rest of number is 4 digits (e.g. '1212')
```

```
    \D*         # optional separator
    (\d*)       # extension is optional and can be any number of digits
    $           # end of string
    ''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()  ①
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                          ②
('800', '555', '1212', '')
```

① Abgesehen davon, dass sich dieser reguläre Ausdruck über mehrere Zeilen erstreckt, ist es immer noch derselbe wie der im letzten Schritt. Es ist daher keine Überraschung, dass er auch dieselben Eingaben verarbeitet.

② Abschließender Funktionstest. Ja, es funktioniert noch. Sie haben es geschafft!

6.7 Zusammenfassung

Dies ist erst die sehr kleine Spitze des Eisbergs. Reguläre Ausdrücke können sehr viel mehr als das. Anders ausgedrückt: auch wenn Sie nun völlig überwältigt von ihnen sind, glauben Sie mir, dass Sie noch nichts davon gesehen haben.

Die folgenden Techniken sollten Sie nun kennen:

- `^` steht für den Beginn eines Strings
- `$` steht für das Ende eines Strings
- `\b` steht für eine Wortgrenze
- `\d` steht für eine Ziffer
- `\D` steht für ein nicht-numerisches Zeichen
- `x?` steht für ein optionales `x`-Zeichen (es steht also für kein oder für ein `x`)
- `x*` steht für kein oder mehrere `x`-Zeichen
- `x+` steht für ein oder mehrere `x`-Zeichen
- `x{n,m}` steht für ein `x`-Zeichen, das mindestens `n`-mal, aber nicht öfter als `m`-mal vorkommen darf
- `(a|b|c)` steht für `a` *oder* `b` *oder* `c`
- `(x)` ist allgemein eine zu merkende Gruppe. Sie erhalten den gefundenen Wert, indem Sie die `groups()`-Methode des von `re.search` zurückgegebenen Objekts aufrufen.

Auch wenn reguläre Ausdrücke sehr mächtig sind, sind sie dennoch nicht für jedes Problem die passende Lösung. Sie sollten so viel über sie lernen, dass Sie beurteilen können, wann sie angebracht sind, wann sie Ihre Probleme lösen können und wann Sie mehr Probleme verursachen als sie lösen.

Kapitel 7
Closures und Generatoren

7.1 Abtauchen

Aus unerfindlichen Gründen war ich schon immer von Sprachen fasziniert. Nicht von Programmiersprachen. Nun, doch, von Programmiersprachen, aber auch von natürlichen Sprachen. Zum Beispiel Englisch. Englisch ist eine schizophrene Sprache, die Wörter aus dem Deutschen, Französischen, Spanischen und Lateinischen (um nur ein paar zu nennen) ausleiht. Eigentlich ist „ausleihen" nicht das richtige Wort. Sagen wir „ausplündern". Oder vielleicht „assimilieren" – wie die Borg (aus Star Trek; Anm. d. Übers.). Ja, das ist gut.

Wir sind die Borg. Eure linguistischen und etymologischen Besonderheiten werden zu unseren eigenen hinzugefügt. Widerstand ist zwecklos.

In diesem Kapitel werden Sie etwas über den Plural von Substantiven lernen. Außerdem auch etwas über Funktionen die andere Funktionen zurückgeben, erweiterte reguläre Ausdrücke und Generatoren. Doch lassen Sie uns zuerst darüber reden, wie man Substantive im Plural bildet.

Sind Sie in einem englischsprachigen Land aufgewachsen oder haben Englisch in der Schule gelernt, dann sind Ihnen diese Grundregeln vielleicht vertraut:

- Endet ein Wort auf S, X, oder Z, hängen Sie ES an. *Bass* wird zu *basses*, *fax* wird zu *faxes* und *waltz* wird zu *waltzes*.
- Endet ein Wort mit einem lauthaften H, hängen Sie ES an. Endet es mit einem stummen H, hängen Sie nur ein S. Was ist ein lauthaftes H? Eines, das in Kombination mit anderen Buchstaben zu hören ist. *Coach* wird also zu *coaches* und *rash* zu *rashes*, da man das CH und das SH bei der Aussprache hören kann. *Cheetah* dagegen wird zu *cheetahs*, da das H stumm bleibt.
- Endet ein Wort auf ein Y, das wie ein I klingt, ändern Sie das Y in IES. Tritt das Y in Kombination mit einem Vokal auf, so dass es anders klingt, hängen Sie einfach ein S daran. *Vacancy* wird somit zu *vacancies*, doch *day* wird zu *days*.
- Sollte das alles nicht klappen, hängen Sie einfach ein S an und hoffen Sie das Beste.

(Ich weiß, dass es seine Vielzahl von Ausnahmen gibt. *Man* wird zu *men* und *woman* zu *women*, aber *human* wird zu *humans*. *Mouse* wird zu *mice* und *louse* zu *lice*, doch *house* wird zu *houses*. *Knife* wird zu *knives* und *wife* zu *wives*, aber *lowlife*

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_7, © Springer-Verlag Berlin Heidelberg 2010

wird zu *lowlifes*. Über Wörter, die ihr eigener Plural sind möchte ich gar nicht erst reden: *sheep*, *deer* und *haiku* zum Beispiel.)

Andere Sprachen haben natürlich völlig andere Regeln.

Lassen Sie uns eine Bibliothek für Python entwerfen, die englische Substantive in den Plural setzt. Wir beginnen mit diesen vier Regeln, doch behalten Sie im Hinterkopf, dass eine Erweiterung unausweichlich ist.

7.2 Nutzen wir reguläre Ausdrücke!

Sie haben also Wörter vor sich, was – zumindest im Englischen – bedeutet, dass Sie Zeichenstrings vor sich haben. Sie haben Regeln, die besagen, dass Sie verschiedene Buchstabenkombinationen finden und mit diesen verschiedene Dinge tun müssen. Das klingt ganz nach einem Job für reguläre Ausdrücke!

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):              ①
        return re.sub('$', 'es', noun)         ②
    elif re.search('[^aeioudgkprt]h$', noun):
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

① Dies ist ein regulärer Ausdruck, der jedoch eine Syntax verwendet, die Sie im vorherigen Kapitel nicht kennen gelernt haben. Die eckigen Klammern bedeuten „finde genau eines dieser Zeichen“. `[sxz]` bedeutet also „`s` *oder* `x` *oder* `z`“, doch nur eins davon. Das `$`-Zeichen sollten Sie kennen; es steht für das Ende des Strings. Zusammengenommen prüft dieser reguläre Ausdruck, ob ein Substantiv auf ein `s`, `x` oder `z` endet.

② Diese `re.sub()`-Funktion führt String-Ersetzungen basierend auf regulären Ausdrücken aus.

Sehen wir uns diese Ersetzungen einmal näher an.

```
>>> import re
>>> re.search('[abc]', 'Mark')    ①
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark')  ②
'Mork'
>>> re.sub('[abc]', 'o', 'rock')  ③
'rook'
>>> re.sub('[abc]', 'o', 'caps')  ④
'oops'
```

① Enthält der String `Mark` ein `a`, `b` oder `c`? Ja, er enthält ein `a`.

② Okay, dann finde nun `a`, `b` oder `c` und ersetze es mit `o`. `Mark` wird zu `Mork`.

③ Die gleiche Funktion verwandelt `rock` in `rook`.

④ Nun denken Sie vielleicht, dass dies `caps` in `oaps` umwandelt, doch das ist nicht der Fall. `re.sub()` ersetzt alle Vorkommen, nicht nur das erste. Dieser reguläre Ausdruck macht aus `caps` daher `oops`, da sowohl das `c`, als auch das `a` in ein `o` umgewandelt werden.

Doch nun zurück zur `plural()`-Funktion ….

```
def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)          ①
    elif re.search('[^aeioudgkprt]h$', noun):   ②
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):         ③
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

① Hier ersetzen Sie das Ende des Strings (`$`) durch den String `es`. Sie fügen also `es` am Ende des Strings hinzu. Dasselbe Ergebnis könnten Sie auch durch Stringverkettung erzielen, z. B. `Substantiv + 'es'`, doch ich habe beschlossen für jede der Regeln reguläre Ausdrücke zu verwenden. Die Gründe dafür werden später in diesem Kapitel klar werden.

② Sehen Sie genau hin. Dies ist eine neue Variante. Das `^`-Zeichen als erstes Zeichen innerhalb der eckigen Klammern bedeutet etwas Besonderes: Negation. `[^abc]` bedeutet „ein einzelnes Zeichen, abgesehen von `a`, `b` oder `c`“. `[^aeioudgkprt]` steht also für jedes Zeichen, außer `a`, `e`, `i`, `o`, `u`, `d`, `g`, `k`, `p`, `r` oder `t`. Diesem Buchstaben muss ein `h` folgen und dann das Ende des Strings. Damit haben Sie Wörter vor sich, die mit einem hörbaren `H` enden.

③ So ähnlich ist es auch hier: finde Wörter, die mit einem `Y` enden, bei denen der Buchstabe vor dem `Y` kein `a`, `e`, `i`, `o` oder `u` ist. Das sind die Wörter, die auf ein wie ein `I` klingendes `Y` enden.

Sehen wir uns die Negation regulärer Ausdrücke einmal näher an.

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy')  ①
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy')      ②
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita')     ③
>>>
```

① `vacancy` wird von diesem regulären Ausdruck erfasst, da es auf `cy` endet und `c` weder ein `a`, noch ein `e`, `i`, `o` oder `u` ist.

② `boy` passt nicht, da es auf `oy` endet und Sie explizit gesagt haben, dass das Zeichen vor dem `y` kein `o` sein darf. `day` wird ebenfalls nicht erkannt, da es auf `ay` endet.

③ `pita` passt nicht, da es nicht auf `y` endet.

```
>>> re.sub('y$', 'ies', 'vacancy')                ①
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy')   ②
'vacancies'
```

① Dieser reguläre Ausdruck macht aus `vacancy` `vacancies` und aus `agency` `agencies`. Das ist genau das, was Sie wollten. Beachten Sie aber, dass er auch aus `boy` `boies` machen würde. Das wird innerhalb der Funktion aber niemals geschehen, da Sie ja `re.search` benutzen, um herauszufinden, ob `re.sub` durchgeführt werden soll.

② Nebenbei möchte ich noch darauf hinweisen, dass man diese beiden regulären Ausdrücke (einer, um herauszufinden, ob die Regel zutrifft und ein weiterer, um sie dann anzuwenden) zu einem kombinieren kann. Hier sehen Sie, wie das aussehen würde. Das meiste davon sollte Ihnen bereits bekannt sein: Sie verwenden eine sich zu merkende Gruppe (siehe: *Fallbeispiel: Telefonnummern gliedern*). Diese Gruppe wird benutzt, um das Zeichen vor dem Buchstaben `y` zu speichern. Innerhalb des Ersetzungsstrings verwenden Sie eine neue Syntax, `\1`, was Folgendes bedeutet: „Hey, füge die erste Gruppe die du dir gemerkt hast hier ein!“ Im vorliegenden Fall haben Sie sich das `c` vor dem `y` gemerkt. Wenn Sie die Ersetzung dann ausführen, ersetzen Sie das `c` durch ein `c` und das `y` durch `ies`. (Ist mehr als eine Gruppe vorhanden, können Sie `\2`, `\3` usw. benutzen.)

Ersetzungen mithilfe regulärer Ausdrücke sind sehr mächtig. Die `\1`-Syntax führt dazu, dass sie sogar noch mächtiger werden. Doch das komplette Verfahren in einen regulären Ausdruck zu packen ist sehr viel schwerer zu lesen und die Übereinstimmung mit den aufgestellten Regeln zur Bildung des Plurals lässt sich nicht auf Anhieb erkennen. Die ursprünglich erstellten Regeln lauteten etwa wie folgt: „Endet das Wort auf S, X oder Z, dann hänge ES an.“ Wenn Sie sich nun die Funktion ansehen, erkennen Sie zwei Zeilen Code die besagen „endet das Wort auf S, X oder Z, dann hänge ES an“. Direkter geht es kaum.

7.3 Eine Funktionsliste

Sie werden nun eine Abstraktionsebene einfügen. Sie haben begonnen mit einer Liste von Regeln: *wenn dies, mache jenes, sonst gehe zur nächsten Regel*. Wir werden nun einen Teil des Programms komplizierter gestalten, um einen anderen Teil zu vereinfachen.

```
import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):                                  ①
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):                                  ②
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return True

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),                    ③
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )

def plural(noun):
    for matches_rule, apply_rule in rules:          ④
        if matches_rule(noun):
            return apply_rule(noun)
```

① Jetzt hat jede Übereinstimmungsregel ihre eigene Funktion, die das Ergebnis des Aufrufs der `re.search()`-Funktion zurückgibt.

② Auch jede Anwendungsregel hat ihre eigene Funktion zum Aufrufen der `re.sub()`-Funktion, welche die passende Plural-Regel anwendet.

③ Anstelle einer einzelnen Funktion (`plural()`) mit mehreren Regeln, haben Sie nun die Datenstruktur `rules`, die eine Reihe von Funktionspaaren enthält.

④ Da wir die Regeln nun in eine eigene Datenstruktur ausgelagert haben, können wir die neue `plural()`-Funktion auf wenige Codezeilen reduzieren. Mithilfe der `for`-Schleife können Sie zwei Funktionen auf einmal (eine zur Prüfung und eine zur Anwendung der Regeln) aus der `rules`-Struktur holen. Beim ersten Durchlauf

der `for`-Schleife wird `matches_rule` zu `match_sxz` und `apply_rule` zu `apply_sxz`. Beim zweiten Durchlauf (sollten Sie so weit kommen) wird `matches_rule` zu `match_h` und `apply_rule` zu `apply_h`. Die Funktion gibt in jedem Fall etwas zurück, da die letzte Übereinstimmungsregel (`match_default`) einfach `True` zurückgibt, womit auch die entsprechende Anwendungsregel (`apply_default`) immer angewandt wird.

Der Grund warum diese Vorgehensweise funktioniert, ist dass alles in Python ein Objekt ist (siehe: *Ihr erstes Python-Programm*), auch Funktionen. Die Datenstruktur `rules` enthält Funktionen – keine Funktionsnamen, sondern richtige Funktionsobjekte. Wenn Sie innerhalb der `for`-Schleife zugewiesen werden, sind `matches_rule` und `apply_rule` tatsächlich Funktionen die Sie aufrufen können. Beim ersten Durchlauf der `for`-Schleife bedeutet dies, dass die Funktion `matches_sxz(noun)`, und wenn diese einen Treffer zurückgibt, `apply_sxz(noun)` aufgerufen wird.

Wenn Sie diese zusätzliche Abstraktionsebene verwirrend finden, versuchen Sie die Funktion zu entwirren, um die Äquivalenz zu erkennen. Die komplette `for`-Schleife ist äquivalent zu Folgendem:

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

Der Vorteil ist hierbei, dass die `plural()`-Funktion nun vereinfacht ist. Sie nimmt eine Folge von Regeln entgegen, die irgendwo anders definiert sind und durchläuft diese in generischer Art und Weise.

- Hole eine Übereinstimmungsregel.
- Trifft die Regel zu? Dann rufe die Anwendungsregel auf und gebe das Ergebnis zurück.
- Kein Treffer? Gehe zu Schritt 1.

Die Regeln könnten irgendwo, irgendwie definiert sein. Die `plural()`-Funktion schert sich nicht darum.

War es sinnvoll diese Abstraktionsebene einzufügen? Nun, bisher noch nicht. Nehmen wir einmal an, wir wollten eine neue Regel zu der Funktion hinzufügen. Beim ersten Beispiel bedeutete dies, eine neue `if`-Anweisung zur `plural()`-Funktion hinzuzufügen. In diesem zweiten Beispiel müssten wir zwei Funktionen, `match_foo()` und `apply_foo()`, hinzufügen und die `rules`-Struktur aktualisieren, um festzulegen, wann die neuen Funktionen aufgerufen werden sollen.

Doch dies ist in Wirklichkeit auch nur ein Schritt hin zum nächsten Abschnitt. Weiter geht's ….

7.4 Eine Musterliste

Es ist gar nicht nötig für jedes Übereinstimmungs- und Anwendungsregel eine eigene Funktion anzulegen. Sie rufen sie niemals direkt auf; Sie fügen sie zur `rules`-Folge hinzu und rufen Sie darüber auf. Außerdem basiert jede Funktion auf einem von zwei Mustern. Alle Übereinstimmungsfunktionen rufen `re.search()` auf und alle Anwendungsfunktionen rufen `re.sub()` auf. Wir sollten die Muster ausklammern, damit das Definieren neuer Regeln einfacher wird.

```
import re

def build_match_and_apply_functions(pattern, search, replace):
    def matches_rule(word):                                    ①
        return re.search(pattern, word)
    def apply_rule(word):                                      ②
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)                          ③
```

① `build_match_and_apply_functions()` ist eine Funktion, die andere Funktionen dynamisch aufbaut. Sie nimmt `pattern`, `search` und `replace` entgegen und definiert eine `matches_rule()`-Funktion, welche `re.search()` mit dem Muster (`pattern`), das an die `build_match_and_apply_functions()`-Funktion übergeben wurde, und dem Wort (`word`), das der zu erstellenden `matches_rule()`-Funktion übergeben wurde, aufruft. Oh je!

② Der Aufbau der Anwendungsfunktion erfolgt auf die gleiche Weise. Die Anwendungsfunktion nimmt einen Parameter entgegen und ruft `re.sub()` mit dem `search`- und dem `replace`-Parameter, die der `build_match_and_apply_functions()`-Funktion übergeben wurden sowie mit dem Wort (`word`), das der zu erstellenden `apply_rule()`-Funktion übergeben wurde, auf. Diese Vorgehensweise des Verwendens von außenliegenden Parametern innerhalb einer dynamischen Funktion wird *Closures* genannt. Im Grunde definieren Sie innerhalb der zu erstellenden Anwendungsfunktion Konstanten: Sie nimmt einen Parameter (`word`) entgegen, arbeitet dann jedoch mit diesem und zwei weiteren Werten (`search` und `replace`), welche bei der Definition der Anwendungsfunktion festgelegt wurden.

③ Schließlich gibt die `build_match_and_apply_functions()`-Funktion ein Tupel von zwei Werten zurück: Die zwei Funktionen die Sie gerade erstellt haben. Die Konstanten die Sie innerhalb dieser Funktionen definiert haben (`pattern` in der `match_rule()`- sowie `search` und `replace` in der `apply_rule()`-Funktion) bleiben bei diesen Funktionen, und das selbst dann, wenn Sie aus `build_match_and_apply_functions()` zurückkehren. Das ist unglaublich toll.

Sollte Ihnen dies unglaublich verwirrend vorkommen (und das sollte es, da es wirklich seltsames Zeug ist), so könnte es klarer werden, wenn Sie sehen, wie wir es benutzen.

```
patterns = \                                                    ①
  (
    ('[sxz]$',           '$',  'es'),
    ('[^aeioudgkprt]h$', '$',  'es'),
    ('(qu|[^aeiou])y$',  'y$', 'ies'),
    ('$',                '$',  's')                             ②
  )
rules = [build_match_and_apply_functions(pattern, search, replace)  ③
         for (pattern, search, replace) in patterns]
```

① Unsere Plural-Regeln sind nun als ein Tupel von Tupeln aus Strings (nicht Funktionen) definiert. Der erste String jeder Gruppe ist das Suchmuster, das Sie in `re.search()` verwenden werden, um herauszufinden, ob diese Regel zutrifft. Der zweite und dritte String jeder Gruppe sind die Ausdrücke zum Suchen und Ersetzen, die Sie in `re.sub()` verwenden, um die Regeln anzuwenden und damit ein Substantiv in den Plural zu setzen.

② Bei dieser Ausweichregel gibt es einen kleinen Unterschied. Im vorherigen Beispiel gab die `match_default()`-Funktion einfach `True` zurück, d. h., wenn keine der spezifischeren Regeln zutraf, wurde einfach ein `s` an das Ende des Wortes angehängt. Dieses Beispiel hat eine ähnliche Funktion. Der letzte reguläre Ausdruck sieht nach, ob das Wort ein Ende hat (`$` sucht das Ende eines Strings), und da natürlich jeder String ein Ende hat, selbst ein leerer, trifft dieser Ausdruck immer zu. Somit erfüllt die Funktion den gleichen Zweck wie die `match_default()`-Funktion, die immer `True` zurückgegeben hat: Wenn keine der spezifischen Regeln zutrifft hängt sie ein `s` an das Ende des Wortes an.

③ Diese Zeile ist Zauberei. Sie übernimmt die String-Folge in `patterns` und wandelt sie in eine Folge von Funktionen um. Wie macht sie das? Durch Zuordnung der Strings zur `build_match_and_apply_functions()`-Funktion. Das heißt sie nimmt jede String-Dreiergruppe und ruft die `build_match_and_apply_functions()`-Funktion mit diesen Strings als Argumenten auf. Die `build_match_and_apply_functions()`-Funktion gibt ein Tupel von zwei Funktionen zurück. Das bedeutet, dass `rules` zum Schluss funktionell mit dem vorherigen Beispiel identisch ist: Eine Tupel-Liste, bei der alle inneren Tupel ein Funktionen-Paar bilden. Die erste Funktion ist die Übereinstimmungsfunktion, die `re.search()` aufruft, die zweite Funktion ist die Anwendungsfunktion, die `re.sub()` aufruft.

Abgerundet wird diese Version des Skripts durch den Haupteinstiegspunkt, die `plural()`-Funktion.

```
def plural(noun):
    for matches_rule, apply_rule in rules:  ①
        if matches_rule(noun):
            return apply_rule(noun)
```

① Da die `rules`-Liste dieselbe ist wie im letzten Beispiel (ist sie wirklich), sollte es Sie nicht überraschen, dass die `plural()`-Funktion sich überhaupt nicht verän-

dert hat. Sie ist vollständig generisch; sie holt sich eine Liste von Regel-Funktionen und ruft sie der Reihe nach auf. Dabei ist es egal, wie die Regeln definiert sind. Im vorherigen Beispiel waren sie als eigenständige Funktionen definiert. Nun werden sie dynamisch durch Zuordnung der Ausgabe der `build_match_and_apply_functions()`-Funktion zu einer Liste von *Raw-Strings* gebildet. Das spielt keine Rolle; die `plural()`-Funktion funktioniert immer noch genauso wie vorher.

7.5 Eine Musterdatei

Sie haben nun den ganzen doppelten Code beseitigt und genug Abstraktion hinzugefügt, die es ermöglicht, dass die Regeln zur Pluralbildung in einer Liste von Strings definiert sind. Der nächste logische Schritt ist es, diese Strings zu nehmen und in eine eigene Datei auszulagern, in der sie getrennt vom sie nutzenden Code gepflegt werden können.

Erstellen wir zuerst einmal eine Textdatei, welche die Regeln enthält die Sie möchten. Keine raffinierten Datenstrukturen, sondern einfach nur durch Whitespace getrennte Strings in drei Spalten. Nennen wie die Datei `plural4-rules.txt`.

```
[sxz]$                $    es
[^aeioudgkprt]h$      $    es
[^aeiou]y$           y$    ies
$                     $    s
```

Lassen Sie uns nun schauen, wie Sie diese Regel-Datei verwenden können.

```
import re

def build_match_and_apply_functions(pattern, search, replace):  ①
    def matches_rule(word):
        return re.search(pattern, word)
    def apply_rule(word):
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)

rules = []
with open('plural4-rules.txt', encoding='utf-8') as pattern_file:  ②
    for line in pattern_file:                                      ③
        pattern, search, replace = line.split(None, 3)             ④
        rules.append(build_match_and_apply_functions(              ⑤
                pattern, search, replace))
```

① Die `build_match_and_apply_functions()`-Funktion hat sich nicht verändert. Sie verwenden immer noch *Closures*, um dynamisch zwei Funktionen zu erstellen, die in der äußeren Funktion definierte Variablen nutzen.

② Die globale `open()`-Funktion öffnet eine Datei und gibt ein Datei-Objekt zurück. In diesem Fall enthält die Datei die wir öffnen die Suchmuster-Strings zum Pluralisieren von Substantiven. Die `with`-Anweisung erstellt einen sogenannten Kontext: Endet der `with`-Block, schließt Python die Datei automatisch, und das auch dann, wenn innerhalb des `with`-Blocks eine Ausnahme ausgelöst wurde. Im Kapitel „Dateien“ werden Sie mehr über `with`-Blöcke und Datei-Objekte erfahren.

③ Der Ausdruck `for line in <fileobject>` liest Zeile für Zeile die Daten aus der geöffneten Datei und weist den Text dann der Variable `line` zu. Im Kapitel *„Dateien“* werden Sie mehr über das Auslesen von Dateien erfahren.

④ Jede Zeile der Datei enthält eigentlich drei Werte; diese sind aber durch Whitespace getrennt (egal ob durch Tabulatoren oder Leerzeichen). Zum Aufteilen der Werte verwenden wir die String-Methode `split()`. Das erste Argument der `split()`-Methode ist `None`, d. h. „trenne bei jedwedem Whitespace (egal ob Tabulatoren oder Leerzeichen)“. Das zweite Argument ist `3`, d. h. „trenne dreimal bei Whitespace und ignoriere den Rest der Zeile“. Eine Zeile wie `[sxz]$ $ es` wird so aufgeteilt in die Liste `['[sxz]$', '$', 'es']`, d. h. `pattern` wird zu `'[sxz]$'`, `search` zu `'$'` und `replace` zu `'es'`. Das ist ganz schön viel für eine kleine Zeile Code.

⑤ Schließlich übergeben Sie `pattern`, `search` und `replace` an die `build_match_and_apply_functions()`-Funktion, die ein Tupel von Funktionen zurückgibt. Dieses Tupel hängen Sie an die `rules`-Liste an, wodurch `rules` schlussendlich die von der `plural()`-Funktion erwarteten Funktionen gespeichert hat.

Nun haben Sie die Regeln zur Pluralbildung in eine separate Datei ausgelagert und können diese unabhängig vom sie nutzenden Code warten und pflegen. Code ist Code, Daten sind Daten, und das Leben ist schön.

7.6 Generatoren

Wäre es nicht großartig, eine generische `plural()`-Funktion zu haben, die die Regeldatei analysiert? *Hole die Regeln, suche nach einem Treffer, wende die passende Veränderung an und gehe zur nächsten Regel*. Das ist alles, was die `plural()`-Funktion tun muss und was sie tun sollte.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)
            yield build_match_and_apply_functions(pattern, search, replace)

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

Wie um Himmels willen funktioniert das? Sehen wir uns zunächst ein interaktives Beispiel an.

```
>>> def make_counter(x):
...     print('entering make_counter')
...     while True:
...         yield x                    ①
...         print('incrementing x')
...         x = x + 1
...
>>> counter = make_counter(2)          ②
>>> counter                            ③
<generator object at 0x001C9C10>
>>> next(counter)                      ④
entering make_counter
2
>>> next(counter)                      ⑤
incrementing x
3
>>> next(counter)                      ⑥
incrementing x
4
```

① Am Vorhandensein des Keywords `yield` in `make_counter` erkennen Sie, dass dies keine gewöhnliche Funktion ist. Es handelt sich hierbei um eine spezielle Art von Funktion, welche einen Wert nach dem anderen generiert. Der Aufruf der Funktion gibt einen Generator zurück, der verwendet werden kann, um aufeinanderfolgende Werte für `x` zu generieren.

② Zum Erstellen einer Instanz des `make_counter`-Generators rufen Sie ihn einfach wie jede andere Funktion auf. Beachten Sie, dass der Code der Funktion nicht wirklich ausgeführt wird. Das erkennen Sie daran, dass die `make_counter()`-Funktion `print()` aufruft, doch bisher noch keine Ausgabe erfolgt ist.

③ Die `make_counter()`-Funktion gibt ein Generator-Objekt zurück.

④ Die Funktion `next()` übernimmt ein Generator-Objekt und gibt dessen nächsten Wert zurück. Beim ersten Aufruf von `next()` mit dem `counter`-Generator wird der Code in `make_counter()` bis zur ersten `yield`-Anweisung ausgeführt und dann der gelieferte (engl. *yielded*) Wert zurückgegeben. In diesem Fall ist dies `2`, da der Generator durch den Aufruf von `make_counter(2)` erstellt wurde.

⑤ Beim erneuten Aufruf von `next()` mit demselben Generator-Objekt setzt die Funktion genau da an, wo sie vorher aufgehört hat und wird wieder bis zur `yield`-Anweisung ausgeführt. Alle lokalen Variablen werden beim Erreichen von `yield` gespeichert und bei `next()` wiederhergestellt. Die nächste Codezeile ruft `print()` auf und gibt `incrementing x` aus. Danach folgt die Anweisung `x = x + 1`. Dann wird erneut die `while`-Schleife durchlaufen. Bei `yield x` wird alles gespeichert und der aktuelle Wert von `x` (jetzt `3`) zurückgegeben.

⑥ Rufen Sie `next(counter)` abermals auf, passiert all dies noch einmal; diesmal ist `x` jedoch `4`.

Da `make_counter` als Endlosschleife aufgesetzt wurde, könnten Sie dies theoretisch für immer fortsetzen und jedes mal würde `x` erhöht und dessen Wert ausgegeben. Wir sollten uns aber nun nützlichere Verwendungsmöglichkeiten für Generatoren ansehen.

7.6.1 Ein Fibonacci-Generator

```
def fib(max):
    a, b = 0, 1           ①
    while a < max:
        yield a           ②
        a, b = b, a + b   ③
```

① Die Fibonacci-Folge ist eine Zahlenfolge, bei der jede Zahl die Summe der zwei vorhergehenden Zahlen ist. Sie beginnt mit 0 und 1, geht erst langsam, dann sehr schnell nach oben. Zu Beginn der Folge benötigen Sie zwei Variablen: `a` beginnt bei `0` und `b` beginnt bei `1`.

② `a` ist die aktuelle Zahl der Folge, also verwenden wir `yield` mit `a`.

③ `b` ist die nächste Zahl der Folge; weisen Sie sie also `a` zu, doch berechnen Sie auch den nächsten Wert (`a + b`) und weisen Sie diesen `b` zur späteren Verwendung zu. Beachten Sie, dass dies parallel geschieht: ist `a` `3` und `b` `5`, dann setzt `a, b = b, a +b` `a` auf `5` (der vorige Wert von `b`) und `b` auf `8` (die Summe der vorherigen Werte von `a` und `b`).

Nun haben Sie also eine Funktion, die aufeinanderfolgende Fibonacci-Zahlen ausgibt. Klar, Sie könnten das auch durch Rekursion erreichen, doch unsere Vorgehensweise ist lesbarer. Außerdem funktioniert sie auch mit `for`-Schleifen.

```
>>> from fibonacci import fib
>>> for n in fib(1000):      ①
...     print(n, end=' ')    ②
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> list(fib(1000))          ③
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

① Einen Generator wie `fib()` können Sie ohne Umschweife in einer `for`-Schleife verwenden. Die `for`-Schleife ruft zum Abrufen der Werte vom `fib()`-Generator automatisch die `next()`-Funktion auf und weist sie der Indexvariable der `for`-Schleife (`n`) zu.

② Bei jedem Durchlauf der `for`-Schleife erhält `n` einen neuen Wert von der `yield`-Anweisung in `fib()`. Alles was Sie tun müssen, ist diesen Wert auszuge-

ben. Stehen `fib()` keine Zahlen mehr zur Verfügung (`a` wird größer als `max`, was hier bei `1000` der Fall ist), wird die `for`-Schleife problemlos verlassen.

③ Das ist ein nützlicher Ausdruck: Übergeben Sie den Generator an die `list()`-Funktion und sie wird den gesamten Generator durchlaufen (genau wie die `for`-Schleife im vorherigen Beispiel) und eine Liste aller Werte zurückgeben.

7.6.2 Ein Generator für Plural-Regeln

Sehen wir uns erneut `plural5.py` und die Funktionsweise dieser Version der `plural()`-Funktion an.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)                    ①
            yield build_match_and_apply_functions(pattern, search, replace)   ②


def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):                    ③
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {0}'.format(noun))
```

① Das ist keine Zauberei. Erinnern Sie sich daran, dass die Zeilen der Regel-Datei drei durch Whitespace getrennte Werte enthalten? Hier benutzen Sie nun `line.split(None, 3)`, um an die drei Spalten zu gelangen und sie drei lokalen Variablen zuzuweisen.

② Nun verwenden Sie `yield`. Worauf aber wenden Sie `yield` an? Auf zwei Funktionen, die dynamisch von unserem alten Freund, `build_match_and_apply_functions()`, gebildet wurden. `rules()` ist somit ein Generator, der Übereinstimmungs- und Anwendungsfunktionen nach Bedarf liefert.

③ Da `rules()` ein Generator ist, können Sie ihn direkt in der `for`-Schleife verwenden. Beim ersten Durchlauf der `for`-Schleife wird die `rules()`-Funktion aufgerufen, die die Musterdatei öffnet, die erste Zeile daraus liest, dynamisch eine Übereinstimmungs- und eine Anwendungsfunktion aus den Mustern in dieser bildet und schließlich `yield` auf diese Funktionen anwendet. Beim zweiten Durchlauf wird genau dort weitergemacht, wo die `rules()`-Funktion vorher verlassen wurde (also in der Mitte der `for line in pattern_file`-Schleife). Zuerst wird dann die nächste Zeile der Datei gelesen (die immer noch geöffnet ist), dann werden dynamisch eine weitere Übereinstimmungs- und eine weitere Anwendungsfunktion basierend auf den Mustern in dieser Zeile gebildet und `yield` darauf angewendet.

Was haben Sie nun gegenüber Stufe 4 gewonnen? Startzeit. Bei Stufe 4, als Sie das `plural4`-Modul importierten, wurde die komplette Musterdatei gelesen und

eine Liste aller möglichen Regeln gebildet, bevor Sie überhaupt daran denken konnten, die `plural()`-Funktion aufzurufen. Mit Generatoren können Sie alles etwas langsamer angehen: Sie lesen die erste Regel, erstellen Funktionen und probieren diese aus; wenn das funktioniert müssen Sie den Rest der Datei gar nicht erst lesen und auch keine anderen Funktionen mehr erstellen.

Was haben Sie verloren? Performance! Bei jedem Aufruf der `plural()`-Funktion beginnt der `rules()`-Generator von vorne. Die Musterdatei muss erneut geöffnet und von Beginn an, Zeile für Zeile, gelesen werden.

Was, wenn Sie das beste beider Welten vereinen könnten: minimale Startzeit (beim Import keinen Code ausführen) und maximale Performance (Funktionen nicht wieder und wieder bilden). Oh, und natürlich sollen die Regeln weiterhin in einer eigenen Datei ausgelagert sein (Code ist Code und Daten sind Daten).

Um dies zu erreichen, müssen Sie Ihren eigenen Iterator erstellen. Doch bevor Sie dies tun, müssen Sie erst einmal Python-Klassen kennenlernen.

Kapitel 8
Klassen und Iteratoren

8.1 Los geht's

Generatoren sind eigentlich nur Spezialfälle von *Iteratoren*. Ein möglicher Weg einen Iterator zu erstellen ohne einen Iterator zu erstellen ist eine Funktion die Werte *yieldet*. Ich zeige Ihnen, was ich damit meine.

Erinnern Sie sich an den Fibonacci-Generator? Hier ist dieser als Iterator:

```
class Fib:
    '''iterator that yields numbers in the Fibonacci sequence'''

    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

Sehen wir uns eine Zeile nach der anderen an.

```
class Fib:
```

`class`? Was bedeutet `class`?

M. Pilgrim, *Python 3 - Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_8, © Springer-Verlag Berlin Heidelberg 2010

8.2 Klassen definieren

Python ist vollständig objektorientiert: Sie können Ihre eigenen Klassen definieren, von diesen oder den integrierten Klassen erben und Ihre definierten Klassen instanziieren.

Die Definition einer Klasse ist in Python sehr einfach. Wie bei Funktionen gibt es auch bei Klassen keine Interface-Definition. Definieren Sie einfach die Klasse und fangen Sie an zu programmieren. Eine Python-Klasse beginnt mit dem Keyword `class`, gefolgt vom Namen der Klasse. Das ist auch schon alles, da eine Klasse nicht unbedingt von einer anderen Klasse erben muss.

```
class PapayaWhip:   ①
    pass            ②
```

① Der Name dieser Klasse ist `PapayaWhip` und sie erbt nicht von einer anderen Klasse. Klassennamen werden gewöhnlich mit Großbuchstaben begonnen, `JedesWortWieHier`, das ist jedoch nur eine Konvention und keine Bedingung.

② Sie haben es vermutlich bereits erahnt, aber alles innerhalb einer Klasse wird eingerückt, genau wie der Code in einer Funktion, einer `if`-Anweisung, einer `for`-Schleife oder jedem anderen Codeblock. Die erste nicht eingerückte Zeile gehört nicht mehr zur Klasse.

Die `PapayaWhip`-Klasse definiert weder Methoden, noch Attribute, doch es muss etwas innerhalb der Definition stehen, daher die `pass`-Anweisung. `pass` ist ein reserviertes Wort und bedeutet *weiter geht's, hier gibt's nichts zu sehen*. Es ist eine Anweisung, die nichts macht und damit ein guter Platzhalter beim Rohentwurf einer Funktion oder Klasse.

☞ Die `pass`-Anweisung in Python entspricht den leeren geschweiften Klammern in Java oder C.

Viele Klassen erben von anderen Klassen, diese jedoch nicht. Viele Klassen definieren Methoden, diese tut das jedoch nicht. Eine Python-Klasse muss einen Namen haben, sonst nichts. C++-Programmierer finden es vielleicht seltsam, dass Python-Klassen keinen expliziten Konstruktor oder Destruktor haben. Auch wenn es nicht nötig ist, können Python-Klassen etwas enthalten, was einem Konstruktor ähnelt: die `__init__()`-Methode.

8.2.1 Die __init__()-Methode

Dieses Beispiel zeigt die Initialisierung der `Fib`-Klasse mithilfe der `__init__()`-Methode.

```
class Fib:
    '''iterator that yields numbers in the Fibonacci sequence'''  ①

    def __init__(self, max):                                      ②
```

① Klassen können (und sollten), genau wie Module und Funktionen, `docstrings` haben.

② Die `__init__()`-Methode wird aufgerufen, sobald eine Instanz der Klasse erstellt wird. Man könnte zu dem – falschen – Schluss gelangen, dies als Konstruktor der Klasse zu bezeichnen. Es sieht aus wie ein C++-Konstruktor (die `__init__()`-Methode ist die erste innerhalb der Klasse definierte Methode), verhält sich wie einer (es ist der erste Codeschnipsel, der beim Instanziieren einer Klasse ausgeführt wird) und hört sich sogar an wie einer. Falsch ist dies aber, da das Objekt beim Aufruf der `__init__()`-Methode bereits besteht und Sie somit auch schon eine gültige Referenz auf die neu erstellte Instanz der Klasse besitzen.

Das erste Argument jeder Klassenmethode, also auch der `__init__()`-Methode, ist immer eine Referenz auf die aktuelle Instanz der Klasse. Dieses Argument wird per Konvention `self` genannt. Dieses Argument erfüllt in Python die Rolle des `this` in C++ oder Java. `self` ist allerdings *kein* reserviertes Wort, sondern lediglich eine Konvention. Nichtsdestotrotz sollten Sie als Bezeichnung in jedem Falle `self` verwenden; dies ist eine sehr strenge Konvention.

Innerhalb der `__init__()`-Methode verweist `self` auf das neu erstellte Objekt; innerhalb anderer Klassenmethoden verweist es auf die Instanz, deren Methode aufgerufen wurde. Sie müssen `self` zwar bei der Definition der Methode explizit angeben, doch müssen Sie dies nicht tun, wenn Sie die Methode dann aufrufen; Python fügt es automatisch hinzu.

8.3 Klassen instanziieren

Das Instanziieren von Klassen ist in Python sehr einfach. Rufen Sie die Klasse wie eine Funktion auf und übergeben Sie ihr die Argumente, die die `__init__()`-Methode benötigt. Der Rückgabewert ist das neu erstellte Objekt.

```
>>> import fibonacci2
>>> fib = fibonacci2.Fib(100)  ①
>>> fib                        ②
<fibonacci2.Fib object at 0x00DB8810>
>>> fib.__class__              ③
<class 'fibonacci2.Fib'>
>>> fib.__doc__                ④
'iterator that yields numbers in the Fibonacci sequence'
```

① Hier erstellen Sie eine Instanz der `Fib`-Klasse (definiert im Modul `fibonacci2`) und weisen diese neu erstellte Instanz der Variable `fib` zu. Sie übergeben einen Parameter, `100`, welcher zum `max`-Argument in der `__init__()`-Methode von `Fib` wird.

② `fib` ist nun eine Instanz der `Fib`-Klasse.

③ Jede Klasseninstanz besitzt ein integriertes Attribut, `__class__`, welches die Klasse des Objekts enthält. Die Java-Programmierer unter Ihnen kennen vielleicht die Klasse `Class`, die Methoden wie `getName()` und `getSuperclass()` bereitstellt, um Metadaten über ein Objekt zu erhalten. In Python sind diese Metadaten über Attribute zugänglich, doch das Prinzip dahinter ist dasselbe.

④ Der `docstring` einer Instanz ist Ihnen auf die gleiche Art zugänglich, wie dies bei Funktionen und Modulen der Fall ist. Alle Instanzen einer Klasse teilen sich denselben `docstring`.

☞ In Python rufen Sie zum Erstellen einer neuen Klasseninstanz die Klasse einfach wie eine Funktion auf. Ein `new`-Operator wie in C++ oder Java existiert nicht.

8.4 Instanzvariablen

Auf zur nächsten Zeile:

```
class Fib:
    def __init__(self, max):
        self.max = max      ①
```

① Was ist `self.max`? Es ist eine Instanzvariable. Sie ist von `max`, dem Argument der `__init__()`-Methode, völlig unabhängig. `self.max` ist innerhalb der Instanz global; alle Methoden können also auf sie zugreifen.

```
class Fib:
    def __init__(self, max):
        self.max = max              ①
    .
    .
    .
    def __next__(self):
        fib = self.a
        if fib > self.max:          ②
```

① `self.max` wird innerhalb der `__init__()`-Methode definiert …

② … und innerhalb der `__next__()`-Methode referenziert.

Instanzvariablen gelten für *eine* Klasseninstanz. Erstellen Sie beispielsweise zwei Instanzen von `Fib` mit verschiedenen Maximalwerten, so behalten sie beide ihren eigenen Wert.

```
>>> import fibonacci2
>>> fib1 = fibonacci2.Fib(100)
>>> fib2 = fibonacci2.Fib(200)
>>> fib1.max
100
>>> fib2.max
200
```

8.5 Ein Fibonacci-Iterator

Jetzt sind Sie bereit zu lernen, wie man einen Iterator erstellt. Ein Iterator ist eine Klasse, die eine `__iter__()`-Methode definiert.

```
class Fib:                                          ①
    def __init__(self, max):                        ②
        self.max = max

    def __iter__(self):                             ③
        self.a = 0
        self.b = 1
        return self

    def __next__(self):                             ④
        fib = self.a
        if fib > self.max:
            raise StopIteration                     ⑤
        self.a, self.b = self.b, self.a + self.b
        return fib                                  ⑥
```

① Um einen Iterator zu erstellen, muss `fib` eine Klasse, keine Funktion, sein.

② Durch Aufrufen von `Fib(max)` wird eine Instanz dieser Klasse erstellt und ihre `__init__()`-Methode mit dem Argument `max` aufgerufen. Die `__init__()`-Methode speichert den Maximalwert als Instanzvariable, so dass andere Methoden sie später referenzieren können.

③ Die Methode `__iter__()` wird aufgerufen, sobald jemand `iter(fib)` aufruft. (Wie Sie gleich sehen werden, ruft eine `for`-Schleife sie automatisch auf, doch Sie können sie auch manuell aufrufen.) Nach der Initialisierung (in diesem Fall das Zurücksetzen von `self.a` und `self.b`, unseren zwei Zählern) kann die `__iter__()`-Methode jedes beliebige Objekt zurückgeben, das eine `__next__()`

-Methode implementiert. In diesem Fall (und in den meisten Fällen) gibt `__iter__()` einfach `self` zurück, da diese Klasse ihre eigene `__next__()`-Methode implementiert.

④ Die `__next__()`-Methode wird immer dann aufgerufen, wenn jemand `next()` auf einen Iterator einer Instanz einer Klasse aufruft. Dies werden Sie gleich besser verstehen.

⑤ Löst die `__next__()`-Methode eine `StopIteration`-Ausnahme aus, signalisiert dies dem Aufrufer, dass die Iteration beendet ist. Dies ist jedoch kein Fehler, sondern ein gewöhnlicher Zustand, der lediglich bedeutet, dass der Iterator keine Werte mehr generieren kann. Ist der Aufrufer eine `for`-Schleife, wird diese die `StopIteration`-Ausnahme bemerken und die Schleife problemlos beenden. (Mit anderen Worten: *die Ausnahme verschlucken*.) Dieser Hauch von Zauberei ist das wichtige am Verwenden von Iteratoren in `for`-Schleifen.

⑥ Zur Ausgabe des nächsten Wertes gibt die `__next__()`-Methode eines Iterators einfach den Wert zurück (`return`). Verwenden Sie hier nicht `yield`; dies wird nur bei Generatoren verwendet. Hier erstellen Sie Ihren eigenen Iterator; verwenden Sie also `return`.

Sind Sie nun völlig verwirrt? Ausgezeichnet! Sehen wir uns an, wie man diesen Iterator aufruft:

```
>>> from fibonacci2 import Fib
>>> for n in Fib(1000):
...     print(n, end=' ')
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Es ist genau dasselbe! Byte für Byte identisch zum Aufruf des Fibonacci-Generators. Doch wieso?

`for`-Schleifen enthalten einen Hauch von Magie. Sehen Sie sich an, was passiert:

- Die `for`-Schleife ruft `Fib(1000)` wie gezeigt auf. Dies gibt eine Instanz der `Fib`-Klasse zurück. Nennen wir sie `fib_inst`.
- Nun ruft die `for`-Schleife insgeheim `iter(fib_inst)` auf, was ein Iteratorobjekt zurückgibt. Nennen wir dieses `fib_iter`. In diesem Fall gilt `fib_iter==fib_inst`, da die `__iter__()`-Methode `self` zurückgibt, die `for`-Schleife dies jedoch nicht weiß (oder es nicht beachtet).
- Um den Iterator zu durchlaufen ruft die `for`-Schleife `next(fib_iter)` auf, was wiederum die `__next__()`-Methode des `fib_iter`-Objekts aufruft, welche die nächste Zahl der Fibonacci-Folge berechnet und einen Wert zurückgibt. Die `for`-Schleife nimmt diesen Wert entgegen, weist ihn `n` zu und führt den Innenteil der `for`-Schleife für diesen Wert `n` aus.
- Woher weiß die `for`-Schleife, wann sie stoppen muss? Gut, dass Sie fragen! Sobald `next(fib_iter)` eine `StopIteration`-Ausnahme auslöst, verschluckt die `for`-Schleife diese Ausnahme und beendet sich selbst problemlos. (Jede andere Ausnahme wird weitergegeben und wie gewöhnlich ausgelöst.)

Und wo haben Sie solch eine `StopIteration`-Ausnahme gesehen? Natürlich in der `__next__()`-Methode!

8.6 Ein Iterator für Plural-Regeln

Kommen wir nun zum großen Finale. Lassen Sie uns den Generator für Plural-Regeln umschreiben zu einem Iterator.

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def __init__(self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')
        self.cache = []

    def __iter__(self):
        self.cache_index = 0
        return self

    def __next__(self):
        self.cache_index += 1
        if len(self.cache) >= self.cache_index:
            return self.cache[self.cache_index - 1]

        if self.pattern_file.closed:
            raise StopIteration

        line = self.pattern_file.readline()
        if not line:
            self.pattern_file.close()
            raise StopIteration

        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(
            pattern, search, replace)
        self.cache.append(funcs)
        return funcs

rules = LazyRules()
```

Dies ist eine Klasse, die `__iter__()` und `__next__()` implementiert und daher als Iterator verwendet werden kann. Sie instanziieren die Klasse und weisen diese Instanz `rules` zu. Dies geschieht nur einmal beim Import.

Sehen wir uns die Klasse Schritt für Schritt an.

```
class LazyRules:
    rules_filename = 'plural6-rules.txt'

    def   init  (self):
        self.pattern_file = open(self.rules_filename, encoding='utf-8')    ①
        self.cache = []                                                    ②
```

① Beim Instanziieren der Klasse `LazyRules` öffnen wir die Musterdatei, lesen jedoch noch nicht daraus. (Das kommt später.)

② Die `__iter__()`-Methode wird aufgerufen, nachdem Sie die Klasse instanziiert, `rules` zugewiesen und `iter(rules)` zum Erstellen des Iterators aufgerufen haben. Sie würde abermals aufgerufen werden, wenn Sie einen neuen Iterator vom selben `rules`-Objekt erstellen würden.

Bevor wir fortfahren, sollten wir uns `rules_filename` näher ansehen. Diese Variable wird nicht innerhalb der `__iter__()`-Methode definiert. Tatsächlich wird sie in gar keiner Methode definiert. Sie wird auf Klassenebene definiert. Sie ist somit eine Klassenvariable, und obwohl Sie auf sie zugreifen können wie auf eine Instanzvariable (`self.rules_filename`), wird sie von allen Instanzen der `LazyRules`-Klasse verwendet.

```
>>> import plural6
>>> r1 = plural6.LazyRules()
>>> r2 = plural6.LazyRules()
>>> r1.rules_filename                                    ①
'plural6-rules.txt'
>>> r2.rules_filename
'plural6-rules.txt'
>>> r2.rules_filename = 'r2-override.txt'                ②
>>> r2.rules_filename
'r2-override.txt'
>>> r1.rules_filename
'plural6-rules.txt'
>>> r2.__class__.rules_filename                          ③
'plural6-rules.txt'
>>> r2.__class__.rules_filename = 'papayawhip.txt'       ④
>>> r1.rules_filename
'papayawhip.txt'
>>> r2.rules_filename                                    ⑤
'r2-overridetxt'
```

① Jede Instanz der Klasse erbt das Attribut `rules_filename` mit dem von der Klasse definierten Wert.

② Eine Änderung des Attributwerts in einer Instanz hat weder Einfluss auf die anderen Instanzen …

③ … noch verändert dies das Klassenattribut. Auf das Klassenattribut (im Gegensatz zu einem Attribut einer einzelnen Instanz) können Sie durch Verwendung des Attributs `__class__` zum Zugriff auf die Klasse selbst zugreifen.

④ Ändern Sie das Klassenattribut, so werden alle Instanzen, die diesen Wert erben (wie hier `r1`), davon beeinflusst.

⑤ Instanzen die dieses Attribut überschrieben haben (wie hier `r2`) werden nicht beeinflusst.

Doch nun zurück zu unserem Hauptschauplatz.

```
    def __iter__(self):                    ①
        self.cache_index = 0
        return self                        ②
```

① Die `__iter__()`-Methode wird jedes mal aufgerufen, wenn jemand – sagen wir eine `for`-Schleife – `iter(rules)` aufruft.

② Schließlich gibt die `__iter__()`-Methode `self` zurück, was bedeutet, dass diese Klasse während der Iteration selbst für die Rückgabe ihrer Werte sorgt.

```
    def __next__(self):                                    ①
        .
        .
        .
        pattern, search, replace = line.split(None, 3)
        funcs = build_match_and_apply_functions(           ②
            pattern, search, replace)
        self.cache.append(funcs)                           ③
        return funcs
```

① Die `__next__()`-Methode wird immer dann aufgerufen, wenn jemand – sagen wir eine `for`-Schleife – `next(rules)` aufruft. Diese Methode ist nur dann sinnvoll, wenn wir am Ende beginnen und uns nach vorne durcharbeiten. Genau das tun wir jetzt.

② Der letzte Teil dieser Funktionen sollte Ihnen bekannt vorkommen. Die Funktion `build_match_and_apply_functions()` hat sich nicht verändert; sie ist genau so, wie sie immer war.

③ Der einzige Unterschied besteht darin, dass wir die Übereinstimmungs- und Anwendungsfunktionen (welche im Tupel `funcs` gespeichert werden) vor der Rückgabe in `self.cache` speichern müssen.

Gehen wir zurück …

```
def __next__(self):
    .
    .
    .
    line = self.pattern_file.readline()  ①
    if not line:                         ②
        self.pattern_file.close()
        raise StopIteration              ③
    .
    .
    .
```

① Dies ist ein kleiner Kniff bei der Arbeit mit Dateien. Die Methode `readline()` (beachten Sie: Singular, *nicht* `readlines()`) liest genau eine Zeile aus einer geöffneten Datei. Genauer gesagt ist es immer die nächste Zeile. (Datei-Objekte sind ebenfalls Iteratoren! Iteratoren wohin man sieht ….)

② Hat `readline()` eine Zeile zum Lesen, so wird `line` kein leerer String sein. Selbst wenn die Datei eine leere Zeile enthielte, wäre `line` der String `'\n'` (ein Zeilenumbruch). Ist `line` wirklich ein leerer String, bedeutet dies, dass in der Datei keine Zeilen mehr zum Lesen vorhanden sind.

③ Haben wir das Ende der Datei erreicht, sollten wir die Datei schließen und die `StopIteration`-Ausnahme auslösen. Wir sind hierher gelangt, da wir eine Übereinstimmungs- und eine Anwendungsfunktion für die nächste Regel brauchten. Die nächste Regel kommt aus der nächsten Zeile der Datei … doch es gibt keine nächste Zeile! Daher haben wir auch keinen Wert, den wir zurückgeben könnten. Die Iteration ist somit beendet. (Die Party ist vorbei ….)

Gehen wir zurück zum Beginn der `__next__()`-Methode …

```
def __next__(self):
    self.cache_index += 1
    if len(self.cache) >= self.cache_index:
        return self.cache[self.cache_index - 1]     ①

    if self.pattern_file.closed:
        raise StopIteration                         ②
    .
    .
    .
```

① `self.cache` wird eine Liste der Funktionen, die wir zum Prüfen der Übereinstimmung und zum Anwenden der einzelnen Regeln benötigen. (Das sollte Ihnen

bekannt vorkommen!) `self.cache_index` verfolgt, welches zwischengespeicherte Element wir als Nächstes zurückgeben müssen. Ist der Cache noch nicht ausgeschöpft (d. h., ist die Länge von `self.cache` größer als `self.cache_index`), haben wir getroffen! Hurra! Nun können wir die Übereinstimmungs- und Anwendungsfunktionen aus dem Cache zurückgeben, statt sie selbst von Grund auf zu erstellen.

② Wenn wir jedoch keinen Treffer im Cache landen und das Dateiobjekt geschlossen wurde (was weiter unten in der Methode passieren könnte, wie Sie im vorherigen Codeschnipsel gesehen haben), können wir nichts mehr tun. Ist die Datei geschlossen, bedeutet dies, dass sie ausgeschöpft ist – wir haben jede Zeile der Musterdatei gelesen und die Übereinstimmungs- und Anwendungsfunktionen für jedes Muster erstellt und zwischengespeichert. Die Datei ist ausgeschöpft; der Cache ist ausgeschöpft; ich bin erschöpft. Warten Sie! Bleiben Sie am Ball, wir haben es fast geschafft.

Hier sehen Sie, *was wann* passiert:

- Wenn das Modul importiert wird, erstellt es eine Instanz der `LazyRules`-Klasse (`rules`), welche die Musterdatei öffnet, jedoch nicht daraus liest.
- Wenn sie nach der ersten Übereinstimmungs- und Anwendungsfunktion gefragt wird, überprüft sie ihren Cache, der aber leer ist. Es wird daher eine einzelne Zeile aus der Musterdatei gelesen, die Übereinstimmungs- und Anwendungsfunktionen werden erstellt und dann zwischengespeichert.
- Nehmen wir an, dass die erste Regel übereinstimmt. Wenn dies der Fall ist, werden keine weiteren Übereinstimmungs- und Anwendungsfunktionen erstellt und keine weiteren Zeilen aus der Musterdatei gelesen.
- Nehmen wir außerdem an, dass der Aufrufer die `plural()`-Funktion abermals aufruft, um ein anderes Wort in den Plural zu setzen. Die `for`-Schleife innerhalb der `plural()`-Funktion ruft `iter(rules)` auf, was den Cache-Index, jedoch *nicht* das geöffnete Dateiobjekt, zurücksetzt.
- Beim ersten Durchlauf fragt die `for`-Schleife `rules` nach einem Wert, der die `__next__()`-Methode startet. Dieses mal ist der Cache mit einem Paar Übereinstimmungs- und Anwendungsfunktionen ausgestattet, die den Mustern der ersten Zeile der Musterdatei entsprechen. Da diese während der Pluralisierung des vorhergehenden Wortes gebildet und zwischengespeichert wurden, werden sie nun aus dem Cache geholt. Der Cache-Index wird erhöht und die geöffnete Datei gar nicht angerührt.
- Nehmen wir nun an, dass die erste Regel nicht passt. Die `for`-Schleife beginnt von neuem und fragt `rules` wieder nach einem Wert. Dies löst die `__next__()`-Methode erneut aus. Diesmal ist der Cache ausgeschöpft – er enthielt nur ein Element und wir fragen nach einem zweiten – also fährt die `__next__()`-Methode fort. Sie liest eine weitere Zeile aus der geöffneten Datei, bildet aus den Mustern Übereinstimmungs- und Anwendungsfunktionen und speichert diese zwischen.
- Der *Lesen-Bilden-und-Zwischenspeichern-Vorgang* wird so lange fortgeführt, wie die aus der Musterdatei gelesenen Regeln nicht zu dem Wort passen, das wir

in den Plural setzen möchten. Finden wir vor dem Erreichen des Dateiendes eine übereinstimmende Regel, verwenden wir diese und beenden den Lesevorgang. (Die Datei bleibt weiterhin geöffnet.) Der Dateizeiger bleibt an der Stelle an der wir gestoppt haben und wartet auf die nächste `readline()`-Anweisung. Der Cache erhält dadurch mehr Elemente und wenn wir ein neues Wort pluralisieren wollen und von vorne beginnen, wird jedes dieser Elemente ausprobiert, bevor die nächste Zeile der Musterdatei gelesen wird.

Wir haben den Himmel der Pluralisierung erreicht:

1. **Minimale Startzeit:** Beim `import` wird lediglich eine einzelne Klasse instanziiert und eine Datei geöffnet (jedoch nicht gelesen).
2. **Maximale Performance:** Das vorherige Beispiel las die Datei bei jedem neuen zu pluralisierenden Wort und erstellte dynamisch die nötigen Funktionen. Die neue Version nutzt einen Cache, um die Funktionen nach ihrer Erstellung zwischenzuspeichern. Sie liest die Musterdatei höchstens einmal komplett, egal wie viele Wörter Sie pluralisieren.
3. **Trennung von Code und Daten:** Alle Muster werden in einer eigenen Datei gespeichert. Code ist Code und Daten sind Daten, und niemals sollen sich die beiden begegnen.

Kapitel 9
Erweiterte Iteratoren

9.1 Los geht's

`HAWAII + IDAHO + IOWA + OHIO == STATES`. Oder anders ausgedrückt: `510199 + 98153 + 9301 + 3593 == 621246`. Spreche ich in Rätseln? Ja, denn es *ist* ein Rätsel.

Ich werde es Ihnen genau darlegen.

```
HAWAII + IDAHO + IOWA + OHIO == STATES
510199 + 98153 + 9301 + 3593 == 621246
H = 5
A = 1
W = 0
I = 9
D = 8
O = 3
S = 6
T = 2
E = 4
```

Solche Rätsel werden als *Alphametiken* bezeichnet. Die Buchstaben ergeben Wörter, doch wenn Sie jeden Buchstaben durch eine Ziffer von `0-9` ersetzen, kommt eine mathematische Gleichung heraus. Nun geht es darum, herauszufinden, welche Ziffer für welchen Buchstaben steht. Jede Ziffer muss genau einem Buchstaben zugeordnet werden und kein Wort darf mit der Ziffer `0` beginnen.

Das bekannteste alphametische Rätsel lautet: `SEND + MORE = MONEY`.

In diesem Kapitel sehen wir uns ein unglaubliches von Raymond Hettinger entwickeltes Python-Programm an, das alphametische Rätsel in gerade einmal 14 Zeilen Code löst.

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_9, © Springer-Verlag Berlin Heidelberg 2010

```
import re
import itertools

def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
    assert len(unique_characters) <= 10, 'Too many letters'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + \
        ''.join(unique_characters - first_letters)
    characters = tuple(ord(c) for c in sorted_characters)
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):
        if zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters, guess)))
            if eval(equation):
                return equation

if __name__ == '__main__':
    import sys
    for puzzle in sys.argv[1:]:
        print(puzzle)
        solution = solve(puzzle)
        if solution:
            print(solution)
```

Sie können das Programm von der Kommandozeile aus starten. Unter Linux würde dies wie folgt aussehen. (Das kann, je nach Rechnergeschwindigkeit, einen Moment dauern. Haben Sie Geduld!)

```
you@localhost:~/diveintopython3/examples$ python3 alphametics.py
"HAWAII + IDAHO + IOWA + OHIO == STATES"
HAWAII + IDAHO + IOWA + OHIO = STATES
510199 + 98153 + 9301 + 3593 == 621246
you@localhost:~/diveintopython3/examples$ python3 alphametics.py
"I + LOVE + YOU == DORA"
I + LOVE + YOU == DORA
1 + 2784 + 975 == 3760
you@localhost:~/diveintopython3/examples$ python3 alphametics.py
"SEND + MORE == MONEY"
SEND + MORE == MONEY
9567 + 1085 == 10652
```

9.2 Alle Vorkommen eines Musters finden

Als Erstes sucht dieser Alphametik-Löser alle Buchstaben (`A-Z`) in dem Rätsel.

```
>>> import re
>>> re.findall('[0-9]+', '16 2-by-4s in rows of 8')  ①
['16', '2', '4', '8']
>>> re.findall('[A-Z]+', 'SEND + MORE == MONEY')     ②
['SEND', 'MORE', 'MONEY']
```

① Das Modul `re` ist Pythons Implementierung regulärer Ausdrücke. Es beinhaltet eine raffinierte Funktion namens `findall()`, die ein Muster und einen String übernimmt und alle Vorkommen des Musters innerhalb des Strings sucht. Im vorliegenden Fall werden Zahlenfolgen gesucht. Die `findall()`-Funktion gibt eine Liste aller dem Muster entsprechenden Substrings zurück.

② Hier werden Buchstabenfolgen gesucht. Der Rückgabewert ist wieder eine Liste und jedes Element der Liste ist ein dem Muster entsprechender String.

Hier nun ein weiteres Beispiel, das Ihre grauen Zellen anregen wird.

```
>>> re.findall(' s.*? s', "The sixth sick sheikh's sixth sheep's sick.")
[' sixth s', " sheikh's s", " sheep's s"]
```

Überrascht? Der reguläre Ausdruck sucht nach einem Leerzeichen, einem `s`, der kürzest möglichen Zeichenfolge (`.*?`), einem Leerzeichen und wieder nach einem `s`. Wenn ich mir den Eingabestring ansehe, erkenne ich fünf Treffer:

1. The **sixth s**ick sheikh's sixth sheep's sick.
2. The sixth **sick s**heikh's sixth sheep's sick.
3. The sixth sick **sheikh's s**ixth sheep's sick.
4. The sixth sick sheikh's **sixth s**heep's sick.
5. The sixth sick sheikh's sixth **sheep's s**ick.

(Das sechste Schaf des sechsten kranken Scheichs ist krank.)

Die Funktion `re.findall()` hat jedoch nur drei Treffer zurückgegeben; den ersten, den dritten und den fünften. Warum? Weil keine sich überlappenden Treffer zurückgegeben werden. Der erste Treffer überlappt den zweiten Treffer; daher wird der erste Treffer zurückgegeben und der zweite Treffer übersprungen. Der dritte Treffer überlappt den vierten Treffer, also wird der dritte zurückgegeben und der vierte übersprungen. Dann wird der fünfte Treffer zurückgegeben. Drei Treffer, nicht fünf!

Das hat nichts mit unserem Alphametik-Löser zu tun. Ich fand es nur interessant.

9.3 Die einmaligen Elemente einer Folge finden

Sets machen es sehr einfach die einmaligen Elemente einer Folge zu finden.

```
>>> a_list = ['The', 'sixth', 'sick', "sheik's", 'sixth', "sheep's", 'sick']
>>> set(a_list)                                    ①
{'sixth', 'The', "sheep's", 'sick', "sheik's"}
>>> a_string = 'EAST IS EAST'
>>> set(a_string)                                  ②
{'A', ' ', 'E', 'I', 'S', 'T'}
>>> words = ['SEND', 'MORE', 'MONEY']
>>> ''.join(words)                                 ③
'SENDMOREMONEY'
>>> set(''.join(words))                            ④
{'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

① Haben wir eine Liste aus mehreren Strings, so gibt die `set()`-Funktion ein Set aus den einmaligen Strings der Liste zurück. Dies ist einleuchtend, wenn Sie es sich wie eine `for`-Schleife vorstellen. Nimm das erste Element der Liste und füge es in das Set ein. Zweites. Drittes. Viertes. Fünftes – Moment, dieses Element ist bereits im Set vorhanden, und da Python keine doppelten Werte in einem Set erlaubt, ist es auch zum Schluss nur einmal enthalten. Sechstes. Siebtes – schon wieder ein doppelter Wert, also wird es nur einmal aufgenommen. Das Resultat? Alle einmaligen Elemente der Liste, *ohne* Duplikate. Die ursprüngliche Liste muss dazu nicht einmal sortiert werden.

② Dieselbe Vorgehensweise funktioniert auch mit Strings, denn ein String ist nur eine Zeichenfolge.

③ `''.join(a_list)` verkettet die einzelnen Strings einer Liste zu einem einzigen String.

④ Haben wir also eine Liste von Strings, so gibt diese Codezeile alle einmaligen Zeichen aller Strings, *ohne* Duplikate, zurück.

Der Alphametik-Löser verwendet dieses Verfahren, um eine Liste aller einmaligen Zeichen des Rätsels zu erhalten.

```
unique_characters = set(''.join(words))
```

Diese Liste wird später verwendet, um den Zeichen während des Iterierens durch die möglichen Lösungen Ziffern zuzuweisen.

9.4 Bedingungen aufstellen

Wie viele andere Programmiersprachen besitzt auch Python eine `assert`-Anweisung. Sehen wir uns an, wie diese funktioniert.

```
>>> assert 1 + 1 == 2                                          ①
>>> assert 1 + 1 == 3                                          ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 2 + 2 == 5, "Only for very large values of 2"       ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Only for very large values of 2
```

① Auf die `assert`-Anweisung folgt ein beliebiger Python-Ausdruck. Im vorliegenden Fall ergibt der Ausdruck `1 + 1 == 2` `True`, also tut die `assert`-Anweisung gar nichts.

② Ergibt der Ausdruck dagegen `False`, so löst die `assert`-Anweisung einen `AssertionError` aus.

③ Sie können eine lesbare Nachricht einfügen, die angezeigt wird, wenn der `AssertionError` auftritt.

Diese Codezeile …

```
assert len(unique_characters) <= 10, 'Too many letters'
```

… entspricht somit diesen Zeilen:

```
if len(unique_characters) > 10:
    raise AssertionError('Too many letters')
```

Der Alphametik-Löser verwendet genau diese `assert`-Anweisung, um schon zu Beginn sicherzustellen, dass das Rätsel nicht mehr als zehn einmalige Buchstaben besitzt. Da jeder Buchstabe genau einer Ziffer zugewiesen ist und es nur zehn Ziffern gibt, kann ein Rätsel mit mehr als zehn einmaligen Buchstaben keine Lösung haben.

9.5 Generator-Ausdrücke

Ein Generator-Ausdruck ist wie eine Generator-Funktion ohne die Funktion.

```
>>> unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
>>> gen = (ord(c) for c in unique_characters)  ①
>>> gen                                        ②
<generator object <genexpr> at 0x00BADC10>
>>> next(gen)                                  ③
69
>>> next(gen)
68
>>> tuple(ord(c) for c in unique_characters)   ④
(69, 68, 77, 79, 78, 83, 82, 89)
```

① Ein Generator-Ausdruck funktioniert wie eine anonyme Funktion die Werte hervorbringt. Der Ausdruck selbst sieht aus wie eine List Comprehension, ist jedoch in runde statt in eckige Klammern eingeschlossen.

② Der Generator-Ausdruck gibt einen Iterator zurück.

③ Durch den Aufruf von `next(gen)` wird der nächste Wert vom Iterator zurückgegeben.

④ Wenn Sie mögen, können Sie alle möglichen Werte durchlaufen und ein Tupel, eine Liste oder ein Set zurückgeben, indem Sie den Generator-Ausdruck `tuple()`, `list()` oder `set()` übergeben. Sie benötigen hier keine zusätzlichen Klammern – übergeben Sie der `tuple()`-Funktion einfach den Ausdruck `ord(c) for c in unique_characters`; Python erkennt automatisch, dass es sich dabei um einen Generator-Ausdruck handelt.

☞ Die Verwendung eines Generator-Ausdrucks anstelle einer List Comprehension kann sowohl den Prozessor, als auch den Arbeitsspeicher entlasten. Erstellen Sie eine Liste, die Sie danach nicht mehr benötigen (weil Sie sie z. B. `tuple()` oder `set()` übergeben), verwenden Sie lieber einen Generator-Ausdruck!

Eine weitere Möglichkeit dasselbe zu erreichen besteht darin, eine Generator-Funktion zu benutzen:

```
def ord_map(a_string):
    for c in a_string:
        yield ord(c)

gen = ord_map(unique_characters)
```

Der Generator-Ausdruck ist kompakter, entspricht funktionell aber der längeren Version.

9.6 Permutationen berechnen … Auf die faule Art!

Was sind Permutationen überhaupt? Permutationen sind ein mathematisches Konzept. Deren genaue Erklärung würde hier zu weit führen, weshalb ich Sie auf Wikipedia oder ein Mathematik-Buch verweise.

Der Grundgedanke ist der, dass man eine Liste von Dingen (Zahlen, Buchstaben, Tanzbären) hat und alle Möglichkeiten findet, diese in kleinere Listen aufzuteilen. Alle kleineren Listen haben dieselbe Größe, die mindestens `1` und höchstens die Anzahl aller Elemente sein kann. Oh, und es gibt keine Wiederholungen. Mathematiker sagen Dinge wie „suchen wir die Permutationen dreier verschiedener Elemente und nehmen uns dazu zwei auf einmal vor“, was Folgendes bedeutet: Wir haben eine Folge von drei Elementen und wollen alle möglichen geordneten Paare finden.

```
>>> import itertools                                    ①
>>> perms = itertools.permutations([1, 2, 3], 2)       ②
>>> next(perms)                                         ③
(1, 2)
>>> next(perms)
(1, 3)
>>> next(perms)
(2, 1)                                                  ④
>>> next(perms)
(2, 3)
>>> next(perms)
(3, 1)
>>> next(perms)
(3, 2)
>>> next(perms)                                         ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

① Das Modul `itertools` enthält viel cooles Zeug, so auch eine `permutations()`-Funktion, die uns die harte Arbeit beim Suchen nach Permutationen abnimmt.

② Die `permutations()`-Funktion übernimmt eine Folge (hier eine Liste aus drei Ganzzahlen) und eine Zahl, welche die Anzahl der Elemente angibt, die Sie in jeder der kleineren Gruppen haben möchten. Die Funktion gibt einen Iterator zurück, den Sie in einer `for`-Schleife oder an jedem anderen iterierenden Ort verwenden können. Hier durchlaufe ich den Iterator schrittweise, um alle Werte zu zeigen.

③ Die erste Permutation von `[1, 2, 3]` ist `(1, 2)`, wenn man zwei Werte auf einmal nimmt.

④ Beachten Sie, dass die Permutationen geordnet sind: `(2, 1)` ist nicht dasselbe wie `(1, 2)`.

⑤ Das war's! Dies sind alle Permutationen, die sich aus `[1, 2, 3]` bilden lassen, wenn man zwei Werte auf einmal nimmt. Paare wie `(1, 1)` oder `(2, 2)` tauchen niemals auf, da sich die Werte darin wiederholen und es somit keine gültigen Permutationen sind. Gibt es keine weiteren Permutationen mehr, löst der Iterator eine `StopIteration`-Ausnahme aus.

Die `permutations()`-Funktion kann nicht nur eine Liste übernehmen. Alle Folgen sind möglich – auch ein String.

```
>>> import itertools
>>> perms = itertools.permutations('ABC', 3)  ①
>>> next(perms)
('A', 'B', 'C')                               ②
>>> next(perms)
('A', 'C', 'B')
>>> next(perms)
('B', 'A', 'C')
>>> next(perms)
('B', 'C', 'A')
>>> next(perms)
('C', 'A', 'B')
>>> next(perms)
('C', 'B', 'A')
>>> next(perms)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> list(itertools.permutations('ABC', 3))    ③
[('A', 'B', 'C'), ('A', 'C', 'B'),
 ('B', 'A', 'C'), ('B', 'C', 'A'),
 ('C', 'A', 'B'), ('C', 'B', 'A')]
```

① Ein String ist lediglich eine Zeichenfolge. Im Zusammenhang mit Permutationen entspricht der String `'ABC'` der Liste `['A', 'B', 'C']`.

② Nehmen wir drei Werte auf einmal, so ist die erste Permutation von `['A', 'B', 'C']` `('A', 'B', 'C')`. Es existieren fünf weitere Permutationen – dieselben drei Zeichen in jeder erdenklichen Reihenfolge.

③ Da die `permutations()`-Funktion immer einen Iterator zurückgibt, besteht eine einfache Möglichkeit des Debuggens der Permutation im Übergeben des Iterators an die integrierte `list()`-Funktion, die dann sofort alle Permutationen anzeigt.

9.7 Anderes cooles Zeug im Modul `itertools`

```
>>> import itertools
>>> list(itertools.product('ABC', '123'))   ①
[('A', '1'), ('A', '2'), ('A', '3'),
 ('B', '1'), ('B', '2'), ('B', '3'),
 ('C', '1'), ('C', '2'), ('C', '3')]
>>> list(itertools.combinations('ABC', 2))  ②
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

① Die Funktion `itertools.product()` gibt einen Iterator zurück, der das kartesische Produkt zweier Folgen enthält.

② Die Funktion `itertools.combinations()` gibt einen Iterator zurück, der alle möglichen Kombinationen der vorhandenen Folge mit der angegebenen Länge enthält. Dies ist wie die Funktion `itertools.permutations()`, abgesehen davon, dass die Kombinationen keine Duplikate enthalten. Es treten also nicht mehrmals dieselben Werte in anderer Reihenfolge auf. `itertools.permutations('ABC', 2)` gibt also sowohl `('A', 'B')` als auch `('B', 'A')` (neben anderen) zurück. `itertools.combinations('ABC', 2)` gibt dagegen nicht `('B', 'A')` zurück, da es ein Duplikat von `('A', 'B')` in anderer Reihenfolge ist.

```
>>> names = list(open('examples/favorite-people.txt', encoding='utf-8'))①
>>> names
['Dora\n', 'Ethan\n', 'Wesley\n', 'John\n', 'Anne\n',
'Mike\n', 'Chris\n', 'Sarah\n', 'Alex\n', 'Lizzie\n']
>>> names = [name.rstrip() for name in names]                          ②
>>> names
['Dora', 'Ethan', 'Wesley', 'John', 'Anne',
'Mike', 'Chris', 'Sarah', 'Alex', 'Lizzie']
>>> names = sorted(names)                                              ③
>>> names
['Alex', 'Anne', 'Chris', 'Dora', 'Ethan',
'John', 'Lizzie', 'Mike', 'Sarah', 'Wesley']
>>> names = sorted(names, key=len)                                     ④
>>> names
['Alex', 'Anne', 'Dora', 'John', 'Mike',
'Chris', 'Ethan', 'Sarah', 'Lizzie', 'Wesley']
```

① Dieser Ausdruck gibt eine Liste der Zeilen in einer Textdatei zurück.

② Leider enthält der Ausdruck `list(open(filename))` (in diesem Beispiel) auch die Zeilenumbrüche am Ende jeder Zeile. Die List Comprehension nutzt die Stringmethode `rstrip()`, um anhängenden Whitespace aus jeder Zeile zu löschen. (Strings besitzen auch eine `lstrip()`-Methode, die führenden Whitespace löscht und eine `strip()`-Methode, die beide Arten löscht.)

③ Die Funktion `sorted()` übernimmt eine Liste und gibt diese sortiert wieder zurück. Standardmäßig sortiert sie alphabetisch.

④ Doch die `sorted()`-Funktion kann auch eine Funktion als Schlüssel übernehmen. Dann wird nach diesem Schlüssel sortiert. Im vorliegenden Fall ist `len()` die Sortierfunktion, also wird nach `len(jedes Element)` sortiert. Die kürzeren Namen kommen zuerst, dann die längeren und schließlich die längsten.

Was hat das alles mit dem `itertools`-Modul zu tun? Gut, dass Sie fragen!

```
…Fortsetzung der vorherigen interaktiven Shell…
>>> import itertools
>>> groups = itertools.groupby(names, len)  ①
>>> groups
<itertools.groupby object at 0x00BB20C0>
>>> list(groups)
[(4, <itertools._grouper object at 0x00BA8BF0>),
 (5, <itertools._grouper object at 0x00BB4050>),
 (6, <itertools._grouper object at 0x00BB4030>)]
>>> groups = itertools.groupby(names, len)   ②
>>> for name_length, name_iter in groups:    ③
...     print('Names with {0:d} letters:'.format(name_length))
...     for name in name_iter:
...         print(name)
...
Names with 4 letters:
Alex
Anne
Dora
John
Mike
Names with 5 letters:
Chris
Ethan
Sarah
Names with 6 letters:
Lizzie
Wesley
```

① Die Funktion `itertools.groupby()` übernimmt eine Folge sowie eine Schlüsselfunktion und gibt einen Iterator zurück, der Paare generiert. Jedes dieser Paare enthält das Ergebnis von `schluessel_funktion(jedes Element)` und einen weiteren Iterator, der alle Elemente enthält, die dieses Schlüsselergebnis gemeinsam nutzen.

② Der Aufruf der `list()`-Funktion hat den Iterator „erschöpft“, d. h. Sie haben bereits jedes Element des Iterators generiert, um die Liste zu erstellen. Sie können einen Iterator nicht zurücksetzen, wenn er einmal erschöpft ist. Möchten Sie den Iterator erneut durchlaufen, so müssen Sie `itertools.groupby()` noch einmal aufrufen und so einen neuen Iterator erzeugen.

③ In diesem Beispiel haben wir bereits eine Liste von nach Länge sortierten Namen. `itertools.groupby(names, len)` fügt alle Namen aus vier Buchstaben in einen Iterator ein, alle Namen aus fünf Buchstaben in einen weiteren Iterator und so weiter. Die `groupby()`-Funktion ist völlig generisch; sie könnte Strings nach ihrem Anfangsbuchstaben gruppieren, Zahlen nach der Anzahl der Faktoren oder nach jeder anderen Schlüsselfunktion, die Sie sich vorstellen können.

☞ Die Funktion `itertools.groupby()` funktioniert nur, wenn die Eingabefolge bereits durch die Gruppierungsfunktion sortiert wurde. Im obigen Beispiel haben Sie eine Namensliste mit der `len()`-Funktion gruppiert. Das funktionierte nur, weil die Eingabeliste bereits nach Länge sortiert war.

Sehen Sie genau hin!

```
>>> list(range(0, 3))
[0, 1, 2]
>>> list(range(10, 13))
[10, 11, 12]
>>> list(itertools.chain(range(0, 3), range(10, 13)))        ①
[0, 1, 2, 10, 11, 12]
>>> list(zip(range(0, 3), range(10, 13)))                    ②
[(0, 10), (1, 11), (2, 12)]
>>> list(zip(range(0, 3), range(10, 14)))                    ③
[(0, 10), (1, 11), (2, 12)]
>>> list(itertools.zip_longest(range(0, 3), range(10, 14)))  ④
[(0, 10), (1, 11), (2, 12), (None, 13)]
```

① Die Funktion `itertools.chain()` übernimmt zwei Iteratoren und gibt einen Iterator zurück, der alle Elemente des ersten Iterators, gefolgt von allen Elementen des zweiten Iterators enthält. (Die Funktion kann eine beliebige Anzahl an Iteratoren übernehmen und verkettet sie alle in der Reihenfolge, in der sie übergeben wurden.)

② Die Funktion `zip()` tut etwas recht Alltägliches, was jedoch sehr nützlich sein kann: Sie übernimmt eine beliebige Anzahl an Folgen und gibt einen Iterator zurück, der wiederum Tupel der ersten Elemente jeder Folge, dann der zweiten Elemente jeder Folge, dann der dritten und so weiter zurückgibt.

③ Die `zip()`-Funktion stoppt am Ende der kürzesten Folge. `range(10, 14)` enthält vier Elemente `(10, 11, 12, 13)`, `range(0, 3)` dagegen nur drei, also gibt die `zip()`-Funktion einen Iterator aus drei Elementen zurück.

④ Die Funktion `itertools.zip_longest()` stoppt am Ende der längsten Folge und fügt für alle Elemente die über das Ende der kürzeren Folgen hinausgehen `None`-Werte ein.

Na schön, das war ja alles ganz interessant, aber was hat das mit unserem Alphametik-Löser zu tun? Sehen Sie es sich selbst an:

```
>>> characters = ('S', 'M', 'E', 'D', 'O', 'N', 'R', 'Y')
>>> guess = ('1', '2', '0', '3', '4', '5', '6', '7')
>>> tuple(zip(characters, guess))  ①
(('S', '1'), ('M', '2'), ('E', '0'), ('D', '3'),
 ('O', '4'), ('N', '5'), ('R', '6'), ('Y', '7'))
>>> dict(zip(characters, guess))   ②
{'E': '0', 'D': '3', 'M': '2', 'O': '4',
 'N': '5', 'S': '1', 'R': '6', 'Y': '7'}
```

① Hier haben wir eine Buchstaben- und eine Ziffernliste (dargestellt als Strings aus jeweils einem Zeichen). Die `zip()`-Funktion erstellt daraus geordnete Paare aus je einem Buchstaben und einer Ziffer.

② Was ist daran so toll? Nun, diese Datenstruktur eignet sich perfekt dazu, sie der `dict()`-Funktion zu übergeben und so ein Dictionary zu erstellen, das die Buchstaben als Schlüssel und die zugehörigen Ziffern als Werte benutzt. (Das ist nicht die einzige Möglichkeit dies zu erreichen. Sie könnten auch eine Dictionary Comprehension verwenden, um das Dictionary direkt zu erstellen.) Auch wenn die angezeigte Reihenfolge der Paare im Dictionary eine andere ist (Dictionarys haben sowieso keine „Reihenfolge“) als die ursprüngliche, so stimmt die Buchstaben-Ziffern-Zuordnung doch mit der Reihenfolge der `characters`- und `guess`-Folgen überein.

Der Alphametik-Löser verwendet diese Vorgehensweise zum Erstellen eines Dictionary, das für jede mögliche Lösung die Buchstaben des Rätsels auf die Ziffern der Lösung abbildet.

```
characters = tuple(ord(c) for c in sorted_characters)
digits = tuple(ord(c) for c in '0123456789')
...
for guess in itertools.permutations(digits, len(characters)):
    ...
    equation = puzzle.translate(dict(zip(characters, guess)))
```

Doch was macht diese `translate()`-Methode? Aha, nun kommen wir zum wirklich spaßigen Teil!

9.8 Eine neue Art der String-Manipulation

Strings haben in Python sehr viele Methoden. Einige dieser Methoden haben Sie bereits im Kapitel *Strings* kennen gelernt: `lower()`, `count()` und `format()`. Nun möchte ich Ihnen eine mächtige aber recht unbekannte Methode vorstellen: `translate()`.

```
>>> translation_table = {ord('A'): ord('O')}  ①
>>> translation_table                         ②
{65: 79}
>>> 'MARK'.translate(translation_table)       ③
'MORK'
```

① Eine String-Umwandlung beginnt mit einer Umwandlungstabelle. Diese Tabelle ist ein einfaches Dictionary, das einem Zeichen ein anderes zuordnet. Naja,

„Zeichen“ ist nicht korrekt – die Umwandlungstabelle ordnet ein Byte einem anderen zu.

② Bedenken Sie, dass Bytes in Python 3 Ganzzahlen sind. Die Funktion `ord()` gibt den ASCII-Wert eines Zeichens zurück, welcher im Falle der Buchstaben von `A-Z` immer ein Byte von `65` bis `90` ist.

③ Wird die `translate()`-Methode auf einen String angewendet, so übernimmt sie eine Umwandlungstabelle und wendet diese auf den String an. Dadurch werden alle Vorkommen der Schlüssel der Tabelle durch die zugeordneten Werte ersetzt. Hier wird also `MARK` zu `MORK`.

Was hat das mit der Lösung alphametischer Rätsel zu tun? Wie sich herausstellt, sehr viel!

```
>>> characters = tuple(ord(c) for c in 'SMEDONRY')       ①
>>> characters
(83, 77, 69, 68, 79, 78, 82, 89)
>>> guess = tuple(ord(c) for c in '91570682')            ②
>>> guess
(57, 49, 53, 55, 48, 54, 56, 50)
>>> translation_table = dict(zip(characters, guess))     ③
>>> translation_table
{68: 55, 69: 53, 77: 49, 78: 54, 79: 48, 82: 56, 83: 57, 89: 50}
>>> 'SEND + MORE == MONEY'.translate(translation_table)  ④
'9567 + 1085 == 10652'
```

① Wir nutzen einen Generator-Ausdruck, um die Byte-Werte jedes Zeichens des Strings zu berechnen. `characters` ist ein Beispiel des Wertes von `sorted_characters` in der Funktion `alphametics.solve()`.

② Wir verwenden einen weiteren Generator-Ausdruck zum Berechnen der Byte-Werte jeder Ziffer des Strings. Das Ergebnis, `guess`, entspricht der Form nach der Rückgabe der Funktion `itertools.permutations()` in der `alphametics.solve()`-Funktion.

③ Die Umwandlungstabelle wird durch das „Zusammenzippen“ von `characters` und `guess` und dem anschließenden Erstellen eines Dictionary aus der Folge der Paare generiert. Genau dies tut die `alphametics.solve()`-Funktion innerhalb der `for`-Schleife.

④ Schlussendlich übergeben wir diese Umwandlungstabelle an die `translate()`-Methode des ursprünglichen Strings des Rätsels. Dadurch wird jeder Buchstabe des Strings in die zugeordnete Ziffer umgewandelt (basierend auf den Buchstaben in `characters` und den Ziffern in `guess`). Das Ergebnis ist ein gültiger Python-Ausdruck als String.

Das ist ziemlich eindrucksvoll. Doch was können Sie mit einem String, der ein gültiger Python-Ausdruck ist, tun?

9.9 Herausfinden, ob ein beliebiger String ein Python-Ausdruck ist

Dies ist nun das letzte Puzzlestück des Rätsels (besser gesagt, das letzte Puzzlestück des Rätsel-Lösers). Nach all den String-Manipulationen stehen wir nun vor einem String wie `'9567 + 1085 == 10652'`. Doch es ist ein String und was bringt uns ein String? Nutzen wie `eval()`, Pythons allumfassendes Auswertungs-Tool.

```
>>> eval('1 + 1 == 2')
True
>>> eval('1 + 1 == 3')
False
>>> eval('9567 + 1085 == 10652')
True
```

Warten Sie, es gibt noch mehr! Die `eval()`-Funktion ist nicht auf boolesche Ausdrücke beschränkt. Sie kann mit *jedem beliebigen* Python-Ausdruck umgehen und *jeden beliebigen* Datentyp zurückgeben.

```
>>> eval('"A" + "B"')
'AB'
>>> eval('"MARK".translate({65: 79})')
'MORK'
>>> eval('"AAAAA".count("A")')
5
>>> eval('["*"] * 5')
['*', '*', '*', '*', '*']
```

Halt! Das ist noch nicht alles.

```
>>> x = 5
>>> eval("x * 5")          ①
25
>>> eval("pow(x, 2)")      ②
25
>>> import math
>>> eval("math.sqrt(x)")   ③
2.2360679774997898
```

① Der von `eval()` übernommene Ausdruck kann globale Variablen referenzieren, die außerhalb von `eval()` definiert wurden. Wird `eval()` innerhalb einer Funktion aufgerufen, so ist auch das Referenzieren lokaler Variablen kein Problem.

② Auch Funktionen lassen sich nutzen.

③ Und sogar Module.

He, warten Sie einen Moment …

```
>>> import subprocess
>>> eval("subprocess.getoutput('ls ~')")                    ①
'Desktop         Library         Pictures \
 Documents       Movies          Public   \
 Music           Sites'
>>> eval("subprocess.getoutput('rm /some/random/file')")    ②
```

① Mithilfe des Moduls `subprocess` können Sie beliebige Shell-Befehle (Kommandozeilen-Befehle) ausführen und das Ergebnis als Python-String erhalten.

② Diese Shell-Befehle können dauerhafte Folgen haben.

Es ist sogar weit schlimmer. Es existiert eine globale `__import__()`-Funktion, die einen Modulnamen als String übernimmt, dieses Modul importiert und eine Referenz darauf zurückgibt. Im Zusammenspiel mit `eval()` kann man so einen einzigen Ausdruck basteln, der all Ihre Dateien ausradiert.

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')")
```

Stellen Sie sich nun die Ausgabe von `'rm -rf ~'` vor. Nun, es gäbe gar keine Ausgabe, aber auch keine Ihrer Dateien mehr.

`eval()` ist ÜBEL!

Das Üble daran ist die Auswertung beliebiger Ausdrücke aus unsicheren Quellen. Sie sollten `eval()` nur mit sicheren Eingaben nutzen. Natürlich fragt man sich nun, was *sicher* bedeutet. Eines kann ich Ihnen mit Sicherheit sagen: Sie sollten diesen Alphametik-Löser NICHT als tollen, kleinen Webservice ins Internet stellen. Denken Sie bloß nicht „die Funktion führt sehr viele String-Manipulationen aus, bevor überhaupt etwas ausgewertet wird. Ich kann mir nicht vorstellen, wie jemand das ausnutzen könnte.“ Irgendjemand WIRD herausfinden, wie er schadhaften ausführbaren Code durch die ganzen String-Manipulationen schleust. Dann können Sie Ihrem Server „Lebewohl!“ sagen.

Es gibt doch sicher eine Möglichkeit Ausdrücke sicher auszuwerten!? Kann man `eval()` in einem geschützten Bereich ausführen, in dem es keinen Schaden anrichten kann? Naja, nicht ganz.

```
>>> x = 5
>>> eval("x * 5", {}, {})                ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined
>>> eval("x * 5", {"x": x}, {})          ②
>>> import math
>>> eval("math.sqrt(x)", {"x": x}, {})   ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'math' is not defined
```

① Der zweite und dritte an `eval()` übergebene Parameter stehen für den globalen und den lokalen Namensbereich zur Auswertung des Ausdrucks. In diesem Fall sind sie beide leer, d. h., wenn der String `''x * 5''` ausgewertet wird, ist `x` weder im globalen, noch im lokalen Namensbereich vorhanden. `eval()` löst eine Ausnahme aus.

② Sie können auswählen, welche Werte der globale Namensbereich enthalten soll, indem Sie sie einzeln angeben. Dann werden diese – und nur diese – während der Auswertung zur Verfügung stehen.

③ Auch wenn Sie das `math`-Modul gerade erst importiert haben, so ist es nicht im an die `eval()`-Funktion übergebenen Namensbereich vorhanden. Die Auswertung schlägt fehl.

Hmm, das war einfach. Lassen Sie mich jetzt einen Webservice daraus machen!

```
>>> eval("pow(5, 2)", {}, {})                    ①
25
>>> eval("__import__('math').sqrt(5)", {}, {})   ②
2.2360679774997898
```

① Obwohl Sie ein leeres Dictionary für den globalen und den lokalen Namensbereich übergeben haben, stehen Pythons integrierte Funktionen während der Auswertung trotzdem zur Verfügung. `pow(5, 2)` funktioniert also, da `5` und `2` Literale sind und `pow()` eine integrierte Funktion ist.

② Leider (wenn Sie nicht wissen warum „leider“, lesen Sie weiter) ist auch die `__import__()`-Funktion eine integrierte Funktion; auch sie lässt sich hier also problemlos nutzen.

Ja, das bedeutet, dass Sie weiterhin schlimme Dinge anrichten können, selbst wenn Sie beim Aufruf von `eval()` leere Dictionarys für den globalen und den lokalen Namensbereich verwenden:

```
>>> eval("__import__('subprocess').getoutput('rm /some/random/file')", {}, {})
```

Oh, ich bin wirklich froh, dass ich keinen Webservice aus unserem Alphametik-Löser gemacht habe. Gibt es *irgendeine* Möglichkeit, `eval()` sicher zu benutzen? Nicht ganz.

```
>>> eval("__import__('math').sqrt(5)",
...      {"__builtins__":None}, {})          ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
>>> eval("__import__('subprocess').getoutput('rm -rf /')",
...      {"__builtins__":None}, {})          ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name '__import__' is not defined
```

① Zum sicheren Auswerten unsicherer Ausdrücke müssen Sie ein globales Namensbereich-Dictionary definieren, das `''__builtins__''` `None` zuordnet, Pythons Null-Wert. Intern sind die „integrierten" Funktionen in einem Pseudo-Modul namens `''__builtins__''` enthalten. Dieses Pseudo-Modul (d. h. die integrierten Funktionen) wird den auszuwertenden Ausdrücken zur Verfügung gestellt, es sei denn, Sie überschreiben es.

② Stellen Sie sicher, dass Sie `__builtins__` überschrieben haben, nicht `__builtin__`, `__built-ins__` oder irgendeine andere Variante. Diese funktionieren zwar genauso gut, bergen aber große Risiken.

Ist `eval()` jetzt also sicher? Nicht ganz.

```
>>> eval("2 ** 2147483647",
...      {"__builtins__":None}, {})        ①
```

① Auch ohne Zugriff auf `__builtins__` können Sie immer noch eine Denial-of-Service-Attacke ausführen. Der Versuch die `2147483647te` Potenz von `2` zu berechnen lastet Ihren Prozessor für einige Zeit zu 100% aus. (Sollten Sie dies in der interaktiven Shell ausprobieren, drücken Sie mehrmals `Strg-C` um es zu beenden.) Der Ausdruck wird zwar irgendwann einen Wert zurückgeben, doch in der Zwischenzeit kann Ihr Server nichts weiter tun.

Es ist also möglich unsichere Python-Ausdrücke sicher auszuwerten, solange man eine Definition von „sicher" verwendet, die im Alltag nicht sehr nützlich ist. Es ist problemlos möglich, wenn Sie einfach ein bisschen damit herumspielen wollen und es ist ebenso problemlos möglich, wenn Sie nur sichere Eingaben übergeben. Doch alles andere bedeutet Ärger.

9.10 Alles zusammenfügen

Fassen wir zusammen: Dieses Programm löst alphametische Rätsel durch *Brute Force*, d. h. es probiert alle möglichen Lösungen aus. Um dies zu erreichen …

1. Sucht es alle Buchstaben des Rätsels mithilfe der Funktion `re.findall()`
2. Sucht es alle einmaligen Buchstaben des Rätsels mithilfe von Sets und der `set()`-Funktion
3. Überprüft es mithilfe einer `assert`-Anweisung, ob mehr als zehn einmalige Buchstaben vorhanden sind (d. h. das Rätsel ist definitiv unlösbar)
4. Konvertiert es die Buchstaben mithilfe eines Generator-Objekts in ihren jeweiligen ASCII-Code
5. Berechnet es alle möglichen Lösungen mithilfe der Funktion `itertools.permutations()`
6. Konvertiert es jede mögliche Lösung mithilfe der Stringmethode `translate()` in einen Python-Ausdruck

7. Prüft es jede mögliche Lösung durch Auswerten des Python-Ausdrucks mithilfe der `eval()`-Funktion
8. Gibt es die erste Lösung zurück, deren Auswertung `True` ergibt

… in lediglich 14 Codezeilen.

Kapitel 10
Unit Testing

10.1 Los geht's (noch nicht)

In diesem Kapitel werden wir einige Dienstprogramme zum Konvertieren von römischen Zahlen schreiben und debuggen. Wie man römische Zahlen erstellt und überprüft haben Sie bereits in Kap. 6 (*Reguläre Ausdrücke – Fallbeispiel: Römische Zahlen*) gesehen. Nun überlegen wir uns, was wir tun müssen, um sowohl in die eine, als auch in die andere Richtung zu konvertieren.

Die Regeln für römische Zahlen führen uns zu einer Reihe interessanter Beobachtungen:

1. Es gibt nur eine korrekte Art, eine bestimmte Zahl als römische Zahl darzustellen.
2. Andersherum gilt dasselbe: Ist ein Zeichenstring eine gültige römische Zahl, so stellt dieser String nur eine Zahl dar (d. h. er kann nur auf eine Art interpretiert werden).
3. Es gibt nur einen begrenzten Zahlenbereich, der mit römischen Zahlen ausgedrückt werden kann, genauer gesagt die Zahlen von `1` bis `3999`. Die Römer nutzten verschiedene Verfahren zur Darstellung größerer Zahlen. So schrieben Sie z. B. einen Balken über eine Zahl, um anzuzeigen, dass der normale Wert mit `1000` multipliziert werden sollte. In diesem Kapitel einigen wir uns aber darauf, dass die römischen Zahlen von `1` bis `3999` reichen.
4. Mit römischen Zahlen lässt sich keine `0` darstellen.
5. Mit römischen Zahlen lassen sich keine negativen Zahlen darstellen.
6. Mit römischen Zahlen lassen sich keine Brüche darstellen.

Legen wir also fest, was das Modul `roman.py` tun soll. Es wird zwei Hauptfunktionen enthalten, `to_roman()` und `from_roman()`. Die `to_roman()`-Funktion soll eine Ganzzahl zwischen `1` und `3999` übernehmen und die entsprechende römische Zahl als String zurückgeben ….

Halt. Nun machen wir etwas Unerwartetes: Wir schreiben einen Test, der überprüft, ob die `to_roman()`-Funktion tut, was sie soll. Sie haben richtig gelesen: Wir werden Code schreiben, der Code testet, den wir noch gar nicht geschrieben haben.

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_10, © Springer-Verlag Berlin Heidelberg 2010

Dies nennt man *Unit Testing*. Die beiden Konvertierungsfunktionen – `to_roman()`, später auch `from_roman()` – können als eine Einheit (*Unit*) geschrieben und getestet werden, die unabhängig von einem größeren, sie importierenden Programm ist. Python stellt ein Framework zum Unit Testing bereit – ein Modul, das bezeichnenderweise `unittest` heißt.

Unit Testing ist ein wichtiger Teil einer testfixierten Entwicklungsstrategie. Sie sollten Unit Tests frühzeitig schreiben (am besten vor dem Code, den sie prüfen) und immer aktualisieren, wenn sich der Code oder die Voraussetzungen ändern. Unit Testing ist kein Ersatz für Funktions- oder Systemtests, doch es ist in allen Phasen der Entwicklung ein wichtiger Teil:

- Vor dem Schreiben des Codes werden Sie gezwungen, Ihre Anforderungen auszuarbeiten.
- Während Sie den Code schreiben schützt Unit Testing Sie vor dem Übercoden. Werden alle Testfälle erfolgreich durchlaufen, ist die Funktion komplett.
- Beim Refactoring von Code stellen Unit Tests sicher, dass sich die neue Version genau wie die alte Version verhält.
- Bei der Wartung des Codes schützen Unit Tests Sie vor Anschuldigungen anderer, die behaupten, Ihre letzte Änderung hätte ihren alten Code zerstört. („Aber lieber Herr Kollege, alle Unit Tests wurden erfolgreich durchlaufen, bevor ich den neuen Code eingebaut habe ….“)
- Beim Schreiben von Code im Team erhöht Unit Testing die Sicherheit, dass Ihr beigesteuerter Code nicht den Code eines anderen zerstört, da Sie zuerst dessen Unit Tests laufen lassen können. (So etwas habe ich schon bei Code-Sprints gesehen. Ein Team teilt die Aufgaben auf, jeder erhält die Spezifikationen seiner Aufgabe, schreibt Unit Tests dafür und teilt diese mit dem Rest des Teams. Auf diese Weise programmiert niemand etwas, das nicht mit dem Code der anderen Teammitglieder funktioniert.) (Bei Code-Sprints geht es darum, bestimmte Ziele in einer bestimmten Zeit zu erreichen; Anm. d. Übers.)

10.2 Eine Frage

Ein Testfall beantwortet eine Frage über den Code den er testet. Ein Testfall sollte …

- … ohne menschliche Eingaben lauffähig sein. Unit Testing bedeutet Automatisierung.
- … selbst entscheiden können – also ohne dass ein Mensch die Ergebnisse auswertet – ob die getestete Funktion den Test erfolgreich durchlaufen hat oder nicht.
- … getrennt von anderen Testfällen laufen (auch wenn sie dieselben Funktionen prüfen). Jeder Testfall ist eine Insel.

Lassen Sie uns auf Basis dessen den ersten Testfall für die erste Anforderung erstellen:

1. Die `to_roman()`-Funktion soll die römischen Zahlen für alle Ganzzahlen zwischen `1` und `3999` zurückgeben.

Es ist nicht sofort ersichtlich, wie dieser Code … nun, irgendetwas tut. Es wird eine Klasse ohne `__init__()`-Methode definiert. Die Klasse hat eine andere Methode, doch diese wird niemals aufgerufen. Das Skript enthält einen `__main__`-Block, doch verweist weder auf die Klasse, noch auf deren Methode. Aber es tut etwas, versprochen.

```
import roman1
import unittest

class KnownValues(unittest.TestCase):                    ①
    known_values = ( (1, 'I'),
                     (2, 'II'),
                     (3, 'III'),
                     (4, 'IV'),
                     (5, 'V'),
                     (6, 'VI'),
                     (7, 'VII'),
                     (8, 'VIII'),
                     (9, 'IX'),
                     (10, 'X'),
                     (50, 'L'),
                     (100, 'C'),
                     (500, 'D'),
                     (1000, 'M'),
                     (31, 'XXXI'),
                     (148, 'CXLVIII'),
                     (294, 'CCXCIV'),
                     (312, 'CCCXII'),
                     (421, 'CDXXI'),
                     (528, 'DXXVIII'),
                     (621, 'DCXXI'),
                     (782, 'DCCLXXXII'),
                     (870, 'DCCCLXX'),
                     (941, 'CMXLI'),
                     (1043, 'MXLIII'),
                     (1110, 'MCX'),
                     (1226, 'MCCXXVI'),
                     (1301, 'MCCCI'),
                     (1485, 'MCDLXXXV'),
                     (1509, 'MDIX'),
                     (1607, 'MDCVII'),
                     (1754, 'MDCCLIV'),
                     (1832, 'MDCCCXXXII'),
                     (1993, 'MCMXCIII'),
```

```
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))           ②

    def test_to_roman_known_values(self):          ③
        '''to_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman1.to_roman(integer)      ④
            self.assertEqual(numeral, result)      ⑤

if __name__ == '__main__':
    unittest.main()
```

① Um den ersten Testfall zu schreiben, müssen wir unsere Klasse von der Klasse `TestCase` des `unittest`-Moduls ableiten. Diese Klasse stellt viele nützliche Methoden bereit, die Sie in Ihrem Testfall nutzen können, um bestimmte Bedingungen zu prüfen.

② Dies ist eine Liste von Ganzzahl/Zahlzeichen-Paaren, die ich manuell geprüft habe. Sie enthält die niedrigsten zehn Zahlen, die höchste Zahl, jede Zahl die aus einem Zeichen in römischen Zahlen besteht und eine zufällige Auswahl anderer gültiger Zahlen. Sie müssen nicht jede mögliche Eingabe prüfen, doch Sie sollten versuchen, alle Grenzfälle zu testen.

③ Jeder Test erhält eine eigene Methode, die weder Parameter übernehmen, noch einen Wert zurückgeben darf. Wird die Methode normal – also ohne Auslösen

einer Ausnahme – beendet, so wird der Test als erfolgreich durchlaufen angesehen; löst die Methode eine Ausnahme aus, gilt der Test als fehlgeschlagen.

④ Hier rufen Sie die `to_roman()`-Funktion auf. (Die wurde zwar bisher noch nicht geschrieben, doch später wird diese Zeile sie aufrufen.) Beachten Sie, dass Sie nun die API der `to_roman()`-Funktion definiert haben: Sie übernimmt eine Ganzzahl (die umzuwandelnde Zahl) und gibt einen String zurück (die römische Zahl). Ist die API anders, wird der Test als fehlgeschlagen angesehen. Beachten Sie außerdem, dass Sie beim Aufruf von `to_roman()` keine Ausnahmen behandeln. Das ist so beabsichtigt. `to_roman()` sollte keine Ausnahme auslösen, wenn Sie sie mit gültigen Eingaben aufrufen, und die vorliegenden Eingabewerte sind alle gültig. Löst `to_roman()` eine Ausnahme aus, so gilt dieser Test als fehlgeschlagen.

⑤ Wenn wir davon ausgehen, dass die `to_roman()`-Funktion korrekt aufgerufen und erfolgreich durchlaufen wurde sowie einen Wert zurückgegeben hat, müssen wir nun prüfen, ob sie einen richtigen Wert zurückgegeben hat. Dies ist eine übliche Frage und die `TestCase`-Klasse stellt eine Methode bereit, `assertEqual`, mit der wir prüfen können, ob zwei Werte gleich sind. Stimmt das von `to_roman()` zurückgegebene Ergebnis (`result`) nicht mit dem erwarteten Wert (`numeral`) überein, löst `assertEqual` eine Ausnahme aus und der Test schlägt fehl. Stimmen beide Werte überein, tut `assertEqual` nichts. Stimmt jeder von `to_roman()` zurückgegebene Wert mit dem erwarteten Wert überein, löst `assertEqual` niemals eine Ausnahme aus und `test_to_roman_known_values` wird normal beendet. Damit hat `to_roman()` den Test erfolgreich durchlaufen.

Sobald Sie den Testfall erstellt haben, können Sie mit der Programmierung der `to_roman()`-Funktion beginnen. Zunächst sollten Sie sie als leere Funktion implementieren und sicherstellen, dass der Test fehlschlägt. Wird der Test vor dem Schreiben des Codes erfolgreich durchlaufen, bedeutet dies, dass Ihr Code überhaupt nicht getestet wird! Unit Testing ist wie ein Tanz: Tests führen, Code folgt. Schreiben Sie einen Test der fehlschlägt und programmieren Sie dann solange, bis er erfolgreich durchlaufen wird.

```
# roman1.py

def to_roman(n):
    '''convert integer to Roman numeral'''
    pass  ①
```

①Hier wollen wir nun die API der `to_roman()`-Funktion definieren, doch nicht die eigentliche Funktion schreiben. (Unser Test muss erst einmal fehlschlagen.) Wir nutzen `pass`, um nichts zu tun.

Führen Sie `romantest1.py` auf der Kommandozeile aus, um den Test zu starten. Rufen Sie das Skript mit der Option `-v` auf, erhalten Sie ausführlichere Ausgaben, so dass Sie genau sehen können was bei jedem Testfall geschieht. Mit etwas Glück sollte Ihre Ausgabe aussehen wie diese:

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)                ①
to_roman should give known result with known input ... FAIL      ②

======================================================================
FAIL: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest1.py", line 73, in test_to_roman_known_values
    self.assertEqual(numeral, result)
AssertionError: 'I' != None                                      ③

----------------------------------------------------------------------
Ran 1 test in 0.016s                                             ④

FAILED (failures=1)                                              ⑤
```

① Beim Ausführen des Skripts wird `unittest.main()` gestartet, was wiederum jeden Testfall startet. Jeder Testfall ist eine Methode innerhalb einer Klasse in `romantest.py`. Die Gliederung dieser Testfälle spielt keine Rolle; sie können alle eine einzelne Methode beinhalten oder Sie können eine Klasse haben, die mehrere Methoden enthält. Die einzige Voraussetzung ist die, dass jede Testklasse von `unittest.TestCase` erben muss.

② Für jeden Testfall gibt das `unittest`-Modul den `docstring` der Methode und das Ergebnis des Tests aus. Wie erwartet schlägt dieser Test fehl.

③ Für jeden fehlgeschlagenen Testfall zeigt `unittest` genaue Informationen darüber an, was zum Fehler geführt hat. Im vorliegenden Fall löste `assertEqual()` eine `AssertionError`-Ausnahme aus, da `to_roman(1)` eigentlich `'I'` zurückgeben sollte, dies aber nicht getan hat. (Da es keine explizite `return`-Anweisung gibt, hat die Funktion `None` zurückgegeben.)

④ Nach den Detailangaben jedes Tests zeigt `unittest` an, wie viele Tests ausgeführt wurden und wie viel Zeit sie beansprucht haben.

⑤ Der Testdurchlauf ist fehlgeschlagen, da mindestens ein Testfall nicht erfolgreich durchlaufen wurde. `unittest` unterscheidet dabei zwischen Fehlschlägen und Fehlern. Ein Fehlschlag ist der wegen einer nicht wahren Bedingung oder einer nicht ausgelösten Ausnahme fehlgeschlagene Aufruf einer `assertXYZ`-Methode, wie `assertEqual` oder `assertRaises`. Ein Fehler ist jede andere Art einer im getesteten Code oder im Unit-Testfall ausgelösten Ausnahme.

Nun können wir endlich die `to_roman()`-Funktion schreiben.

```
roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
```

```
                    ('L',  50),
                    ('XL', 40),
                    ('X',  10),
                    ('IX', 9),
                    ('V',  5),
                    ('IV', 4),
                    ('I',  1))                ①

def to_roman(n):
    '''convert integer to Roman numeral'''
    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:                   ②
            result += numeral
            n -= integer
    return result
```

① `roman_numeral_map` ist ein Tupel von Tupeln, das drei Dinge definiert: die Zeichendarstellung der grundlegendsten römischen Zahlen; die Reihenfolge der römischen Zahlen (in absteigender Reihenfolge von `M` hinunter zu `I`); der Wert jeder dieser römischen Zahlen. Jedes innere Tupel ist ein Paar aus `(Zahl, Wert)`. Nicht nur römische Zahlen aus einem Zeichen, sondern auch solche aus zwei Zeichen, wie `CM` („einhundert weniger als tausend“) werden angegeben. Das macht den Code der `to_roman()`-Funktion einfacher.

② Hier zahlt sich unsere Datenstruktur von `roman_numeral_map` aus, da wir keine besondere Logik für die Subtraktionsregel benötigen. Zum Konvertieren in römische Zahlen müssen wir einfach nur `roman_numeral_map` durchlaufen und nach dem größten Ganzzahlwert suchen, der kleiner oder gleich der Eingabe ist. Ist dieser Wert gefunden, fügen wir die römische Zahl zum Ende der Ausgabe hinzu, ziehen den entsprechenden Ganzzahlwert von der Eingabe ab und wiederholen den Vorgang solange, bis die Bedingung der `while`-Schleife nicht mehr erfüllt ist.

Sollte Ihnen immer noch nicht klar sein, wie die `to_roman()`-Funktion funktioniert, fügen Sie am Ende der `while`-Schleife eine `print()`-Anweisung ein:

```
while n >= integer:
    result += numeral
    n -= integer
    print('subtracting {0} from input, adding {1} to
output'.format(integer, numeral))
```

Nun sieht die Ausgabe so aus:

```
>>> import roman1
>>> roman1.to_roman(1424)
```

```
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

Die `to_roman()`-Funktion scheint – zumindest bei dieser manuellen Überprüfung – korrekt zu funktionieren. Doch wird auch unser Testfall erfolgreich durchlaufen?

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok    ①

----------------------------------------------------------------------
Ran 1 test in 0.016s

OK
```

① Hurra! Die `to_roman()`-Funktion durchläuft den Testfall mit *bekannten Werten* erfolgreich. Der Testfall ist nicht allumfassend, doch immerhin werden Eingaben überprüft, die alle römischen Zahlen aus einem Zeichen, den größtmöglichen Wert (`3999`) und die längste römische Zahl (`3888`) hervorbringen. Sie können sich an dieser Stelle ziemlich sicher sein, dass die Funktion mit allen guten Eingaben korrekt funktioniert.

„Gute“ Eingaben? Hmm. Was ist mit schlechten Eingaben?

10.3 Anhalten und Alarm schlagen

Es genügt natürlich nicht, zu testen, ob Funktionen korrekt funktionieren, wenn man ihnen gute Eingaben übergibt; Sie müssen auch testen, ob sie fehlschlagen, wenn man ihnen schlechte Eingaben übergibt. *Dazu* müssen sie noch in *der* Weise fehlschlagen, wie Sie es erwarten.

```
>>> import roman1
>>> roman1.to_roman(4000)
'MMMM'
>>> roman1.to_roman(5000)
'MMMMM'
>>> roman1.to_roman(9000)  ①
'MMMMMMMMM'
```

① Das ist bestimmt nicht das, was Sie wollten – es ist ja nicht mal eine gültige römische Zahl! Tatsächlich entspricht keine dieser Zahlen unserem Eingabebereich

(1–3999), und dennoch gibt die Funktion einen falschen Wert zurück. Heimlich, still und leise schlechte Werte zurückgeben ist schleeeeeecht! Schlägt ein Programm fehl, ist es besser, wenn das schnell und laut geschieht. „Anhalten und Alarm schlagen“ eben. Python erreicht dies durch das Auslösen einer Ausnahme.

Die Frage, die Sie sich nun stellen sollten, lautet: „Wie kann ich das als zu testende Anforderung ausdrücken?“ Wie wär’s hiermit:

- Die `to_roman()`-Funktion soll einen `OutOfRangeError` auslösen, wenn die übergebene Ganzzahl größer als `3999` ist.

Wie würde dieser Test aussehen?

```
class ToRomanBadInput(unittest.TestCase):                                    ①
    def test_too_large(self):                                                ②
        '''to_roman should fail with large input'''
        self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)     ③
```

① Wie beim vorherigen Testfall erstellen Sie auch hier eine Klasse, die von `unittest.TestCase` erbt. Eine Klasse kann zwar mehr als einen Test enthalten (wie Sie später in diesem Kapitel sehen werden), doch ich habe mich hier dazu entschieden, eine neue Klasse zu erstellen, da dieser Test anders als der letzte ist. Wir werden alle Tests für gute Eingaben in einer Klasse halten und alle Tests für schlechte Eingaben in einer anderen.

② Der Test selbst ist wieder eine Methode der Klasse, deren Name mit `test` beginnt.

③ Die Klasse `unittest.TestCase` stellt die Methode `assertRaises` bereit, welche die folgenden Argumente übernimmt: die erwartete Ausnahme, die zu testende Funktion und die Argumente, die der Funktion übergeben werden. (Übernimmt die zu testende Funktion mehr als ein Argument, übergeben Sie all diese in der korrekten Reihenfolge an `assertRaises`; sie werden dann an die zu testende Funktion weitergegeben.)

Sehen Sie sich die letzte Codezeile genau an. Anstatt `to_roman()` direkt aufzurufen und manuell zu prüfen, ob die Funktion eine Ausnahme auslöst (durch das Umschließen mit einem `try...except`-Block), macht `assertRaises` all das für uns. Sie müssen lediglich die zu erwartende Ausnahme (`roman2.OutOfRangeError`), die Funktion (`to_roman()`) und die Argumente der Funktion (`4000`) angeben. Die `assertRaises`-Methode sorgt für das Aufrufen von `to_roman()` und prüft, ob `roman2.OutOfRangeError` ausgelöst wird.

Achten Sie auch darauf, dass Sie `to_roman()` selbst als Argument übergeben. Sie rufen die Funktion nicht auf und übergeben auch nicht ihren Namen als String. Habe ich in der letzten Zeit darauf hingewiesen, wie praktisch es ist, dass alles in Python ein Objekt ist?

Was passiert also nun, wenn Sie unsere Testsuite mit diesem neuen Test starten?

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
```

```
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ERROR                    ①

======================================================================
ERROR: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AttributeError: 'module' object has no attribute 'OutOfRangeError'  ②

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (errors=1)
```

① Sie sollten mit diesem Fehlschlag gerechnet haben (da Sie bisher keinen Code zum erfolgreichen Durchlaufen geschrieben haben), doch … es ist kein „Fehlschlag“, sondern ein „Fehler“. Das ist ein kleiner, aber feiner Unterschied. Ein Unit-Test hat drei Rückgabewerte: `PASS`, `FAIL` und `ERROR`. `PASS` bedeutet natürlich, dass der Test erfolgreich durchlaufen wurde – der Code hat genau das getan, was Sie erwartet haben. `FAIL` ist das, was gerade eben passiert ist – der Code wurde ausgeführt, doch das Ergebnis war nicht das, was Sie erwartet haben. `ERROR` bedeutet, dass der Code nicht einmal korrekt ausgeführt wurde.

② Warum wurde der Code nicht korrekt ausgeführt? Die Traceback-Angaben verraten uns alles. Das von Ihnen getestete Modul besitzt keine Ausnahme namens `OutOfRangeError`. Sie haben der `assertRaises()`-Methode diese Ausnahme übergeben, weil Sie wollen, dass die Funktion diese Ausnahme bei einem zu großen Eingabewert auslöst. Die Ausnahme selbst existiert jedoch gar nicht, also schlug die `assertRaises()`-Methode fehl. Die `to_roman()`-Funktion konnte nicht getestet werden; so weit kam es gar nicht.

Zur Lösung dieses Problems müssen Sie die `OutOfRangeError`-Ausnahme in `roman2.py` definieren.

```
class OutOfRangeError(ValueError):        ①
    pass                                  ②
```

① Ausnahmen sind Klassen. Ein „out of range“-Fehler ist eine Art Wertefehler – der Wert des Arguments liegt außerhalb des zulässigen Bereichs. Diese Ausnahme erbt daher von der integrierten Ausnahme `ValueError`. Das ist nicht unbedingt nötig (sie könnte auch von der Klasse `Exception` erben), aber es fühlt sich richtig an.

② Ausnahmen tun eigentlich überhaupt nichts. Sie benötigen aber mindestens eine Codezeile zum Erstellen einer Klasse. `pass` tut nichts, ist jedoch eine Codezeile, also haben wir nun eine Klasse.

Starten Sie die Testsuite erneut.

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... FAIL          ①

======================================================================
FAIL: to_roman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AssertionError: OutOfRangeError not raised by to_roman  ②

----------------------------------------------------------------------
Ran 2 tests in 0.016s

FAILED (failures=1)
```

① Der neue Test wird immer noch nicht erfolgreich durchlaufen, doch es wird zumindest kein Fehler mehr zurückgegeben. Stattdessen schlägt der Test fehl. Das nenne ich Fortschritt! Das bedeutet nämlich, dass die `assertRaises()`-Methode diesmal erfolgreich aufgerufen wurde und unser Unit Test-Framework nun die `to_roman()`-Funktion getestet hat.

② Natürlich löst `to_roman()` nicht die `OutOfRangeError`-Ausnahme aus, da Sie ihr bisher nicht mitgeteilt haben, dass sie das tun soll. Das sind tolle Neuigkeiten! Es bedeutet, dass der Testfall funktioniert – vor dem Schreiben des Codes schlägt er fehl.

Nun können Sie *den* Code schreiben, der zum erfolgreichen Durchlaufen des Tests führt.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if n > 3999:
        raise OutOfRangeError('number out of range (must be less than 4000)')  ①

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

① Das ist recht simpel: Ist die übergebene Eingabe (`n`) größer als `3999`, wird eine `OutOfRangeError`-Ausnahme ausgelöst. Der Unit Test überprüft nicht den für Menschen lesbaren String, der unsere Ausnahme begleitet, auch wenn dies mit einem weiteren Test möglich wäre.

Wird unser Test nun erfolgreich durchlaufen? Finden wir es heraus!

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok      ①

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

① Hurra! Beide Tests waren erfolgreich. Da Sie immer wieder zwischen Testen und Programmieren hin- und hergesprungen sind, können Sie sicher sein, dass die beiden gerade geschriebenen Codezeilen für das erfolgreiche Durchlaufen des zweiten Tests gesorgt haben. Da steckt einiges an Arbeit hinter, die sich aber über die Lebensdauer Ihres Codes bezahlt macht.

10.4 Wieder anhalten und wieder Alarm

Neben zu großen Zahlen müssen Sie auch Zahlen testen, die zu klein sind. Wie wir am Anfang des Kapitels in unseren Anforderungen festgelegt haben, können römische Zahlen weder `0`, noch negative Zahlen ausdrücken.

```
>>> import roman2
>>> roman2.to_roman(0)
''
>>> roman2.to_roman(-1)
''
```

Das ist nicht gut. Lassen Sie uns nun Tests für jede dieser Bedingungen hinzufügen.

```
class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 4000)   ①

    def test_zero(self):
        '''to_roman should fail with 0 input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)      ②

    def test_negative(self):
        '''to_roman should fail with negative input'''
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)     ③
```

① Die `test_too_large()`-Methode hat sich nicht verändert. Ich gebe sie hier mit an, damit Sie sehen, wo der neue Code hingehört.

② Das ist ein neuer Test: die Methode `test_zero()`. Genau wie die `test_too_large()`-Methode teilt auch diese Methode der in `unittest.TestCase` definierten `assertRaises()`-Methode mit, dass sie `to_roman()` mit dem Parameter `0` aufrufen und prüfen soll, ob die passende Ausnahme, `OutOfRangeError`, ausgelöst wird.

③ Die Methode `test_negative()` ist dazu fast identisch, abgesehen davon, dass hier `-1` an die `to_roman()`-Funktion übergeben wird. Löst einer dieser neuen Tests keine `OutOfRangeError`-Ausnahme aus (entweder, weil die Funktion tatsächlich einen Wert zurückliefert, oder weil eine andere Ausnahme ausgelöst wird), wird der Test als fehlgeschlagen angesehen.

Prüfen Sie nun, ob die Tests fehlschlagen:

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... FAIL

======================================================================
FAIL: to_roman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 86, in test_negative
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman

======================================================================
FAIL: to_roman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest3.py", line 82, in test_zero
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=2)
```

Ausgezeichnet. Beide Tests sind wie erwartet fehlgeschlagen. Sehen wir uns nun den Code an, und was wir tun können, damit die Tests erfolgreich durchlaufen werden.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 4000):                                          ①
        raise OutOfRangeError('number out of range (must be 1..3999)')  ②

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

① Das ist eine schöne Abkürzung in Python: mehrere Vergleiche auf einmal. Der Code entspricht `if not ((0 < n) and (n < 4000))`, doch er ist viel lesbarer. Diese eine Codezeile sollte sowohl zu große Eingaben, als auch negative und die `0` abfangen.

② Wenn Sie Ihre Bedingungen ändern, sollten Sie sicherstellen, dass Sie Ihre Fehler-Strings ebenfalls anpassen. Das `unittest`-Framework schert sich nicht darum, doch manuelles Debugging wird erheblich erschwert, wenn Ihr Code fehlerhaft beschriebene Ausnahmen auslöst.

Ich könnte Ihnen jetzt an einer ganzen Reihe unzusammenhängender Beispiele zeigen, dass mehrere Vergleiche auf einmal funktionieren, doch stattdessen starte ich einfach den Unit Test und beweise es.

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.016s

OK
```

10.5 Noch eine Kleinigkeit

Es gab noch eine weitere Anforderung: mit Zahlen umgehen, die keine Ganzzahlen sind.

```
>>> import roman3
>>> roman3.to_roman(0.5)  ①
''
>>> roman3.to_roman(1.0)  ②
'I'
```

① Oh, das ist schlecht.

② Oh, das ist sogar noch schlechter. In beiden Fällen sollte eine Ausnahme ausgelöst werden. Doch stattdessen bekommen wir falsche Ergebnisse.

Eine Überprüfung auf *nichtganzzahlige* Zahlen ist nicht schwierig. Zunächst definieren wir eine `NotIntegerError`-Ausnahme.

```
# roman4.py
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
```

Dann schreiben wir einen Testfall, um die `NotIntegerError`-Ausnahme zu prüfen.

```
class ToRomanBadInput(unittest.TestCase):
    .
    .
    .
    def test_non_integer(self):
        '''to_roman should fail with non-integer input'''
        self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
```

Nun schauen wir, ob der Test auch fehlschlägt.

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

======================================================================
FAIL: to_roman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest4.py", line 90, in test_non_integer
    self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
AssertionError: NotIntegerError not raised by to_roman
```

```
----------------------------------------------------------------------
Ran 5 tests in 0.000s

FAILED (failures=1)
```

Schreiben wir den Code zum erfolgreichen Durchlaufen des Tests.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 4000):
        raise OutOfRangeError('number out of range (must be 1..3999)')
    if not isinstance(n, int):                                          ①
        raise NotIntegerError('non-integers can not be converted')      ②

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

① Die integrierte Funktion `isinstance()` prüft, ob eine Variable von einem bestimmten Datentyp ist.

② Ist das Argument `n` kein `int`, wird unsere neue `NotIntegerError`-Ausnahme ausgelöst.

Schließlich stellen wir sicher, dass der Code den Test tatsächlich erfolgreich durchläuft.

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Die `to_roman()`-Funktion durchläuft alle Tests erfolgreich. Weitere Tests fallen mir nicht mehr ein, also kümmern wir uns nun um `from_roman()`.

10.6 Eine erfreuliche Symmetrie

Einen String von einer römischen Zahl in eine Ganzzahl (einen Integer) umzuwandeln, klingt schwieriger, als eine Ganzzahl in eine römische Zahl umzuwandeln. Und tatsächlich besteht das Problem der Gültigkeitsprüfung. Es ist extrem einfach, zu prüfen, ob eine Ganzzahl größer als `0` ist; es ist aber wesentlich schwerer, zu prüfen, ob ein String eine gültige römische Zahl ist. Zum Glück haben wir aber in Kap. 6 einen regulären Ausdruck konstruiert, der genau dies tut. Das wäre also schon erledigt.

Wir stehen nun vor dem Problem, den String selbst umzuwandeln. Wie wir später sehen werden, ist es dank unserer Datenstruktur, die einzelnen römischen Zahlen Ganzzahlen zuordnet, recht einfach die `from_roman()`-Funktion zu erstellen.

Doch zuerst: die Tests. Wir benötigen einen Test für bekannte Werte, der die Genauigkeit stichprobenartig prüft. Unsere Testsuite enthält bereits eine Zuordnung bekannter Werte; nutzen wir sie.

```
    def test_from_roman_known_values(self):
        '''from_roman should give known result with known input'''
        for integer, numeral in self.known_values:
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

Hier gibt es eine erfreuliche Symmetrie. `from_roman()` ist das genaue Gegenstück zu `to_roman()`. Die eine Funktion wandelt Ganzzahlen in speziell formatierte Strings um, die andere konvertiert speziell formatierte Strings in Ganzzahlen. Theoretisch sollten wir `to_roman()` eine Zahl übergeben können, um einen String zu erhalten; diesen String sollten wie dann `from_roman()` übergeben können, um wieder dieselbe Zahl zu erhalten.

```
n = from_roman(to_roman(n)) for all values of n
```

In diesem Fall steht „all values“ (*alle Werte*) für eine beliebige Zahl zwischen `1` und `3999`, da dies der für die `to_roman()`-Funktion gültige Zahlenbereich ist. Wir können diese Symmetrie in einem Testfall ausdrücken, der alle Werte von `1` bis `3999` durchläuft, `to_roman()` aufruft, `from_roman()` aufruft und dann prüft, ob die Ausgabe der ursprünglichen Eingabe entspricht.

```
class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 4000):
            numeral = roman5.to_roman(integer)
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

Diese neuen Tests werden nicht fehlschlagen. Da wir bisher keine `from_roman()` -Funktion definiert haben, treten nur Fehler auf.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
E.E....
======================================================================
ERROR: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 78, in test_from_roman_known_values
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

======================================================================
ERROR: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 103, in test_roundtrip
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'

----------------------------------------------------------------------
Ran 7 tests in 0.019s

FAILED (errors=2)
```

Eine kurze Funktionsskizze löst dieses Problem.

```
# roman5.py
def from_roman(s):
    '''convert Roman numeral to integer'''
```

(Haben Sie es bemerkt? Ich habe eine Funktion definiert, die nichts als einen `doc-string` enthält. Das ist in Python möglich. Manche Programmierer schwören darauf: „Skizziere nicht; dokumentiere!“)

Nun schlagen die Testfälle fehl.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
F.F....
======================================================================
FAIL: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 79, in test_from_roman_known_values
    self.assertEqual(integer, result)
AssertionError: 1 != None

======================================================================
FAIL: test roundtrip (  main  .RoundtripCheck)
from roman(to roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest5.py", line 104, in test roundtrip
    self.assertEqual(integer, result)
AssertionError: 1 != None

----------------------------------------------------------------------
Ran 7 tests in 0.002s

FAILED (failures=2)
```

Es ist an der Zeit die `from_roman()`-Funktion zu schreiben.

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:  ①
            result += integer
            index += len(numeral)
    return result
```

① Das Muster ist dasselbe wie bei der `to_roman()`-Funktion. Wir durchlaufen die Datenstruktur der römischen Zahlen (ein Tupel aus Tupeln).

Sollte es sich Ihnen nicht erschließen, wie `from_roman()` funktioniert, fügen Sie am Ende der `while`-Schleife eine `print`-Anweisung ein:

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print('found', numeral, 'of length', len(numeral), ', adding', integer)
```

Führen Sie die Funktion nun aus.

```
>>> import roman5
>>> roman5.from_roman('MCMLXXII')
found M of length 1, adding 1000
found CM of length 2, adding 900
found L of length 1, adding 50
found X of length 1, adding 10
found X of length 1, adding 10
found I of length 1, adding 1
found I of length 1, adding 1
1972
```

Starten wir die Tests erneut.

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
.......
----------------------------------------------------------------------
------
Ran 7 tests in 0.060s

OK
```

Hier gibt es interessante Neuigkeiten. Die `from_roman()`-Funktion funktioniert, zumindest für alle bekannten Werte, mit guten Eingaben. Außerdem wurde auch unser neuer Test erfolgreich durchlaufen. Im Zusammenspiel mit den Tests der bekannten Werte können Sie recht sicher sein, dass sowohl `to_roman()`, als auch `from_roman()` mit allen guten Werten korrekt funktionieren. (Das ist keine Garantie; es ist theoretisch möglich, dass `to_roman()` einen Bug hat, der für bestimmte Eingaben die falschen römischen Zahlen liefert, und dass `from_roman()` einen wechselseitigen Bug hat, der die falschen Ganzzahlen für genau diese römischen Zahlen produziert. Je nach Ihrer Anwendung und Ihren Anforderungen könnte das ein Problem sein; sollte dem so sein, schreiben Sie umfassendere Testfälle.)

10.7 Noch mehr schlechte Eingaben

Die `from_roman()`-Funktion funktioniert nun also mit guten Eingaben korrekt. Jetzt ist es an der Zeit, auch das letzte Puzzleteil einzufügen: `from_roman()` muss mit schlechten Eingaben umgehen können. Wir müssen also eine Möglichkeit finden, zu bestimmen, ob ein String eine gültige römische Zahl ist. Dies ist grundsätzlich schwieriger als die Gültigkeitsprüfung numerischer Eingaben in der `to_roman()`-Funktion, doch wir haben ein mächtiges Werkzeug zur Hand: reguläre Ausdrücke.

Wie Sie bereits bei unserem Fallbeispiel: Römische Zahlen (*Kap. 6*) gesehen haben, gibt es einige einfache Regeln zum Aufbau römischer Zahlen, die die Buchstaben M, D, C, L, X, V und I verwenden. Sehen wir uns diese Regeln noch einmal an:

- Manchmal werden die Zeichen zusammengezählt. I ist 1, II ist 2 und III ist 3. VI ist 6 („5 und 1“), VII ist 7 und VIII ist 8.
- Die Zehner-Zeichen (I, X, C und M) können bis zu drei Mal wiederholt werden. Bei 4 müssen Sie vom nächsthöheren Fünfer-Zeichen subtrahieren. Sie können 4 nicht als IIII darstellen, sondern nur als IV („1 weniger als 5“). Die Zahl 40 wird als XL geschrieben („10 weniger als 50“), 41 als XLI, 42 als XLII, 43 als XLIII und schließlich 44 als XLIV („10 weniger als 50, und dann 1 weniger als 5“).
- Manchmal müssen Zeichen subtrahiert werden. So müssen Sie z. B. bei 9 verfahren. Hier müssen Sie vom nächsthöheren Zehner-Zeichen subtrahieren: 8 ist VIII, doch 9 ist IX („1 weniger als 10“), nicht VIIII (da das Zeichen I nicht vier Mal wiederholt werden darf). Die Zahl 90 wird somit als XC dargestellt und 900 als CM.
- Die Fünfer-Zeichen dürfen niemals wiederholt werden. Die Zahl 10 wird immer als X dargestellt, niemals als VV. Die Zahl 100 ist immer C, niemals LL.
- Römische Zahlen werden von links nach rechts gelesen. Die Reihenfolge der Zeichen spielt daher eine sehr wichtige Rolle. DC ist 600; CD ist eine völlig andere Zahl (400, „100 weniger als 500“). CI ist 101; IC ist nicht einmal eine gültige römische Zahl (man kann 1 nicht direkt von 100 subtrahieren, sondern müsste XCIX schreiben, „10 weniger als 100, und dann 1 weniger als 10“).

Ein nützlicher Test würde daher sicherstellen, dass die `from_roman()`-Funktion fehlschlägt, wenn Sie Ihr Strings mit zu vielen wiederholten Zeichen übergeben. Wie viel „zu viel“ ist, hängt von der Zahl ab.

```
class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Ein weiterer sinnvoller Test wäre, zu prüfen, dass bestimmte Muster nicht wiederholt vorkommen. IX ist beispielsweise 9, doch IXIX ist niemals gültig.

```
  def test_repeated_pairs(self):
        '''from_roman should fail with repeated pairs of numerals'''
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Ein dritter Test könnte sicherstellen, dass die Zahlen in der korrekten Reihenfolge – vom größten zum kleinsten Wert – erscheinen. So ist z. B. CL 150, während LC niemals gültig ist, da das Zeichen für 50 niemals vor dem Zeichen für 100 stehen darf. Dieser Test enthält eine zufällige Auswahl an ungültigen Zeichenfolgen: I vor M, V vor C und so weiter.

```
    def test_malformed_antecedents(self):
        '''from_roman should fail with malformed antecedents'''
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Jeder dieser Tests setzt voraus, dass die `from_roman()`-Funktion eine neue Ausnahme auslöst, `InvalidRomanNumeralError`, die wir bisher noch nicht definiert haben.

```
# roman6.py
class InvalidRomanNumeralError(ValueError): pass
```

All diese drei Tests sollten fehlschlagen, da die `from_roman()`-Funktion noch keine Gültigkeitsprüfung enthält. (Wenn Sie jetzt nicht fehlschlagen, was zum Teufel testen sie dann?)

```
you@localhost:~/diveintopython3/examples$ python3 romantest6.py
FFF.......
======================================================================
FAIL: test_malformed_antecedents (__main__.FromRomanBadInput)
from_roman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 113, in test_malformed_antecedents
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman,
s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_repeated_pairs (__main__.FromRomanBadInput)
from_roman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 107, in test_repeated_pairs
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman,
s)
AssertionError: InvalidRomanNumeralError not raised by from_roman

======================================================================
FAIL: test_too_many_repeated_numerals (__main__.FromRomanBadInput)
from_roman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest6.py", line 102, in test_too_many_repeated_numerals
    self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman,
s)
AssertionError: InvalidRomanNumeralError not raised by from_roman
```

```
----------------------------------------------------------------------
Ran 10 tests in 0.058s

FAILED (failures=3)
```

Schön. Nun müssen wir lediglich noch den regulären Ausdruck zur Gültigkeitsprüfung römischer Zahlen zu `from_roman()` hinzufügen.

```
roman_numeral_pattern = re.compile('''
    ^                   # beginning of string
    M{0,3}              # thousands - 0 to 3 Ms
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    ''', re.VERBOSE)


def from_roman(s):
    '''convert Roman numeral to integer'''
    if not roman_numeral_pattern.search(s):
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))

    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index : index + len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

Und die Tests erneut laufen lassen …

```
you@localhost:~/diveintopython3/examples$ python3 romantest7.py
..........
----------------------------------------------------------------------
------
Ran 10 tests in 0.066s

OK
```

Und der Preis für den enttäuschendsten Abschluss des Jahres geht an … das Wörtchen „`OK`“, das vom `unittest`-Modul angezeigt wird, wenn alle Tests erfolgreich durchlaufen wurden.

Kapitel 11
Refactoring

11.1 Los geht's

Auch wenn Ihre Unit Tests noch so umfangreich sind: Bugs kommen vor. Was meine ich mit „Bug"? Ein Bug ist ein Testfall, den Sie noch nicht geschrieben haben.

```
>>> import roman7
>>> roman7.from_roman('') ①
0
```

① Das ist ein Bug. Ein leerer String sollte eine `InvalidRomanNumeralError`-Ausnahme auslösen, genau wie jede andere Zeichenfolge, die keine gültige römische Zahl darstellt.

Nach dem Reproduzieren des Bugs, aber vor dessen Beheben, sollten Sie einen Testfall schreiben, der fehlschlägt, um den Bug aufzuzeigen.

```
class FromRomanBadInput(unittest.TestCase):
    .
    .
    .
    def testBlank(self):
        '''from_roman should fail with blank string'''
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, '') ①
```

① Das ist ziemlich einfach. Rufen Sie `from_roman()` mit einem leeren String auf und sorgen Sie dafür, dass dies eine `InvalidRomanNumeralError`-Ausnahme auslöst. Das Auffinden des Bugs war der schwierige Teil; den Code danach zu testen ist der einfache Teil.

Da Ihr Code einen Bug enthält und Sie nun einen Testfall haben, der diesen Bug testet, schlägt der Testfall fehl.

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_11, © Springer-Verlag Berlin Heidelberg 2010

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... FAIL
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
FAIL: from_roman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest8.py", line 117, in test_blank
    self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman,
'')
AssertionError: InvalidRomanNumeralError not raised by from_roman

----------------------------------------------------------------------
Ran 11 tests in 0.171s

FAILED (failures=1)
```

Jetzt können Sie den Bug beheben.

```
def from_roman(s):
    '''convert Roman numeral to integer'''
    if not s:    ①

        raise InvalidRomanNumeralError('Input can not be blank')
    if not re.search(romanNumeralPattern, s):
            raise   InvalidRomanNumeralError('Invalid   Roman   numeral:
{}'.format(s))  ②

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

① Es werden gerade einmal zwei Codezeilen benötigt: eine ausdrückliche Überprüfung auf einen leeren String und eine `raise`-Anweisung.

② Ich glaube, ich habe es in diesem Buch bisher nicht erwähnt, daher hier nun Ihre letzte Lektion zur Stringformatierung. Ab Python 3.1 ist es möglich die Zahlen bei den Indizes einer Formatangabe wegzulassen. Sie können also den ersten Parameter der `format()`-Methode statt mit `{0}` auch einfach mit `{}` ansprechen; Python setzt den korrekten Index für Sie ein. Die Anzahl der Argumente ist dabei unerheblich: das erste `{}` wird zu `{0}`, das zweite `{}` wird zu `{1}` usw.

```
you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... ok  ①
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 11 tests in 0.156s

OK  ②
```

① Der neue Testfall wird nun erfolgreich durchlaufen, der Bug ist also behoben.

② Alle anderen Testfälle werden auch weiterhin erfolgreich durchlaufen. Das Beheben des Bugs hat somit nicht zu Problemen beim Rest des Codes geführt.

Das Programmieren auf diese Art macht das Beheben von Bugs nicht einfacher. Simple Bugs (wie der vorliegende) erfordern simple Testfälle; komplexe Bugs erfordern komplexe Testfälle. Es scheint in einer Umgebung mit vielen Tests länger zu dauern, einen Bug zu beheben, da Sie innerhalb des Codes genau angeben müssen, wo das Problem liegt (um den Testfall zu schreiben) und dieses dann beheben müssen. Wird der Testfall dann nicht sofort erfolgreich durchlaufen, müssen Sie herausfinden, ob die Behebung falsch ist, oder ob der Testfall selbst einen Bug enthält. Doch trotzdem zahlt sich dieses Vorgehen, das Hin- und Herspringen zwischen Testcode und getestetem Code, langfristig aus, weil es so wahrscheinlicher ist, dass Bugs bereits beim ersten Versuch korrekt behoben werden können. Da Sie außerdem alle Tests, inklusive dem neuen, erneut laufen lassen können, ist es unwahrscheinlicher, dass Sie beim Fehlerbereinigen des neuen Codes Probleme im alten Code verursachen. Der Unit Test von heute ist der Regressionstest von morgen. (*Regressionstest*: Wiederholung von Testfällen, um Seiteneffekte bereits getesteter Codeteile zu finden; Anm. d. Übers.)

11.2 Mit sich ändernden Anforderungen umgehen

Auch wenn Sie sich noch so sehr anstrengen und alles unternehmen, um die Anforderungen der Benutzer Ihres Programms von Beginn an genau zu bestimmen, werden sich diese Anforderungen zweifellos ändern. Die meisten Benutzer wissen nicht was sie wollen, bis sie es sehen. Doch selbst wenn sie es wissen, können sie Ihnen dies meist nicht so genau mitteilen, dass es für Sie nützlich wäre. Und selbst wenn sie es können, wollen sie ohnehin in der nächsten Version neue Features sehen. Seien Sie also dazu bereit, Ihre Testfälle an die sich ändernden Anforderungen anzupassen.

Nehmen wir z. B. an, dass Sie den Zahlenbereich der Funktionen zur Umwandlung in römische Zahlen erweitern wollten. Normalerweise kann kein Zeichen mehr als dreimal hintereinander wiederholt werden. Die Römer ließen jedoch eine Ausnahme zu: man kann vier `M`-Zeichen hintereinander verwenden, um `4000` darzustellen. Führen Sie diese Änderung durch, können Sie den Zahlenbereich von `1..3999` zu `1..4999` erweitern. Doch zunächst müssen Sie einige Änderungen an Ihren Testfällen vornehmen.

```
class KnownValues(unittest.TestCase):
    known_values = ( (1, 'I'),
                     .
                     .
                     .
                    (3999, 'MMMCMXCIX'),
                    (4000, 'MMMM'),                                    ①
                    (4500, 'MMMMD'),
                    (4888, 'MMMMDCCCLXXXVIII'),
                    (4999, 'MMMMCMXCIX') )

class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        '''to_roman should fail with large input'''
        self.assertRaises(roman8.OutOfRangeError, roman8.to_roman, 5000)  ②

    .
    .
    .

class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):     ③
            self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_roman, s)

    .
    .
    .
```

```
class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        '''from_roman(to_roman(n))==n for all n'''
        for integer in range(1, 5000):                                    ④
            numeral = roman8.to_roman(integer)
            result = roman8.from_roman(numeral)
            self.assertEqual(integer, result)
```

① Die bereits bestehenden bekannten Werte ändern sich nicht (es sind nach wie vor vernünftige Testwerte), doch Sie müssen ein paar hinzufügen. Ich habe hier folgende Werte eingefügt: `4000` (der kürzeste), `4500` (der zweitkürzeste), `4888` (der längste) und `4999` (der größte).

② Die Definition der „zu großen Eingabe“ hat sich geändert. Dieser Test rief `to_roman()` vormals mit dem Wert `4000` auf und erwartete einen Fehler; da `4000-4999` nun gute Werte darstellen, müssen Sie hier `5000` einsetzen.

③ Auch die Definition von „zu vielen wiederholten Zeichen“ hat sich geändert. Dieser Test rief `to_roman()` vormals mit `'MMMM'` auf und erwartete einen Fehler; nun ist `MMMM` jedoch eine gültige römische Zahl und wir müssen den angegebenen Wert auf `'MMMMM'` erhöhen.

④ Der Funktionstest durchläuft jede Zahl des Zahlenbereichs, von `1` bis `3999`. Da sich der Zahlenbereich nun erweitert hat, muss die `for`-Schleife aktualisiert werden und alle Zahlen von `1` bis `4999` durchlaufen.

Jetzt sind Ihre Testfälle auf dem aktuellen Stand, doch Ihr Code ist es nicht. Einige Testfälle sollten daher fehlschlagen.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ERROR          ①
to_roman should give known result with known input ... ERROR            ②
from_roman(to_roman(n))==n for all n ... ERROR                          ③
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

======================================================================
ERROR: from_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 82, in test_from_roman_known_values
    result = roman9.from_roman(numeral)
  File "C:\home\diveintopython3\examples\roman9.py", line 60, in from_roman
```

```
    raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
roman9.InvalidRomanNumeralError: Invalid Roman numeral: MMMM

======================================================================
ERROR: to_roman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 76, in test_to_roman_known_values
    result = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)

======================================================================
ERROR: from_roman(to_roman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "romantest9.py", line 131, in testSanity
    numeral = roman9.to_roman(integer)
  File "C:\home\diveintopython3\examples\roman9.py", line 42, in to_roman
    raise OutOfRangeError('number out of range (must be 0..3999)')
roman9.OutOfRangeError: number out of range (must be 0..3999)

----------------------------------------------------------------------
Ran 12 tests in 0.171s

FAILED (errors=3)
```

① Der Test der bekannten Werte von `from_roman()` schlägt fehl, sobald er auf `'MMMM'` trifft, da `from_roman()` immer noch davon ausgeht, dass es sich dabei um eine ungültige römische Zahl handelt.

② Der Test der bekannten Werte von `to_roman()` schlägt fehl, sobald er auf `4000` trifft, da `to_roman()` immer noch glaubt, `4000` sei außerhalb des gültigen Zahlenbereichs.

③ Auch dieser Test schlägt beim Treffen auf `4000` fehl, da `to_roman()` nach wie vor davon ausgeht, dass der Wert außerhalb des gültigen Zahlenbereichs liegt.

Jetzt, wo Sie basierend auf den neuen Anforderungen fehlschlagende Testfälle haben, können Sie sich Gedanken über die Änderung des Codes machen, um ihn in Einklang mit den Testfällen zu bringen. (Es ist zu Beginn des Schreibens von Unit Tests schwer zu begreifen, dass der zu testende Code den Testfällen niemals „voraus“ ist. Solange er hinterherhinkt haben Sie immer noch Arbeit vor sich. Hat der Code die Testfälle dann irgendwann eingeholt, hören Sie auf zu programmieren. Haben Sie sich erst einmal daran gewöhnt, werden Sie sich fragen, wie Sie jemals ohne Unit Tests auskommen konnten.)

```
roman_numeral_pattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 Ms  ①
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),
                        #            or 500-800 (D, followed by 0 to 3 Cs)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),
                        #        or 50-80 (L, followed by 0 to 3 Xs)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),
                        #        or 5-8 (V, followed by 0 to 3 Is)
    $                   # end of string
    ''', re.VERBOSE)


def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):                        ②
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if not isinstance(n, int):
        raise NotIntegerError('non-integers can not be converted')

    result = ''
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result


def from_roman(s):
    .
    .
    .
```

① An der `from_roman()`-Funktion müssen Sie überhaupt nichts ändern. Sie müssen lediglich eine kleine Änderung an `roman_numeral_pattern` durchführen. Wenn Sie genau hinsehen, werden Sie feststellen, dass ich die maximale Zahl an `M`-Zeichen von `3` auf `4` erhöht habe. Damit erlauben wir nun `4999` statt `3999`. Die `from_roman()`-Funktion selbst ist vollständig generisch; sie schaut nur nach sich wiederholenden Zeichen und addiert diese, ohne sich jedoch um die Anzahl der Wiederholungen zu kümmern. Mit `'MMMM'` konnte die Funktion vorher nur nicht umgehen, weil Sie sie durch den regulären Ausdruck daran gehindert haben.

② Die `to_roman()`-Funktion benötigt lediglich eine kleine Änderung in der Bereichsüberprüfung. Wo Sie vorher `0 < n < 4000` überprüft haben, prüfen Sie nun `0 < n < 5000`. Außerdem müssen Sie die Fehlermitteilung ändern, damit sie den neuen Zahlenbereich angibt (`1..4999` statt `1..3999`). Sonst müssen Sie keinerlei Änderungen an der Funktion vornehmen.

Sie glauben vielleicht nicht, dass diese zwei kleinen Änderungen alles sind. Sie müssen mir nicht blind vertrauen. Sehen Sie selbst.

```
you@localhost:~/diveintopython3/examples$ python3 romantest9.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.203s

OK    ①
```

① Alle Testfälle werden erfolgreich durchlaufen.

Umfangreiche Unit Tests bedeuten auch, dass Sie sich niemals auf einen Programmierer verlassen müssen, der sagt: „Vertrauen Sie mir“.

11.3 Refactoring

Das Beste an umfassenden Unit Tests ist nicht das gute Gefühl, das Sie haben, wenn schließlich alle Testfälle erfolgreich waren; es ist auch nicht das Gefühl, das Sie haben, wenn jemand Sie beschuldigt, seinen Code zerstört zu haben und Sie beweisen können, dass dem nicht so ist. Das Beste an Unit Tests ist, dass sie gnadenloses Refactoring betreiben können.

Refactoring bedeutet, dass Sie funktionierenden Code verwenden und diesen noch besser machen. „Besser“ heißt meistens „schneller“, kann aber auch „speicherschonender“, „festplattenschonender“ oder einfach „schöner“ bedeuten. Was auch immer es für Sie, Ihr Projekt, in Ihrer Umgebung bedeutet: Refactoring ist wichtig für die langfristige Funktionstüchtigkeit eines jeden Programms.

In unserem Fall bedeutet „besser“ sowohl „schneller“ als auch „einfacher zu pflegen“. Vor allem die `from_roman()`-Funktion ist aufgrund des riesigen regulären Ausdrucks langsamer und komplexer als ich es gerne hätte. Nun könnten Sie denken: „Okay, der reguläre Ausdruck ist lang und grässlich, aber wie sonst sollte ich prüfen, ob ein beliebiger String eine gültige römische Zahl ist?“

Ich antworte darauf: Es gibt lediglich 5.000 von ihnen; warum erstellen Sie keine Lookup-Tabelle? Diese Idee erscheint sogar noch besser, wenn Sie bedenken, dass Sie überhaupt keine regulären Ausdrücke verwenden müssen. Wenn Sie die Lookup-Tabelle zur Umwandlung von Ganzzahlen in römische Zahlen anlegen, können Sie auch gleichzeitig die umgekehrte Lookup-Tabelle zur Umwandlung römischer Zahlen in Ganzzahlen erstellen. Zum Zeitpunkt des Überprüfens, ob ein String eine gültige römische Zahl ist, haben Sie alle gültigen römischen Zahlen gesammelt. Die „Gültigkeitsprüfung“ wird auf ein einziges Nachschlagen in einem Dictionary reduziert.

Das Beste daran: Sie haben bereits einen kompletten Satz an Unit Tests. Sie können die Hälfte des Modulcodes ändern; die Unit Tests bleiben gleich. Sie können also – sich selbst und anderen – beweisen, dass der neue Code genauso gut funktioniert wie der alte.

```
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
class InvalidRomanNumeralError(ValueError): pass

roman_numeral_map = (('M',  1000),
                     ('CM', 900),
                     ('D',  500),
                     ('CD', 400),
                     ('C',  100),
                     ('XC', 90),
                     ('L',  50),
                     ('XL', 40),
                     ('X',  10),
                     ('IX', 9),
                     ('V',  5),
                     ('IV', 4),
                     ('I',  1))

to_roman_table = [ None ]
from_roman_table = {}

def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]

def from_roman(s):
    '''convert Roman numeral to integer'''
```

```
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]

def build_lookup_tables():
    def to_roman(n):
        result = ''
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)
        to_roman_table.append(roman_numeral)
        from_roman_table[roman_numeral] = integer

build_lookup_tables()
```

Sehen wir uns diesen Code in kleinen Häppchen an. Die wohl wichtigste Zeile ist die letzte:

```
build_lookup_tables()
```

Dies ist, wie unschwer zu erkennen, ein Funktionsaufruf. Er ist jedoch nicht von einer `if`-Anweisung umgeben. Dies ist kein `if __name__ == '__main__'` -Block; die Funktion wird beim Importieren des Moduls aufgerufen. (Module werden nur einmal importiert. Dabei werden Sie zwischengespeichert. Importieren Sie also ein bereits importiertes Modul erneut, passiert nichts. Dieser Code wird also nur beim ersten Importieren aufgerufen.)

Was macht die Funktion `build_lookup_tables()` denn nun? Gut, dass Sie fragen.

```
to_roman_table = [ None ]
from_roman_table = {}
.
.
.
def build_lookup_tables():
    def to_roman(n):                                    ①
        result = ''
```

```
        for numeral, integer in roman_numeral_map:
            if n >= integer:
                result = numeral
                n -= integer
                break
        if n > 0:
            result += to_roman_table[n]
        return result

    for integer in range(1, 5000):
        roman_numeral = to_roman(integer)          ②
        to_roman_table.append(roman_numeral)       ③
        from_roman_table[roman_numeral] = integer
```

① Das ist cleveres Programmieren … vielleicht zu clever. Die `to_roman()`-Funktion wird weiter oben definiert; sie schlägt Werte in der Lookup-Tabelle nach und gibt diese zurück. Die `build_lookup_tables()`-Funktion redefiniert die `to_roman()`-Funktion, so dass Sie nun etwas tut (wie in den vorherigen Beispielen, bevor wir die Lookup-Tabelle hinzugefügt haben). Innerhalb der `build_lookup_tables()`-Funktion führt der Aufruf von `to_roman()` zur Ausführung der redefinierten Version. Wird die `build_lookup_tables()`-Funktion verlassen, verschwindet die redefinierte Version – sie ist nur im lokalen Sichtbarkeitsbereich der `build_lookup_tables()`-Funktion definiert.

② Diese Codezeile ruft die redefinierte `to_roman()`-Funktion auf, die dann die römische Zahl berechnet.

③ Ist das Ergebnis berechnet, wird die Ganzzahl und die entsprechende römische Zahl zu beiden Lookup-Tabellen hinzugefügt.

Sind die Lookup-Tabellen einmal erstellt, ist der Rest des Codes einfach und schnell.

```
def to_roman(n):
    '''convert integer to Roman numeral'''
    if not (0 < n < 5000):
        raise OutOfRangeError('number out of range (must be 1..4999)')
    if int(n) != n:
        raise NotIntegerError('non-integers can not be converted')
    return to_roman_table[n]        ①


def from_roman(s):
    '''convert Roman numeral to integer'''
    if not isinstance(s, str):
        raise InvalidRomanNumeralError('Input must be a string')
    if not s:
        raise InvalidRomanNumeralError('Input can not be blank')
    if s not in from_roman_table:
        raise InvalidRomanNumeralError('Invalid Roman numeral: {0}'.format(s))
    return from_roman_table[s]      ②
```

① Nachdem wir die schon bekannten Begrenzungen überprüft haben, sucht die `to_roman()`-Funktion in der Lookup-Tabelle nach dem passenden Wert und gibt diesen zurück.

② Auf die gleiche Weise ist auch die `from_roman()`-Funktion nun auf das Überprüfen einiger Begrenzungen und eine Codezeile reduziert. Keine regulären Ausdrücke mehr. Keine Schleifen mehr. Einfacher geht es nicht.

Aber funktioniert es auch? Ja, ja das tut es. Ich kann es beweisen.

```
you@localhost:~/diveintopython3/examples$ python3 romantest10.py -v
from_roman should fail with blank string ... ok
from_roman should fail with malformed antecedents ... ok
from_roman should fail with non-string input ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.031s

OK    ①
```

Nicht dass Sie gefragt hätten, aber der Code ist dazu noch schnell! Fast zehnmal so schnell wie vorher. Natürlich benötigt diese Version länger zum Importieren, doch da der Importvorgang lediglich einmal durchgeführt wird, wird die erhöhte Startzeit durch all die Aufrufe von `to_roman()` und `from_roman()` amortisiert. Da die Tests einige Tausend Funktionsaufrufe ausführen, machen sich die Einsparungen schnell bemerkbar.

Die Moral von der Geschichte'?

- Einfachheit ist eine Tugend.
- Besonders dann, wenn reguläre Ausdrücke im Spiel sind.
- Unit Tests ermöglichen Ihnen umfassendes Refactoring.

11.4 Zusammenfassung

Unit Testing ist, korrekt implementiert, ein mächtiges Konzept, das bei einem langfristigen Projekt die Wartungskosten reduzieren und die Flexibilität erhöhen kann. Unit Testing ist allerdings kein magischer Problemlöser. Gute Testfälle zu schreiben

ist harte Arbeit. Diese auch noch aktuell zu halten erfordert außerdem viel Disziplin (vor allem dann, wenn die Benutzer der Software nach Fehlerbereinigungen schreien). Unit Testing ist kein Ersatz für andere Testformen, wie Funktionstests, Integrierungstests und Tests zur Akzeptanz der Benutzer. Doch Unit Testing ist möglich und funktioniert, und wenn Sie es einmal in Aktion gesehen haben, werden Sie nie mehr ohne es programmieren wollen.

Diese wenigen Kapitel haben Ihnen Sie in die Grundlagen eingeführt. Vieles davon war nicht einmal Python-spezifisch. Unit Testing-Frameworks sind für viele Programmiersprachen verfügbar und die Verwendung all dieser setzt Ihr Verständnis der folgenden Grundkonzepte voraus:

- Entwicklung von Testfällen, die spezifisch, automatisiert und unabhängig sind
- Schreiben der Testfälle vor dem Schreiben des zu testenden Codes
- Schreiben von Testfällen die gute Eingaben testen und auf korrekte Ergebnisse prüfen
- Schreiben von Testfällen die schlechte Eingaben testen und auf korrekte Fehlerangaben prüfen
- Schreiben und aktualisieren der Testfälle zur Anpassung an neue Anforderungen
- Gnadenloses Refactoring zur Verbesserung der Performance, Skalierbarkeit, Lesbarkeit, Wartungsfähigkeit oder jeglicher anderer -keit, die Sie sich vorstellen können.

Kapitel 12
Dateien

12.1 Los geht's

Mein Windows-Laptop hatte 38.493 Dateien auf der Festplatte, bevor ich Python 3 darauf installierte. Die Installation von Python 3 hat beinahe 3.000 Dateien hinzugefügt. Dateien sind die wichtigste Speichermöglichkeit jedes großen Betriebssystems; dieses Konzept ist so tief in uns verwurzelt, dass die meisten Menschen Probleme haben, sich eine Alternative vorzustellen. Ihr Computer erstickt geradezu an Dateien.

12.2 Aus Textdateien lesen

Bevor Sie aus einer Datei lesen können, müssen Sie diese erstmal öffnen. Das Öffnen von Dateien in Python könnte nicht einfacher sein:

```
a_file = open('examples/chinese.txt', encoding='utf-8')
```

Pythons integrierte Funktion `open()` übernimmt einen Dateinamen als Argument. Hier lautet der Dateiname `'examples/chinese.txt'`. Fünf Dinge sind an diesem Dateinamen interessant:

1. Es ist nicht nur ein Dateiname; es ist eine Kombination aus Verzeichnispfad und Dateiname. Theoretisch könnte eine Funktion zum Öffnen einer Datei auch zwei Argumente übernehmen – einen Verzeichnispfad und einen Dateinamen – doch die `open()`-Funktion übernimmt nur eins. Wenn Sie in Python einen „Dateinamen" verwenden, können Sie Teile eines Verzeichnispfads oder den kompletten Pfad mit angeben.
2. Im Verzeichnispfad kommen normale Schrägstriche (Slashes) zur Anwendung, obwohl ich nicht gesagt habe, welches Betriebssystem verwendet wird. Windows verwendet umgekehrte Schrägstriche (Backslashes) zur Angabe von Unterverzeichnissen, während Mac OS X und Linux normale Slashes benutzen.

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_12, © Springer-Verlag Berlin Heidelberg 2010

In Python macht das keinen Unterschied; normale Slashes funktionieren selbst unter Windows.
3. Der Verzeichnispfad beginnt nicht mit einem Slash oder einem Laufwerksbuchstaben; es ist daher also ein relativer Pfad. Relativ zu was? Geduld, junger Padawan.
4. Es ist ein String. Alle modernen Betriebssysteme (sogar Windows!) verwenden Unicode zur Speicherung von Dateinamen und Verzeichnissen. Python 3 unterstützt Nicht-ASCII-Pfadnamen vollständig.
5. Die Datei muss nicht auf Ihrer lokalen Festplatte liegen. Sie könnten auch ein Netzwerklaufwerk eingehängt haben. Die „Datei“ könnte sogar Teil eines vollständig virtuellen Dateisystems sein. Erkennt Ihr Computer es als Datei und können Sie darauf zugreifen wie auf eine Datei, so kann Python sie auch öffnen.

Der Aufruf der `open()`-Funktion endet aber nicht mit dem Dateinamen. Es ist noch ein anderes Argument, namens `encoding`, vorhanden. Oh nein, das kommt mir erschreckend bekannt vor.

12.2.1 Die Zeichencodierung zeigt ihre hässliche Fratze

Bytes sind Bytes; Zeichen sind eine Abstraktion (siehe *Kap. 4*). Ein String ist eine Folge von Unicode-Zeichen. Eine Datei auf der Festplatte ist dagegen keine Folge von Unicode-Zeichen, sondern eine Bytefolge. Wie wandelt Python aber eine „Textdatei“ von einer Bytefolge in eine Zeichenfolge um? Es decodiert die Bytes auf Basis eines bestimmten Zeichencodierungsalgorithmus und gibt eine Folge von Unicode-Zeichen zurück (auch bekannt als String).

```
# Dieses Beispiel wurde unter Windows erstellt. Andere Plattformen
# verhalten sich aus Gründen, die ich weiter unten darstelle,
# vielleicht anders.
>>> file = open('examples/chinese.txt')
>>> a_string = file.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python31\lib\encodings\cp1252.py", line 23, in decode
    return codecs.charmap_decode(input,self.errors,decoding_table)[0]
UnicodeDecodeError: 'charmap' codec can't decode byte 0x8f in position
28: character maps to <undefined>
>>>
```

Was ist da gerade passiert? Sie haben keine Zeichencodierung angegeben, also verwendet Python die voreingestellte Codierung. Welche ist die voreingestellte Codierung? Wenn Sie sich die Traceback-Angaben genau ansehen, können Sie erkennen,

dass der Fehler in `cp1252.py` auftritt; Python verwendet als Codierung also standardmäßig `CP-1252`. (`CP-1252` ist unter Windows eine geläufige Codierung.) Die Zeichen der `CP-1252`-Codierung unterstützen die Zeichen dieser Datei nicht, daher schlägt der Lesevorgang mit einem `UnicodeDecodeError` fehl.

Doch es ist sogar noch schlimmer! Die voreingestellte Codierung ist plattformabhängig. Dieser Code könnte also auf Ihrem Computer funktionieren (wenn Ihre Standardcodierung UTF-8 ist), doch auf einem anderen nicht (wenn die Standardcodierung dort z. B. `CP-1252` ist).

☞ Müssen Sie die voreingestellte Zeichencodierung herausfinden, importieren Sie dazu das Modul `locale` und rufen `locale.getpreferredencoding()` auf. Auf meinem Windows-Laptop gibt die Funktion `'cp1252'` zurück, auf meinem Linux-Rechner aber `'UTF8'`. Nicht mal im eigenen Haus kann ich für Einheitlichkeit sorgen! Ihre Ergebnisse können anders aussehen (selbst unter Windows), je nach Ihrer verwendeten Betriebssystem-Version und Ihren Spracheinstellungen. Deshalb ist es so wichtig, dass Sie bei jedem Öffnen einer Datei die Zeichencodierung angeben.

12.2.2 Streamobjekte

Bis jetzt wissen wir lediglich, dass Python eine integrierte Funktion namens `open()` besitzt. Die `open()`-Funktion gibt ein Streamobjekt zurück, das Methoden und Attribute besitzt, mit denen wir Informationen über einen Zeichenstrom erhalten und diesen manipulieren können.

```
>>> a_file = open('examples/chinese.txt', encoding='utf-8')
>>> a_file.name            ①
'examples/chinese.txt'
>>> a_file.encoding        ②
'utf-8'
>>> a_file.mode            ③
'r'
```

① Das Attribut `name` enthält den Namen, den Sie der `open()`-Funktion beim Öffnen der Datei übergeben haben. Er wird nicht zu einer absoluten Pfadangabe gemacht.

② Das Attribut `encoding` enthält die Codierung, die Sie der `open()`-Funktion übergeben haben. Haben Sie die Codierung beim Öffnen der Datei nicht angegeben (schlechter Entwickler!), dann enthält das Attribut den Wert von `locale.getpreferredencoding()`.

③ Das Attribut `mode` enthält den Modus, in dem die Datei geöffnet wurde. Sie können der `open()`-Funktion einen optionalen `mode`-Parameter übergeben. Da Sie beim Öffnen dieser Datei keinen Modus angegeben haben, verwendet Python `'r'`, d. h. „im Textmodus zum Lesen öffnen“. Wie Sie später in diesem Kapitel

noch sehen werden, dient der Dateimodus verschiedenen Zwecken. Sie können in eine Datei schreiben, Daten an eine Datei anhängen, oder eine Datei im Binärmodus öffnen (dann haben Sie es mit Bytes statt Strings zu tun).

12.2.3 Daten aus einer Textdatei lesen

Nachdem Sie eine Datei zum Lesen geöffnet haben, möchten Sie vielleicht irgendwann auch daraus lesen.

```
>>> a_file = open('examples/chinese.txt', encoding='utf-8')
>>> a_file.read()          ①
'Dive Into Python 是为有经验的程序员编写的一本 Python 书。\n'
>>> a_file.read()          ②
''
```

① Haben Sie einmal eine Datei (mit der korrekten Codierung) geöffnet, so können Sie daraus einfach über die `read()`-Methode des Streamobjekts lesen. Das Ergebnis ist ein String.

② Überraschenderweise löst das erneute Lesen der Datei keine Ausnahme aus. Python sieht das Lesen über das Dateiende hinaus nicht als Fehler an, sondern gibt einfach einen leeren String zurück.

Was müssen Sie tun, wenn Sie die Datei erneut lesen möchten?

```
# Fortsetzung des vorherigen Beispiels
>>> a_file.read()          ①
''
>>> a_file.seek(0)         ②
0
>>> a_file.read(16)        ③
'Dive Into Python'
>>> a_file.read(1)         ④
' '
>>> a_file.read(1)
'是'
>>> a_file.tell()          ⑤
20
```

① Da Sie sich immer noch am Dateiende befinden, liefern weitere Aufrufe der `read()`-Methode des Streamobjekts nur leere Strings.

② Mit der `seek()`-Methode können Sie zu einer bestimmten Byteposition innerhalb der Datei springen.

③ Die `read()`-Methode kann einen optionalen Parameter übernehmen: die Anzahl der zu lesenden Zeichen.

④ Wenn Sie möchten, können Sie sogar Zeichen für Zeichen aus der Datei lesen.

⑤ `16 + 1 + 1 = … 20?`

Versuchen wir das noch mal!

```
# Fortsetzung des vorherigen Beispiels
>>> a_file.seek(17)         ①
17
>>> a_file.read(1)          ②
'是'
>>> a_file.tell()           ③
20
```

① Zum `17ten` Byte gehen.

② Ein Zeichen lesen.

③ Nun sind Sie beim `20sten` Byte.

Haben Sie's bemerkt? Sowohl die `seek()`-, als auch die `tell()`-Methode zählen immer in Bytes; da wir aber eine Textdatei geöffnet haben, zählt die `read()`-Methode in Zeichen. Chinesische Zeichen benötigen in UTF-8 codiert mehrere Bytes. Die englischen Zeichen in der Datei benötigen jeweils nur ein Byte. Sie könnten daher annehmen, dass die `seek()`- und die `read()`-Methode dasselbe zählen. Das stimmt jedoch nur für manche Zeichen.

Doch es wird noch schlimmer!

```
>>> a_file.seek(18)          ①
18
>>> a_file.read(1)           ②
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a_file.read(1)
  File "C:\Python31\lib\codecs.py", line 300, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode byte 0x98 in position 0:
unexpected code byte
```

① Gehe zum `18ten` Byte und versuche ein Zeichen zu lesen.

② Wieso schlägt dies fehl? Weil es beim `18ten` Byte kein Zeichen gibt. Das nächstliegende Zeichen beginnt beim `17ten` Byte (und belegt drei Bytes). Der Versuch, ein Zeichen dazwischen zu lesen schlägt mit einem `UnicodeDecodeError` fehl.

12.2.4 Dateien schließen

Geöffnete Dateien verbrauchen Systemressourcen. Außerdem können je nach verwendetem Modus andere Programme u. U. nicht auf diese Dateien zugreifen. Daher ist es wichtig, die Dateien zu schließen, sobald Sie sie fertig bearbeitet haben.

```
# Fortsetzung des vorherigen Beispiels
>>> a_file.close()
```

Naja, ziemlich enttäuschend.

Das Streamobjekt `a_file` existiert nach wie vor. Der Aufruf seiner `close()`-Methode zerstört nicht das Objekt selbst. Doch es ist nicht mehr sehr nützlich.

```
# Fortsetzung des vorherigen Beispiels
>>> a_file.read()                                    ①
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    a_file.read()
ValueError: I/O operation on closed file.
>>> a_file.seek(0)                                   ②
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    a_file.seek(0)
ValueError: I/O operation on closed file.
>>> a_file.tell()                                    ③
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    a_file.tell()
ValueError: I/O operation on closed file.
>>> a_file.close()                                   ④
>>> a_file.closed                                    ⑤
True
```

① Aus einer geschlossenen Datei können Sie nicht lesen; der Versuch löst eine `IOError`-Ausnahme aus.

② Auch `seek()` können Sie bei einer geschlossenen Datei nicht anwenden.

③ Auch die `tell()`-Methode schlägt fehl, da es bei einer geschlossenen Datei keine aktuelle Position gibt.

④ Überraschenderweise löst der Aufruf der `close()`-Methode hier keine Ausnahme aus. Es geschieht einfach gar nichts.

⑤ Geschlossene Streamobjekte besitzen ein nützliches Attribut: `closed` bestätigt Ihnen, dass die Datei geschlossen ist.

12.2.5 Automatisches Schließen von Dateien

Streamobjekte besitzen eine `close()`-Methode; doch was geschieht, wenn Ihr Code einen Bug enthält und abstürzt, bevor `close()` aufgerufen wurde? Diese Datei könnte theoretisch sehr viel länger geöffnet bleiben als nötig. Beim Debuggen auf Ihrem eigenen Computer ist das keine große Sache, wohl aber auf einem im Betrieb befindlichen Server.

In Python 2 gab es dafür eine Lösung: den `try..finally`-Block. Dieser funktioniert auch in Python 3 noch und wahrscheinlich werden Sie ihn im Code anderer Personen oder in älterem Code, der zu Python 3 portiert wurde, auch hin und wieder sehen. Doch mit Python 2.5 wurde ein besserer Lösungsansatz eingeführt, welcher nun auch in Python 3 bevorzugt wird: die `with`-Anweisung.

```
with open('examples/chinese.txt', encoding='utf-8') as a_file:
    a_file.seek(17)
    a_character = a_file.read(1)
    print(a_character)
```

In diesem Code wird zwar `open()`, aber niemals `a_file.close()` aufgerufen. Die `with`-Anweisung leitet – wie eine `if`-Anweisung oder eine `for`-Schleife – einen neuen Codeblock ein. Innerhalb dieses Codeblocks können Sie `a_file` wie das vom `open()`-Aufruf zurückgegebene Streamobjekt verwenden. Ihnen stehen alle Streamobjekt-Methoden zur Verfügung – `seek()`, `read()`, was immer beliebt. Wird der `with`-Block verlassen, ruft Python automatisch `a_file.close()` auf.

Hier nun der Clou: Python schließt die Datei auch dann, wenn der `with`-Block durch eine unbehandelte Ausnahme „verlassen“ wird. Richtig, selbst wenn Ihr Code eine Ausnahme auslöst und Ihr ganzes Programm auf unsanfte Art beendet wird, wird diese Datei geschlossen. Garantiert!

☞ Aus technischer Sicht wird durch die `with`-Anweisung ein *Laufzeitkontext* erstellt. In den vorliegenden Beispielen verhält sich das Streamobjekt wie ein Kontextmanager. Python erstellt das Streamobjekt `a_file` und teilt diesem mit, dass ein Laufzeitkontext beginnt. Wird der `with`-Block beendet, teilt Python dem Streamobjekt mit, dass der Laufzeitkontext verlassen wird, woraufhin das Streamobjekt seine eigene `close()`-Methode aufruft. Sehen Sie sich *Anhang B, „Kontextmanager“* für nähere Informationen hierzu an.

Die `with`-Anweisung kann nicht nur für Dateien verwendet werden. Es ist ein generisches Framework, das Laufzeitkontexte erstellt und Objekten mitteilt, dass ein Laufzeitkontext beginnt. Ist das Objekt ein Streamobjekt, werden nützliche Dateioperationen ausgeführt (wie das automatische Schließen der Datei). Dieses Verhalten ist aber nicht in der `with`-Anweisung definiert, sondern im Streamobjekt selbst. Sie können Kontextmanager auf viele Arten verwenden, die nichts mit Datei-

en zu tun haben. Sie können sogar eigene Kontextmanager erstellen, wie Sie später in diesem Kapitel sehen werden.

12.2.6 Daten zeilenweise lesen

Eine „Zeile“ einer Textdatei ist genau das, was Sie sich darunter vorstellen – Sie tippen ein paar Wörter ein und drücken `EINGABE`, schon befinden Sie sich in einer neuen Zeile. Eine Textzeile ist eine Zeichenfolge, die begrenzt wird von … was genau? Nun, das ist kompliziert, da Textdateien verschiedene Zeichen zur Markierung des Zeilenendes verwenden können. Jedes Betriebssystem hat seine eigenen Konventionen. Manche benutzen ein Wagenrücklauf-, andere ein Zeilenvorschub-Zeichen und wieder andere verwenden sowohl das eine, als auch das andere Zeichen am Ende einer Zeile.

Glücklicherweise kann Python Zeilenenden automatisch verarbeiten. Wenn Sie sagen, „Ich möchte diese Textdatei Zeile für Zeile lesen“, dann findet Python heraus, welche Art von Zeilenendung diese Textdatei benutzt und alles funktioniert tadellos.

☞ Sollten Sie eine genauere Kontrolle über die Behandlung des Zeilenendes benötigen, können Sie der `open()`-Funktion den optionalen Parameter `newline` übergeben. Sehen Sie sich dazu die Dokumentation zur `open()`-Funktion an.

Wie liest man denn nun eine Datei Zeile für Zeile? Es ist so einfach. Es ist wunderschön.

```
line_number = 0
with open('examples/favorite-people.txt', encoding='utf-8') as a_file:     ①
    for a_line in a_file:                                                   ②
        line_number += 1
        print('{:>4} {}'.format(line_number, a_line.rstrip()))              ③
```

① Durch Verwendung der `with`-Notation können wir die Datei sicher öffnen und überlassen das Schließen Python.

② Zum zeilenweisen Lesen der Datei verwenden wir eine `for`-Schleife. Das ist alles. Neben der Bereitstellung von Methoden wie `read()` ist das Streamobjekt auch ein Iterator, der bei jeder Abfrage eines Wertes eine einzelne Zeile ausgibt.

③ Mithilfe der Stringmethode `format()` können wir die Zeilennummer und die Zeile selbst ausgeben. Die Formatangabe `{:>4}` bedeutet „gebe dieses Argument rechtsbündig mit vier Stellen aus“. Die Variable `a_line` enthält die komplette Zeile, inklusive Zeilenumbruch usw. Die Stringmethode `rstrip()` entfernt den nachfolgenden Whitespace, inklusive der Zeilenumbruch-Zeichen.

Die Ausgabe:

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
   1 Dora
   2 Ethan
   3 Wesley
   4 John
   5 Anne
   6 Mike
   7 Chris
   8 Sarah
   9 Alex
  10 Lizzie
```

Haben Sie diesen Fehler erhalten?

```
you@localhost:~/diveintopython3$ python3 examples/oneline.py
Traceback (most recent call last):
  File "examples/oneline.py", line 4, in <module>
    print('{:>4} {}'.format(line_number, a_line.rstrip()))
ValueError: zero length field name in format
```

Wenn ja, dann verwenden Sie möglicherweise Python 3.0. Sie sollten wirklich eine Aktualisierung auf Python 3.1 durchführen.

Python 3.0 unterstützt zwar Stringformatierung, dies aber nur mit der ausdrücklichen Angabe nummerierter Formatangaben. Python 3.1 dagegen erlaubt innerhalb der Formatangaben das Weglassen der Indizes. Hier nun zum Vergleich die Python 3.0-konforme Version des Codes:

```
print('{0:>4} {1}'.format(line_number, a_line.rstrip()))
```

12.3 In Textdateien schreiben

In Textdateien zu schreiben funktioniert fast so, wie aus ihnen zu lesen. Zunächst öffnen Sie eine Datei und erhalten so ein Streamobjekt. Dann verwenden Sie die Methoden des Streamobjekts, um Daten in die Datei zu schreiben. Danach schließen Sie die Datei.

Verwenden Sie die `open()`-Funktion und geben Sie den Schreibmodus an, um eine Datei zum Schreiben zu öffnen. Zum Schreiben gibt es zwei Dateimodi:

- Im Modus „Schreiben“ (engl. *write*) wird die Datei überschrieben. Übergeben Sie dazu `mode='w'` an die `open()`-Funktion.
- Im Modus „Anhängen“ (engl. *append*) werden die Daten ans Dateiende angehängt. Übergeben Sie dazu `mode='a'` an die `open()`-Funktion.

Bei beiden Modi wird die Datei automatisch erstellt, sollte sie nicht bereits vorhanden sein. Sie brauchen sich also nicht um die Überprüfung der Existenz der Datei zu kümmern. Öffnen Sie sie einfach und fangen Sie an zu schreiben.

Nach dem Schreibvorgang sollten Sie die Datei immer schließen, um sicherzustellen, dass die Datei für andere Anwendungen verfügbar ist und die Daten auch wirklich auf die Festplatte geschrieben werden. Genau wie beim Lesevorgang können Sie auch hier die `close()`-Methode des Streamobjekts aufrufen, oder die `with`-Anweisung verwenden, wenn Sie wollen, dass Python die Datei für Sie schließt. Ich wette, Sie ahnen bereits, welche Vorgehensweise ich empfehle.

```
>>> with open('test.log', mode='w', encoding='utf-8') as a_file:  ①
...     a_file.write('test succeeded')                            ②
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
test succeeded
>>> with open('test.log', mode='a', encoding='utf-8') as a_file:  ③
...     a_file.write('and again')
>>> with open('test.log', encoding='utf-8') as a_file:
...     print(a_file.read())
test succeededand again                                           ④
```

① Wir beginnen mit der Erstellung der neuen Datei `test.log` (oder dem Überschreiben der bereits vorhandenen Datei) und öffnen diese zum Schreiben. Der Parameter `mode='w'` bedeutet „zum Schreiben öffnen“. Ja, das ist so gefährlich, wie es klingt. Ich hoffe, der vorherige Inhalt der Datei (sofern vorhanden) hat Ihnen nicht viel bedeutet, denn dieser Inhalt ist nun verloren.

② Mithilfe der `write()`-Methode des von der `open()`-Funktion zurückgegebenen Streamobjekts können Sie Daten zu der gerade geöffneten Datei hinzufügen. Nach der Beendung des `with`-Blocks schließt Python die Datei automatisch.

③ Das hat Spaß gemacht; machen wir’s noch mal. Doch nun hängen wir mit `mode='a'` Daten an die Datei an, statt sie zu überschreiben. Anhängen hat niemals Einfluss auf den vorhandenen Inhalt der Datei.

④ Nun befinden sich sowohl die erste von Ihnen geschriebene Zeile als auch die angehängte Zeile in der Datei `test.log`. Beachten Sie auch das Fehlen von Zeilenumbrüchen. Da Sie diese nicht ausdrücklich in die Datei geschrieben haben, sind sie nicht vorhanden. Einen Zeilenumbruch erreichen Sie durch die Eingabe von `'\n'`. Da Sie das nicht getan haben, steht der komplette Text in der Datei auf einer Zeile.

12.3.1 Schon wieder Zeichencodierung

Haben Sie den Parameter `encoding` bemerkt, den wir der `open()`-Funktion beim Öffnen der Datei übergeben haben? Dieser Parameter ist sehr wichtig; lassen

Sie ihn nicht weg! Wie Sie zu Beginn dieses Kapitels gesehen haben, enthalten Dateien keine Strings, sondern Bytes. Das Lesen eines „Strings“ aus einer Textdatei funktioniert nur deshalb, weil Sie Python mitgeteilt haben, welche Codierung verwendet werden soll. Das Schreiben einer Textdatei stellt uns vor dasselbe Problem, nur andersherum. Sie können keine Zeichen in eine Datei schreiben; Zeichen sind eine Abstraktion. Zum Schreiben in die Datei muss Python wissen, wie Ihr String in eine Bytefolge umgewandelt werden muss. Und dies erreicht man auf sichere Art und Weise nur durch die Angabe des `encoding`-Parameters beim Öffnen der Datei zum Schreiben.

12.4 Binärdateien

Nicht alle Dateien enthalten Text. Manche enthalten z. B. Fotos.

```
>>> an_image = open('examples/beauregard.jpg', mode='rb')        ①
>>> an_image.mode                                                ②
'rb'
>>> an_image.name                                                ③
'examples/beauregard.jpg'
>>> an_image.encoding                                            ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: '_io.BufferedReader' object has no attribute 'encoding'
```

① Das Öffnen einer Binärdatei ist sehr einfach. Der einzige Unterschied zum Öffnen einer Datei im Textmodus ist der, dass der `mode`-Parameter ein `'b'` enthält.

② Das Streamobjekt, das Sie beim Öffnen einer Datei im Binärmodus erhalten, besitzt viele der bekannten Attribute, darunter auch `mode`. `mode` repräsentiert den Parameter `mode`, den Sie der `open()`-Funktion übergeben haben.

③ Binäre Streamobjekte haben, genau wie Streamobjekte für Text, ein `name`-Attribut.

④ Einen Unterschied gibt es: Ein binäres Streamobjekt besitzt kein `encoding`-Attribut. Das ist einleuchtend, oder? Sie lesen (oder schreiben) Bytes, keine Strings, also muss Python keine Umwandlung durchführen. Was Sie aus einer Binärdatei lesen, ist genau das, was Sie hineingeschrieben haben. Es ist keine Umwandlung nötig.

Habe ich darauf hingewiesen, dass Sie Bytes lesen? Ja, genau das tun Sie.

```
# Fortsetzung des vorherigen Beispiels
>>> an_image.tell()
0
>>> data = an_image.read(3)  ①
```

```
>>> data
b'\xff\xd8\xff'
>>> type(data)                ②
<class 'bytes'>
>>> an_image.tell()           ③
3
>>> an_image.seek(0)
0
>>> data = an_image.read()
>>> len(data)
3150
```

① Wie von Textdateien bekannt, können Sie auch Binärdateien Stück für Stück lesen. Doch es gibt einen entscheidenden Unterschied …

② … Sie lesen Bytes, keine Strings. Da Sie die Datei im Binärmodus geöffnet haben, übernimmt die `read()`-Methode die Anzahl der zu lesenden Bytes, nicht die der zu lesenden Zeichen.

③ Das bedeutet, dass es niemals eine unerwartete Diskrepanz zwischen der an die `read()`-Methode übergebenen Zahl und dem von der `tell()`-Methode zurückgegebenen Positionsindex gibt. Die `read()`-Methode liest Bytes und sowohl die `seek()`- als auch die `tell()`-Methode verfolgen die Anzahl der gelesenen Bytes. Bei Binärdateien stimmen diese Zahlen immer überein.

12.5 Streamobjekte aus anderen Quellen als Dateien

Stellen Sie sich vor, Sie schreiben eine Bibliothek und eine Ihrer Bibliotheksfunktionen liest Daten aus einer Datei. Die Funktion könnte einfach einen Dateinamen als String übernehmen, diese Datei zum Lesen öffnen, sie lesen und vor dem Beenden schließen. Doch so sollten Sie es nicht machen … Stattdessen sollte Ihre API ein beliebiges Streamobjekt übernehmen.

Im einfachsten Fall ist ein Streamobjekt alles, was eine `read()`-Methode besitzt, die einen optionalen `size`-Parameter (`size` = die Anzahl der zu lesenden Zeichen oder Bytes; Anm. d. Übers.) übernimmt und einen String zurückgibt. Wird die `read()`-Methode ohne `size`-Parameter aufgerufen, so soll sie alles aus der Eingabequelle lesen und die kompletten Daten als einen einzigen Wert zurückgeben. Wird sie hingegen mit `size`-Parameter aufgerufen wird, liest sie so viel wie angegeben und gibt nur diese Daten zurück. Wird sie erneut aufgerufen, fährt sie dort fort, wo sie die Datei vorher verlassen hat und gibt das nächste Datenstück zurück.

Das klingt genau nach dem Streamobjekt, das Sie beim Öffnen einer echten Datei erhalten. Doch Sie beschränken sich nicht auf echte Dateien. Die zu lesende Eingabequelle könnte alles sein: eine Webseite, ein String im Speicher oder sogar die Ausgabe eines anderen Programms. Solange Ihre Funktionen ein Streamobjekt

übernehmen und einfach dessen `read()`-Methode aufrufen, können Sie mit jeder Art von Eingabequelle umgehen, die sich wie eine Datei verhält. Sie benötigen dann keinen Code, der jede Art von Eingabe einzeln behandelt.

```
>>> a_string = 'PapayaWhip is the new black.'
>>> import io                                    ①
>>> a_file = io.StringIO(a_string)               ②
>>> a_file.read()                                ③
'PapayaWhip is the new black.'
>>> a_file.read()                                ④
''
>>> a_file.seek(0)                               ⑤
0
>>> a_file.read(10)                              ⑥
'PapayaWhip'
>>> a_file.tell()
10
>>> a_file.seek(18)
18
>>> a_file.read()
'new black.'
```

① Das Modul `io` definiert die Klasse `StringIO`, mit der Sie einen String im Speicher wie eine Datei behandeln können.

② Um ein Streamobjekt aus einem String zu erstellen, müssen Sie eine Instanz der `io.StringIO()`-Klasse erzeugen und ihr den zu verwendenden String übergeben. Nun haben Sie ein Streamobjekt, mit dem Sie allerhand anstellen können.

③ Der Aufruf der `read()`-Methode „liest“ die gesamte „Datei“, was im Falle eines `StringIO`-Objekts bedeutet, dass einfach der ursprüngliche String zurückgegeben wird.

④ Wie bei einer echten Datei führt der erneute Aufruf der `read()`-Methode zur Rückgabe eines leeren Strings.

⑤ Mithilfe der `seek()`-Methode des `StringIO`-Objekts können Sie, ebenfalls wie bei einer echten Datei, zum Anfang des Strings springen.

⑥ Sie können den String auch Stück für Stück lesen, indem Sie der `read()`-Methode einen `size`-Parameter übergeben.

☞ Mit `io.StringIO` können Sie einen String wie eine Textdatei behandeln. Mithilfe der ebenfalls verfügbaren Klasse `io.BytesIO` können Sie außerdem ein Bytearray wie eine Binärdatei handhaben.

12.5.1 Umgang mit komprimierten Dateien

Pythons Standardbibliothek enthält Module, mit denen komprimierte Dateien gelesen und geschrieben werden können. Dazu stehen verschiedene Komprimierungs-

methoden zur Verfügung; die zwei bekanntesten auf Nicht-Windows-Systemen sind gzip und bzip2. (Vielleicht sind Ihnen auch schon PKZIP- und GNU-Tar-Archive begegnet. Python enthält auch Module zur Unterstützung dieser Methoden.)

Das Modul `gzip` erlaubt Ihnen die Erstellung eines Streamobjekts zum Lesen oder Schreiben einer gzip-komprimierten Datei. Dieses Streamobjekt unterstützt die `read()`-Methode (wenn Sie es zum Lesen geöffnet haben) wie auch die `write()`-Methode (wenn Sie es zum Schreiben geöffnet haben). Sie können also die Ihnen bereits bekannten Methoden nutzen, um direkt eine gzip-komprimierte Datei zu lesen oder zu schreiben, ohne erst eine temporäre Datei mit den noch nicht komprimierten Daten zu erstellen.

Außerdem unterstützt es auch die `with`-Anweisung; Sie können Python also automatisch die gzip-komprimierte Datei schließen lassen, wenn Sie mit ihr fertig sind.

```
you@localhost:~$ python3

>>> import gzip
>>> with gzip.open('out.log.gz', mode='wb') as z_file:          ①
...     z_file.write('A nine mile walk is no joke, especially in the
rain.'.encode('utf-8'))
...
>>> exit()

you@localhost:~$ ls -l out.log.gz                               ②
-rw-r--r--  1 mark mark    79 2009-07-19 14:29 out.log.gz
you@localhost:~$ gunzip out.log.gz                              ③
you@localhost:~$ cat out.log                                    ④
A nine mile walk is no joke, especially in the rain.
```

① Sie sollten gzip-Dateien immer im Binärmodus öffnen. (Beachten Sie das `'b'` im Argument `mode`.)

② Dieses Beispiel habe ich unter Linux erstellt. Dieser Befehl bewirkt, dass in der Python Shell detaillierte Angaben zu der gerade erstellten gzip-komprimierten Datei ausgegeben werden. Man erkennt, dass die Datei existiert (gut), und dass sie 79 B groß ist. Das ist sogar größer, als der String mit dem wir begonnen haben! Das gzip-Dateiformat beinhaltet einen Kopfbereich mit fester Größe; dieser Kopfbereich enthält Metadaten über die Datei. Die Komprimierung ist also für sehr kleine Dateien nicht effizient.

③ Der Befehl `gunzip` (*gesprochen* „dschi-anzipp“) dekomprimiert die Datei und speichert ihren Inhalt in einer neuen Datei mit demselben Namen, aber ohne die Erweiterung `.gz`.

④ Der Befehl `cat` zeigt den Inhalt einer Datei an. Diese Datei enthält den String, den Sie ursprünglich direkt von der Python-Shell in die komprimierte Datei `out.log.gz` geschrieben haben.

12.6 Standardeingabe, -ausgabe und -fehler

Kommandozeilen-Profis kennen das Konzept von Standardeingabe, Standardausgabe und Standardfehler bereits. Dieser Abschnitt ist für alle anderen.

Standardausgabe und Standardfehler (meist abgekürzt als `stdout` und `stderr`) sind sogenannte *Pipes*, die in jedem UNIX-ähnlichen System, also auch Mac OS X und Linux, vorhanden sind. Wenn Sie `print()` aufrufen, wird die gewünschte Ausgabe an die `stdout`-Pipe geschickt. Stürzt Ihr Programm ab und gibt Traceback-Angaben aus, wird diese Ausgabe an die `stderr`-Pipe gesendet. Per Voreinstellung sind diese Pipes einfach mit dem Terminalfenster verbunden, in dem Sie gerade arbeiten; gibt Ihr Programm etwas aus, sehen Sie diese Ausgabe in Ihrem Terminalfenster, und auch wenn Ihr Programm abstürzt, sehen Sie die Traceback-Angaben in Ihrem Terminalfenster. In der grafischen Python-Shell sind `stdout` und `stderr` auf Ihr „interaktives Fenster“ voreingestellt.

```
>>> for i in range(3):
...     print('PapayaWhip')        ①
PapayaWhip
PapayaWhip
PapayaWhip
>>> import sys
>>> for i in range(3):
... sys.stdout.write('is the')     ②
is theis theis the
>>> for i in range(3):
... sys.stderr.write('new black')  ③
new blacknew blacknew black
```

① Hier gibt’s nichts Überraschendes. Lediglich eine `print()`-Anweisung innerhalb einer Schleife.

② `stdout` ist ein im Modul `sys` definiertes Streamobjekt. Der Aufruf seiner `write()`-Funktion gibt den String aus, den Sie ihr übergeben. Das ist das, was die `print()`-Funktion tatsächlich tut; sie fügt einen Zeilenumbruch am String-Ende hinzu und ruft dann `sys.stdout.write()` auf.

③ Im einfachsten Fall senden `sys.stdout` und `sys.stderr` ihre Ausgaben an dieselbe Stelle: an die Python IDE (sofern Sie sich in einer befinden) oder an das Terminal (wenn Sie Python auf der Kommandozeile ausführen). Der Standardfehler-Stream fügt, genau wie der Standardausgabe-Stream, keine Zeilenumbrüche ein. Möchten Sie Zeilenumbrüche haben, müssen Sie diese mithilfe von `'\n'` (dem Wagenrücklauf-Zeichen) selbst einfügen.

`sys.stdout` und `sys.stderr` sind Streamobjekte, auf die man allerdings nur schreibend zugreifen kann. Der Versuch ihre `read()`-Methoden aufzurufen endet immer mit einem `IOError`.

```
>>> import sys
>>> sys.stdout.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: not readable
```

12.6.1 Die Standardausgabe umleiten

`sys.stdout` und `sys.stderr` sind Streamobjekte, selbst wenn sie nur den Schreibmodus unterstützen. Doch es sind keine Konstanten; es sind Variablen. Das heißt, dass Sie ihnen neue Werte – jedes beliebige andere Streamobjekt – zuweisen können, um ihre Ausgabe umzuleiten.

```
import sys

class RedirectStdoutTo:
    def __init__(self, out_new):
        self.out_new = out_new

    def __enter__(self):
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):
        sys.stdout = self.out_old

print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Sehen Sie sich das an:

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
A
C
you@localhost:~/diveintopython3/examples$ cat out.log
B
```

Haben Sie diesen Fehler erhalten:

```
you@localhost:~/diveintopython3/examples$ python3 stdout.py
  File "stdout.py", line 15
    with open('out.log', mode='w', encoding='utf-8') as a_file, Redi-
rectStdoutTo(a_file):
                                                               ^
SyntaxError: invalid syntax
```

Wenn ja, nutzen Sie vermutlich Python 3.0. Sie sollten wirklich eine Aktualisierung auf Python 3.1 durchführen.

Python 3.0 unterstützt zwar die `with`-Anweisung, doch jede Anweisung kann hier nur einen Kontextmanager verwenden. Python 3.1 erlaubt Ihnen dagegen das Verketten mehrerer Kontextmanager in einer einzigen `with`-Anweisung.

Sehen wir uns zuerst den letzten Teil an.

```
print('A')
with open('out.log', mode='w', encoding='utf-8') as a_file, RedirectStdoutTo(a_file):
    print('B')
print('C')
```

Das ist eine recht komplizierte `with`-Anweisung. Ich werde Sie etwas klarer formulieren.

```
with open('out.log', mode='w', encoding='utf-8') as a_file:
    with RedirectStdoutTo(a_file):
        print('B')
```

Wie die neue Fassung zeigt, haben wir in Wirklichkeit zwei `with`-Anweisungen; eine innerhalb des Gültigkeitsbereichs der anderen. Die „äußere“ `with`-Anweisung sollte Ihnen bekannt vorkommen: Sie öffnet eine in UTF8 codierte Textdatei namens `out.log` zum Schreiben und weist das Streamobjekt einer Variable namens `a_file` zu. Doch das ist hier nicht das einzig Seltsame.

```
with RedirectStdoutTo(a_file):
```

Wo ist der `as`-Teil? Die `with`-Anweisung benötigt eigentlich gar keinen. Genau wie Sie eine Funktion aufrufen und deren Rückgabewert ignorieren können, können Sie auch eine `with`-Anweisung ohne Zuweisung des `with`-Kontexts an eine Variable verwenden. Im vorliegenden Fall sind Sie lediglich an den Seiteneffekten des `RedirectStdoutTo`-Kontexts interessiert.

Welche Seiteneffekte sind das? Sehen Sie sich die Klasse `RedirectStdoutTo` einmal an. Diese Klasse ist ein selbst definierter Kontextmanager. Jede Klasse kann als Kontextmanager fungieren, indem Sie zwei besondere Methoden definieren: `__enter__()` und `__exit__()`.

```
class RedirectStdoutTo:
    def __init__(self, out_new):     ①
        self.out_new = out_new

    def __enter__(self):              ②
        self.out_old = sys.stdout
        sys.stdout = self.out_new

    def __exit__(self, *args):        ③
        sys.stdout = self.out_old
```

① Die `__init__()`-Methode wird sofort nach dem Erstellen einer Instanz aufgerufen. Sie übernimmt einen Parameter: das Streamobjekt, das Sie für die Dauer des Kontexts als Standardausgabe verwenden möchten. Diese Methode speichert das Streamobjekt in einer Instanzvariablen, so dass andere Methoden es später nutzen können.

② Die `__enter__()`-Methode ist eine besondere Klassenmethode; Python ruft sie auf, sobald ein Kontext beginnt (z. B. zu Beginn einer `with`-Anweisung). Die Methode speichert den aktuellen Wert von `sys.stdout` in `self.out_old` und leitet die Standardausgabe dann durch Zuweisung von `self.out_new` an `sys.stdout` um.

③ Die `__exit__()`-Methode ist eine weitere spezielle Klassenmethode; Python ruft sie auf, sobald ein Kontext verlassen wird (z. B. am Ende der `with`-Anweisung). Die Methode setzt die Standardausgabe wieder auf den ursprünglichen Wert, indem `self.out_old` `sys.stdout` zugewiesen wird.

Alles zusammen sieht dann so aus:

```
print('A')                ①

with open('out.log', mode='w', encoding='utf-8') as a_file, Re-
directStdoutTo(a_file):    ②
    print('B')             ③

print('C')                 ④
```

① Diese Ausgabe erfolgt im „interaktiven Fenster“ der IDE (oder im Terminal, wenn das Skript von der Kommandozeile aus gestartet wurde).

② Diese `with`-Anweisung übernimmt eine durch Kommas getrennte Kontextliste. Diese Liste verhält sich wie eine Reihe verschachtelter `with`-Blöcke. Der erste angeführte Kontext ist der „äußere“ Block; der Letzte ist der „innere“ Block. Der erste Kontext öffnet eine Datei; der zweite Kontext leitet `sys.stdout` auf das im ersten Kontext erstellte Streamobjekt um.

③ Da die `print()`-Anweisung mit den von der `with`-Anweisung erstellten Kontexten ausgeführt wird, erfolgt die Ausgabe nicht auf den Bildschirm, sondern in die Datei `out.log`.

④ Der `with`-Block wurde beendet. Python hat jedem Kontextmanager mitgeteilt, das zu tun, was sie beim Verlassen eines Kontexts tun sollen. Die Kontextmanager bilden einen *LIFO-Stack* (*last in, first out; zuletzt hinein, zuerst hinaus*). Beim Verlassen setzte der zweite Kontext `sys.stdout` wieder auf den ursprünglichen Wert; dann wurde vom ersten Kontext die Datei `out.log` geschlossen. Da die Standardausgabe wieder zurückgesetzt wurde, erfolgt die Ausgabe der `print()`-Funktion nun wieder auf den Bildschirm.

Die Umleitung des Standardfehler-Streams erfolgt auf die gleiche Weise. Verwenden Sie einfach `sys.stderr`, anstelle von `sys.stdout`.

Kapitel 13
XML

13.1 Los geht's

Die meisten Kapitel dieses Buches drehten sich bisher um Beispielcode. Doch bei XML geht es nicht um Code; es geht um Daten. Ein großer Einsatzbereich von XML liegt in sogenannten *Feeds*, die die aktuellsten Artikel eines Blogs, Forums oder jeder anderen häufig aktualisierten Webseite auflisten. Die meisten bekannten Blog-Programme können Feeds erzeugen und sie aktualisieren, sobald ein neuer Artikel, Thread oder Blogeintrag veröffentlicht wird. Sie können einen Feed „abonnieren", um ständig auf dem Laufenden zu bleiben. Mithilfe eines Feed-Readers wie Google Reader können Sie auch Feeds von verschiedenen Blogs verwalten.

Hier also nun die XML-Daten, mit denen wir im Verlauf dieses Kapitels arbeiten werden. Es handelt sich um einen Feed – genauer gesagt um einen *Atom-Feed*. (*Atom* ist das Format.)

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  <link rel='self' type='application/atom+xml' href='http://diveintomark.org/feed/'/>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Dive into history, 2009 edition</title>
    <link rel='alternate' type='text/html'
          href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-
edition'/>
    <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
    <updated>2009-03-27T21:56:07Z</updated>
```

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_13, © Springer-Verlag Berlin Heidelberg 2010

```
    <published>2009-03-27T17:20:42Z</published>
    <category scheme='http://diveintomark.org' term='diveintopython'/>
    <category scheme='http://diveintomark.org' term='docbook'/>
    <category scheme='http://diveintomark.org' term='html'/>
  <summary type='html'>Putting an entire chapter on one page sounds
    bloated, but consider this &amp;mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds&amp;hellip;
    On dialup.</summary>
  </entry>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Accessibility is a harsh mistress</title>
    <link rel='alternate' type='text/html'
          href='http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-
mistress'/>
    <id>tag:diveintomark.org,2009-03-21:/archives/20090321200928</id>
    <updated>2009-03-22T01:05:37Z</updated>
    <published>2009-03-21T20:09:28Z</published>
    <category scheme='http://diveintomark.org' term='accessibility'/>
    <summary type='html'>The accessibility orthodoxy does not permit people to
         question the value of features that are rarely useful and rarely
used.</summary>
  </entry>
  <entry>
    <author>
      <name>Mark</name>
    </author>
    <title>A gentle introduction to video encoding, part 1: container formats</title>
    <link rel='alternate' type='text/html'
            href='http://diveintomark.org/archives/2008/12/18/give-part-1-container-
formats'/>
    <id>tag:diveintomark.org,2008-12-18:/archives/20081218155422</id>
    <updated>2009-01-11T19:39:22Z</updated>
    <published>2008-12-18T15:54:22Z</published>
    <category scheme='http://diveintomark.org' term='asf'/>
    <category scheme='http://diveintomark.org' term='avi'/>
    <category scheme='http://diveintomark.org' term='encoding'/>
    <category scheme='http://diveintomark.org' term='flv'/>
    <category scheme='http://diveintomark.org' term='GIVE'/>
    <category scheme='http://diveintomark.org' term='mp4'/>
    <category scheme='http://diveintomark.org' term='ogg'/>
    <category scheme='http://diveintomark.org' term='video'/>
    <summary type='html'>These notes will eventually become part of a
      tech talk on video encoding.</summary>
  </entry>
</feed>
```

13.2 Ein XML-Crashkurs

Wenn Sie sich mit XML bereits auskennen, können Sie diesen Abschnitt überspringen.

XML ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten. Ein XML-Dokument besteht aus einem oder mehreren Elementen, die durch *Start-* und *End-Tags* begrenzt werden. Dies ist ein vollständiges (wenn auch ziemlich langweiliges) XML-Dokument:

```
<foo>   ①
</foo>  ②
```

① Dies ist das Start-Tag des Elements `foo`.

② Dies ist das dazugehörige End-Tag. Wie die Klammern beim Schreiben, in der Mathematik oder bei Quellcode müssen auch die Start- und End-Tags ausgeglichen sein. Jedes Start-Tag muss durch ein End-Tag abgeschlossen werden.

Elemente können beliebig weit verschachtelt werden. Ein Element mit der Bezeichnung `bar` innerhalb des `foo`-Elements wird Unterelement oder Kindelement von `foo` genannt.

```
<foo>
  <bar></bar>
</foo>
```

Das erste Element eines jeden XML-Dokuments wird *Wurzelelement* genannt. Ein XML-Dokument kann lediglich ein Wurzelelement besitzen. Der folgende Text ist *kein* XML-Dokument, da zwei Wurzelelemente enthalten sind:

```
<foo></foo>
<bar></bar>
```

Elemente können *Attribute* haben. Dies sind Name-Wert-Paare, die innerhalb des Start-Tags, getrennt durch Whitespace, aufgelistet werden. Attributnamen können innerhalb eines Elements nicht wiederholt werden. Attributwerte müssen in Anführungszeichen geschrieben werden. Sie können sowohl einfache, als auch doppelte Anführungszeichen benutzen.

```
<foo lang='en'>                          ①
  <bar id='papayawhip' lang="fr"></bar>  ②
</foo>
```

① Das `foo`-Element besitzt ein Attribut namens `lang`. Der Wert von `lang` ist `en`.

② Das `bar`-Element besitzt zwei Attribute, `id` und `lang`. Der Wert von `lang` ist `fr`. Dies führt *nicht* zu Problemen mit dem `foo`-Element. Jedes Element hat seine eigenen Attribute.

Hat ein Element mehr als ein Attribut, so spielt die Reihenfolge der Attribute keine Rolle. Die Attribute eines Elements bilden eine ungeordnete Menge von Schlüsseln und Werten, so wie ein Dictionary. Die Anzahl der Attribute die Sie für jedes Element definieren können, ist unbegrenzt.

Elemente können einen Textinhalt haben:

```
<foo lang='en'>
  <bar lang='fr'>PapayaWhip</bar>
</foo>
```

Elemente die keinen Text und kein Kindelement enthalten, sind *leer*.

```
<foo></foo>
```

Leere Elemente kann man auch kürzer schreiben. Durch Einfügen eines `/`-Zeichens in das Start-Tag können Sie das End-Tag ganz weglassen. Das gerade gezeigte XML-Dokument könnte also auch so geschrieben werden:

```
<foo/>
```

So wie Funktionen in unterschiedlichen *Modulen* deklariert werden können, können XML-Elemente in unterschiedlichen *Namensbereichen* deklariert werden. Namensbereiche sehen gewöhnlich aus wie URLs. Zum Definieren eines *Standardnamensbereichs* verwenden Sie eine `xmlns`-Deklaration. Eine Namensbereichsdeklaration sieht aus wie ein Attribut, erfüllt aber einen anderen Zweck.

```
<feed xmlns='http://www.w3.org/2005/Atom'>  ①
  <title>dive into mark</title>             ②
</feed>
```

① Das Element `feed` befindet sich im Namensbereich `http://www.w3.org/2005/Atom`.

② Das Element `title` befindet sich ebenfalls im Namensbereich `http://www.w3.org/2005/Atom`. Die Namensbereichsdeklaration wirkt sich auf das Element in der die Deklaration stattfindet, wie auch auf alle Kindelemente aus.

Sie können zum Definieren eines Namensbereichs auch eine `xmlns:Präfix`-Deklaration verwenden und so den Namensbereich mit einem *Präfix* verbinden. Dann muss jedes Element in diesem Namensbereich ausdrücklich mit diesem Präfix deklariert werden.

```
<atom:feed xmlns:atom='http://www.w3.org/2005/Atom'>  ①
  <atom:title>dive into mark</atom:title>             ②
</atom:feed>
```

① Das Element `feed` befindet sich im Namensbereich `http://www.w3.org/2005/Atom`.

② Das Element `title` befindet sich ebenfalls im Namensbereich `http://www.w3.org/2005/Atom`.

Der XML-Parser betrachtet die beiden vorherigen XML-Dokumente als *gleich*. *Namensbereich + Elementname = XML-Gleichheit*. Präfixe sind nur dazu da, auf Namensbereiche zu verweisen; der Name eines Präfixes (`atom:`) ist irrelevant. Die Namensbereiche stimmen überein, die Elementnamen stimmen überein, die Attribute (oder das Fehlen eben dieser) stimmen überein und der Textinhalt jedes Elements stimmt überein; die XML-Dokumente sind daher gleich.

XML-Dokumente können in der ersten Zeile – noch vor dem Wurzelelement – Informationen zur Zeichencodierung enthalten.

```
<?xml version='1.0' encoding='utf-8'?>
```

Jetzt wissen Sie genug über XML, um loszulegen!

13.3 Der Aufbau eines Atom-Feeds

Stellen Sie sich ein Weblog vor, oder eigentlich jede Webseite, die häufig aktualisiert wird, wie CNN.com. Die Seite selbst hat einen Titel („CNN.com“), einen Untertitel („Breaking News, U.S., World, Weather, Entertainment & Video News“), ein Datum der letzten Aktualisierung („updated 12:43 p.m. EDT, Sat May 16, 2009“) und eine Liste von Artikeln, die zu verschiedenen Zeiten veröffentlicht wurden. Jeder Artikel hat wiederum einen Titel, ein Datum der ersten Veröffentlichung (vielleicht auch ein Datum der letzten Aktualisierung, wenn eine Korrektur durchgeführt wurde) und eine einmalige URL.

Das Atom-Format dient dazu, all diese Informationen in einem Standardformat zu erhalten. Mein Weblog und CNN.com mögen sehr verschieden in Design, Größe und Zielgruppe sein, doch sie haben beide dieselbe Grundstruktur. CNN.com hat einen Titel; mein Blog hat einen Titel. CNN.com veröffentlicht Artikel; ich veröffentliche Artikel.

Auf der obersten Ebene befindet sich das Wurzelelement, das bei jedem Atom-Feed gleich ist: das `feed`-Element im Namensbereich `http://www.w3.org/2005/Atom`.

```
<feed xmlns='http://www.w3.org/2005/Atom'          ①
      xml:lang='en'>                               ②
```

① `http://www.w3.org/2005/Atom` ist der Atom-Namensbereich.

② Jedes Element kann ein `xml:lang`-Attribut enthalten, das die Sprache des Elements und seiner Kindelemente angibt. In unserem Fall wird das `xml:lang`-Attribut nur einmal im Wurzelelement deklariert, unser gesamter Feed verwendet also die englische Sprache.

Ein Atom-Feed enthält einige Informationen über den Feed selbst. Diese werden als Kindelemente des `feed`-Elements deklariert.

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>                                          ①
  <subtitle>currently between addictions</subtitle>                      ②
  <id>tag:diveintomark.org,2001-07-29:/</id>                             ③
  <updated>2009-03-27T21:56:07Z</updated>                                ④
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>  ⑤
```

① Der Titel dieses Feeds lautet `dive into mark`.

② Der Untertitel des Feeds lautet `currently between addictions`.

③ Jeder Feed braucht eine einmalige ID.

④ Dieser Feed wurde zuletzt am 27. März 2009 um 21:56 GMT aktualisiert. Dies ist normalerweise das Datum der letzten Änderung des aktuellsten Artikels.

⑤ Jetzt wird's interessant. Das Element `link` hat keinen Textinhalt, doch drei Attribute: `rel`, `type` und `href`. Der Wert `rel` enthält die Art des Links; `rel='alternate'` bedeutet, dass dies ein Link zu einer alternativen Darstellung dieses Feeds ist. `type='text/html'` heißt, dass dies ein Link zu einer HTML-Seite ist. Das Attribut `href` enthält das Ziel des Links.

Nun wissen wir also, dass dieser Feed zu einer Seite namens „dive into mark" gehört, die unter `http://www.diveintomark.org` verfügbar ist und zuletzt am 27. März 2009 aktualisiert wurde.

☞ Auch wenn die Reihenfolge der Elemente in manchen XML-Dokumenten eine Rolle spielt, so ist sie bei einem Atom-Feed bedeutungslos.

Auf die Metadaten des Feeds folgt eine Liste der aktuellsten Artikel. Ein Artikel sieht wie folgt aus:

```
<entry>
  <author>                                                               ①
    <name>Mark</name>
    <uri>http://diveintomark.org/</uri>
  </author>
  <title>Dive into history, 2009 edition</title>                         ②
  <link rel='alternate' type='text/html'                                 ③
        href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-
edition'/>
  <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>      ④
  <updated>2009-03-27T21:56:07Z</updated>                                ⑤
  <published>2009-03-27T17:20:42Z</published>
  <category scheme='http://diveintomark.org' term='diveintopython'/>     ⑥
  <category scheme='http://diveintomark.org' term='docbook'/>
  <category scheme='http://diveintomark.org' term='html'/>
  <summary type='html'>Putting an entire chapter on one page sounds      ⑦
    bloated, but consider this &amp;mdash; my longest chapter so far
    would be 75 printed pages, and it loads in under 5 seconds&amp;hellip;
    On dialup.</summary>
</entry>                                                                 ⑧
```

① Das Element `author` nennt den Autor des Artikels: irgendein Typ, den Sie unter `http://www.diveintomark.org` finden können. (Dieser Link entspricht dem alternativen Link in den Metadaten des Feeds, doch das muss nicht so sein. Viele Blogs haben mehrere Autoren, von denen jeder eine eigene persönliche Webseite besitzt.)

② Das Element `title` nennt den Titel des Artikels, „Dive into history, 2009 edition“.

③ Das Element `link` enthält die Adresse der HTML-Version dieses Artikels.

④ Einträge benötigen, genau wie Feeds, eine einmalige ID.

⑤ Einträge haben zwei Daten: das Datum der ersten Veröffentlichung (`published`) und das Datum der letzten Änderung (`updated`).

⑥ Einträge können einer beliebigen Zahl an Kategorien zugeordnet werden. Dieser Artikel ist unter `diveintopython`, `docbook` und `html` abgelegt.

⑦ Das Element `summary` enthält eine kurze Zusammenfassung des Artikels. (Es gibt auch ein Element namens `content`, das hier nicht gezeigt wird; mit diesem Element können Sie den kompletten Artikel in den Feed einfügen.) Dieses `summary`-Element hat außerdem das Atom-spezifische Attribut `type='html'`, das angibt, dass diese Zusammenfassung HTML ist und nicht einfacher Text. Das ist wichtig, da der Artikel HTML-spezifische Zeichen enthält (`&mdash;` und `&hellip;`), die als „–“ und „…“ – und nicht etwa direkt wie angegeben – dargestellt werden sollen.

⑧ Schließlich folgt das End-Tag des `entry`-Elements, welches das Ende der Metadaten dieses Artikels anzeigt.

13.4 XML parsen

Python kann XML-Dokumente auf unterschiedliche Arten verarbeiten. Es besitzt traditionelle DOM- und SAX-Parser, doch ich möchte mich auf eine andere Bibliothek namens ElementTree konzentrieren.

```
>>> import xml.etree.ElementTree as etree    ①
>>> tree = etree.parse('examples/feed.xml')  ②
>>> root = tree.getroot()                    ③
>>> root                                     ④
<Element {http://www.w3.org/2005/Atom}feed at cd1eb0>
```

① Die Bibliothek ElementTree ist Teil von Pythons Standardbibliothek und befindet sich in `xml.etree.ElementTree`.

② Der Haupteinstiegspunkt der ElementTree-Bibliothek ist die Funktion `parse()`, die einen Dateinamen oder ein dateiähnliches Objekt übernimmt. Diese Funktion verarbeitet das komplette Dokument auf einmal.

③ Die `parse()`-Funktion gibt ein Objekt zurück, welches das komplette Dokument repräsentiert. Dies ist nicht das Wurzelelement. Um eine Referenz auf das Wurzelelement zu erhalten, rufen Sie die Methode `getroot()` auf.

④ Wie erwartet ist das Wurzelelement das Element `feed` im Namensbereich `http://www.w3.org/2005/Atom`. Die Stringdarstellung dieses Objekts untermauert einen wichtigen Punkt: Ein XML-Element ist eine Kombination aus seinem Namensbereich und seinem Tagnamen (auch *lokaler Name* genannt). Jedes Element dieses Dokuments befindet sich im Atom-Namensbereich, also wird das Wurzelelement als `{http://www.w3.org/2005/Atom}feed` angezeigt.

☞ ElementTree stellt XML-Elemente als `{Namensbereich}lokalerName` dar. Dieses Format werden Sie in der ElementTree-API einige Male sehen und verwenden.

13.4.1 Elemente sind Listen

In der ElementTree-API verhält sich ein Element wie eine Liste. Die Listenelemente sind dessen Kindelemente.

```
# Fortsetzung des vorherigen Beispiels
>>> root.tag                          ①
'{http://www.w3.org/2005/Atom}feed'
>>> len(root)                         ②
8
>>> for child in root:                ③
...   print(child)                     ④
...
<Element {http://www.w3.org/2005/Atom}title at e2b5d0>
<Element {http://www.w3.org/2005/Atom}subtitle at e2b4e0>
<Element {http://www.w3.org/2005/Atom}id at e2b6c0>
<Element {http://www.w3.org/2005/Atom}updated at e2b6f0>
<Element {http://www.w3.org/2005/Atom}link at e2b4b0>
<Element {http://www.w3.org/2005/Atom}entry at e2b720>
<Element {http://www.w3.org/2005/Atom}entry at e2b510>
<Element {http://www.w3.org/2005/Atom}entry at e2b750>
```

① Da wir das vorherige Beispiel fortführen, ist das Wurzelelement immer noch `{http://www.w3.org/2005/Atom}feed`.

② Die „Länge“ des Wurzelelements ist die Anzahl der Kindelemente.

③ Sie können das Element selbst als Iterator verwenden, um alle seine Kindelemente zu durchlaufen.

④ An der Ausgabe können Sie sehen, dass es tatsächlich acht Kindelemente gibt: die Metadaten (`title`, `subtitle`, `id`, `updated` und `link`), gefolgt von den drei `entry`-Elementen.

Sie haben es sich vielleicht schon gedacht, doch ich möchte es noch einmal ausdrücklich erwähnen: Die Liste der Kindelemente enthält nur *direkte* Kindelemente. Jedes der `entry`-Elemente enthält eigene Kindelemente, die aber in dieser Liste nicht eingeschlossen sind. Sie wären in der Liste der Kindelemente jedes `entry`-Elements enthalten, aber nicht in der Liste der Kindelemente des Feeds. Es gibt verschiedene Möglichkeiten, unabhängig von der Verschachtelungstiefe Elemente zu finden; zwei dieser Möglichkeiten werden wir uns später in diesem Kapitel ansehen.

13.4.2 Attribute sind Dictionarys

XML ist nicht nur eine Ansammlung von Elementen; jedes Element kann außerdem auch seine eigenen Attribute besitzen. Haben Sie erstmal eine Referenz auf ein bestimmtes Element, können Sie dessen Attribute einfach als Dictionary abrufen.

```
# Fortsetzung des vorherigen Beispiels
>>> root.attrib                              ①
{'{http://www.w3.org/XML/1998/namespace}lang': 'en'}
>>> root[4]                                  ②
<Element {http://www.w3.org/2005/Atom}link at e181b0>
>>> root[4].attrib                           ③
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}
>>> root[3]                                  ④
<Element {http://www.w3.org/2005/Atom}updated at e2b4e0>
>>> root[3].attrib                           ⑤
{}
```

① Die Eigenschaft `attrib` ist ein Dictionary der Elementattribute. Der ursprüngliche Code war hier `<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>`. Das Präfix `xml:` verweist auf einen integrierten Namensbereich, den jedes XML-Dokument verwenden kann, ohne ihn deklarieren zu müssen.

② Das fünfte Kindelement – `[4]` in einer auf `0` basierenden Liste – ist das Element `link`.

③ Das `link`-Element besitzt drei Attribute: `href`, `type` und `rel`.

④ Das vierte Kindelement – `[3]` in einer auf `0` basierenden Liste – ist das Element `updated`.

⑤ Das `updated`-Element besitzt keine Attribute, `.attrib` ist hier daher ein leeres Dictionary.

13.5 Innerhalb eines XML-Dokuments nach Knoten suchen

Bisher haben wir mit diesem XML-Dokument „von oben nach unten“ gearbeitet. Wir haben mit dem Wurzelelement begonnen, seine Kindelemente abgerufen und so weiter. Doch die Verwendung von XML setzt häufig voraus, dass man bestimmte Elemente findet. Etree kann auch das.

```
>>> import xml.etree.ElementTree as etree
>>> tree = etree.parse('examples/feed.xml')
>>> root = tree.getroot()
>>> root.findall('{http://www.w3.org/2005/Atom}entry')    ①
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> root.tag
'{http://www.w3.org/2005/Atom}feed'
>>> root.findall('{http://www.w3.org/2005/Atom}feed')     ②
[]
>>> root.findall('{http://www.w3.org/2005/Atom}author')   ③
[]
```

① Die Methode `findall()` sucht alle Kindelemente, die zu der angegebenen Abfrage passen.

② Jedes Element – das Wurzelelement, genauso wie auch die Kindelemente – besitzt eine `findall()`-Methode. Sie sucht innerhalb der Kindelemente nach allen passenden Elementen. Doch warum gibt es keine Ergebnisse? Auch wenn es vielleicht nicht offensichtlich ist, so durchsucht diese Abfrage lediglich die Kindelemente. Da das Wurzelelement `feed` keine Kindelemente namens `feed` besitzt, gibt die Abfrage eine leere Liste zurück.

③ Dieses Ergebnis wird Sie vermutlich auch überraschen. Es ist ein Element namens `author` in diesem Dokument vorhanden; tatsächlich sind es sogar drei (eines in jedem `entry`). Doch diese `author`-Elemente sind keine *direkten* Kindelemente des Wurzelelements; es sind „Enkelelemente“ (Kindelemente von Kindelementen). Wollen Sie `author`-Elemente unabhängig von der Verschachtelungstiefe suchen, können Sie das tun, doch die Abfrage sieht dann geringfügig anders aus.

```
>>> tree.findall('{http://www.w3.org/2005/Atom}entry')    ①
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> tree.findall('{http://www.w3.org/2005/Atom}author')   ②
[]
```

① Aus Gründen der Bequemlichkeit besitzt das `tree`-Objekt (das von `etree.parse()` zurückgegeben wurde) einige Methoden, die die Methoden des Wurzelelements widerspiegeln. Die Ergebnisse sind dieselben, als hätten Sie `tree.getroot().findall()` aufgerufen.

② Vielleicht kommt es für Sie überraschend, dass diese Abfrage nicht die `author`-Elemente des Dokuments findet. Warum nicht? Weil dies lediglich die Kurzschreibweise für `tree.getroot().findall('{http://www.w3.org/2005/Atom} author')` ist, was bedeutet „suche alle `author`-Elemente, die Kindelemente des Wurzelelements sind“. Die `author`-Elemente sind keine Kindelemente des Wurzelelements; sie sind Kindelemente der `entry`-Elemente. Daher liefert die Abfrage keinerlei Treffer.

Es gibt auch eine Methode namens `find()`, die nur das erste passende Element zurückgibt. Dies ist nützlich, wenn Sie nur einen Treffer erwarten oder Sie sich nur für den ersten von mehreren möglichen Treffern interessieren.

```
>>> entries = tree.findall('{http://www.w3.org/2005/Atom}entry')          ①
>>> len(entries)
3
>>> title_element = entries[0].find('{http://www.w3.org/2005/Atom}title')  ②
>>> title_element.text
'Dive into history, 2009 edition'
>>> foo_element = entries[0].find('{http://www.w3.org/2005/Atom}foo')      ③
>>> foo_element
>>> type(foo_element)
<class 'NoneType'>
```

① Dies haben Sie bereits im vorherigen Beispiel gesehen. Hier werden alle `atom:entry`-Elemente gesucht.

② Die `find()`-Methode übernimmt eine ElementTree-Abfrage und gibt das erste passende Element zurück.

③ In diesem Eintrag existiert kein Element namens `foo`, also wird `None` zurückgegeben.

☞ Im Zusammenhang mit der `find()`-Methode unterläuft Programmierern häufig ein Fehler, der auch Ihnen irgendwann passieren wird. In einem booleschen Kontext verwendet, ergeben ElementTree-Objekte `False`, wenn sie keine Kindelemente besitzen (`len(element)` ist also `0`). Das bedeutet, dass `if element.find('...')` nicht etwa prüft, ob die `find()`-Methode ein passendes Element gefunden hat; es wird stattdessen geprüft, ob das passende Element irgendwelche Kindelemente besitzt! Zum Testen, ob die `find()`-Methode ein Element zurückgegeben hat, verwenden Sie `if element.find('...') is not None`.

Es *gibt* eine Möglichkeit nach Kindelementen, Enkelelementen und allen anderen Elementen mit beliebiger Verschachtelungstiefe zu suchen.

```
>>> all_links = tree.findall('//{http://www.w3.org/2005/Atom}link')  ①
>>> all_links
[<Element {http://www.w3.org/2005/Atom}link at e181b0>,
 <Element {http://www.w3.org/2005/Atom}link at e2b570>,
 <Element {http://www.w3.org/2005/Atom}link at e2b480>,
 <Element {http://www.w3.org/2005/Atom}link at e2b5a0>]
>>> all_links[0].attrib                                              ②
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[1].attrib                                              ③
{'href':'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[2].attrib
{'href':'http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-
mistress',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[3].attrib
{'href':'http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats',
 'type': 'text/html',
 'rel': 'alternate'}
```

① Diese Abfrage – `//{ http://www.w3.org/2005/Atom }link` – ähnelt unseren letzten Beispielen, abgesehen von den beiden Schrägstrichen am Anfang der Abfrage. Diese beiden Schrägstriche bedeuten „suche nicht nur direkte Kindelemente; ich will alle Elemente, egal wie tief sie verschachtelt sind“. Das Ergebnis ist daher eine Liste von vier `link`-Elementen und nicht nur von einem.

② Das erste Ergebnis ist ein direktes Kindelement des Wurzelelements. Wie Sie an den Attributen sehen können, ist dies der alternative Link, der auf die HTML-Version der vom Feed beschriebenen Webseite verweist.

③ Die anderen drei Ergebnisse sind alternative Links auf Eintragsebene. Jeder `entry` besitzt ein `link`-Kindelement, und aufgrund der beiden Schrägstriche am Anfang der Abfrage wird nach allen davon gesucht.

Die `findall()`-Methode von ElementTree ist sehr mächtig, doch die Syntax der Abfragen ist manchmal etwas seltsam. Offiziell bietet sie „begrenzte Unterstützung für XPath-Ausdrücke“. XPath ist ein W3C-Standard zur Abfrage von XML-Dokumenten. Die Abfragesyntax von ElementTree ähnelt XPath genug, um einfache Suchen durchzuführen, doch es ähnelt ihm so wenig, dass es Sie verärgern könnte, wenn Sie XPath bereits kennen. Sehen wir uns nun eine XML-Bibliothek eines Drittanbieters an, die die ElementTree-API um vollständige Xpath-Unterstützung erweitert.

13.6 Noch mehr XML

`lxml` ist eine Open-Source-Bibliothek, die auf dem beliebten *libxml2*-Parser aufbaut. Sie stellt eine zu 100% kompatible ElementTree-API bereit und erweitert diese durch vollständige XPath 1.0-Unterstützung und andere Feinheiten. Installationsprogramme für Windows sind verfügbar; Linux-Benutzer sollten immer versuchen, auf distributionseigene Werkzeuge wie `yum` oder `apt-get` zurückzugreifen, um vorkompilierte Anwendungen aus ihren Repositorys zu installieren. Andernfalls müssen Sie `lxml` von Hand installieren.

```
>>> from lxml import etree                      ①
>>> tree = etree.parse('examples/feed.xml')     ②
>>> root = tree.getroot()                       ③
>>> root.findall('{http://www.w3.org/2005/Atom}entry')   ④
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
```

① Einmal importiert, bietet `lxml` dieselbe API wie die integrierte ElementTree-Bibliothek.

② Die `parse()`-Funktion: wie bei ElementTree.

③ Die `getroot()`-Methode: auch wie bei ElementTree.

④ Die `findall()`-Methode: ganz genau so.

Bei großen XML-Dokumenten ist `lxml` erheblich schneller als die integrierte ElementTree-Bibliothek. Verwenden sie nur die ElementTree-API, wollen aber die schnellstmögliche Implementierung, können Sie versuchen `lxml` zu importieren und falls das nicht klappt auf die integrierte ElementTree-Bibliothek zurückgreifen.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

`lxml` ist jedoch weitaus mehr, als nur ein schnelleres ElementTree. Seine `findall()`-Methode unterstützt sehr viel komplexere Ausdrücke.

```
>>> import lxml.etree                                                    ①
>>> tree = lxml.etree.parse('examples/feed.xml')
>>> tree.findall('//{http://www.w3.org/2005/Atom}*[@href]')              ②
[<Element {http://www.w3.org/2005/Atom}link at eeb8a0>,
 <Element {http://www.w3.org/2005/Atom}link at eeb990>,
 <Element {http://www.w3.org/2005/Atom}link at eeb960>,
 <Element {http://www.w3.org/2005/Atom}link at eeb9c0>]
```

```
>>>
tree.findall("//{http://www.w3.org/2005/Atom}*[@href='http://diveintomark.org/']") ③
[<Element {http://www.w3.org/2005/Atom}link at eeb930>]
>>> NS = '{http://www.w3.org/2005/Atom}'
>>> tree.findall('//{NS}author[{NS}uri]'.format(NS=NS))                          ④
[<Element {http://www.w3.org/2005/Atom}author at eeba80>,
 <Element {http://www.w3.org/2005/Atom}author at eebba0>]
```

① In diesem Beispiel verwende ich `import lxml.etree` (statt beispielsweise `from lxml import etree`), um hervorzuheben, dass diese Eigenschaften `lxml`-spezifisch sind.

② Diese Abfrage sucht alle Elemente innerhalb des Atom-Namensbereichs, im kompletten Dokument, die ein `href`-Attribut besitzen. `//` am Anfang der Abfrage bedeutet „im kompletten Dokument (nicht nur als Kindelemente des Wurzelelements)“. `{http://www.w3.org/2005/Atom}` bedeutet „nur Elemente im Atom-Namensbereich“. `*` heißt „Elemente mit beliebigem lokalen Namen“. `[@href]` schließlich steht für „hat ein `href`-Attribut“.

③ Diese Abfrage sucht alle Atom-Elemente mit einem `href`-Attribut, dessen Wert `http://diveintomark.org/` ist.

④ Nachdem wir schnell eine Stringformatierung durchgeführt haben (da die Abfrage sonst sehr lang würde), sucht diese Abfrage nach `author`-Elementen, die ein `uri`-Element als Kindelement enthalten. Es werden lediglich zwei `author`-Elemente – das im ersten und das im zweiten `entry` – zurückgegeben. Das `author`-Element des dritten `entrys` enthält nur einen `name`, keinen `uri`.

Das ist Ihnen noch nicht genug? `lxml` integriert auch die Unterstützung für beliebige XPath 1.0-Ausdrücke ein. Ich werde hier nicht in die Tiefe gehen, da man über die XPath-Syntax ein eigenes Buch schreiben könnte. Ich werde Ihnen aber zeigen, wie sich XPath in `lxml` einfügt.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed.xml')
>>> NSMAP = {'atom': 'http://www.w3.org/2005/Atom'}                    ①
>>> entries = tree.xpath("//atom:category[@term='accessibility']/..",  ②
...     namespaces=NSMAP)
>>> entries                                                            ③
[<Element {http://www.w3.org/2005/Atom}entry at e2b630>]
>>> entry = entries[0]
>>> entry.xpath('./atom:title/text()', namespaces=NSMAP)               ④
['Accessibility is a harsh mistress']
```

① Zum Ausführen von XPath-Abfragen an Elemente in einem Namensbereich, müssen Sie eine Präfixzuweisung definieren. Dies geschieht einfach anhand eines Dictionarys.

② Das ist eine Xpath-Abfrage. Der Xpath-Ausdruck sucht nach `category`-Elementen (im Atom-Namensbereich), die ein `term`-Attribut mit dem Wert

`accessibility` enthalten. Doch dies ist nicht das eigentliche Ergebnis der Abfrage. Sehen Sie sich das Ende des Abfrage-Strings an; haben Sie das `/..` bemerkt? Es bedeutet „und gebe dann das Elternelement des gerade gefundenen `category`-Elements zurück“. Diese einzelne XPath-Abfrage sucht also nach allen Einträgen mit diesem Kindelement: `<category term='accessibility'>`.

③ Die Funktion `xpath()` gibt eine Liste von ElementTree-Objekten zurück. Im vorliegenden Dokument gibt es nur einen Eintrag mit einer `category`, deren `term` `accessibility` ist.

④ XPath-Ausdrücke geben aber nicht immer eine Liste von Elementen zurück. Das DOM (*Document Object Model*) eines geparsten XML-Dokuments enthält gar keine Elemente; es enthält Knoten. Je nach ihrem Typ können Knoten Elemente, Attribute oder sogar Textinhalt sein. Das Ergebnis einer XPath-Abfrage ist eine Liste von Knoten. Die vorliegende Abfrage gibt eine Liste von Textknoten zurück: den Textinhalt (`text()`) des `title`-Elements (`atom:title`), das ein Kindelement des aktuellen Elements ist (`./`).

13.7 XML erzeugen

Python unterstützt nicht nur das Parsen bereits vorhandener XML-Dokumente. Sie können XML-Dokumente auch von Grund auf neu erstellen.

```
>>> import xml.etree.ElementTree as etree
>>> new_feed = etree.Element('{http://www.w3.org/2005/Atom}feed',   ①
...     attrib={'{http://www.w3.org/XML/1998/namespace}lang': 'en'})②
>>> print(etree.tostring(new_feed))                                 ③
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

① Zur Erstellung eines neuen Elements müssen Sie die Klasse `Element` instanziieren. Sie übergeben dabei den Elementnamen (Namensbereich + Lokaler Name) als erstes Argument. Die vorliegende Anweisung erstellt so ein `feed`-Element im Atom-Namensbereich. Dies ist damit das Wurzelelement unseres neuen Dokuments.

② Um dem neu erstellten Element Attribute hinzuzufügen, übergeben Sie mithilfe des `attrib`-Arguments ein Dictionary aus Attributnamen und -werten. Beachten Sie dabei, dass der Attributname das ElementTree-Format haben solle, `{namensbereich}lokalerName`.

③ Sie können jedes beliebige Element (und seine Kindelemente) zu jeder Zeit serialisieren, indem Sie die `tostring()`-Funktion aus ElementTree verwenden.

Hat Sie diese Serialisierung überrascht? Die Art wie ElementTree in einem Namensbereich befindliche XML-Elemente serialisiert ist zwar technisch exakt, aber nicht optimal. Unser XML-Beispieldokument am Beginn dieses Kapitels definierte einen Standardnamensbereich (`xmlns= 'http://www.w3.org/2005/Atom'`). Die Definition eines Standardnamensbereichs ist sinnvoll bei Dokumen-

ten – wie Atom-Feeds –, bei denen sich jedes Element im selben Namensbereich befindet. So müssen Sie nur einmal den Namensbereich deklarieren und können fortan jedes Element nur mit seinem lokalen Namen deklarieren (`<feed>`, `<link>`, `<entry>`). Präfixe müssen nur angegeben werden, wenn Sie Elemente aus einem anderen Namensbereich deklarieren möchten.

Ein XML-Parser „sieht“ keinen Unterschied zwischen einem XML-Dokument mit einem Standardnamensbereich und einem XML-Dokument mit einem vorangestellten (unter Verwendung eines Präfixes) Namensbereich. Das sich ergebende DOM dieser Serialisierung:

```
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

ist somit identisch zum DOM dieser Serialisierung:

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

Der einzige Unterschied besteht darin, dass die zweite Serialisierung um einige Zeichen kürzer ist. Würden wir unseren kompletten Beispiel-Feed neu schreiben und dabei das Präfix `ns0:` in jedes Start- und End-Tag einfügen, so erhielten wir *4 Zeichen pro Start-Tag ∗ 79 Tags + 4 Zeichen für die Deklaration des Namensbereichs selbst*; das ergibt *320 Zeichen*. Bei UTF-8-Codierung wären das 320 zusätzliche Bytes. (Nach einer gzip-Komprimierung wären es nur noch 21 B, aber 21 B sind immerhin 21 B.) Vielleicht ist Ihnen das egal, doch bei einem Atom-Feed, der unter Umständen bei jeder Aktualisierung einige tausend Mal heruntergeladen wird, kommen so sehr viele gesparte Bytes zusammen.

Diese feinen Kontrollmöglichkeiten über die Serialisierung von in einem Namensbereich befindlichen Elementen bietet die integrierte ElementTree-Bibliothek nicht, doch `lxml` tut es.

```
>>> import lxml.etree
>>> NSMAP = {None: 'http://www.w3.org/2005/Atom'}                    ①
>>> new_feed = lxml.etree.Element('feed', nsmap=NSMAP)               ②
>>> print(lxml.etree.tounicode(new_feed))                            ③
<feed xmlns='http://www.w3.org/2005/Atom'/>
>>> new_feed.set('{http://www.w3.org/XML/1998/namespace}lang', 'en')④
>>> print(lxml.etree.tounicode(new_feed))
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

① Definieren Sie zunächst eine Namensbereichszuweisung als Dictionary. Dictionary-Werte sind Namensbereiche; Dictionary-Schlüssel sind Präfixe. Verwendet man None als Präfix, erhält man in der Folge einen Standardnamensbereich.

② Nun können Sie bei der Erstellung eines Elements das `lxml`-spezifische Argument `nsmap` übergeben, das dafür sorgt, dass `lxml` Ihre definierten Namensbereichspräfixe beachtet.

③ Wie erwartet definiert diese Serialisierung den Atom-Namensbereich als Standardnamensbereich und deklariert das `feed`-Element ohne ein Präfix.

④ Upps, wir haben vergessen, das `xml:lang`-Attribut einzufügen. Mit der `set()`-Methode können Sie jederzeit jedem beliebigen Element Attribute hinzufügen. Die Methode übernimmt zwei Argumente: den Attributnamen im Element-Tree-Format und den Attributwert. (Diese Methode ist nicht `lxml`-spezifisch. Der einzige `lxml`-spezifische Teil dieses Beispiels war das `nsmap`-Argument, das die Namensbereichspräfixe der serialisierten Ausgabe steuert.)

Sind XML-Dokumente auf ein Element pro Dokument beschränkt? Nein, natürlich nicht. Sie können ganz einfach Kindelemente erzeugen.

```
>>> title = lxml.etree.SubElement(new_feed, 'title',          ①
...     attrib={'type':'html'})                               ②
>>> print(lxml.etree.tounicode(new_feed))                     ③
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type='html'/></feed>
>>> title.text = 'dive into &hellip;'                         ④
>>> print(lxml.etree.tounicode(new_feed))                     ⑤
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type='html'>dive into
&amp;hellip;</title></feed>
>>> print(lxml.etree.tounicode(new_feed, pretty_print=True))  ⑥
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
<title type='html'>dive into&amp;hellip;</title>
</feed>
```

① Um ein Kindelement eines bereits vorhandenen Elements zu erstellen, instanziieren Sie die Klasse `SubElement`. Die benötigten Argumente sind das Elternelement (in unserem Fall `new_feed`) und der Name des neuen Elements. Da das Kindelement die Namensbereichszuordnung seines Elternelements erbt, müssen Sie den Namensbereich oder das Präfix hier nicht noch einmal deklarieren.

② Sie können auch ein Attribut-Dictionary übergeben. Schlüssel sind Attributnamen; Werte sind Attributwerte.

③ Das neue Element `title` wurde, wie erwartet, im Atom-Namensbereich erstellt und als Kindelement des `feed`-Elements eingefügt. Da das `title`-Element keinen Textinhalt und keine Kindelemente besitzt, serialisiert `lxml` es als ein leeres Element (mithilfe des `/>`-Kürzels).

④ Um den Textinhalt eines Elements festzulegen, geben Sie einfach einen Wert für die Eigenschaft `.text` an.

⑤ Nun ist das `title`-Element inklusive seinem Textinhalt serialisiert. Textinhalt, der *Kleiner-als-* oder *Und-*Zeichen enthält, muss beim Serialisieren mit Escape-Sequenzen versehen werden. `lxml` kümmert sich automatisch darum.

⑥ Setzen Sie `pretty_print` auf `True`, werden nach End-Tags und Start-Tags von Elementen mit Kindelementen aber ohne Textinhalt Zeilenumbrüche eingefügt. Die Ausgabe wird dadurch lesbarer.

☞ Vielleicht möchten Sie auch `xmlwitch`, eine weitere Drittanbieter-Bibliothek zur XML-Erzeugung ausprobieren. Sie macht umfassenden Gebrauch von `with`-Anweisungen, um den Code lesbarer zu machen.

13.8 Beschädigtes XML parsen

Der XML-Spezifikation zufolge sollen alle mit dieser übereinstimmenden XML-Parser eine „rigorose Fehlerbehandlung" einsetzen. D. h. sie sollen sofort Alarm schlagen, wenn sie einen Fehler in der *Wohlgeformtheit* des Dokuments finden. Fehler in der Wohlgeformtheit schließen falsch platzierte Start- und End-Tags, undefinierte Entitäten, ungültige Unicode-Zeichen und eine Reihe weiterer Regelverletzungen ein. Dies steht in krassem Kontrast zu anderen Formaten wie HTML – Ihr Browser zeigt eine Webseite auch dann an, wenn vergessen wurde, ein HTML-Tag zu schließen, oder ein Und-Zeichen als Escape-Sequenz anzugeben. (Es ist ein weit verbreiterter Irrtum, dass HTML keine Fehlerbehandlung besitzt. Die HTML-Fehlerbehandlung ist sogar sehr klar definiert, doch wesentlich komplizierter als „anhalten und Alarm schlagen".)

Es gibt Leute (dazu gehöre auch ich), die glauben, dass es ein Fehler der XML-Erfinder war, rigorose Fehlerbehandlung zu fordern. Verstehen Sie mich nicht falsch; ich kann verstehen, dass man der Verlockung erliegt, die Regeln zur Fehlerbehandlung zu vereinfachen. Doch das Konzept der „Wohlgeformtheit" ist in der Praxis schwieriger umzusetzen als man glaubt. Das gilt besonders für XML-Dokumente (wie Atom-Feeds), die im Internet über HTTP veröffentlicht werden. Auch wenn XML mittlerweile erwachsen geworden ist (die rigorose Fehlerbehandlung wurde 1997 standardisiert), zeigen Studien immer wieder, dass eine hohe Zahl der Atom-Feeds im Internet mit Fehlern bei der Wohlgeformtheit zu kämpfen haben.

Ich habe also sowohl theoretische als auch praktische Gründe, XML-Dokumente „um jeden Preis" zu parsen. Es soll nicht beim ersten Fehler in der Wohlgeformtheit angehalten und Alarm geschlagen werden. Wenn Sie dies auch so handhaben möchten, kann Ihnen `lxml` dabei helfen.

Hier ist ein Teil eines beschädigten XML-Dokuments. Ich habe den Fehler in der Wohlgeformtheit hervorgehoben.

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into &hellip;</title>
...
</feed>
```

Dies ist ein Fehler, weil `&hellip;` in XML nicht definiert ist. (Es ist in HTML definiert.) Wenn Sie versuchen diesen beschädigten Feed mit den Voreinstellungen zu parsen, erstickt `lxml` an dem undefinierten Gebilde.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed-broken.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lxml.etree.pyx", line 2693, in lxml.etree.parse
(src/lxml/lxml.etree.c:52591)
  File "parser.pxi", line 1478, in lxml.etree._parseDocument
(src/lxml/lxml.etree.c:75665)
  File "parser.pxi", line 1507, in
lxml.etree._parseDocumentFromURL (src/lxml/lxml.etree.c:75993)
  File "parser.pxi", line 1407, in lxml.etree._parseDocFromFile
(src/lxml/lxml.etree.c:75002)
  File "parser.pxi", line 965, in
lxml.etree._BaseParser._parseDocFromFile
(src/lxml/lxml.etree.c:72023)
  File "parser.pxi", line 539, in
lxml.etree._ParserContext._handleParseResultDoc
(src/lxml/lxml.etree.c:67830)
  File "parser.pxi", line 625, in lxml.etree._handleParseResult
(src/lxml/lxml.etree.c:68877)
  File "parser.pxi", line 565, in lxml.etree._raiseParseError
(src/lxml/lxml.etree.c:68125)
lxml.etree.XMLSyntaxError: Entity 'hellip' not defined, line 3,
column 28
```

Um das XML-Dokument trotz des Fehlers in der Wohlgeformtheit zu verarbeiten, müssen Sie einen eigenen XML-Parser erstellen.

```
>>> parser = lxml.etree.XMLParser(recover=True)                  ①
>>> tree = lxml.etree.parse('examples/feed-broken.xml', parser)  ②
>>> parser.error_log                                             ③
examples/feed-broken.xml:3:28:FATAL:PARSER:ERR_UNDECLARED_ENTITY: Entity
'hellip' not defined
>>> tree.findall('{http://www.w3.org/2005/Atom}title')
[<Element {http://www.w3.org/2005/Atom}title at ead510>]
>>> title = tree.findall('{http://www.w3.org/2005/Atom}title')[0]
>>> title.text                                                   ④
'dive into '
>>> print(lxml.etree.tounicode(tree.getroot()))                  ⑤
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into </title>
.
. [aufgrund der Übersichtlichkeit gekürzt]
.
```

① Zum Erstellen eines eigenen Parsers müssen Sie die Klasse `lxml.etree.XMLParser` instanziieren. Diese Klasse kann eine Reihe verschiedener Argumente übernehmen. Wird sind nur an einem interessiert: `recover`. Setzen wir dieses

Argument auf `True`, so wird der XML-Parser versuchen, sich von Fehlern in der Wohlgeformtheit zu „erholen“ (engl. *to recover*).

② Um nun ein XML-Dokument mit Ihrem eigenen Parser zu verarbeiten, übergeben Sie der `parse()`-Funktion das `parser`-Objekt als zweites Argument. Beachten Sie, dass `lxml` keine Ausnahme wegen des undefinierten `&hellip;` auslöst.

③ Der Parser protokolliert die gefundenen Fehler in der Wohlgeformtheit in einem Log. (Das trifft auch dann zu, wenn `recover` auf `True` gesetzt wurde.)

④ Da der Parser nicht weiß, wie er mit dem undefinierten `&hellip;` verfahren soll, hat er es einfach ausgelassen. Der Textinhalt des `title`-Elements wird zu `'dive into '`.

⑤ Wie Sie an der Serialisierung sehen können, wurde `&hellip;` nicht verschoben; es wurde einfach ausgelassen.

Ich wiederhole noch einmal, dass es *keine Garantie der Interoperabilität* „sich erholender“ XML-Parser gibt. Ein anderer Parser könnte `&hellip;` als HTML erkennen und es durch `&hellip;` ersetzen. Ist das „besser“? Vielleicht. Ist es „korrekter“? Nein, beides ist gleichermaßen falsch. Richtig wäre es (gemäß der XML-Spezifikation), anzuhalten und Alarm zu schlagen. Haben Sie entschieden, dass Sie das nicht wollen, müssen Sie selbst sehen, wie Sie zurechtkommen.

Kapitel 14
Python-Objekte serialisieren

14.1 Los geht's

Das Konzept der Serialisierung ist recht einfach. Stellen Sie sich vor, Sie hätten eine Datenstruktur im Speicher, die Sie sichern, wiederverwenden und an jemand anders senden wollen. Wie würden Sie das tun? Nun, das hängt ganz davon ab, wie Sie diese Datenstruktur speichern, wiederverwenden und an wen Sie sie senden wollen. Viele Computerspiele erlauben Ihnen das Speichern beim Beenden des Spiels und das Wiederaufnehmen des Spiels an dieser Stelle beim nächsten Start. (Viele andere Computerprogramme machen es genauso.) In diesem Fall muss eine Datenstruktur, die „Ihren Fortschritt bisher" festhält, beim Beenden auf der Festplatte gesichert und beim erneuten Starten geladen werden. Die Daten werden nur von dem Programm verwendet, von dem sie auch erstellt wurden. Sie werden nicht über ein Netzwerk geschickt und niemals von etwas anderem als dem erstellenden Programm gelesen. Was die Interoperabilitätsfrage betrifft, beschränkt sie sich darauf, sicherzustellen, dass spätere Programmversionen die von früheren Versionen geschriebenen Daten lesen können.

Für solche Fälle ist das Modul `pickle` (dt. *pökeln*, *einlegen*) vorzüglich geeignet. Es ist Teil der Standardbibliothek und somit immer verfügbar. Es ist außerdem schnell; der Großteil des Moduls ist in C geschrieben, genau wie der Python-Interpreter selbst. Es kann beliebig komplexe Datenstrukturen speichern.

Was kann das `pickle`-Modul speichern?

- Alle von Python unterstützen nativen Datentypen: boolesche Werte, Ganzzahlen, Fließkommazahlen, komplexe Zahlen, Strings, `bytes`-Objekte, Bytearrays und `None`.
- Listen, Tupel, Dictionarys und Sets, die eine Kombination der nativen Datentypen enthalten.
- Listen, Tupel, Dictionarys und Sets, die eine Kombination aus Listen, Tupeln, Dictionarys und Sets enthalten, die wiederum eine Kombination der nativen Datentypen enthalten (und so weiter, bis zu der von Python unterstützten größtmöglichen Verschachtelung).
- Funktionen, Klassen und Klasseninstanzen (mit Einschränkungen).

Sollte Ihnen das nicht genügen, können Sie das `pickle`-Modul auch erweitern.

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_14, © Springer-Verlag Berlin Heidelberg 2010

14.1.1 Eine kurze Bemerkung zu den Beispielen dieses Kapitels

In diesem Kapitel machen wir Gebrauch von zwei Python-Shells. Im Verlauf dieses Kapitels müssen Sie zwischen diesen hin- und herspringen, während ich Ihnen das `pickle`- und das `json`-Modul vorstelle.

Um uns die Arbeit zu erleichtern, öffnen wir die Python-Shell und definieren die folgende Variable:

```
>>> shell = 1
```

Lassen Sie das Fenster geöffnet. Öffnen Sie nun eine weitere Python-Shell und definieren Sie die folgende Variable:

```
>>> shell = 2
```

Im Verlauf dieses Kapitels benutze ich die `shell`-Variable, um die jeweils zur Anwendung kommende Python-Shell anzuzeigen.

14.2 Daten in einer `pickle`-Datei speichern

Das `pickle`-Modul arbeitet mit Datenstrukturen. Lassen Sie uns eine erstellen.

```
>>> shell                                                                    ①
1
>>> entry = {}                                                               ②
>>> entry['title'] = 'Dive into history, 2009 edition'
>>> entry['article_link'] = 'http://diveintomark.org/archives/2009/03/27/dive-into-
history-2009-edition'
>>> entry['comments_link'] = None
>>> entry['internal_id'] = b'\xDE\xD5\xB4\xF8'
>>> entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> entry['published'] = True
>>> import time
>>> entry['published_date'] = time.strptime('Fri Mar 27 22:20:42 2009')      ③
>>> entry['published_date']
time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20,
tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1)
```

① Geben Sie diesen Code in Python-Shell `#1` ein.

② Ich möchte gerne ein Dictionary erstellen, das irgendetwas Nützliches – wie einen Eintrag in einem Atom-Feed – repräsentiert. Doch ich will außerdem sicherstellen, dass es verschiedene Datentypen enthält, damit ich Ihnen zeigen kann, was das `pickle`-Modul alles beherrscht. Interpretieren Sie nicht zu viel in diese Werte hinein.

③ Das Modul time enthält eine Datenstruktur (time_struct) zur Darstellung eines Zeitpunktes (auf eine Millisekunde genau) und Funktionen zur Manipulation von Zeitstrukturen. Die Funktion strptime() übernimmt einen formatierten String und wandelt diesen in eine time_struct um. Dieser String hat das Standardformat, doch mithilfe von Formatangaben können Sie das Format beeinflussen.

Das ist ein sehr schönes Dictionary. Lassen Sie es uns in einer Datei speichern.

```
>>> shell                                          ①
1
>>> import pickle
>>> with open('entry.pickle', 'wb') as f:         ②
...     pickle.dump(entry, f)                      ③
...
```

① Wir befinden uns immer noch in Python-Shell #1.

② Verwenden Sie die open()-Funktion zum Öffnen einer Datei. Setzen Sie den Dateimodus auf 'wb', um die Datei zum Schreiben im Binärmodus zu öffnen. Schließen Sie dies in eine with-Anweisung ein, um sicherzustellen, dass die Datei automatisch geschlossen wird, wenn Sie sie fertig bearbeitet haben.

③ Die Funktion dump() des pickle-Moduls übernimmt eine serialisierbare Datenstruktur, serialisiert diese unter Verwendung der aktuellsten Version des „Pökel-Protokolls“ als binäres, Python-spezifisches Format und speichert sie in einer geöffneten Datei.

Dieser letzte Satz ist sehr wichtig.

- Das pickle-Modul übernimmt eine Datenstruktur und speichert diese in einer Datei.
- Um dies zu erreichen, wird die Datenstruktur unter Verwendung eines bestimmten Datenformats serialisiert.
- Dieses „Pökel-Protokoll“ ist Python-spezifisch; die *Kompatibilität* mit anderen Sprachen ist *nicht garantiert*. Sie könnten mit der entry.pickle-Datei vermutlich in Perl, PHP, Java oder einer anderen Sprache nichts Sinnvolles anfangen.
- Nicht jede von Pythons Datenstrukturen kann vom pickle-Modul serialisiert werden. Das „Pökel-Protokoll“ hat sich einige Male beim Hinzufügen neuer Datentypen zu Python verändert, doch es gibt immer noch Grenzen für seine Verwendung.
- Aufgrund dieser Änderungen kann selbst die Kompatibilität zwischen zwei verschiedenen Python-Versionen *nicht garantiert* werden. Neuere Versionen unterstützen zwar die älteren Formate zur Serialisierung, doch ältere Versionen unterstützen nicht die neueren Formate (da sie die neuen Datentypen nicht unterstützen).
- Sofern Sie es nicht anders angeben, verwenden die Funktionen des pickle-Moduls die neueste Version des „Pökel-Protokolls“. Dies stellt sicher, dass Sie die größtmögliche Flexibilität bei den zur Serialisierung verfügbaren Datentypen haben. Allerdings bedeutet es auch, dass die entstandene Datei nicht von älteren Python-Versionen gelesen werden kann, die die neueste Version des „Pökel-Protokolls“ nicht unterstützen.

- Die neueste Version des „Pökel-Protokolls“ nutzt ein Binärformat. Öffnen Sie Ihre `pickle`-Dateien also im Binärmodus, sonst werden Ihre Daten während des Schreibens zerstört.

14.3 Daten aus einer `pickle`-Datei lesen

Wechseln Sie nun auf die zweite Python-Shell – d. h. nicht die, in der Sie das `entry`-Dictionary erstellt haben.

```
>>> shell                                    ①
2
>>> entry                                    ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import pickle
>>> with open('entry.pickle', 'rb') as f:    ③
...     entry = pickle.load(f)               ④
...
>>> entry                                    ⑤
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link':
 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22,
tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
```

① Dies ist Python-Shell `#2`.

② Hier wurde keine `entry`-Variable definiert. Sie haben eine `entry`-Variable in Python-Shell `#1` definiert, doch dies ist eine völlig andere Umgebung mit einem eigenen Status.

③ Öffnen Sie die in Python-Shell `#1` erstellte `entry.pickle`-Datei. Das `pickle`-Modul nutzt ein binäres Datenformat, weshalb Sie `pickle`-Dateien immer im Binärmodus öffnen sollten.

④ Die Funktion `pickle.load()` übernimmt ein Streamobjekt, liest die serialisierten Daten aus diesem Stream, erstellt ein neues Python-Objekt, baut die serialisierten Daten in diesem neuen Objekt wieder auf und gibt das Objekt dann zurück.

⑤ Die `entry`-Variable ist nun ein Dictionary mit bekannten Schlüsseln und Werten.

Der `pickle.dump()`/`pickle.load()`-Zyklus ergibt eine neue Datenstruktur, die mit der ursprünglichen übereinstimmt.

```
>>> shell                                    ①
1
>>> with open('entry.pickle', 'rb') as f:    ②
...     entry2 = pickle.load(f)              ③
...
>>> entry2 == entry                          ④
True
>>> entry2 is entry                          ⑤
False
>>> entry2['tags']                           ⑥
('diveintopython', 'docbook', 'html')
>>> entry2['internal_id']
b'\xDE\xD5\xB4\xF8'
```

① Gehen Sie zurück zu Python-Shell #1.

② Öffnen Sie die entry.pickle-Datei.

③ Laden Sie die serialisierten Daten in eine neue Variable: entry2.

④ Python bestätigt, dass die beiden Dictionarys, entry und entry2, gleich sind. In dieser Shell haben Sie entry von Grund auf aufgebaut. Sie haben mit einem leeren Dictionary begonnen und per Hand bestimmten Schlüsseln bestimmte Werte zugewiesen. Dann haben Sie dieses Dictionary serialisiert und in der entry.pickle-Datei gespeichert. Nun haben Sie die serialisierten Daten aus dieser Datei gelesen und so eine *perfekte Kopie* der ursprünglichen Datenstruktur erstellt.

⑤ Gleich ist nicht identisch. Sie haben eine *perfekte Kopie* der ursprünglichen Datenstruktur erstellt. Aber dennoch ist es lediglich eine Kopie.

⑥ Aus Gründen, die erst später in diesem Kapitel klar werden, möchte ich darauf hinweisen, dass der Wert des Schlüssels 'tags' ein Tupel und der Wert des Schlüssels 'internal_id' ein bytes-Objekt ist.

14.4 pickle ohne Datei

Die Beispiele des letzten Abschnitts haben Ihnen gezeigt, wie man serialisierte Python-Objekte direkt in einer Datei auf der Festplatte ablegt. Doch was, wenn Sie gar keine Datei wollen oder brauchen? Sie können die serialisierten Daten auch in einem bytes-Objekt im Speicher ablegen.

```
>>> shell
1
>>> b = pickle.dumps(entry)      ①
>>> type(b)                      ②
<class 'bytes'>
>>> entry3 = pickle.loads(b)     ③
>>> entry3 == entry              ④
True
```

① Die Funktion `pickle.dumps()` (beachten Sie das `'s'` am Ende des Funktionsnamens) führt dieselbe Serialisierung wie die `pickle.dump()`-Funktion aus; statt aber das Streamobjekt zu übernehmen und die serialisierten Daten in eine Datei auf der Festplatte zu schreiben, gibt sie die serialisierten Daten einfach zurück.

② Da das „Pökel-Protokoll“ ein binäres Datenformat nutzt, gibt die `pickle.dumps()`-Funktion ein `bytes`-Objekt zurück.

③ Die Funktion `pickle.loads()` (beachten Sie auch hier das `'s'` am Ende des Funktionsnamens) führt dieselbe Deserialisierung aus wie die `pickle.load()`-Funktion; statt aber ein Streamobjekt zu übernehmen und die serialisierten Daten aus einer Datei zu lesen, übernimmt sie ein `bytes`-Objekt, das, so wie das von der `pickle.sumps()`-Funktion zurückgegebene, serialisierte Daten enthält.

④ Das Ergebnis ist dasselbe: eine perfekte Kopie des ursprünglichen Dictionarys.

14.5 Bytes und Strings zeigen ein weiteres Mal ihre hässlichen Fratzen

Das „Pökel-Protokoll“ existiert bereits viele Jahre lang und es ist über die Zeit gereift, wie auch Python gereift ist. Es gibt mittlerweile vier verschiedene Versionen des „Pökel-Protokolls“.

- Python 1.x besaß zwei „Pökel-Protokolle“, eines in einem textbasierten Format („Version 0“) und eines in einem binären Format („Version 1“).
- Python 2.3 führte ein neues „Pökel-Protokoll“ ein („Version 2“), das mit der neuen Funktionalität in Pythons Klassenobjekten umgehen konnte.
- Python 3.0 führte ein weiteres „Pökel-Protokoll“ ein („Version 3“), das ausdrücklich `bytes`-Objekte und Bytearrays unterstützt. Es hat ein binäres Format.

Schauen Sie doch, der Unterschied zwischen Bytes und Strings zeigt wieder einmal seine hässliche Fratze. (Sollten Sie nun überrascht sein, haben Sie nicht aufgepasst.) Praktisch bedeutet dies, dass Python 3 zwar Daten lesen kann, die mit Version 2 des Protokolls „gepökelt“ wurden, Python 2 aber keine Daten lesen kann, die mit Version 3 des Protokolls „gepökelt“ wurden.

14.6 `pickle`-Dateien debuggen

Wie sieht das „Pökel-Protokoll“ überhaupt aus? Verlassen wir für einen Moment die Python-Shell und sehen uns die von uns erstellte `entry.pickle`-Datei an.

```
you@localhost:~/diveintopython3/examples$ ls -l entry.pickle
-rw-r--r-- 1 you  you  358 Aug  3 13:34 entry.pickle
you@localhost:~/diveintopython3/examples$ cat entry.pickle
comments_linkqNXtagsqXdiveintopythonqXdocbookqXhtmlq?qX publishedq?
XlinkXJhttp://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition
q   Xpublished_dateq
ctime
struct_time
?qRqXtitleqXDive into history, 2009 editionqu.
```

Das war nicht gerade wahnsinnig hilfreich. Sie können zwar die Strings erkennen, aber die anderen Datentypen werden zu nicht-druckbaren (oder zumindest nicht-lesbaren) Zeichen. Die einzelnen Felder werden nicht durch Tabulatoren oder Leerzeichen begrenzt. Dies ist kein Format, das Sie gerne selbst debuggen würden.

```
>>> shell
1
>>> import pickletools
>>> with open('entry.pickle', 'rb') as f:
...     pickletools.dis(f)
    0: \x80 PROTO      3
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: X        BINUNICODE 'published_date'
   25: q        BINPUT     1
   27: c        GLOBAL     'time struct_time'
   45: q        BINPUT     2
   47: (        MARK
   48: M            BININT2    2009
   51: K            BININT1    3
   53: K            BININT1    27
   55: K            BININT1    22
   57: K            BININT1    20
   59: K            BININT1    42
   61: K            BININT1    4
   63: K            BININT1    86
   65: J            BININT     -1
   70: t            TUPLE      (MARK at 47)
   71: q        BINPUT     3
   73: }        EMPTY_DICT
   74: q        BINPUT     4
   76: \x86     TUPLE2
   77: q        BINPUT     5
   79: R        REDUCE
   80: q        BINPUT     6
   82: X        BINUNICODE 'comments_link'
  100: q        BINPUT     7
  102: N        NONE
  103: X        BINUNICODE 'internal_id'
  119: q        BINPUT     8
  121: C        SHORT_BINBYTES 'Þ\xd5´ø'
  127: q        BINPUT     9
  129: X        BINUNICODE 'tags'
  138: q        BINPUT     10
  140: X        BINUNICODE 'diveintopython'
  159: q        BINPUT     11
  161: X        BINUNICODE 'docbook'
  173: q        BINPUT     12
```

```
  175: X        BINUNICODE 'html'
  184: q        BINPUT     13
  186: \x87     TUPLE3
  187: q        BINPUT     14
  189: X        BINUNICODE 'title'
  199: q        BINPUT     15
  201: X        BINUNICODE 'Dive into history, 2009 edition'
  237: q        BINPUT     16
  239: X        BINUNICODE 'article_link'
  256: q        BINPUT     17
  258: X        BINUNICODE 'http://diveintomark.org/archives/2009/03/27/dive-into-
history-2009-edition'
  337: q        BINPUT     18
  339: X        BINUNICODE 'published'
  353: q        BINPUT     19
  355: \x88     NEWTRUE
  356: u        SETITEMS   (MARK at 5)
  357: .    STOP
highest protocol among opcodes = 3
```

Der interessanteste Teil dieser Disassemblierung ist die letzte Zeile, da diese die zum Speichern der Datei verwendete Version des Protokolls enthält. Das Protokoll enthält keine besondere Versionskennzeichnung. Um herauszufinden, welche Version des Protokolls zum Speichern einer `pickle`-Datei verwendet wurde, müssen Sie sich die sogenannten „*opcodes*“ in der „gepökelten“ Datei ansehen und wissen, welche *opcodes* in welcher Version des Protokolls hinzugekommen sind. Die Funktion `pickle.dis()` macht genau das und zeigt dann das Ergebnis in der letzten Zeile der Disassemblierung an. Hier eine Funktion, die nur die Versionsnummer zurückgibt, ohne etwas anzuzeigen:

```
import pickletools

def protocol_version(file_object):
    maxproto = -1
    for opcode, arg, pos in pickletools.genops(file_object):
        maxproto = max(maxproto, opcode.proto)
    return maxproto
```

Und hier nun in Aktion:

```
>>> import pickleversion
>>> with open('entry.pickle', 'rb') as f:
...     v = pickleversion.protocol_version(f)
>>> v
3
```

14.7 Serialisierte Python-Objekte in anderen Sprachen lesbar machen

Das vom `pickle`-Modul eingesetzte Datenformat ist Python-spezifisch. Es wird gar nicht erst der Versuch unternommen, es zu anderen Sprachen kompatibel zu machen. Wenn Sie sprachübergreifende Kompatibilität benötigen, müssen Sie sich andere Serialisierungsformate ansehen. Eines dieser Formate ist *JSON*. „JSON" steht für „JavaScript Object Notation"; doch lassen Sie sich von dem Namen nicht täuschen – JSON ist ausdrücklich für die Verwendung unter verschiedenen Programmiersprachen ausgelegt.

Die Standardbibliothek von Python 3 enthält ein `json`-Modul. Wie das `pickle`-Modul besitzt auch das `json`-Modul Funktionen zum Serialisieren von Datenstrukturen, Speichern der serialisierten Daten auf der Festplatte, Laden der serialisierten Daten von der Festplatte und Deserialisieren der Daten in ein neues Python-Objekt hinein. Doch es gibt auch einige wesentliche Unterschiede. Zuallererst ist das JSON-Datenformat textbasiert, nicht binär. *RFC 4627* definiert das JSON-Format und gibt an, wie verschiedene Arten von Daten als Text codiert werden müssen. Ein boolescher Wert wird so beispielsweise entweder als `'false'` oder als `'true'` gespeichert. Bei allen JSON-Werten wird zwischen Groß- und Kleinschreibung unterschieden.

Zweitens stehen wir, wie bei allen textbasierten Formaten, vor dem Whitespace-Problem. JSON erlaubt beliebig viel Whitespace (Leerzeichen, Tabulatoren, Wagenrückläufe und Zeilenvorschübe) zwischen den Werten. Dieser Whitespace ist „unbedeutend"; JSON-Encoder können so viel oder so wenig Whitespace einfügen, wie sie wollen, denn JSON-Decoder müssen den Whitespace zwischen den Werten ignorieren. Sie können Ihre JSON-Daten also übersichtlich gestalten, indem Sie Werte innerhalb anderer Werte mit Einzügen verschachteln, so dass Sie die Daten in einem Browser oder Texteditor lesen können. Pythons `json`-Modul bietet Optionen, um diese Gestaltung während des Codierens durchzuführen.

Drittens stehen wir vor dem immerwährenden Problem der Zeichencodierung. JSON codiert Werte als einfachen Text, doch wie Sie wissen, gibt es so etwas wie „einfachen Text" (engl. *plain text*) nicht. (siehe *Kap. 5*) JSON muss in einer Unicode-Codierung gespeichert werden (UTF-32, UTF-16 oder, standardmäßig, UTF-8). Abschnitt 3 des *RFC 4627* gibt an, wie man bestimmen kann, welche Codierung verwendet wird.

14.8 Daten in einer JSON-Datei speichern

JSON sieht einer manuell in JavaScript definierten Datenstruktur auffallend ähnlich. Das kommt nicht von ungefähr; Sie können sogar JavaScripts `eval()`-Funktion zum „Decodieren" JSON-serialisierter Daten verwenden. (Die üblichen Einschränkungen bezüglich unsicherer Eingaben treffen zwar auch hier zu, doch es

geht darum, dass JSON gültiges JavaScript ist.) Aus diesem Grund kommt JSON Ihnen vielleicht bekannt vor.

```
>>> shell
1
>>> basic_entry = {}                                                    ①
>>> basic_entry['id'] = 256
>>> basic_entry['title'] = 'Dive into history, 2009 edition'
>>> basic_entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> basic_entry['published'] = True
>>> basic_entry['comments_link'] = None
>>> import json
>>> with open('basic.json', mode='w', encoding='utf-8') as f:    ②
...     json.dump(basic_entry, f)                                       ③
```

① Wir werden eine neue Datenstruktur erstellen, statt die bereits vorhandene `entry`-Datenstruktur wiederzuverwenden. Später in diesem Kapitel werden wir erfahren, was passiert, wenn wir versuchen, die komplexere Datenstruktur in JSON zu codieren.

② JSON ist ein textbasiertes Format, weshalb Sie diese Datei im Textmodus öffnen und eine Zeichencodierung angeben müssen. Mit UTF-8 können Sie nie falsch liegen.

③ Wie das `pickle`-Modul definiert auch das `json`-Modul eine `dump()`-Funktion, die eine Datenstruktur und ein beschreibbares Streamobjekt übernimmt. Die `dump()`-Funktion serialisiert die Datenstruktur und schreibt sie in ein Streamobjekt. Führen wir dies innerhalb einer `with`-Anweisung aus, stellen wir sicher, dass die Datei geschlossen wird, wenn wir fertig mit ihr sind.

Wie sieht die sich ergebende JSON-Serialisierung also nun aus?

```
you@localhost:~/diveintopython3/examples$ cat basic.json
{"published": true, "tags": ["diveintopython", "docbook",
"html"], "comments_link": null,
"id": 256, "title": "Dive into history, 2009 edition"}
```

Das ist doch wesentlich lesbarer als eine `pickle`-Datei. Doch JSON kann auch beliebigen Whitespace zwischen den Werten enthalten und das `json`-Modul stellt uns eine einfache Möglichkeit zur Nutzung dieses Vorteils bereit, so dass wir noch lesbarere JSON-Dateien erstellen können.

```
>>> shell
1
>>> with open('basic-pretty.json', mode='w', encoding='utf-8') as f:
...     json.dump(basic_entry, f, indent=2)     ①
```

① Übergeben Sie der `json.dump()`-Funktion einen `indent`-Parameter (dt. *Einzug*), gestaltet sie die JSON-Datei lesbarer, erhöht aber dadurch die Dateigröße. Der `indent`-Parameter ist eine Ganzzahl. `0` bedeutet „setze jeden Wert in eine eigene Zeile“. Eine Zahl größer als `0` bedeutet „setze jeden Wert auf eine eigene Zeile und füge einen Einzug aus so vielen Leerzeichen wie angegeben ein“.

Das Ergebnis sieht so aus:

```
you@localhost:~/diveintopython3/examples$ cat basic-pretty.json
{
  "published": true,
  "tags": [
    "diveintopython",
    "docbook",
    "html"
  ],
  "comments_link": null,
  "id": 256,
  "title": "Dive into history, 2009 edition"
}
```

14.9 Entsprechungen der Python-Datentypen in JSON

Da JSON nicht Python-spezifisch ist, gibt es einige Diskrepanzen in der Deckung mit Pythons Datentypen. Manche unterscheiden sich nur im Namen, doch zwei wichtige Datentypen Pythons fehlen ganz. Schauen Sie doch mal, ob Sie sie finden (Tab. 14.1).

Haben Sie bemerkt was fehlt? Tupel und Bytes! JSON besitzt einen *Array*-Datentyp, den das `json`-Modul Pythons *Liste* zuordnet. JSON hat aber keinen eigenen Typ für „eingefrorene Arrays“ (Tupel). Und während JSON Strings ausgezeichnet unterstützt, bietet es *keine* Unterstützung für `bytes`-Objekte oder Bytearrays.

Tab. 14.1 Deckung der Datentypen

JSON	Python 3
Objekt	Dictionary
Array	Liste
String	String
Ganzzahl	Ganzzahl
Reelle Zahl	Fließkommazahl
`true`	`True`
`false`	`False`
`null`	`None`

14.10 Von JSON nicht unterstützte Datentypen serialisieren

Auch wenn JSON keine integrierte Unterstützung für Bytes bietet, heißt das nicht, dass Sie keine `bytes`-Objekte serialisieren können. Das `json`-Modul stellt Erweiterungen zum Codieren und Decodieren unbekannter Datentypen zur Verfügung. (Mit „unbekannt“ meine ich „in JSON nicht definiert“. Offensichtlich kennt das `json`-Modul Bytearrays, doch es wird von den Beschränkungen der JSON-Spezifikation eingeengt.) Möchten Sie Bytes oder andere von JSON von Haus aus nicht unterstützte Datentypen codieren, müssen Sie eigene Encoder und Decoder für diese Typen bereitstellen.

```
>>> shell
1
>>> entry                                                    ①
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-
history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27,
tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
>>> import json
>>> with open('entry.json', 'w', encoding='utf-8') as f:  ②
...     json.dump(entry, f)                                ③
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "C:\Python31\lib\json\encoder.py", line 170, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: b'\xDE\xD5\xB4\xF8' is not JSON serializable
```

① Also gut, sehen wir uns die `entry`-Datenstruktur noch einmal an. In ihr ist alles enthalten: ein boolescher Wert, ein `None`-Wert, ein String, ein Tupel von Strings, ein `bytes`-Objekt und eine `time`-Struktur.

② Ich habe bereits darauf hingewiesen, doch ich kann es gar nicht oft genug sagen: JSON ist ein textbasiertes Format. Öffnen Sie JSON-Dateien immer im Textmodus mit der Zeichencodierung UTF-8.

③ Hm, das ist nicht gut. Was ist passiert?

Sehen Sie sich an, was passiert ist: die `json.dump()`-Funktion hat versucht, das `bytes`-Objekt `b'\xDE\xD5\xB4\xF8'` zu serialisieren, doch der Versuch schlug fehl, da JSON `bytes`-Objekte nicht unterstützt. Wenn das Speichern von Bytes aber nun für Sie wichtig ist, können Sie Ihr eigenes „Mini-Serialisierungsformat" definieren.

```
def to_json(python_object):                                          ①
    if isinstance(python_object, bytes):                             ②
        return {'__class__': 'bytes',
                '__value__': list(python_object)}                    ③
    raise TypeError(repr(python_object) + ' is not JSON serializable')  ④
```

① Zum Definieren Ihres eigenen „Mini-Serialisierungsformats" für einen Datentyp, den JSON von Haus aus nicht unterstützt, definieren Sie einfach eine Funktion, die ein Python-Objekt als Parameter übernimmt. Dieses Python-Objekt ist das Objekt, das die `json.dump()`-Funktion selbst nicht serialisieren kann – in diesem Fall also das `bytes`-Objekt `b'\xDE\xD5\xB4\xF8'`.

② Ihre eigene Serialisierungsfunktion sollte den Typ des Python-Objekts prüfen, das ihr von der `json.dump()`-Funktion übergeben wurde. Wenn Ihre Funktion lediglich einen Datentyp serialisiert ist das nicht unbedingt nötig, doch so wird klar, welchen Fall Ihre Funktion abdeckt. Außerdem wird auch eine Erweiterung der Funktion einfacher, falls Sie später Serialisierungen für weitere Datentypen hinzufügen müssen.

③ Im vorliegenden Fall habe ich mich dazu entschieden, das `bytes`-Objekt in ein Dictionary umzuwandeln. Der Schlüssel `__class__` wird den ursprünglichen Datentyp enthalten (als String: `'bytes'`), der Schlüssel `__value__` den eigentlichen Wert. Natürlich kann das kein `bytes`-Objekt sein; es geht ja gerade darum, dass es in etwas umgewandelt wird, das von JSON serialisiert werden kann. Ein `bytes`-Objekt ist lediglich eine Folge von Ganzzahlen; jede dieser Ganzzahlen liegt irgendwo im Bereich von `0-255`. Wir können die `list()`-Funktion verwenden, um das `bytes`-Objekt in eine Liste aus Ganzzahlen zu konvertieren. `b'\xDE\xD5\xB4\xF8'` wird so zu `[222, 213, 180, 248]`. (Rechnen Sie nach! Es stimmt! Das hexadezimale Byte `\xDE` ist dezimal `222`, `\xD5` ist `213` usw.)

④ Diese Zeile ist wichtig. Die Datenstruktur, die Sie serialisieren, könnte Datentypen enthalten, die weder der ursprüngliche JSON-Serialisierer, noch Ihr eigener Serialisierer verarbeiten können. In einem solchen Fall muss Ihr Serialisierer einen `TypeError` auslösen, damit die `json.dump()`-Funktion erfährt, dass Ihr eigener Serialisierer den Datentyp nicht unterstützt.

Das war's! Mehr müssen Sie nicht tun. Diese Serialisierungsfunktion gibt ein Dictionary zurück, keinen String. Sie übernehmen nicht die ganze Arbeit. Sie wandeln das Objekt nur in einen von JSON unterstützten Datentyp um. Die `json.dump()`-Funktion wird den Rest erledigen.

```
>>> shell
1                                                                    ①
>>> import customserializer                                          ②
>>> with open('entry.json', 'w', encoding='utf-8') as f:             ③
...     json.dump(entry, f, default=customserializer.to_json)
...
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
    json.dump(entry, f, default=customserializer.to_json)
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "/Users/pilgrim/diveintopython3/examples/customserializer.py", line 12, in
to_json
    raise TypeError(repr(python_object) + ' is not JSON serializable')   ④
TypeError: time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22,
tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1) is not JSON serializable
```

① Im Modul `customserializer` haben Sie im vorherigen Beispiel die `to_json()`-Funktion definiert.

② Textmodus, UTF-8-Codierung, bla bla bla. (Sie werden es vergessen! Ich vergesse es manchmal! Alles geht solange gut, bis zu dem Moment, wenn es fehlschlägt; und dann schlägt es richtig fehl.)

③ Das ist der wichtige Teil: Um Ihre eigene Konvertierungsfunktion mit der `json.dump()`-Funktion zu verbinden, müssen Sie dieser Ihre Funktion im Parameter `default` übergeben. (Hurra, alles in Python ist ein Objekt!)

④ Na gut, das hat nicht wirklich funktioniert. Doch sehen Sie sich die Ausnahme an. Die `json.dump()`-Funktion beschwert sich nicht länger darüber, dass sie das `bytes`-Objekt nicht serialisieren kann. Jetzt beschwert sie sich über ein völlig anderes Objekt: das `time.struct_time`-Objekt.

Auch wenn das Auslösen einer anderen Ausnahme nicht wie ein Fortschritt wirkt, ist es dennoch einer. Wir benötigen nur noch eine Anpassung, um weiterzukommen.

```
import time

def to_json(python_object):
    if isinstance(python_object, time.struct_time):          ①
        return {'__class__': 'time.asctime',
                '__value__': time.asctime(python_object)}    ②
    if isinstance(python_object, bytes):
```

```
        return {'__class__': 'bytes',
                '__value__': list(python_object)}
    raise TypeError(repr(python_object) + ' is not JSON serializable')
```

① Wir müssen innerhalb unserer bestehenden Funktion `customserializer.to_json()` prüfen, ob das Python-Objekt (das, mit dem die `json.dump()`-Funktion ein Problem hat) ein `time.struct_time`-Objekt ist.

② Wir machen also etwas, was wir auch schon mit dem `bytes`-Objekt getan haben: Wir konvertieren das `time.struct_time`-Objekt in ein Dictionary, das nur Werte enthält, die in JSON serialisiert werden können. Im vorliegenden Fall ist es am einfachsten, das Objekt mithilfe der Funktion `time.asctime()` in einen String umzuwandeln. Die `time.asctime()`-Funktion wandelt dieses hässliche `time.struct_time`-Objekt in den String `'Fri Mar 27 22:20:42 2009'` um.

Mit diesen beiden selbstdefinierten Umwandlungen sollte sich die komplette `entry`-Datenstruktur ohne weitere Probleme in JSON serialisieren lassen.

```
>>> shell
1
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f, default=customserializer.to_json)
...
```

Die entstandene JSON-Datei:

```
you@localhost:~/diveintopython3/examples$ ls -l example.json
-rw-r--r-- 1 you  you  391 Aug  3 13:34 entry.json
you@localhost:~/diveintopython3/examples$ cat example.json
{"published_date": {"__class__": "time.asctime", "__value__": "Fri Mar 27 22:20:42
2009"},
"comments_link": null, "internal_id": {"__class__": "bytes", "__value__": [222, 213,
180, 248]},
"tags": ["diveintopython", "docbook", "html"], "title": "Dive into history, 2009 edi-
tion",
"article_link": "http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-
edition",
"published": true}
```

14.11 Daten aus einer JSON-Datei laden

Wie das `pickle`-Modul enthält auch das `json`-Modul eine `load()`-Funktion, die ein Streamobjekt übernimmt, JSON-codierte Daten daraus liest und ein neues Python-Objekt erstellt, das die JSON-Datenstruktur widerspiegelt.

```
>>> shell
2
>>> del entry                                                    ①
>>> entry
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import json
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f)                                     ②
...
>>> entry                                                       ③
{'comments_link': None,
 'internal_id': {'__class__': 'bytes', '__value__': [222, 213, 180,
248]},
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-
history-2009-edition',
 'published_date': {'__class__': 'time.asctime', '__value__': 'Fri Mar 27
22:20:42 2009'},
 'published': True}
```

① Wechseln Sie zu Demonstrationszwecken zu Python-Shell `#2` und löschen Sie die `entry`-Datenstruktur, die Sie zu Beginn dieses Kapitels mithilfe des `pickle`-Moduls erstellt haben.

② Im einfachsten Fall funktioniert die `json.load()`-Funktion genau wie die `pickle.load()`-Funktion. Sie übergeben ihr ein Streamobjekt und die Funktion gibt ein neues Python-Objekt zurück.

③ Ich habe eine gute und eine schlechte Nachricht. Zuerst die gute Nachricht: Die `json.load()`-Funktion hat die in Python-Shell `#1` erstellte `entry.json`-Datei erfolgreich gelesen und ein neues Python-Objekt mit diesen Daten erstellt. Die schlechte Nachricht: Die Funktion hat nicht die ursprüngliche `entry`-Datenstruktur wiederhergestellt. Die beiden Werte `'internal_id'` und `'published_date'` wurden als Dictionarys wiederhergestellt – genauer, als die Dictionarys mit JSON-kompatiblen Werten, die Sie in der `to_json()`-Funktion erstellt haben.

`json.load()` weiß nichts von einer Konvertierungsfunktion, die Sie `json.dump()` übergeben haben könnten. Sie brauchen das Gegenstück zur `to_json()`-Funktion – eine Funktion, die ein selbst umgewandeltes JSON-Objekt übernimmt und es wieder in den ursprünglichen Python-Datentyp umwandelt.

```
# fügen Sie dies zu customserializer.py hinzu
def from_json(json_object):                                       ①
    if '__class__' in json_object:                                ②
        if json_object['__class__'] == 'time.asctime':
            return time.strptime(json_object['__value__'])        ③
```

```
        if json_object['__class__'] == 'bytes':
            return bytes(json_object['__value__'])            ④
    return json_object
```

① Diese Konvertierungsfunktion übernimmt ebenfalls einen Parameter und gibt einen Wert zurück. Doch der übernommene Parameter ist kein String; es ist ein Python-Objekt – das Ergebnis der Deserialisierung eines JSON-codierten Strings.

② Sie müssen lediglich überprüfen, ob dieses Objekt den von der `to_json()`-Funktion erstellten `'__class__'`-Schlüssel enthält. Ist dies der Fall, so verrät Ihnen der Wert des `'__class__'`-Schlüssels, wie der Wert wieder in den ursprünglichen Python-Datentyp decodiert werden kann.

③ Zum Decodieren des von der `time.asctime()`-Funktion zurückgegebenen Strings, verwenden Sie die Funktion `time.strptime()`. Diese Funktion übernimmt einen formatierten String (in einem anpassbaren Format, dessen Standard jedoch dem Standard der `time.asctime()`-Funktion entspricht) und gibt ein `time.struct_time`-Objekt zurück.

④ Um eine Liste von Ganzzahlen wieder in ein `bytes`-Objekt umzuwandeln, können Sie die Funktion `bytes()` benutzen.

Das war's! Es gab nur zwei von der `to_json()`-Funktion behandelte Datentypen und diese beiden Datentypen haben wir nun in der `from_json()`-Funktion verarbeitet. Das Ergebnis:

```
>>> shell
2
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f, object_hook=customserializer.from_json)①
...
>>> entry                                                          ②
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link':  'http://diveintomark.org/archives/2009/03/27/dive-into-
history-2009-edition',
 'published_date':  time.struct_time(tm_year=2009,  tm_mon=3,  tm_mday=27,
tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
```

① Um die `from_json()`-Funktion in den Deserialisierungsprozess einzubinden, übergeben Sie sie der `json.load()`-Funktion als `object_hook`-Parameter. Funktionen übernehmen Funktionen; es ist so praktisch!

② Die `entry`-Datenstruktur enthält nun einen `'internal_id'`-Schlüssel, dessen Wert ein `bytes`-Objekt ist. Außerdem enthält sie einen `'published_date'`-Schlüssel, dessen Wert ein `time.struct_time`-Objekt ist.

Ein kleines Problem gibt es aber noch.

```
>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry2 = json.load(f, object_hook=customserializer.from_json)
...
>>> entry2 == entry                      ①
False
>>> entry['tags']                        ②
('diveintopython', 'docbook', 'html')
>>> entry2['tags']                       ③
['diveintopython', 'docbook', 'html']
```

① Selbst nachdem wir die `to_json()`-Funktion in die Serialisierung und die `from_json()`-Funktion in die Deserialisierung eingebunden haben, wird immer noch keine perfekte Kopie der ursprünglichen Datenstruktur erstellt. Warum nicht?

② In der ursprünglichen `entry`-Datenstruktur war der Wert des Schlüssels `'tags'` ein *Tupel* aus drei Strings.

③ In der `entry2`-Datenstruktur dagegen ist der Wert des `'tags'`-Schlüssels eine *Liste* aus drei Strings. JSON unterscheidet nicht zwischen Tupeln und Listen; es besitzt lediglich einen listenähnlichen Datentyp, das Array. Das `json`-Modul wandelt während der Serialisierung sowohl Tupel als auch Listen in JSON-Arrays um. In den meisten Fällen können Sie den Unterschied zwischen Tupeln und Listen ignorieren. Sie sollten dies jedoch bei der Arbeit mit dem `json`-Modul im Hinterkopf behalten.

Kapitel 15
HTTP-Webdienste

15.1 Los geht's

Mithilfe von HTTP-Webdiensten kann man durch Programmierung Daten an einen entfernten Server senden und von diesem abrufen, indem man lediglich die Befehle von HTTP nutzt. Möchten Sie Daten vom Server holen, verwenden Sie `HTTP GET`; möchten Sie Daten an den Server senden, verwenden Sie `HTTP POST`. Einige fortschrittlichere HTTP-Webdienst-APIs bieten auch Befehle zum Erstellen, Verändern und Löschen von Daten, `HTTP PUT` und `HTTP DELETE`. Die Befehle des HTTP-Protokolls (`GET`, `POST`, `PUT` und `DELETE`) können direkt auf die Befehle zum Abrufen, Erstellen, Verändern und Löschen von Daten auf der Anwendungsebene abgebildet werden.

Der Hauptvorteil dieser Vorgehensweise ist Einfachheit; und diese Einfachheit ist sehr beliebt. Daten – normalerweise XML-Daten – können statisch erstellt und gespeichert, oder dynamisch von einem serverseitigen Skript erzeugt werden. Alle großen Programmiersprachen (natürlich auch Python) besitzen eine HTTP-Bibliothek zum Herunterladen dieser Daten. Debuggen ist ebenfalls einfacher; da jede Ressource eines HTTP-Webdienstes eine einmalige Adresse (in Form einer URL) besitzt, können Sie sie in Ihrem Webbrowser laden und sofort die Rohdaten anschauen.

Beispiele für HTTP-Webdienste:

- Google Data APIs erlauben Ihnen die Interaktion mit sehr vielen Google-Diensten, wie z. B. Blogger und YouTube.
- FlickrServices erlauben Ihnen das Hoch- und Runterladen der Fotos von Flickr.
- Twitter API erlaubt Ihnen das Veröffentlichen von Status-Aktualisierungen auf Twitter.
- … und viele mehr.

Python 3 besitzt zwei verschiedene Bibliotheken zur Interaktion mit HTTP-Webdiensten:

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_15, © Springer-Verlag Berlin Heidelberg 2010

- `http.client` ist eine Low-Level-Bibliothek, die *RFC 2616*, das HTTP-Protokoll, implementiert.
- `urllib.request` ist eine Abstraktionsschicht, die auf `http.client` aufbaut. Sie stellt eine API zum Zugriff auf HTTP- und FTP-Server bereit, folgt automatisch HTTP-Weiterleitungen und kommt mit einigen bekannten Methoden der HTTP-Authentifikation zurecht.

Welche Bibliothek sollten Sie nutzen? Keine der bisher genannten. Stattdessen sollten Sie `httplib2`, eine Open-Source-Bibliothek, nutzen, die HTTP umfassender als `http.client` implementiert, aber eine bessere Abstraktion als `urllib.request` bietet.

Um zu verstehen, warum `httplib2` die richtige Wahl ist, müssen Sie erst einmal HTTP verstehen.

15.2 Eigenschaften von HTTP

Es gibt fünf wichtige Eigenschaften, die alle HTTP-Clients unterstützen sollten.

15.2.1 Caching

Bei Webdiensten ist es wichtig zu verstehen, dass ein Netzwerkzugriff unglaublich kostspielig ist. Ich meine damit nicht kostspielig im Sinne von „Dollar und Cent“ (obwohl Bandbreite nicht gratis ist). Ich rede davon, dass es ausgesprochen lange dauert, eine Verbindung aufzubauen, eine Anfrage zu senden und eine Antwort von einem entfernten Server zu erhalten. Selbst bei der schnellsten Verbindung kann die Latenzzeit (die zum Senden einer Anfrage und zum Erhalten der Daten benötigte Zeit) höher sein, als Sie angenommen haben. Ein Router funktioniert nicht richtig, ein Datenpaket geht verloren, ein zwischengeschalteter Proxy wird attackiert – im Internet gibt es nie auch nur eine ruhige Minute, und Sie können wahrscheinlich nichts dagegen tun.

Beim Entwurf von HTTP spielt *Caching* (das Zwischenspeichern von Daten) eine große Rolle. Es gibt eine ganze Geräteklasse („Caching Proxys“ genannt), die nichts anderes tut als zwischen Ihnen und dem Rest der Welt zu sitzen und den Netzwerkzugriff zu reduzieren. Ihre Firma oder Ihr Internetprovider unterhalten sicher „Caching Proxys“, auch wenn Sie nichts davon wissen. Sie funktionieren, weil Caching in das HTTP-Protokoll integriert ist.

Hier nun ein konkretes Beispiel zur Funktionsweise des Cachings. Sie besuchen mit Ihrem Browser `diveintomark.org`. Diese Seite enthält ein Hintergrundbild: `wearehugh.com/m.jpg`. Lädt Ihr Browser dieses Bild herunter, fügt der Server die folgenden HTTP-Header hinzu:

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

Die Header `Cache-Control` und `Expires` teilen Ihrem Browser (und jedem „Caching Proxy“ zwischen Ihnen und dem Server) mit, dass dieses Bild für ein Jahr zwischengespeichert werden kann. Ein Jahr! Und wenn Sie nun im nächsten Jahr eine andere Seite besuchen, die auch einen Link zu diesem Bild enthält, lädt Ihr Browser es aus seinem Cache und erzeugt so überhaupt keine Netzwerkaktivität.

Warten Sie, es wird noch besser. Sagen wir, Ihr Browser löscht das Bild aus irgendeinem Grund aus Ihrem lokalen Zwischenspeicher. Vielleicht war kein Festplattenspeicher mehr verfügbar; vielleicht haben Sie den Zwischenspeicher auch selbst gelöscht. Ganz egal. Die HTTP-Header besagten, dass die Daten auch von öffentlichen „Caching Proxys“ (aufgrund des Keywords `public` im Header `Cache-Control`) zwischengespeichert werden können. „Caching Proxys“ sind so entworfen, dass Sie massig Speicherplatz besitzen; wahrscheinlich sehr viel mehr als Ihrem lokalen Browser zur Verfügung steht.

Unterhält Ihr Unternehmen oder Ihr Internetprovider einen „Caching Proxy“, könnte das Bild dort immer noch zwischengespeichert sein. Besuchen Sie `diveintomark.org` erneut, sucht Ihr Browser in seinem lokalen Cache nach dem Bild, findet es aber nicht. Er sendet eine Netzwerkanfrage, um das Bild herunterzuladen. Hat der „Caching Proxy“ nun aber noch eine Kopie des Bildes, wird er die Anfrage unterbrechen und das Bild aus seinem Cache zur Verfügung stellen. Das bedeutet, dass Ihre Anfrage den entfernten Server nie erreicht; tatsächlich verlässt sie sogar gar nicht Ihr Firmennetzwerk. Dies sorgt für einen schnelleren Download (weniger Netzwerksprünge) und spart Ihrer Firma Geld (weniger Daten werden von außerhalb heruntergeladen).

HTTP-Caching funktioniert nur, wenn jeder seinen Teil dazu beiträgt. Auf der einen Seite müssen Server in ihren Antworten die korrekten Header senden. Auf der anderen Seite müssen Clients diese Header verstehen und beachten, bevor sie dieselben Daten zweimal anfordern. Die Proxys dazwischen sind kein Allheilmittel; sie sind lediglich so clever, wie die Server und Clients es erlauben.

Pythons HTTP-Bibliotheken unterstützen kein Caching, doch `httplib2` schon.

15.2.2 Überprüfen des Datums der letzten Änderung

Manche Daten ändern sich nie, während andere dies ständig tun. Dazwischen liegt eine große Anzahl an Daten, die sich geändert haben könnten, es aber nicht getan haben. Der Feed von CNN.com wird alle paar Minuten aktualisiert, der Feed meines Blogs dagegen könnte sich tage- oder wochenlang nicht ändern. In diesem Fall möchte ich nicht, dass die Clients meinen Feed wochenlang zwischenspeichern, denn wenn ich dann einen neuen Artikel schreibe, würde er vielleicht über Wochen nicht gelesen (da meine Header sagen „überprüfe diesen Feed wochenlang nicht"). Andererseits möchte ich auch nicht, dass die Clients meinen kompletten Feed jede Stunde herunterladen, obwohl er sich nicht geändert hat.

HTTP hat auch dafür eine Lösung. Wenn Sie Daten zum ersten Mal anfragen, kann der Server einen `Last-Modified`-Header zurück senden. Das ist genau das, wonach es klingt: das Datum, an dem die Daten geändert wurden. Das Hintergrundbild von `diveintomark.org` enthält solch einen `Last-Modified`-Header.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

Wenn Sie dieselben Daten nun ein zweites (oder drittes, oder viertes) Mal anfragen, können Sie in Ihrer Anfrage einen `If-Modified-Since`-Header zusammen mit dem letzten vom Server zurückgegebenen Datum senden. Haben sich die Daten seitdem nicht geändert, sendet der Server den HTTP-Statuscode `304`, der bedeutet „diese Daten haben sich seit Ihrer letzten Anfrage nicht geändert". Sie können dies unter Verwendung von `curl` auf der Kommandozeile testen:

```
you@localhost:~$ curl -I -H "If-Modified-Since: Fri, 22 Aug 2008 04:28:16 GMT"
http://wearehugh.com/m.jpg
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

Warum ist das eine Verbesserung? Weil der Server beim Auftreten des Statuscodes `304` die Daten nicht erneut sendet. Sie erhalten lediglich den Statuscode. Selbst dann, wenn Ihre zwischengespeicherte Kopie abgelaufen ist, stellt die Überprüfung des Datums der letzten Änderung sicher, dass Sie dieselben Daten nicht zweimal herunterladen. (Als Zugabe enthält diese `304`-Antwort auch Caching-Header. Die Proxys behalten auch dann eine Kopie der Daten, nachdem sie offiziell „abgelaufen" sind, in der Hoffnung, dass die Daten sich nicht wirklich geändert haben und die nächste Anfrage mit einem `304`-Statuscode und aktualisierten Cache-Informationen beantwortet wird.)

Pythons HTTP-Bibliotheken unterstützen das Überprüfen des Datums der letzten Änderung nicht, doch `httplib2` tut es.

15.2.3 ETags

Mit ETags kann man dasselbe erreichen wie mit der Überprüfung des Datums der letzten Änderung. Mithilfe von ETags sendet der Server einen *Hashcode* in einem `ETag`-Header zusammen mit den angefragten Daten. (Wie dieser Hashcode bestimmt wird, entscheidet allein der Server. Die einzige Voraussetzung ist, dass er sich ändert, wenn sich die Daten ändern.) Das Hintergrundbild von `diveintomark.org` hat einen `ETag`-Header.

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
Cache-Control: max-age=31536000, public
Expires: Mon, 31 May 2010 17:14:04 GMT
Connection: close
Content-Type: image/jpeg
```

Beim erneuten Anfragen derselben Daten fügen Sie den ETag-Hashcode in einem `If-None-Match`-Header in Ihre Anfrage ein. Haben sich die Daten nicht geändert, sendet der Server den Statuscode `304`. Wie bei der Überprüfung des Datums der letzten Änderung sendet der Server *nur* den Statuscode `304`; die Daten selbst werden nicht noch einmal übertragen. Durch das Einfügen des Etag-Hashcodes in Ihre zweite Anfrage teilen Sie dem Server mit, dass er die Daten – sofern sie sich nicht geändert haben – nicht noch ein zweites Mal senden muss, da Sie diese noch vom letzten Mal besitzen.

Wieder benutzen wir `curl`:

```
you@localhost:~$ curl -I -H "If-None-Match: \"3075-ddc8d800\""
http://wearehugh.com/m.jpg  ①
HTTP/1.1 304 Not Modified
Date: Sun, 31 May 2009 18:04:39 GMT
Server: Apache
Connection: close
ETag: "3075-ddc8d800"
Expires: Mon, 31 May 2010 18:04:39 GMT
Cache-Control: max-age=31536000, public
```

① ETags werden häufig in Anführungszeichen gesetzt, doch die Anführungszeichen sind Teil des Wertes. Es sind keine Begrenzer; der einzige Begrenzer im `ETag`-Header ist der Doppelpunkt zwischen `ETag` und `"3075-ddc8d800"`. Das bedeutet, dass Sie die Anführungszeichen auch an den Server senden müssen.

Pythons HTTP-Bibliotheken unterstützen keine ETags, doch `httplib2` schon.

15.2.4 Komprimierung

Redet man über HTTP-Webdienste, so redet man fast immer darüber, textbasierte Daten über das Netz zu schicken. Vielleicht XML, vielleicht JSON, vielleicht auch nur einfachen Text. Text lässt sich unabhängig vom Format stark komprimieren. Der Beispiel-Feed im XML-Kapitel ist unkomprimiert 3.070 B groß, mit gzip komprimiert jedoch nur noch 941 B. Das sind lediglich 30% der ursprünglichen Größe!

HTTP unterstützt verschiedene Komprimierungs-Algorithmen. Die zwei verbreitetsten sind *gzip* und *deflate*. Fragen Sie über HTTP eine Ressource an, können Sie den Server anweisen, sie in einem komprimierten Format zu senden. Dazu fügen Sie einen `Accept-encoding`-Header in Ihre Anfrage ein, der auflistet, welche Algorithmen Sie unterstützen. Unterstützt der Server eines dieser Formate, sendet er Ihnen komprimierte Daten (mit einem `Content-encoding`-Header, der Ihnen den verwendeten Algorithmus mitteilt). Dann liegt es an Ihnen, die Daten zu dekomprimieren.

Pythons HTTP-Bibliotheken unterstützen keine Komprimierung, doch `httplib2` tut es.

15.2.5 Weiterleitungen

Coole URIs ändern sich nicht, doch viele URIs sind wirklich uncool. Webseiten werden umgebaut, Seiten ziehen zu neuen Adressen um. Sogar Webdienste können umstrukturiert werden. Ein Feed unter `http://example.com/index.xml`

könnte zu `http://example.com/xml/atom.xml` umziehen. Eine komplette Domain könnte umziehen, wenn eine Einrichtung erweitert oder neu strukturiert wird; `http://www.example.com/index.xml` wird zu `http://server-farm-1.example.com/index.xml`.

Jedes Mal wenn Sie eine Ressource von einem HTTP-Server anfragen, enthält die Antwort des Servers einen Statuscode. Der Statuscode `200` bedeutet „alles ist normal, hier ist die von Ihnen angeforderte Seite". Der Statuscode `404` bedeutet „Seite nicht gefunden". (`404`-Fehler haben Sie vermutlich beim Surfen im Internet schon gesehen.) Statuscodes im `300`er-Bereich bedeuten irgendeine Art von Weiterleitung.

HTTP besitzt verschiedene Möglichkeiten zum Anzeigen, dass eine Ressource umgezogen ist. Die beiden geläufigsten sind die Statuscodes `302` und `301`. `302` ist eine *temporäre Weiterleitung*; dies bedeutet „upps, das ist vorübergehend dorthin gezogen" (dann wird diese vorübergehende Adresse in einem `Location`-Header angegeben). `301` dagegen ist eine *dauerhafte Weiterleitung*; sie bedeutet „upps, das ist dauerhaft umgezogen" (dann wird die neue Adresse in einem `Location`-Header angegeben). Erhalten Sie einen `302`-Statuscode und eine neue Adresse, sollen Sie laut HTTP-Spezifikation diese neue Adresse benutzen, um das zu bekommen was Sie wollten, doch beim nächsten Mal sollten Sie wieder die alte Adresse ausprobieren. Erhalten Sie dagegen einen `301`-Statuscode und eine neue Adresse, sollen Sie diese neue Adresse ab sofort immer benutzen.

Das Modul `urllib.request` „folgt" Weiterleitungen automatisch, wenn es vom HTTP-Server den passenden Statuscode erhält, teilt Ihnen dies aber nicht mit. Sie erhalten schlussendlich die Daten, die Sie wollten, werden jedoch nie erfahren, dass die zugrundeliegende Bibliothek „hilfsbereiterweise" einer Weiterleitung gefolgt ist. Sie werden also weiterhin die alte Adresse verwenden und jedes Mal zu der neuen Adresse weitergeleitet, und jedes Mal folgt das `urllib.request`-Modul „hilfsbereiterweise" dieser Weiterleitung. Das Modul behandelt dauerhafte Weiterleitungen genauso wie temporäre Weiterleitungen. Das bedeutet zwei Aufrufe statt einem, was sowohl schlecht für den Server als auch schlecht für Sie ist.

`httplib2` bearbeitet dauerhafte Weiterleitungen für Sie. Es teilt Ihnen nicht nur mit, dass eine dauerhafte Weiterleitung vorliegt, sondern verwaltet sie lokal und ändert weitergeleitete URLs automatisch, bevor sie aufgerufen werden.

15.3 Wie man Daten nicht über HTTP abrufen sollte

Nehmen wir an, Sie wollten eine Ressource über HTTP herunterladen, z. B. einen Atom-Feed. Einen Feed laden Sie nicht nur einmal herunter; Sie laden ihn immer und immer wieder herunter. (Die meisten Feed-Reader überprüfen Feeds jede Stunde auf Änderungen.) Machen wir es erstmal auf die schnelle Art und sehen uns dann an, wie wir es besser machen können.

```
>>> import urllib.request
>>> a_url = 'http://diveintopython3.org/examples/feed.xml'
>>> data = urllib.request.urlopen(a_url).read()  ①
>>> type(data)                                   ②
<class 'bytes'>
>>> print(data)
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  …
```

① In Python ist es unglaublich einfach, irgendetwas über HTTP herunterzuladen; tatsächlich wird dazu nur eine Zeile benötigt. Das `urllib.request`-Modul besitzt eine praktische `urlopen()`-Funktion, die die Adresse der gewünschten Seite übernimmt und ein dateiähnliches Objekt zurückgibt, mit dessen `read()`-Methode Sie den Inhalt der Seite erhalten können. Einfacher könnte es gar nicht sein.

② Die `urlopen().read()`-Methode gibt immer ein `bytes`-Objekt zurück, keinen String. Bytes sind Bytes; Zeichen sind eine Abstraktion. HTTP-Server befassen sich nicht mit Abstraktionen. Fordern Sie eine Ressource an, erhalten Sie Bytes. Möchten Sie sie als String, müssen Sie die Zeichencodierung bestimmen und sie in einen String umwandeln.

Was ist hieran denn nun falsch? Bei einem kurzen, einmaligen Einsatz zum Testen oder Entwickeln ist nichts falsch daran. Ich mache es dauernd. Ich wollte den Inhalt des Feeds und den habe ich erhalten. Dasselbe funktioniert mit jeder Webseite. Doch bei einem Webdienst auf den Sie regelmäßig zugreifen möchten (z. B. das stündliche Abrufen dieses Feeds) ist dies ineffizient und unhöflich.

15.4 Was geht über's Netz

Um zu sehen, warum es ineffizient und unhöflich ist, schalten wir die Debugging-Funktionen von Pythons HTTP-Bibliothek ein und sehen uns an, was über das Netzwerk geschickt wird.

```
>>> from http.client import HTTPConnection
>>> HTTPConnection.debuglevel = 1                                    ①
>>> from urllib.request import urlopen
>>> response = urlopen('http://diveintopython3.org/examples/feed.xml')  ②
send: b'GET /examples/feed.xml HTTP/1.1                              ③
Host: diveintopython3.org                                            ④
Accept-Encoding: identity                                            ⑤
```

```
User-Agent: Python-urllib/3.1'                                    ⑥
Connection: close
reply: 'HTTP/1.1 200 OK'
…gekürzt…
```

① Wie ich zu Beginn des Kapitels bereits anmerkte, ist `urllib.request` von einer weiteren Standardbibliothek abhängig: `http.client`. Normalerweise müssen Sie sich nicht direkt mit `http.client` beschäftigen. (Das `urllib.request`-Modul importiert es automatisch.) Hier importieren wir es aber, da wir so das *Debugging-Flag* der Klasse `HTTPConnection` setzen können. Diese Klasse nutzt `urllib.request` zur Verbindung mit einem HTTP-Server.

② Nun, da das Debugging-Flag gesetzt ist, werden in Echtzeit Informationen zur HTTP-Anfrage und -Antwort ausgegeben. Wie Sie sehen können, sendet das `urllib.request`-Modul bei der Anfrage des Atom-Feeds fünf Zeilen an den Server.

③ Die erste Zeile gibt den verwendeten HTTP-Befehl und den Pfad der Ressource (ohne Domainnamen) an.

④ Die zweite Zeile gibt den Domainnamen an, von dem wir diesen Feed anfragen.

⑤ Die dritte Zeile gibt den vom Client unterstützten Komprimierungs-Algorithmus an. Wie ich schon sagte, unterstützt `urllib.request` standardmäßig keine Komprimierung.

⑥ Die vierte Zeile gibt den Namen der anfragenden Bibliothek an. Standardmäßig ist dies `Python-urllib` und eine Versionsnummer. Sowohl `urllib.request` als auch `httplib2` unterstützen das Ändern des User-Agents (Browsertyps), indem man einfach einen `User-Agent`-Header zur Anfrage hinzufügt (das überschreibt den Standardwert).

Sehen wir uns nun an, was der Server als Antwort sendet.

```
# Fortsetzung des vorherigen Beispiels
>>> print(response.headers.as_string())        ①
Date: Sun, 31 May 2009 19:23:06 GMT            ②
Server: Apache
Last-Modified: Sun, 31 May 2009 06:39:55 GMT   ③
ETag: "bfe-93d9c4c0"                           ④
Accept-Ranges: bytes
Content-Length: 3070                           ⑤
Cache-Control: max-age=86400                   ⑥
Expires: Mon, 01 Jun 2009 19:23:06 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml
>>> data = response.read()                     ⑦
>>> len(data)
3070
```

① Die von der `urllib.request.urlopen()`-Funktion zurückgegebene Antwort enthält alle vom Server gesendeten Header. Sie enthält außerdem Methoden zum Herunterladen der eigentlichen Daten; dazu kommen wir gleich.

② Der Server teilt Ihnen mit, wann er die Anfrage bearbeitet hat.

③ Diese Antwort enthält einen `Last-Modified`-Header.

④ Diese Antwort enthält einen `ETag`-Header.

⑤ Die Daten haben eine Größe von 3.070 B. Beachten Sie das Fehlen eines `Content-encoding`-Headers. Ihre Anfrage erlaubte nur unkomprimierte Daten (`Accept-encoding: identity`) und diese Antwort enthält sicherlich unkomprimierte Daten.

⑥ Diese Antwort enthält Caching-Header, die angeben, dass dieser Feed bis zu 24 h (86.400 s) zwischengespeichert werden kann.

⑦ Schließlich laden wir durch den Aufruf von `response.read()` die eigentlichen Daten herunter. Wie Sie an der `len()`-Funktionen erkennen können, lädt dies alle 3.070 B auf einmal herunter.

Wie Sie sehen können, ist dieser Code bereits ineffizient: Er fordert (und erhält) unkomprimierte Daten. Ich weiß genau, dass dieser Server gzip-Komprimierung unterstützt, doch HTTP-Komprimierung muss man ausdrücklich angeben. Wir haben nicht danach gefragt, also haben wir es auch nicht bekommen. Wir laden also 3.070 B herunter, obwohl wir auch lediglich 941 B hätten herunterladen können. Böser Hund, kein Leckerli.

Doch es wird noch schlimmer! Um zu sehen, wie ineffizient dieser Code wirklich ist, fragen wir den Feed ein zweites Mal an.

```
# Fortsetzung des vorherigen Beispiels
>>> response2 = urlopen('http://diveintopython3.org/examples/feed.xml')
send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
Accept-Encoding: identity
User-Agent: Python-urllib/3.1'
Connection: close
reply: 'HTTP/1.1 200 OK'
…gekürzt…
```

Fällt Ihnen an dieser Anfrage etwas Merkwürdiges auf? Sie hat sich nicht verändert! Es ist genau dieselbe Anfrage wie die erste. Kein Anzeichen eines `If-Modified-Since`-Headers. Kein Anzeichen eines `If-None-Match`-Headers. Kein Beachten der Caching-Header. Immer noch keine Komprimierung.

Und was passiert, wenn Sie dasselbe zweimal tun? Sie erhalten zweimal dieselbe Antwort.

```
# Fortsetzung des vorherigen Beispiels
>>> print(response2.headers.as_string())      ①
Date: Mon, 01 Jun 2009 03:58:00 GMT
Server: Apache
Last-Modified: Sun, 31 May 2009 22:51:11 GMT
```

```
ETag: "bfe-255ef5c0"
Accept-Ranges: bytes
Content-Length: 3070
Cache-Control: max-age=86400
Expires: Tue, 02 Jun 2009 03:58:00 GMT
Vary: Accept-Encoding
Connection: close
Content-Type: application/xml
>>> data2 = response2.read()
>>> len(data2)                                    ②
3070
>>> data2 == data                                 ③
True
```

① Der Server sendet immer noch dieselben „klugen“ Header: `Cache-Control` und `Expires`, um Caching zu erlauben; `Last-Modified` und `ETag`, um Änderungen zu verfolgen. Der Header `Vary : Accept-Encoding` weist darauf hin, dass der Server Komprimierung unterstützen würde, wenn Sie danach fragen würden. Doch das haben Sie nicht.

② Wieder einmal werden beim Abrufen der Daten die kompletten 3.070 B heruntergeladen …

③ … dieselben 3.070 B, die Sie schon beim letzten Mal heruntergeladen haben.

HTTP wurde so entworfen, dass es mehr kann als das. `urllib` spricht HTTP so, wie ich Spanisch spreche – genug, um ein bisschen zu verstehen, aber nicht genug, um ein Gespräch zu führen. HTTP ist ein Gespräch. Es ist an der Zeit, eine Bibliothek zu wählen, die fließend HTTP spricht.

15.5 Vorstellung von httplib2

Bevor Sie `httplib2` nutzen können, müssen Sie es installieren. Gehen Sie auf `code.google.com/p/httplib2/` und laden Sie die neueste Version herunter. `httplib2` steht für Python 2.x und Python 3.x zur Verfügung; laden Sie die Python 3-Version herunter, die etwa wie folgt heißt: `httplib2-python3-0.5.0.zip`.

Entpacken Sie das Archiv, öffnen Sie ein Terminal und gehen Sie in das neu erstellte `httplib2`-Verzeichnis. Unter Windows müssen Sie das Startmenü öffnen, `Ausführen...` wählen, `cmd.exe` eingeben und `EINGABE` drücken.

```
c:\Users\pilgrim\Downloads> dir
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\Downloads
```

```
07/28/2009  12:36 PM    <DIR>          .
07/28/2009  12:36 PM    <DIR>          ..
07/28/2009  12:36 PM    <DIR>          httplib2-python3-0.5.0
07/28/2009  12:33 PM            18,997 httplib2-python3-0.5.0.zip
               1 File(s)         18,997 bytes
               3 Dir(s)  61,496,684,544 bytes free

c:\Users\pilgrim\Downloads> cd httplib2-python3-0.5.0
c:\Users\pilgrim\Downloads\httplib2-python3-0.5.0> c:\python31\python.exe setup.py
install
running install
running build
running build_py
running install_lib
creating c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\iri2uri.py -> c:\python31\Lib\site-packages\httplib2
copying build\lib\httplib2\__init__.py -> c:\python31\Lib\site-packages\httplib2
byte-compiling c:\python31\Lib\site-packages\httplib2\iri2uri.py to iri2uri.pyc
byte-compiling c:\python31\Lib\site-packages\httplib2\__init__.py to __init__.pyc
running install_egg_info
Writing c:\python31\Lib\site-packages\httplib2-python3_0.5.0-py3.1.egg-info
```

Unter Mac OS X starten Sie die Anwendung `Terminal.app` in Ihrem Ordner `/Programme/Dienstprogramme/`. Unter Linux müssen Sie die Anwendung `Terminal` starten, die sich gewöhnlich in Ihrem `Anwendungen`-Menü unter `Zubehör` oder `System` befindet.

```
you@localhost:~/Desktop$ unzip httplib2-python3-0.5.0.zip
Archive:  httplib2-python3-0.5.0.zip
  inflating: httplib2-python3-0.5.0/README
  inflating: httplib2-python3-0.5.0/setup.py
  inflating: httplib2-python3-0.5.0/PKG-INFO
  inflating: httplib2-python3-0.5.0/httplib2/__init__.py
  inflating: httplib2-python3-0.5.0/httplib2/iri2uri.py
you@localhost:~/Desktop$ cd httplib2-python3-0.5.0/
you@localhost:~/Desktop/httplib2-python3-0.5.0$ sudo python3 setup.py install
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-3.1
creating build/lib.linux-x86_64-3.1/httplib2
copying httplib2/iri2uri.py -> build/lib.linux-x86_64-3.1/httplib2
copying httplib2/__init__.py -> build/lib.linux-x86_64-3.1/httplib2
running install_lib
creating /usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/iri2uri.py ->
```

```
/usr/local/lib/python3.1/dist-packages/httplib2
copying build/lib.linux-x86_64-3.1/httplib2/__init__.py ->
/usr/local/lib/python3.1/dist-packages/httplib2
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/iri2uri.py to
iri2uri.pyc
byte-compiling /usr/local/lib/python3.1/dist-packages/httplib2/__init__.py to
__init__.pyc
running install_egg_info
Writing /usr/local/lib/python3.1/dist-packages/httplib2-python3_0.5.0.egg-info
```

Zur Nutzung von `httplib2` müssen Sie eine Instanz der Klasse `httplib2.Http` erstellen.

```
>>> import httplib2
>>> h = httplib2.Http('.cache')                                                    ①
>>> response, content = h.request('http://diveintopython3.org/examples/feed.xml')  ②
>>> response.status                                                                ③
200
>>> content[:52]                                                                   ④
b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content)
3070
```

① Die Hauptschnittstelle zu `httplib2` ist das `Http`-Objekt. Aus Gründen die Sie im nächsten Abschnitt kennen lernen werden, sollten Sie beim Erstellen des `Http`-Objekts immer einen Verzeichnisnamen übergeben. Dieses Verzeichnis muss nicht vorhanden sein; `httplib2` erstellt es, wenn nötig.

② Haben Sie ein `Http`-Objekt erstellt, können Sie durch einfachen Aufruf der `request()`-Methode mit der Adresse der gewünschten Daten diese Daten erhalten. Dies sendet eine `HTTP GET`-Anfrage für diese URL. (Später in diesem Kapitel werden Sie erfahren, wie man auch andere HTTP-Anfragen, wie `POST`, senden kann.)

③ Die `request()`-Methode gibt zwei Werte zurück. Der erste Wert ist ein `httplib2.Response`-Objekt, das alle vom Server zurückgesendeten HTTP-Header enthält. Der Statuscode `200` gibt beispielsweise an, dass die Anfrage erfolgreich war.

④ Die Variable `content` enthält die eigentlichen Daten, die vom HTTP-Server gesendet wurden. Die Daten werden als `bytes`-Objekt, nicht als String, zurückgegeben. Möchten Sie sie als String, müssen Sie die Zeichencodierung bestimmen und sie selbst in einen String umwandeln.

☞ Sie benötigen wahrscheinlich nur ein einziges `httplib2.Http`-Objekt. Es gibt Situationen, in denen es sinnvoll ist, mehrere zu erstellen, doch das sollten Sie nur tun, wenn Sie wissen, wozu Sie diese brauchen. „Ich muss Daten von zwei verschiedenen URLs anfragen“ ist kein berechtigter Grund. Verwenden Sie das `Http`-Objekt erneut und rufen Sie die `request()`-Methode einfach zweimal auf.

15.5.1 Ein kleiner Exkurs zur Erklärung, warum `httplib2` Bytes statt Strings zurückgibt

Bytes. Strings. Es nervt. Warum kann `httplib2` nicht „einfach“ die Umwandlung für Sie erledigen? Nun, das ist kompliziert, da die Regeln zur Bestimmung der Zeichencodierung von der Art der angefragten Ressource abhängen. Wie könnte `httplib2` wissen, welche Art von Ressource Sie anfragen? Normalerweise wird dies im HTTP-Header `Content-Type` angegeben; dies ist aber eine optionale Funktion von HTTP, die nicht alle HTTP-Server unterstützen. Ist dieser Header in der HTTP-Antwort nicht enthalten, muss der Client raten, welche Art Ressource es ist. (Und das ist nie perfekt.)

Wenn Sie die zu erwartende Art der Ressource kennen (in diesem Fall ein XML-Dokument), könnten Sie das zurückgegebene `bytes`-Objekt vielleicht einfach der `xml.etree.ElementTree.parse()`-Funktion übergeben. Dies funktioniert nur, wenn das XML-Dokument Informationen zu seiner eigenen Zeichencodierung enthält (so wie dieses); dies ist jedoch optional und nicht bei jedem XML-Dokument der Fall. Enthält ein XML-Dokument keine Informationen zur Zeichencodierung, so muss sich der Client den `Content-Type`-HTTP-Header ansehen, der einen `charset`-Parameter enthalten kann.

Doch es ist schlimmer als das. Informationen zur Zeichencodierung können nun an zwei Stellen vorkommen: innerhalb des XML-Dokuments selbst und innerhalb des `Content-Type`-Headers. Welche Stelle hat hier Vorrang? Ist der im `Content-Type`-Header angegebene Inhaltstyp `application/xml`, `application/xml-dtd`, `application/xml-external-parsed-entity` oder irgendein Untertyp von `application/xml`, wie `application/atom+xml`, `application/rss+xml` oder `application/rdf+xml`, so ist die Zeichencodierung gemäß *RFC 3023*

1. die im `charset`-Parameter des `Content-Type`-Headers angegebene Codierung *oder*
2. die im `encoding`-Attribut der XML-Deklaration innerhalb des Dokuments angegebene Codierung *oder*
3. UTF-8.

Ist der im `Content-Type`-Header angegebene Inhaltstyp dagegen `text/xml`, `text/xml-external-parsed-entity` oder ein Untertyp, wie `text/Irgendetwas+xml`, dann wird das `encoding`-Attribut der XML-Deklaration innerhalb des Dokuments ignoriert und die Zeichencodierung ist

1. die im `charset`-Parameter des `Content-Type`-Headers angegebene Codierung oder
2. `us-ascii`.

15.5.2 Wie `httplib2` mit Caching umgeht

Erinnern Sie sich, dass ich im vorherigen Abschnitt gesagt habe, Sie sollten ein `httplib2.Http`-Objekt immer mit einem Verzeichnisnamen erstellen? Der Grund dafür ist das Caching.

```
# Fortsetzung des vorherigen Beispiels
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml')①
>>> response2.status                                                    ②
200
>>> content2[:52]                                                      ③
b"<?xml version='1.0' encoding='utf-8'?>\r\n<feed xmlns="
>>> len(content2)
3070
```

① Das sollte Sie nicht überraschen. Genau das haben wir schon beim letzten Mal gemacht; wir verstauen das Ergebnis lediglich in zwei neuen Variablen.

② Der HTTP-Status ist wieder einmal `200`, genau wie beim letzten Mal.

③ Der heruntergeladene Inhalt ist ebenfalls derselbe wie beim letzten Mal.

Was soll das Ganze dann? Beenden Sie Ihre interaktive Shell und starten Sie sie erneut, dann zeige ich es Ihnen.

```
# Bitte verlassen Sie die interaktive Shell
# und starten Sie eine neue.
>>> import httplib2
>>> httplib2.debuglevel = 1          ①
>>> h = httplib2.Http('.cache')      ②
>>> response, content = h.request('http://diveintopython3.org/examples/feed.xml')③
>>> len(content)                     ④
3070
>>> response.status                  ⑤
200
>>> response.fromcache               ⑥
True
```

① Schalten wir Debugging ein und sehen uns an, was über's Netz geht. Dies ist das `httplib2`-Äquivalent zum Anschalten des Debuggings in `http.client`. `httplib2` gibt alle zum Server gesendeten Daten und einige vom Server zurückgesendete wichtige Informationen aus.

② Erstellen Sie ein `httplib2.Http`-Objekt mit demselben Verzeichnisnamen wie zuvor.

③ Fragen Sie dieselbe URL ab. Es scheint nichts zu passieren. Genauer gesagt: Nichts wird zum Server gesendet und nichts wird vom Server zurückgesendet. Es liegt absolut keine Netzwerkaktivität vor.

④ Dennoch haben wir einige Daten „erhalten" – tatsächlich sogar alle.

⑤ Wir haben auch einen HTTP-Statuscode „erhalten", der anzeigt, dass die „Anfrage" erfolgreich war.

⑥ Der Kniff ist dieser: Die „Antwort" wurde aus dem lokalen Cache von `httplib2` erzeugt. Das Verzeichnis, das Sie beim Erstellen des `httplib2.Http`-Objekts erstellt haben, enthält den Cache aller jemals von `httplib2` ausgeführten Befehle.

☞ Um das `httplib2`-Debugging einzuschalten, müssen Sie eine Konstante auf Modulebene setzen (`httplib2.debuglevel`) und dann ein neues `httplib2.Http`-Objekt erstellen. Wollen Sie das Debugging abschalten, müssen Sie die Konstante ändern und wieder ein neues `httplib2.Http`-Objekt erstellen.

Sie haben vorhin die Daten dieser URL abgefragt. Die Anfrage war erfolgreich (`status: 200`). Die Antwort enthielt nicht nur die Daten des Feeds, sondern auch einige Caching-Header, die mitteilten, dass diese Ressource für bis zu 24 h zwischengespeichert werden kann (`Cache-Control: max-age=86400`, was 24 h in Sekunden entspricht). `httplib2` kennt und beachtet die Caching-Header und hat die Antwort im Verzeichnis `.cache` (das Sie beim Erstellen des `Http`-Objekts übergeben haben) gespeichert. Dieser Cache ist bisher nicht abgelaufen, so dass `httplib2` beim zweiten Anfragen der Daten dieser URL einfach das zwischengespeicherte Ergebnis liefert, ohne überhaupt auf das Netzwerk zuzugreifen.

Ich sage „einfach", doch offensichtlich steckt hinter dieser Einfachheit eine immense Komplexität. `httplib2` arbeitet automatisch und per Voreinstellung mit Caching. Sollten Sie einmal herausfinden müssen, ob eine Antwort aus dem Cache kommt, können Sie `response.fromcache` überprüfen. Ansonsten funktioniert es einfach.

Stellen Sie sich nun vor, Sie wollten den Cache umgehen und die Daten stattdessen ein weiteres Mal vom entfernten Server holen. Browser tun dies manchmal auf ausdrücklichen Wunsch des Benutzers. Durch Drücken von `F5` wird beispielsweise die aktuelle Seite neu geladen, durch Drücken von `Strg+F5` dagegen wird der Cache umgangen und die aktuelle Seite erneut vom entfernten Server geholt. Sie denken nun vielleicht: „Hm, ich lösche einfach die Daten aus dem lokalen Cache und frage sie dann erneut an." Das könnten Sie tun, doch bedenken Sie, dass nicht nur Sie und der entfernte Server beteiligt sein könnten. Was ist mit den dazwischenliegenden Proxy-Servern? Diese entziehen sich Ihrer Kontrolle; sie könnten die Daten immer noch zwischengespeichert haben und werden sie Ihnen vergnügt übergeben, da ihr Cache nach wie vor gültig ist.

Statt Ihren lokalen Cache zu bearbeiten, sollten Sie lieber die Funktionen von HTTP nutzen, um sicherzustellen, dass Ihre Anfrage auch wirklich den entfernten Server erreicht.

```
# Fortsetzung des vorherigen Beispiels
>>> response2, content2 = h.request('http://diveintopython3.org/examples/feed.xml',
...     headers={'cache-control':'no-cache'})  ①
connect: (diveintopython3.org, 80)             ②
send: b'GET /examples/feed.xml HTTP/1.1
Host: diveintopython3.org
user-agent: Python-httplib2/$Rev: 259 $
accept-encoding: deflate, gzip
cache-control: no-cache'
reply: 'HTTP/1.1 200 OK'
…gekürzt…
>>> response2.status
200
>>> response2.fromcache                        ③
False
>>> print(dict(response2.items()))             ④
{'status': '200',
 'content-length': '3070',
 'content-location': 'http://diveintopython3.org/examples/feed.xml',
 'accept-ranges': 'bytes',
 'expires': 'Wed, 03 Jun 2009 00:40:26 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Sun, 31 May 2009 22:51:11 GMT',
 'connection': 'close',
 '-content-encoding': 'gzip',
 'etag': '"bfe-255ef5c0"',
 'cache-control': 'max-age=86400',
 'date': 'Tue, 02 Jun 2009 00:40:26 GMT',
 'content-type': 'application/xml'}
```

① `httplib2` erlaubt Ihnen das Hinzufügen beliebiger HTTP-Header zu jeder ausgehenden Anfrage. Zum Umgehen aller Caches (nicht nur Ihres lokalen Caches, sondern auch der Caching-Proxys zwischen Ihnen und dem entfernten Server) fügen Sie einen `no-cache`-Header zum `headers`-Dictionary hinzu.

② Sie sehen nun, dass `httplib2` eine Netzwerkanfrage aussendet. `httplib2` kennt und beachtet Caching-Header in beiden Richtungen – als Teil der eingehenden Antwort und als Teil der ausgehenden Anfrage. Es hat bemerkt, dass Sie einen `no-cache`-Header hinzugefügt haben und hat daher den lokalen Cache umgangen und zum Anfragen der Daten stattdessen den Weg über das Netzwerk gewählt.

③ Diese Antwort wurde nicht von Ihrem lokalen Cache erzeugt. Natürlich wussten Sie das bereits, da Sie ja die Debugging-Informationen der ausgehenden Anfrage gesehen haben. Es ist aber dennoch schön, es noch einmal bestätigt zu sehen.

④ Die Anfrage war erfolgreich; Sie haben den kompletten Feed ein weiteres Mal vom entfernten Server heruntergeladen. Natürlich hat der Server neben dem eigentlichen Feed auch ein ganzes Arsenal an HTTP-Headern gesendet. Darin sind auch Caching-Header enthalten, die `httplib2` zur Aktualisierung seines lokalen Caches in der Hoffnung nutzt, einen Netzwerkzugriff bei der nächsten Anfrage des Feeds zu vermeiden. HTTP-Caching ist so gestaltet, dass die Zugriffe auf den Cache vermehrt und die Zugriffe auf das Netzwerk reduziert werden. Auch wenn Sie diesmal den Cache umgangen haben, wäre der entfernte Server glücklich darüber, wenn Sie das Ergebnis für das nächste Mal zwischenspeichern würden.

15.5.3 Wie `httplib2` mit `Last-Modified`- und `ETag`-Headern umgeht

Die Caching-Header `Cache-Control` und `Expires` werden als *Frische-Indikator* bezeichnet. Sie teilen den Caches unmissverständlich mit, dass Sie jegliche Netzwerkzugriffe vermeiden können, solange der Cache nicht abgelaufen ist. Dieses Verhalten haben Sie im vorigen Abschnitt gesehen: Ist ein Frische-Indikator angegeben, wird durch `httplib2` beim Abruf zwischengespeicherter Daten keinerlei Netzwerkaktivität erzeugt (es sei denn, Sie umgehen den Cache).

Doch was geschieht, wenn die Daten sich geändert haben könnten, es aber nicht getan haben? HTTP definiert zu diesem Zweck die Header `Last-Modified` und `ETag`. Diese Header werden *Validatoren* genannt. Ist der lokale Cache nicht mehr „frisch“, kann der Client die Validatoren in der nächsten Anfrage mitsenden und so erfahren, ob sich die Daten geändert haben. Haben sich die Daten nicht geändert, sendet der Server den Statuscode `304` und *keine Daten*. Der Weg führt so zwar immer noch über das Netzwerk, aber es werden weniger Daten heruntergeladen.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> response, content = h.request('http://diveintopython3.org/')  ①
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
>>> print(dict(response.items()))                                  ②
{'-content-encoding': 'gzip',
 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
```

```
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 'etag': '"7f806d-1a01-9fb97900"',
 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
 'server': 'Apache',
 'status': '200',
 'vary': 'Accept-Encoding,User-Agent'}
>>> len(content)                                                    ③
6657
```

① Statt des Feeds, laden wir diesmal die *Homepage* der Website herunter, welche im HTML-Format vorliegt. Da Sie diese Seite zum ersten Mal anfragen, sendet `httplib2` in der Anfrage nur ein Minimum an Headern.

② Die Antwort enthält eine Vielzahl an HTTP-Headern … doch keinerlei Informationen zum Caching. Sie enthält aber sowohl einen `ETag`- als auch einen `Last-Modified`-Header.

③ Als ich dieses Beispiel erstellt habe, hatte die Seite eine Größe von `6657` Bytes. Seitdem hat sich die Größe wahrscheinlich geändert; machen Sie sich darüber keine Gedanken.

```
# Fortsetzung des vorherigen Beispiels
>>> response, content = h.request('http://diveintopython3.org/')  ①
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
if-none-match: "7f806d-1a01-9fb97900"                              ②
if-modified-since: Tue, 02 Jun 2009 02:51:48 GMT                   ③
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 304 Not Modified'                                 ④
>>> response.fromcache                                             ⑤
True
>>> response.status                                                ⑥
200
>>> response.dict['status']                                        ⑦
'304'
>>> len(content)                                                   ⑧
6657
```

① Sie senden erneut eine Anfrage mit demselben `Http`-Objekt (und demselben lokalen Cache) an dieselbe Seite.

② `httplib2` sendet im Header `If-None-Match` den `ETag`-Validator zurück an den Server.

③ Im `Last-Modified`-Header sendet `httplib2` außerdem auch den `Last-Modified`-Validator zurück an den Server.

④ Der Server hat sich die Validatoren sowie die angeforderte Seite angesehen und ermittelt, dass sich die Seite seit der letzten Anfrage nicht geändert hat. Er sendet daher den Statuscode `304` und *keine Daten* zurück.

⑤ Wieder beim Client angekommen, bemerkt `httplib2` den `304`-Statuscode und lädt den Inhalt der Seite aus seinem Cache.

⑥ Das könnte etwas verwirrend sein. In Wirklichkeit sind zwei Statuscodes vorhanden – `304` (bei dieser Anfrage vom Server zurückgegeben; hat `httplib2` veranlasst, in seinem Cache nachzuschauen) und `200` (bei der vorherigen Anfrage vom Server zurückgegeben; im Cache von `httplib2` zusammen mit den Daten der Seite gespeichert). `response.status` gibt den Status aus dem Cache zurück.

⑦ Möchten Sie den tatsächlich vom Server zurückgegebenen Statuscode erhalten, können Sie diesen in `response.dict` nachschlagen; dies ist ein Dictionary, das alle vom Server zurückgegebenen Header enthält.

⑧ Sie erhalten die Daten immer noch in der `content`-Variable. Im Allgemeinen müssen Sie nicht wissen, warum eine Antwort aus dem Cache kommt. (Sie bräuchten sich noch nicht einmal Gedanken darüber zu machen, ob die Antwort überhaupt aus dem Cache kam, und das ist gut so.) Zu dem Zeitpunkt, da die `request()`-Methode zu ihrem Aufrufer zurückkehrt, hat `httplib2` seinen Cache bereits aktualisiert und die Daten an Sie zurückgegeben.

15.5.4 Wie httplib2 mit Komprimierung umgeht

HTTP unterstützt zwei Komprimierungsarten. `httplib2` unterstützt beide.

```
>>> response, content = h.request('http://diveintopython3.org/')
connect: (diveintopython3.org, 80)
send: b'GET / HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip                          ①
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
>>> print(dict(response.items()))
{'-content-encoding': 'gzip',                           ②
 'accept-ranges': 'bytes',
 'connection': 'close',
 'content-length': '6657',
 'content-location': 'http://diveintopython3.org/',
 'content-type': 'text/html',
 'date': 'Tue, 02 Jun 2009 03:26:54 GMT',
 'etag': '"7f806d-1a01-9fb97900"',
 'last-modified': 'Tue, 02 Jun 2009 02:51:48 GMT',
 'server': 'Apache',
 'status': '304',
 'vary': 'Accept-Encoding,User-Agent'}
```

① Jedes Mal, wenn `httplib2` eine Anfrage sendet, fügt es einen `Accept-Encoding`-Header ein, der dem Server mitteilt, dass es mit *deflate*- und *gzip*-Komprimierung zurechtkommt.

② Im vorliegenden Fall hat der Server mit gzip-komprimierten Daten geantwortet. Zum Zeitpunkt da die `request()`-Methode zurückkehrt, hat `httplib2` den Nachrichtenkörper bereits dekomprimiert und in der `content`-Variable abgelegt. Wenn Sie wissen möchten, ob die Antwort komprimiert wurde oder nicht, können Sie `response['-content-encloding']` überprüfen; andernfalls brauchen Sie sich keine Gedanken darüber zu machen.

15.5.5 Wie httplib2 mit Weiterleitungen umgeht

HTTP definiert zwei Weiterleitungsarten: *temporäre* und *dauerhafte*. An temporären Weiterleitungen ist nichts Besonderes, außer dass man ihnen folgen soll, was `httplib2` automatisch tut.

```
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> response, content =
h.request('http://diveintopython3.org/examples/feed-302.xml')    ①
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-302.xml HTTP/1.1                      ②
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 302 Found'                                      ③
send: b'GET /examples/feed.xml HTTP/1.1                          ④
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 200 OK'
```

① Unter dieser URL existiert kein Feed. Ich habe meinen Server so eingerichtet, dass er eine temporäre Weiterleitung zur korrekten Adresse vornimmt.

② Das ist die Anfrage.

③ Und das ist die Antwort: `302 Found`. Die Antwort enthält außerdem einen `Location`-Header, der auf die tatsächliche URL verweist, hier aber nicht gezeigt wird.

④ `httplib2` dreht sofort um und „folgt“ der Weiterleitung durch eine weitere Anfrage nach der im `Location`-Header angegebenen URL: `http://diveintopython3.org/examples/feed.xml`.

Das Beispiel zeigt alles, was nötig ist, um einer Weiterleitung zu „folgen“. `httplib2` sendet eine Anfrage nach der gewünschten URL. Der Server sendet eine Antwort zurück, die bedeutet „Nein, hier nicht. Sehen Sie stattdessen dort nach“. `httplib2` sendet daraufhin eine weitere Anfrage nach der neuen URL.

```
# Fortsetzung des vorherigen Beispiels
>>> response                                                          ①
{'status': '200',
 'content-length': '3070',
 'content-location': 'http://diveintopython3.org/examples/feed.xml',②
 'accept-ranges': 'bytes',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'vary': 'Accept-Encoding',
 'server': 'Apache',
 'last-modified': 'Wed, 03 Jun 2009 02:20:15 GMT',
 'connection': 'close',
 '-content-encoding': 'gzip',                                       ③
 'etag': '"bfe-4cbbf5c0"',
 'cache-control': 'max-age=86400',                                  ④
 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'application/xml'}
```

① Die Antwort (`response`), die Sie durch den einmaligen Aufruf der `request()`-Methode erhalten, ist die Antwort der endgültigen URL.

② `httplib2` fügt die endgültige URL dem `response`-Dictionary als `content-location` hinzu. Dies ist kein vorm Server erhaltener Header, sondern `httplib2`-spezifisch.

③ Dieser Feed ist übrigens komprimiert.

④ Und kann zwischengespeichert werden. (Das ist, wie Sie gleich sehen werden, wichtig.)

Die Antwort, die Sie erhalten, gibt Ihnen Informationen zu der endgültigen URL. Was machen Sie aber, wenn Sie mehr Informationen zu den URLs dazwischen – die Sie schließlich an die endgültige URL weitergeleitet haben – bekommen möchten? Mit `httplib2` ist auch das kein Problem.

```
# Fortsetzung des vorherigen Beispiels
>>> response.previous                  ①
{'status': '302',
 'content-length': '228',
 'content-location': 'http://diveintopython3.org/examples/feed-302.xml',
 'expires': 'Thu, 04 Jun 2009 02:21:41 GMT',
 'server': 'Apache',
 'connection': 'close',
 'location': 'http://diveintopython3.org/examples/feed.xml',
 'cache-control': 'max-age=86400',
 'date': 'Wed, 03 Jun 2009 02:21:41 GMT',
 'content-type': 'text/html; charset=iso-8859-1'}
```

```
>>> type(response)                    ②
<class 'httplib2.Response'>
>>> type(response.previous)
<class 'httplib2.Response'>
>>> response.previous.previous        ③
>>>
```

① Das Attribut `response.previous` enthält einen Verweis auf das vorherige Antwort-Objekt, dem `httplib2` gefolgt ist, um zum aktuellen Antwort-Objekt zu gelangen.

② Sowohl `response` als auch `response.previous` sind `httplib2.Response`-Objekte.

③ Mit `response.previous.previous` können Sie die Weiterleitungen also noch weiter zurückverfolgen. (Eine URL leitet an eine zweite weiter, die wiederum zu einer dritten URL weiterleitet. Das könnte durchaus passieren.) Im vorliegenden Fall haben wir den Anfang der Weiterleitungskette bereits erreicht, weshalb das Attribut `None` ist.

Was geschieht, wenn Sie dieselbe URL ein weiteres Mal anfragen?

```
# Fortsetzung des vorherigen Beispiels
>>> response2, content2 =
h.request('http://diveintopython3.org/examples/feed-302.xml')    ①
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-302.xml HTTP/1.1                      ②
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 302 Found'                                      ③
>>> content2 == content                                          ④
True
```

① Dieselbe URL; dasselbe `httplib2.Http`-Objekt (und daher auch derselbe Cache).

② Der `302`-Statuscode wurde nicht zwischengespeichert, also sendet `httplib2` eine weitere Anfrage an die URL.

③ Abermals antwortet der Server mit `302`. Beachten Sie aber, dass es *keine* zweite Anfrage nach der endgültigen URL, `http://diveintopython3.org/examples/feed.xml`, gegeben hat. Diese Antwort befindet sich im Cache (denken Sie an den `Cache-Control`-Header aus dem vorherigen Beispiel). Als `httplib2` den Statuscode `302 Found` erhalten hatte, hat es seinen Cache überprüft. Da der Cache eine aktuelle Kopie von `http://diveintopython3.org/examples/feed.xml` enthielt, wurde die URL nicht noch einmal angefragt.

④ Zum Zeitpunkt der Rückkehr der `request()`-Methode hat sie die Daten des Feeds aus dem Cache gelesen und zurückgegeben. Natürlich sind es dieselben Daten wie beim letzten Mal.

Bei temporären Weiterleitungen müssen Sie also gar nichts Besonderes tun. `httplib2` folgt ihnen automatisch; wir brauchen uns beim Weiterleiten einer URL zu einer anderen nicht mit Komprimierung, Caching, ETags oder anderen HTTP-Funktionen zu beschäftigen.

Dauerhafte Weiterleitungen sind genauso einfach.

```
# Fortsetzung des vorherigen Beispiels
>>> response, content =
h.request('http://diveintopython3.org/examples/feed-301.xml')    ①
connect: (diveintopython3.org, 80)
send: b'GET /examples/feed-301.xml HTTP/1.1
Host: diveintopython3.org
accept-encoding: deflate, gzip
user-agent: Python-httplib2/$Rev: 259 $'
reply: 'HTTP/1.1 301 Moved Permanently'                           ②
>>> response.fromcache                                            ③
True
```

① Wieder einmal existiert diese URL nicht wirklich. Ich habe meinen Server so eingestellt, dass er eine dauerhafte Weiterleitung zu `http://diveintopython3.org/examples/feed.xml` vornimmt.

② Hier ist er: Statuscode `301`. Doch es gab wieder keine Anfrage der weitergeleiteten URL. Warum nicht? Weil sie bereits lokal zwischengespeichert ist.

③ `httplib2` ist der Weiterleitung direkt zu seinem Cache „gefolgt“.

Aber es geht noch weiter.

```
# Fortsetzung des vorherigen Beispiels
>>> response2, content2 =
h.request('http://diveintopython3.org/examples/feed-301.xml')  ①
>>> response2.fromcache                                        ②
True
>>> content2 == content                                        ③
True
```

① Hier sehen Sie den Unterschied zwischen temporären und dauerhaften Weiterleitungen: Ist `httplib2` einmal einer dauerhaften Weiterleitung gefolgt, werden alle weiteren Anfragen dieser URL auf die Ziel-URL umgeschrieben, ohne dass etwas über das Netzwerk gesendet wird. Debugging ist nach wie vor eingeschaltet, und dennoch gibt es keine Anzeige von Netzwerkaktivität.

② Ja, diese Antwort stammt aus dem lokalen Cache.

③ Ja, Sie haben den kompletten Feed (aus dem Cache) erhalten.

HTTP. Es funktioniert.

15.6 Über HTTP-GET hinaus

HTTP-Webdienste sind nicht auf GET-Anfragen beschränkt. Was, wenn Sie etwas Neues erstellen möchten? Wann immer Sie einen Beitrag in einem Forum veröffentlichen, Ihr Blog aktualisieren, oder Ihren Status bei einem Microblogging-Service wie *Twitter* oder *Identi.ca* ändern, verwenden Sie vermutlich HTTP-POST.

Sowohl Twitter als auch Identi.ca bieten eine einfache auf HTTP basierende API zum Veröffentlichen und Aktualisieren Ihres Status mit 140 oder weniger Zeichen. Sehen wir uns die API-Dokumentation von Identi.ca zum Aktualisieren Ihres Status an:

Identi.ca REST-API-Methode: Status/Aktualisierung
Aktualisiert den Status des angemeldeten Benutzers. Benötigt den unten angegebenen status-Parameter. Die Anfrage muss ein POST sein.

URL
`https://identi.ca/api/statuses/update.format`

Formate
`xml`, `json`, `rss`, `atom`

HTTP-Methode(n)
`POST`

Setzt Authentifizierung voraus
Ja

Parameter
`status`. Benötigt. Der Text Ihrer Satus-Aktualisierung. URL-Codierung nach Bedarf.

Wie funktioniert das? Zum Veröffentlichen einer neuen Nachricht auf Identi.ca müssen Sie eine HTTP-POST-Anfrage an `https://identi.ca/api/statuses/update.`*`format`* senden. (`format` ist nicht Teil der URL; Sie müssen dies durch das Datenformat ersetzen, das der Server als Antwort auf Ihre Anfrage senden soll. Möchten Sie die Antwort z. B. in XML, müssen Sie die Anfrage an `https://identi.ca/api/statuses/update.xml` senden.) Die Anfrage muss den Parameter `status` enthalten, der den Text Ihrer Status-Aktualisierung beinhaltet. Die Anfrage muss authentifiziert erfolgen.

Authentifiziert? Ja, genau. Um Ihren Status aktualisieren zu können, müssen Sie nachweisen, wer Sie sind. Identi.ca ist kein Wiki; nur Sie allein können Ihren eigenen Status aktualisieren. Identi.ca nutzt die HTTP-Basisauthentifizierung (auch bekannt als *RFC 2617*) über SSL, um eine sichere und dennoch einfach zu bedienende Authentifizierung zur Verfügung zu stellen. `httplib2` unterstützt sowohl SSL als auch die HTTP-Basisauthentifizierung, das ist also kein Problem.

Eine POST-Anfrage unterscheidet sich von einer GET-Anfrage darin, dass sie die Daten enthält, die Sie an den Server senden möchten. Die einzigen Daten, die

diese API-Methode *benötigt*, ist `status`. `status` sollte URL-codiert sein. Dies ist ein sehr einfaches Serialisierungsformat, das Schlüssel-Wert-Paare (also ein Dictionary) übernimmt und in einen String umwandelt.

```
>>> from urllib.parse import urlencode                ①
>>> data = {'status': 'Test update from Python 3'}  ②
>>> urlencode(data)                                   ③
'status=Test+update+from+Python+3'
```

① Python enthält eine Funktion zur URL-Codierung eines Dictionarys: `urllib.parse.urlencode()`.

② Die Identi.ca-API benötigt ein Dictionary in dieser Form. Es enthält einen Schlüssel, `status`, dessen Wert der Text einer einzelnen Status-Aktualisierung ist.

③ So sieht der URL-codierte String aus. Dieser wird innerhalb der `HTTP-POST`-Anfrage über das Netz an den Identi.ca-API-Server gesendet.

```
>>> from urllib.parse import urlencode
>>> import httplib2
>>> httplib2.debuglevel = 1
>>> h = httplib2.Http('.cache')
>>> data = {'status': 'Test update from Python 3'}
>>> h.add_credentials('diveintomark', 'MY_SECRET_PASSWORD', 'identi.ca')   ①
>>> resp, content = h.request('https://identi.ca/api/statuses/update.xml',
...     'POST',                                                            ②
...     urlencode(data),                                                   ③
...     headers={'Content-Type': 'application/x-www-form-urlencoded'})     ④
```

① So wird die Authentifizierung von `httplib2` gehandhabt. Speichern Sie Ihren Benutzernamen und Ihr Passwort mithilfe der `add_credentials()`-Methode. Versucht `httplib2` die Anfrage zu senden, antwortet der Server mit dem Statuscode `401 Unauthorized` und listet auf, welche Arten der Authentifizierung er unterstützt (im Header `WWW-Authenticate`). `httplib2` erstellt daraufhin automatisch einen `Authorization`-Header und sendet die Anfrage erneut.

② Der zweite Parameter ist die Art der HTTP-Anfrage, hier also `POST`.

③ Der dritte Parameter enthält die an den Server zu sendenden Daten. Wir senden das URL-codierte Dictionary, das die Nachricht enthält.

④ Schließlich müssen wir dem Server noch mitteilen, dass es sich um URL-codierte Daten handelt.

☞ Der dritte Parameter der `add_credentials()`-Methode ist die Domain, für die die Nutzerdaten gelten. Sie sollten diese immer angeben! Wenn Sie die Domain auslassen und das `httplib2.Http`-Objekt später noch einmal bei einer anderen Seite mit Authentifizierung verwenden, könnte `httplib2` die Nutzerdaten der einen Seite an die andere Seite übergeben.

Dies geht über's Netz:

```
# Fortsetzung des vorherigen Beispiels
send: b'POST /api/statuses/update.xml HTTP/1.1
Host: identi.ca
Accept-Encoding: identity
Content-Length: 32
content-type: application/x-www-form-urlencoded
user-agent: Python-httplib2/$Rev: 259 $

status=Test+update+from+Python+3'
reply: 'HTTP/1.1 401 Unauthorized'                          ①
send: b'POST /api/statuses/update.xml HTTP/1.1              ②
Host: identi.ca
Accept-Encoding: identity
Content-Length: 32
content-type: application/x-www-form-urlencoded
authorization: Basic SECRET_HASH_CONSTRUCTED_BY_HTTPLIB2    ③
user-agent: Python-httplib2/$Rev: 259 $

status=Test+update+from+Python+3'
reply: 'HTTP/1.1 200 OK'                                    ④
```

① Nach der ersten Anfrage antwortet der Server mit dem Statuscode `401 Unauthorized`. `httplib2` sendet erst *Authentifizierungsheader*, wenn der Server ausdrücklich danach verlangt. Nun wissen Sie, wie der Server dies tut.

② `httplib2` macht sofort kehrt und fragt dieselbe URL ein zweites Mal an.

③ Diesmal enthält die Anfrage den Benutzernamen und das Passwort. Beides haben Sie mit der `add_credentials()`-Methode hinzugefügt.

④ Es hat funktioniert!

Was sendet der Server nach einer erfolgreichen Anfrage zurück? Das hängt ganz von der Webdienst-API ab. Bei manchen Protokollen (wie dem *Atom Publishing Protocol*) sendet der Server den Statuscode `201 Created` und, innerhalb des `Location`-Headers, den Ort der neu erstellten Ressource. Identi.ca sendet `200 OK` und ein XML-Dokument, das Informationen zu der neu erstellten Ressource enthält.

```
# Fortsetzung des vorherigen Beispiels
>>> print(content.decode('utf-8'))                          ①
<?xml version="1.0" encoding="UTF-8"?>
<status>
 <text>Test update from Python 3</text>                     ②
 <truncated>false</truncated>
 <created_at>Wed Jun 10 03:53:46 +0000 2009</created_at>
 <in_reply_to_status_id></in_reply_to_status_id>
 <source>api</source>
 <id>5131472</id>                                           ③
```

```
 <in_reply_to_user_id></in_reply_to_user_id>
 <in_reply_to_screen_name></in_reply_to_screen_name>
 <favorited>false</favorited>
 <user>
  <id>3212</id>
  <name>Mark Pilgrim</name>
  <screen_name>diveintomark</screen_name>
  <location>27502, US</location>
  <description>tech writer, husband, father</description>
  <profile_image_url>http://avatar.identi.ca/3212-48-20081216000626.png
  </profile_image_url>
  <url>http://diveintomark.org/</url>
  <protected>false</protected>
  <followers_count>329</followers_count>
  <profile_background_color></profile_background_color>
  <profile_text_color></profile_text_color>
  <profile_link_color></profile_link_color>
  <profile_sidebar_fill_color></profile_sidebar_fill_color>
  <profile_sidebar_border_color></profile_sidebar_border_color>
  <friends_count>2</friends_count>
  <created_at>Wed Jul 02 22:03:58 +0000 2008</created_at>
  <favourites_count>30768</favourites_count>
  <utc_offset>0</utc_offset>
  <time_zone>UTC</time_zone>
  <profile_background_image_url></profile_background_image_url>
  <profile_background_tile>false</profile_background_tile>
  <statuses_count>122</statuses_count>
  <following>false</following>
  <notifications>false</notifications>
</user>
</status>
```

① Vergessen Sie nicht, dass die von `httplib2` zurückgegebenen Daten immer Bytes sind, niemals ein String. Um sie ihn einen String umzuwandeln, müssen Sie sie unter Verwendung der passenden Zeichencodierung decodieren. Die Identi.ca-API gibt Ergebnisse immer in UTF-8 zurück. Dieser Teil ist also kein Problem.

② Hier ist der gerade veröffentlichte Nachrichtentext.

③ Dies ist die einmalige ID der neuen Nachricht. Identi.ca erstellt daraus eine URL, mit der man die Nachricht im Browser lesen kann.

15.7 Über `HTTP-POST` hinaus

HTTP ist nicht auf `GET` und `POST` beschränkt. Sicherlich sind dies die meistgenutzten Anfragearten, besonders in Webbrowsern, doch Webdienst-APIs können oft mehr als `GET` und `POST`. `httplib2` ist dafür bereit.

```
# Fortsetzung des vorherigen Beispiels
>>> from xml.etree import ElementTree as etree
>>> tree = etree.fromstring(content)            ①
>>> status_id = tree.findtext('id')             ②
>>> status_id
'5131472'
>>> url =
'https://identi.ca/api/statuses/destroy/{0}.xml'.format(status_id)  ③
>>> resp, deleted_content = h.request(url, 'DELETE')                ④
```

① Der Server hat XML zurückgegeben, richtig? Sie wissen, wie man XML verarbeitet (parst).

② Die Methode `findtext()` sucht das erste Vorkommen des übergebenen Ausdrucks und entnimmt den dazugehörigen Textinhalt. Im vorliegenden Fall suchen wir nur nach dem <id>-Element.

③ Basierend auf dem Textinhalt des <id>-Elements können wir eine URL zum Löschen der gerade erstellten Nachricht erzeugen.

④ Zum Löschen einer Nachricht müssen Sie lediglich eine HTTP-DELETE-Anfrage an die URL senden.

Und das geht über's Netz:

```
send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1      ①
Host: identi.ca
Accept-Encoding: identity
user-agent: Python-httplib2/$Rev: 259 $

'
reply: 'HTTP/1.1 401 Unauthorized'                             ②
send: b'DELETE /api/statuses/destroy/5131472.xml HTTP/1.1      ③
Host: identi.ca
Accept-Encoding: identity
authorization: Basic SECRET_HASH_CONSTRUCTED_BY_HTTPLIB2       ④
user-agent: Python-httplib2/$Rev: 259 $

'
reply: 'HTTP/1.1 200 OK'                                       ⑤
>>> resp.status
200
```

① „Lösche diese Nachricht."

② „Tut mir leid, das kann ich nicht tun."

③ „Nicht autorisiert!? Hmmm. Lösche diese Nachricht bitte, …

④ … hier sind mein Benutzername und mein Passwort."

⑤ „Betrachte es als erledigt!"

Kapitel 16
Fallstudie: `chardet` zu Python 3 portieren

16.1 Los geht's

Unbekannte oder falsche Zeichencodierung ist der Hauptgrund für unlesbaren Text im WWW, in Ihrem E-Mail-Postfach und mit Sicherheit bei jedem Computersystem, das jemals programmiert wurde. In Kap. 5 habe ich über die Geschichte der Zeichencodierungen und die Erfindung von Unicode, „eine Codierung, sie alle zu beherrschen", geschrieben. Ich wäre zutiefst erfreut darüber, nie mehr auch nur ein einziges unlesbares Zeichen auf einer Webseite zu sehen, weil alle Website-Editoren korrekte Codierungsinformationen speicherten, alle Übertragungsprotokolle Unicode kennen und alle mit Text umgehenden Systeme die Genauigkeit beim Umwandeln von Zeichencodierungen beibehalten würden.

Außerdem hätte ich gerne ein Pony.

Ein Unicode-Pony.

Ein Unipony sozusagen.

Ich werde mich mit der automatischen Zeichencodierungserkennung zufrieden geben.

16.2 Was ist die automatische Zeichencodierungserkennung?

Mit ihrer Hilfe ist es möglich, die Zeichencodierung einer in einer unbekannten Codierung vorliegenden Bytefolge zu bestimmen, so dass man den Text lesen kann. Das ist so, als knacke man einen Code ohne den Decodierungsschlüssel zu besitzen.

16.2.1 Ist das nicht unmöglich?

Generell schon. Doch manche Zeichencodierungen sind auf bestimmte Sprachen abgestimmt und Sprachen sind kein Zufallsprodukt. Einige Zeichenfolgen treten gehäuft auf, während andere keinen Sinn ergeben. Eine Person, die fließend Englisch spricht, eine Zeitung öffnet und die Schlagzeile „txzqJv 2!dasd0a QqdKjvz"

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_16, © Springer-Verlag Berlin Heidelberg 2010

vorfindet, erkennt sofort, dass dies kein Englisch ist (auch wenn es englische Buchstaben sind). Ein Computeralgorithmus kann durch das Analysieren vieler „typischer“ Texte diese Art von Sprachgewandtheit simulieren und dadurch eine begründete Vermutung über die Sprache eines Textes abgeben.

Zeichencodierungserkennung ist somit eigentlich Spracherkennung in Verbindung mit dem Wissen, welche Sprachen welche Zeichencodierungen verwenden.

16.2.2 Existiert solch ein Algorithmus?

Aber ja! Alle großen Browser besitzen eine automatische Zeichencodierungserkennung, da das WWW voller Seiten ohne Codierungsinformationen ist. Mozilla Firefox enthält eine Open-Source-Bibliothek zur automatischen Zeichencodierungserkennung. Ich habe diese Bibliothek zu Python 2 portiert und das Modul auf den Namen `chardet` getauft. In diesem Kapitel werden wir das `chardet`-Modul nun Schritt für Schritt von Python 2 zu Python 3 portieren.

16.3 Das `chardet`-Modul

Bevor wir mit der Portierung des Codes beginnen, sollten Sie vielleicht erst einmal verstehen, wie der Code funktioniert. Dies ist ein kleiner Leitfaden zur Navigation innerhalb des Codes. Die `chardet`-Bibliothek ist zu groß, um sie hier abzudrucken; Sie können sie aber unter `chardet.feedparser.org` herunterladen.

Der Haupteinstiegspunkt des Erkennungsalgorithmus ist `universaldetector.py`, worin eine Klasse, `UniversalDetector`, enthalten ist. (Sie könnten auf die Idee kommen, dass der Haupteinstiegspunkt die Funktion `detect` in der Datei `chardet/__init__.py` ist, doch diese Funktion erstellt lediglich ein `UniversalDetector`-Objekt, ruft es auf und gibt sein Ergebnis zurück.)

`UniversalDetector` kann 5 Codierungskategorien verarbeiten:

1. UTF-`n` mit einer *Byte Order Mark* (BOM). Dazu gehören auch UTF-8, Big-Endian- und Little-Endian-Varianten von UTF-16 sowie alle UTF-32-Varianten mit 4 Bytes.
2. Escape-Codierungen, die vollständig kompatibel zu 7-Bit-ASCII sind und Escape-Sequenzen am Beginn von Nicht-ASCII-Zeichen verwenden. Beispiele: `ISO-2022-JP` (Japanisch) und `HZ-GB-2312` (Chinesisch).
3. Multi-Byte-Codierungen, bei denen jedes Zeichen durch eine variable Anzahl an Bytes dargestellt wird. Beispiele: `Big5` (Chinesisch), `SHIFT_JIS` (Japanisch), `EUC-KR` (Koreanisch) und UTF-8 *ohne* BOM.
4. Single-Byte-Codierungen, bei denen jedes Zeichen durch ein Byte dargestellt wird. Beispiele: `KOI8-R` (Russisch), `windows-1255` (Hebräisch) und `TIS-620` (Thailändisch).

5. windows-1252, eine Codierung, die hauptsächlich unter Windows von Leuten aus dem mittleren Management verwendet wird, die eine Zeichencodierung nicht von einem Erdloch unterscheiden könnten.

16.3.1 UTF-n mit einer Byte Order Mark

Beginnt der Text mit einer BOM, können wir davon ausgehen, dass der Text in UTF-8, UTF-16 oder UTF-32 codiert ist. (Die BOM teilt uns genau mit, welche dieser Codierungen angewandt wird; dafür ist sie ja da.) Die Verarbeitung findet in UniversalDetector statt und das Ergebnis wird sofort zurückgegeben.

16.3.2 Escape-Codierungen

Enthält der Text eine erkennbare Escape-Sequenz, könnte dies auf eine Zeichencodierung mit Escape-Sequenzen hindeuten. UniversalDetector erstellt einen EscCharSetProber (in escprober.py definiert) und führt diesem den Text zu. (*prober* = Prüfer, Untersucher usw.)

EscCharSetProber erstellt, basierend auf Modellen von HZ-GB-2312, ISO-2022-CN, ISO-2022-JP und ISO-2022-KR (in escsm.py definiert) eine Reihe von Zustandsautomaten. EscCharSetProber führt den Text jedem dieser Zustandsautomaten Byte für Byte zu. Erkennt einer der Zustandsautomaten zweifelsfrei die Codierung, gibt EscCharSetProber das positive Ergebnis sofort an UniversalDetector zurück, der es wiederum an den Aufrufer zurückgibt. Trifft ein Zustandsautomat auf unerlaubte Zeichen, wird er verlassen und mit den anderen Zustandsautomaten fortgefahren.

16.3.3 Multi-Byte-Codierungen

Keine BOM vorausgesetzt, überprüft UniversalDetector, ob der Text Zeichen mit höherwertigen Bits enthält. Sollte das der Fall sein, werden einige „Prüfer“ zur Erkennung von Multi-Byte-Codierungen, Single-Byte-Codierungen und, als letzter Ausweg, windows-1252 erstellt.

Der Prüfer für Multi-Byte-Codierungen, MBCSGroupProber (in mbcsgroupprober.py definiert), ist in Wirklichkeit lediglich eine Hülle, die eine Gruppe anderer Prüfer verwaltet. Ein Prüfer für jede Multi-Byte-Codierung: Big5, GB2312, EUC-TW, EUC-KR, EUC-JP, SHIFT_JIS und UTF-8. MBCSGroupProber führt den Text jedem dieser codierungsspezifischen Prüfer zu und überprüft die Ergebnisse. Meldet ein Prüfer, dass er eine unerlaubte Bytefolge gefunden hat, wird dieser Prüfer von der weiteren Verarbeitung ausgeschlossen (so

werden z. B. alle nachfolgenden Aufrufe von `UniversalDetector.feed()` diesen Prüfer überspringen). Meldet ein Prüfer dagegen, dass er sich ziemlich sicher ist, die Codierung erkannt zu haben, gibt `MBCSGroupProber` dieses positive Ergebnis an `UniversalDetector` weiter, der es dann dem Aufrufer meldet.

Die meisten Prüfer für Multi-Byte-Codierungen sind von `MultiByteCharSetProber` (in `mbcharsetprober.py` definiert) abgeleitet, hängen einfach den passenden Zustandsautomaten und Verteilungsanalysator an und überlassen `MultiByteCharSetProber` den Rest der Arbeit. `MultiByteCharSetProber` lässt den Text Byte für Byte durch den codierungsspezifischen Zustandsautomaten laufen und sucht nach Bytefolgen, die ein eindeutiges positives oder negatives Ergebnis zur Folge haben. Zur selben Zeit führt `MultiByteCharSetProber` den Text einem codierungsspezifischen Verteilungsanalysator zu.

Die Verteilungsanalysatoren (jeder in `chardistribution.py` definiert) verwenden sprachspezifische Modelle der häufigsten Zeichen der jeweiligen Sprache. Hat `MultiByteCharSetProber` dem Verteilungsanalysator genug Text zugeführt, berechnet dieser eine Bewertung der Übereinstimmung, basierend auf der Zahl der häufig genutzten Zeichen, der Gesamtzahl der Zeichen und einem sprachspezifischen Verteilungsverhältnis. Ist die Übereinstimmung groß genug, gibt `MultiByteCharSetProber` das Ergebnis zurück an `MBCSGroupProber`, der es an `UniversalDetector` zurückgibt und dieser schließlich an den Aufrufer.

Beim Japanischen ist es schwieriger. Die Verteilungsanalyse einzelner Zeichen ist nicht immer ausreichend, um zwischen `EUC-JP` und `SHIFT_JIS` zu unterscheiden, daher nutzt der `SJISProber` (in `sjisprober.py` definiert) auch die Verteilungsanalyse zweier aufeinander folgender Zeichen. `SJISContextAnalysis` und `EUCJPContextAnalysis` (beide in `jpcntx.py` definiert und beide von der allgemeinen Klasse `JapaneseContextAnalysis` abgeleitet) überprüfen die Häufigkeit von Hiragana-Silbenzeichen innerhalb des Texts. Wurde genug Text verarbeitet, geben sie eine Übereinstimmungsrate an `SJISProber` zurück, der beide Analysatoren prüft und die höhere Übereinstimmungsrate an `MBCSGroupProber` zurückgibt.

16.3.4 Single-Byte-Codierungen

Der Prüfer für Single-Byte-Codierungen, `SBCSGroupProber` (in `sbcsgroupprober.py` definiert), ist ebenfalls lediglich eine Hülle, die eine Gruppe anderer Prüfer verwaltet. Ein Prüfer für jede Kombination aus Single-Byte-Codierung und Sprache: `windows-1251`, `KOI8-R`, `ISO-8859-5`, `MacCyrillic`, `IBM855` und `IBM866` (Russisch); `ISO-8859-7` und `windows-1253` (Griechisch); `ISO-8859-5` und `windows-1251` (Bulgarisch); `ISO-8859-2` und `windows-1250` (Ungarisch); `TIS-620` (Thailändisch); `windows-1255` und `ISO-8859-8` (Hebräisch).

`SBCSGroupProber` führt den Text jedem dieser codierungs- und sprachspezifischen Prüfer zu und überprüft die Ergebnisse. Diese Prüfer sind alle als eine einzi-

ge Klasse implementiert, SingleByteCharSetProber (in sbcharsetprober.py definiert), die ein Sprachmodell als Argument übernimmt. Das Sprachmodell definiert die Häufigkeit, mit der verschiedene Folgen von zwei Zeichen in einem typischen Text auftreten. SingleByteCharSetProber verarbeitet den Text und zählt die häufigsten 2-Zeichen-Folgen. Wurde genug Text verarbeitet, berechnet er eine Übereinstimmungsrate, basierend auf der Zahl der häufig genutzten Folgen, der Gesamtzahl der Zeichen und einem sprachspezifischen Verteilungsverhältnis.

Hebräisch wird als Spezialfall behandelt. Basiert der Text auf 2-Zeichen-Verteilungsanalyse, versucht HebrewProber (in hebrewprober.py definiert) zwischen „visuellem Hebräisch“ (der Quelltext wird Zeile für Zeile „rückwärts“ gespeichert und dann wortgetreu von rechts nach links angezeigt) und „logischem Hebräisch“ (der Quelltext wird in Leserichtung gespeichert und dann von rechts nach links wiedergegeben) zu unterscheiden. Da bestimmte Zeichen anders codiert werden, wenn Sie am Ende eines Wortes auftreten, können wir die Richtung des Quelltextes herausfinden und die dazu passende Codierung zurückgeben (windows-1255 für „logisches Hebräisch“ oder ISO-8859-8 für „visuelles Hebräisch“).

16.3.5 windows-1252

Erkennt UniversalDetector ein Zeichen mit einem höherwertigen Bit und keiner der anderen Prüfer gibt ein verlässliches Ergebnis zurück, erstellt er einen Latin1Prober (in latin1prober.py definiert) und versucht englischen Text in der windows-1252-Codierung zu erkennen. Diese Erkennung ist unweigerlich unzuverlässig, da englische Buchstaben in vielen verschiedenen Codierungen auf die gleiche Weise codiert sind. Die einzige Möglichkeit windows-1252 zu erkennen besteht darin, nach Symbolen wie Anführungszeichen, Apostrophen, Copyright-Symbolen usw. zu suchen. Latin1Prober reduziert seine Übereinstimmungsbewertung automatisch, um exakteren Prüfern eine Erkennung zu ermöglichen.

16.4 2to3 ausführen

Wir werden das chardet-Modul nun von Python 2 zu Python 3 portieren. Python 3 enthält ein Skript namens 2to3, das Quellcode von Python 2 als Eingabe übernimmt und soviel wie möglich davon automatisch in Quellcode für Python 3 konvertiert. Manchmal ist das sehr einfach – eine Funktion wurde umbenannt oder in ein anderes Modul verschoben – doch manchmal kann es auch sehr komplex sein. Um zu erfahren, was das Skript alles kann, sehen Sie sich den *Anhang A* an. In diesem Kapitel beginnen wir damit, 2to3 auf das chardet-Paket anzuwenden, doch selbst wenn die automatische Verarbeitung erfolgt ist, haben wir noch viel Arbeit vor uns.

Das chardet-Paket ist auf mehrere im gleichen Verzeichnis liegende Dateien aufgeteilt. Das 2to3-Skript macht das Umwandeln mehrerer Dateien zu einem

Kinderspiel: Übergeben Sie das Verzeichnis einfach als Kommandozeilenargument und `2to3` konvertiert der Reihe nach alle Dateien darin.

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w chardet\
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- chardet\__init__.py (original)
+++ chardet\__init__.py (refactored)
@@ -18,7 +18,7 @@
 __version__ = "1.0.1"

 def detect(aBuf):
-    import universaldetector
+    from . import universaldetector
     u = universaldetector.UniversalDetector()
     u.reset()
     u.feed(aBuf)
--- chardet\big5prober.py (original)
+++ chardet\big5prober.py (refactored)
@@ -25,10 +25,10 @@
 # 02110-1301  USA
 ######################### END LICENSE BLOCK #########################

-from mbcharsetprober import MultiByteCharSetProber
-from codingstatemachine import CodingStateMachine
-from chardistribution import Big5DistributionAnalysis
-from mbcssm import Big5SMModel
+from .mbcharsetprober import MultiByteCharSetProber
+from .codingstatemachine import CodingStateMachine
+from .chardistribution import Big5DistributionAnalysis
+from .mbcssm import Big5SMModel

 class Big5Prober(MultiByteCharSetProber):
     def __init__(self):
--- chardet\chardistribution.py (original)
+++ chardet\chardistribution.py (refactored)
@@ -25,12 +25,12 @@
 # 02110-1301  USA
 ######################### END LICENSE BLOCK #########################

-import constants
-from euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE,
EUCTW_TYPICAL_DISTRIBUTION_RATIO
-from euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE,
EUCKR_TYPICAL_DISTRIBUTION_RATIO
-from gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE,
GB2312_TYPICAL_DISTRIBUTION_RATIO
-from big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE,
BIG5_TYPICAL_DISTRIBUTION_RATIO
-from jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE,
JIS_TYPICAL_DISTRIBUTION_RATIO
+from . import constants
+from .euctwfreq import EUCTWCharToFreqOrder, EUCTW_TABLE_SIZE,
EUCTW_TYPICAL_DISTRIBUTION_RATIO
+from .euckrfreq import EUCKRCharToFreqOrder, EUCKR_TABLE_SIZE,
EUCKR_TYPICAL_DISTRIBUTION_RATIO
+from .gb2312freq import GB2312CharToFreqOrder, GB2312_TABLE_SIZE,
GB2312_TYPICAL_DISTRIBUTION_RATIO
+from .big5freq import Big5CharToFreqOrder, BIG5_TABLE_SIZE,
BIG5_TYPICAL_DISTRIBUTION_RATIO
+from .jisfreq import JISCharToFreqOrder, JIS_TABLE_SIZE,
JIS_TYPICAL_DISTRIBUTION_RATIO

 ENOUGH_DATA_THRESHOLD = 1024
 SURE_YES = 0.99
.
.
```

```
. (so geht es für einige Zeit weiter)
.
.
RefactoringTool: Files that were modified:
RefactoringTool: chardet\__init__.py
RefactoringTool: chardet\big5prober.py
RefactoringTool: chardet\chardistribution.py
RefactoringTool: chardet\charsetgroupprober.py
RefactoringTool: chardet\codingstatemachine.py
RefactoringTool: chardet\constants.py
RefactoringTool: chardet\escprober.py
RefactoringTool: chardet\escsm.py
RefactoringTool: chardet\eucjpprober.py
RefactoringTool: chardet\euckrprober.py
RefactoringTool: chardet\euctwprober.py
RefactoringTool: chardet\gb2312prober.py
RefactoringTool: chardet\hebrewprober.py
RefactoringTool: chardet\jpcntx.py
RefactoringTool: chardet\langbulgarianmodel.py
RefactoringTool: chardet\langcyrillicmodel.py
RefactoringTool: chardet\langgreekmodel.py
RefactoringTool: chardet\langhebrewmodel.py
RefactoringTool: chardet\langhungarianmodel.py
RefactoringTool: chardet\langthaimodel.py
RefactoringTool: chardet\latin1prober.py
RefactoringTool: chardet\mbcharsetprober.py
RefactoringTool: chardet\mbcsgroupprober.py
RefactoringTool: chardet\mbcssm.py
RefactoringTool: chardet\sbcharsetprober.py
RefactoringTool: chardet\sbcsgroupprober.py
RefactoringTool: chardet\sjisprober.py
RefactoringTool: chardet\universaldetector.py
RefactoringTool: chardet\utf8prober.py
```

Wenden Sie `2to3` nun auf die Testdatei `test.py` an.

```
C:\home\chardet> python c:\Python30\Tools\Scripts\2to3.py -w test.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
--- test.py (original)
+++ test.py (refactored)
@@ -4,7 +4,7 @@
 count = 0
 u = UniversalDetector()
 for f in glob.glob(sys.argv[1]):
-    print f.ljust(60),
+    print(f.ljust(60), end=' ')
     u.reset()
     for line in file(f, 'rb'):
         u.feed(line)
@@ -12,8 +12,8 @@
     u.close()
     result = u.result
     if result['encoding']:
-        print result['encoding'], 'with confidence',
result['confidence']
+        print(result['encoding'], 'with confidence',
result['confidence'])
     else:
-        print '******** no result'
+        print('******** no result')
     count += 1
-print count, 'tests'
+print(count, 'tests')
RefactoringTool: Files that were modified:
RefactoringTool: test.py
```

Nun, das war nicht besonders schwierig. Es wurden lediglich ein paar import- und print-Anweisungen konvertiert. Wo wir gerade dabei sind: Was war das bei den Importanweisungen das Problem? Um dies zu verstehen, müssen Sie erst einmal verstehen, wie das chardet-Modul auf mehrere Dateien aufgeteilt ist.

16.5 Mehr-Dateien-Module

chardet ist ein Mehr-Dateien-Modul. Ich hätte mich auch dazu entscheiden können, den kompletten Code in einer Datei unterzubringen (namens chardet.py), doch das habe ich nicht getan. Stattdessen habe ich ein Verzeichnis (namens chardet) und darin die Datei __init__.py erstellt. *Findet Python innerhalb eines Verzeichnisses eine Datei namens __init__.py, nimmt es an, dass alle Dateien in diesem Verzeichnis zum selben Modul gehören*. Der Name des Moduls ist der Name des Verzeichnisses. Dateien innerhalb des Verzeichnisses können auf andere Dateien im selben Verzeichnis oder sogar in Unterverzeichnissen verweisen. (Dazu gleich mehr.) Alle diese Dateien werden jedoch von anderem Python-Code als ein einziges Modul angesehen – als ob alle Funktionen und Klassen sich in einer .py-Datei befänden.

Was steht in der Datei __init__.py? Nichts. Alles. Irgendwas dazwischen. __init__.py muss nichts definieren; die Datei kann leer sein. Sie können aber auch Ihre Haupteinstiegsfunktion darin definieren. Oder alle Funktionen. Oder alle außer einer.

☞ Ein Verzeichnis, das eine __init__.py-Datei enthält, wird immer als Mehr-Dateien-Modul behandelt. Ohne eine __init__.py-Datei ist es jedoch lediglich ein Verzeichnis zusammenhangloser .py-Dateien.

Sehen wir uns an, wie das in der Praxis funktioniert.

```
>>> import chardet
>>> dir(chardet)                ①
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__path__', '__version__', 'detect']
>>> chardet                     ②
<module 'chardet' from 'C:\Python31\lib\site-
packages\chardet\__init__.py'>
```

① Außer den üblichen Klassenattributen befindet sich lediglich eine detect()-Funktion im chardet-Modul.

② Dies ist der erste Hinweis darauf, dass das chardet-Modul mehr ist als eine Datei: Das „Modul“ wird als die __init__.py-Datei im chardet/-Verzeichnis angezeigt.

Sehen wir uns diese `__init__.py`-Datei einmal näher an.

```
def detect(aBuf):                                    ①
    from . import universaldetector                  ②
    u = universaldetector.UniversalDetector()
    u.reset()
    u.feed(aBuf)
    u.close()
    return u.result
```

① Die `__init__.py`-Datei definiert die `detect()`-Funktion, die der Haupteinstiegspunkt der `chardet`-Bibliothek ist.

② Doch die `detect()`-Funktion besitzt so gut wie keinen Code! Sie importiert lediglich das `universaldetector`-Modul und benutzt es. Wo ist aber `universaldetector` definiert?

Die Antwort finden wir in dieser seltsam anmutenden `import`-Anweisung:

```
from . import universaldetector
```

Auf Deutsch: „Importiere das Modul `universaldetector`, dass sich im selben Verzeichnis wie ich befindet". „Ich" ist hier die Datei `chardet/__init__.py`. Dies nennt man *relativen* Import. Auf diese Art können Dateien eines Mehr-Dateien-Moduls gegenseitig auf sich verweisen, ohne dass sie sich über Namenskonflikte mit anderen im Import-Suchpfad angelegten Modulen Gedanken machen müssen. Diese `import`-Anweisung importiert nur das `universaldetector`-Modul im `chardet/`-Verzeichnis.

Diese beiden Konzepte – `__init__.py` und relative Importe – bewirken, dass Sie Ihr Modul in so viele Teile zerlegen können, wie Sie möchten. Das `chardet`-Modul besteht aus 36 `.py`-Dateien – 36! Dennoch müssen Sie `chardet` lediglich importieren und die Funktion `chardet.detect()` aufrufen. Ohne dass Ihr Code es weiß, ist die `detect()`-Funktion in der Datei `chardet/__init__.py` definiert. Ohne Ihr Wissen verwendet die `detect()`-Funktion einen relativen Import, um eine in `chardet/universaldetector.py` definierte Klasse zu referenzieren, die wiederum selbst fünf weitere im `chardet/`-Verzeichnis befindliche Dateien durch relative Importe einbindet.

☞ Sollten Sie sich je in der Lage befinden, eine große Bibliothek in Python zu schreiben (oder, realistischer, feststellen, dass eine kleine Bibliothek zu einer großen geworden ist), nehmen Sie sich die Zeit, sie zu einem Mehr-Dateien-Modul umzugestalten. Dies ist eines der vielen Dinge, die Python wirklich gut kann. Nutzen Sie diesen Vorteil.

16.6 Anpassen, was 2to3 nicht anpassen kann

16.6.1 False ist ungültige Syntax

Jetzt testen wir richtig: Wir lassen die Test-Suite mit der Testdatei laufen. Die Test-Suite testet alle möglichen Code-Verzweigungen; es ist daher eine gute Idee, unseren portierten Code zu testen, um so sicherzustellen, dass sich keine Bugs versteckt haben.

```
C:\home\chardet> python test.py tests\*\*
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    from chardet.universaldetector import UniversalDetector
  File "C:\home\chardet\chardet\universaldetector.py", line 51
    self.done = constants.False
                          ^
SyntaxError: invalid syntax
```

Hm, ein unerwartetes Problem. In Python 3 ist `False` ein reserviertes Wort; Sie können es daher nicht als Variablenname benutzen. Schauen wir uns `constants.py` an, um zu sehen, wo es definiert ist. Dies ist die ursprüngliche Version von `constants.py` (bevor das `2to3`-Skript den Code verändert hat):

```
import __builtin__
if not hasattr(__builtin__, 'False'):
    False = 0
    True = 1
else:
    False = __builtin__.False
    True = __builtin__.True
```

Dieser Codeschnipsel erlaubt es der Bibliothek unter älteren Versionen von Python 2 zu laufen. Vor Python 2.3 gab es in Python keinen integrierten `bool`-Typ. Dieser Code erkennt das Fehlen der integrierten Konstanten `True` und `False` und definiert sie bei Bedarf.

Python 3 wird aber immer einen `bool`-Typ haben; dieser Codeschnipsel ist also unnötig. Die einfachste Lösung besteht darin, alle Vorkommen von `constants.True` und `constants.False` durch `True` bzw. `False` zu ersetzen und diesen unnötigen Code aus `constants.py` zu löschen.

Aus dieser Zeile in `universaldetector.py`:

```
self.done = constants.False
```

wird somit:

```
self.done = False
```

Ist das nicht schön? Der Code ist kürzer und lesbarer geworden.

16.6.2 Kein Modul namens constants

Führen wir test.py erneut aus und schauen, was geschieht.

```
C:\home\chardet> python test.py tests\*\*
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    from chardet.universaldetector import UniversalDetector
  File "C:\home\chardet\chardet\universaldetector.py", line 29,
in <module>
    import constants, sys
ImportError: No module named constants
```

Was ist da los? Kein Modul namens constants? Natürlich gibt es ein Modul namens constants. Es befindet sich in chardet/constants.py.

Erinnern Sie sich, dass das 2to3-Skript die import-Anweisungen angepasst hat? Diese Bibliothek besitzt eine Menge relative Importe – d. h. Module, die andere Module derselben Bibliothek importieren – doch die Logik hinter relativen Importen hat sich in Python 3 geändert. In Python 2 konnten Sie einfach constants importieren und es wurde zuerst im Verzeichnis chardet/ danach gesucht. In Python 3 sind alle import-Anweisungen per Voreinstellung *absolut*. Wollen Sie nun einen relativen Import in Python 3, müssen Sie das ausdrücklich angeben:

```
from . import constants
```

Sollte sich darum nicht eigentlich das 2to3-Skript kümmern? Nun, das hat es; doch diese import-Anweisung verbindet zwei Import-Arten in einer Zeile: den relativen Import des constants-Moduls innerhalb der Bibliothek und den absoluten Import des sys-Moduls, das in der Standardbibliothek vorinstalliert ist. In Python 2 konnten Sie dazu *eine* import-Anweisung benutzen. In Python 3 ist dies nicht möglich und das 2to3-Skript ist nicht clever genug, die import-Anweisung in *zwei* Anweisungen zu teilen.

Sie müssen sie daher manuell teilen. Dieser *zwei-in-einem-Import*:

```
import constants, sys
```

muss zu zwei getrennten import-Anweisungen werden:

```
from . import constants
import sys
```

Diese Art von Problem tritt in der chardet-Bibliothek häufiger auf. Manchmal ist es „import constants, sys“; ein anderes Mal ist es „import constants, re“. Die Anpassung ist immer dieselbe: Verteilen Sie die import-Anweisung auf zwei Zeilen, eine für den *relativen* Import, die andere für den *absoluten* Import.

Weiter geht’s!

16.6.3 Bezeichner 'file' ist nicht definiert

Wieder einmal führen wir test.py, und damit unsere Testfälle, aus …

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    for line in file(f, 'rb'):
NameError: name 'file' is not defined
```

Davon war ich überrascht. Ich nutze diesen Ausdruck schon immer. In Python 2 war file() eine andere Bezeichnung für die open()-Funktion, mit der man Textdateien zum Lesen öffnet. Die file()-Funktion existiert in Python 3 nicht mehr, die open()-Funktion dagegen schon.

Das Problem lässt sich daher am einfachsten dadurch lösen, dass man open() statt file() aufruft.

```
for line in open(f, 'rb'):
```

Das ist alles, was ich dazu sage.

16.6.4 Ein Stringmuster kann nicht auf ein byteartiges Objekt angewandt werden

Jetzt wird's interessant. Und mit „interessant" meine ich „unglaublich verwirrend".

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 98,
in feed
    if self._highBitDetector.search(aBuf):
TypeError: can't use a string pattern on a bytes-like object
```

Zum Debuggen sehen wir uns zunächst einmal an, was self._highBitDetector überhaupt ist. Es wird in der __init__-Methode der UniversalDetector-Klasse definiert:

```
class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(r'[\x80-\xFF]')
```

Damit wird ein regulärer Ausdruck zum Suchen von Nicht-ASCII-Zeichen im Bereich von 128-255(0x80-0xFF) vorkompiliert. Warten Sie, das stimmt nicht ganz; ich muss mich genauer ausdrücken. Dieses Muster sucht Nicht-ASCII-*Bytes* im Bereich von 128-255.

Genau da liegt das Problem.

In Python 2 war ein String ein Bytearray, dessen Zeichencodierung gesondert verfolgt wurde. Wollten Sie, dass Python 2 die Zeichencodierung verfolgt, mussten Sie einen Unicode-String (`u''`) verwenden. In Python 3 dagegen ist ein String immer ein Unicode-String, d. h., ein Array von Unicode-Zeichen (möglicherweise mit variierenden Bytelängen). Da dieser reguläre Ausdruck durch ein Stringmuster definiert ist, kann er auch nur zur Suche eines Strings, also eines Zeichenarrays, verwendet werden. Was wir suchen ist jedoch kein String; es ist ein Bytearray. Wie wir an den Traceback-Angaben erkennen können, trat der Fehler in `universaldetector.py` auf:

```
def feed(self, aBuf):
    .
    .
    .
    if self._mInputState == ePureAscii:
        if self._highBitDetector.search(aBuf):
```

Was ist `aBuf`? Gehen wir dorthin zurück, wo `UniversalDetector.feed()` aufgerufen wird. Eine solche Stelle ist `test.py`.

```
u = UniversalDetector()
.
.
.
for line in open(f, 'rb'):
    u.feed(line)
```

Hier finden wir die Antwort: In der Methode `UniversalDetector.feed()` ist `aBuf` eine aus einer Datei auf der Festplatte gelesene Zeile. Sehen Sie sich die beim Öffnen der Datei verwendeten Parameter genau an: `'rb'`. `'r'` steht für „lesen“; Okay, ganz toll, wir lesen die Datei. Aber `'b'` steht für „binär“. *Ohne* das `'b'`-Flag würde die `for`-Schleife die Datei Zeile für Zeile lesen und jede Zeile – basierend auf der voreingestellten Zeichencodierung des Systems – in einen String, ein Array von Unicode-Zeichen, umwandeln. *Mit* dem `'b'`-Flag dagegen liest die `for`-Schleife die Datei Zeile für Zeile und speichert jede Zeile genau so, wie sie in der Datei vorkommt – als ein Bytearray. Dieses Bytearray wird `UniversalDetector.feed()` und schließlich dem vorkompilierten regulären Ausdruck, `self._highBitDetector`, übergeben, um nach höherwertigen Zeichen zu suchen. Wir haben aber keine Zeichen; wir haben Bytes. Upps.

Der reguläre Ausdruck muss ein Bytearray durchsuchen, kein Zeichenarray.

Hat man das einmal begriffen, ist die Lösung recht einfach. Mit Strings definierte reguläre Ausdrücke können Strings durchsuchen. Mit Bytearrays definierte reguläre Ausdrücke können Bytearrays durchsuchen. Zur Definition eines Bytearray-Musters ändern wir einfach den Typ des zur Definition des regulären Ausdrucks verwendeten Arguments. Wir machen ein Bytearray daraus. (Direkt in der Zeile darunter tritt dasselbe Problem auf.)

```
class UniversalDetector:
      def __init__(self):
-         self._highBitDetector = re.compile(r'[\x80-\xFF]')
-         self._escDetector = re.compile(r'(\033|~{)')
+         self._highBitDetector = re.compile(b'[\x80-\xFF]')
+         self._escDetector = re.compile(b'(\033|~{)')
          self._mEscCharSetProber = None
          self._mCharSetProbers = []
          self.reset()
```

Durchsuchen wir den kompletten Code des Moduls auf weitere Nutzungen des `re`-Moduls, so finden wir zwei weitere in der Datei `charsetprober.py`. Wieder einmal werden reguläre Ausdrücke durch den Code als Strings definiert, aber mit `aBuf`, also einem Bytearray, ausgeführt. Die Lösung ist dieselbe: Wir definieren die Muster des regulären Ausdrucks als Bytearrays.

```
class CharSetProber:
      .
      .
      .
      def filter_high_bit_only(self, aBuf):
-         aBuf = re.sub(r'([\x00-\x7F])+', ' ', aBuf)
+         aBuf = re.sub(b'([\x00-\x7F])+', b' ', aBuf)
          return aBuf

      def filter_without_english_letters(self, aBuf):
-         aBuf = re.sub(r'([A-Za-z])+', ' ', aBuf)
+         aBuf = re.sub(b'([A-Za-z])+', b' ', aBuf)
          return aBuf
```

16.6.5 Implizite Umwandlung eines `'bytes'`-Objekts in `str` nicht möglich

Sehr verworren …

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 100, in feed
    elif (self._mInputState == ePureAscii) and
self._escDetector.search(self._mLastChar + aBuf):
TypeError: Can't convert 'bytes' object to str implicitly
```

Hier treffen Programmierstil und Python-Interpreter unglücklich zusammen. Der `TypeError` könnte überall in dieser Zeile sein, die Traceback-Angaben helfen uns da nicht weiter. Um genauere Angaben zu erhalten, sollten Sie die Zeile teilen:

```
elif (self._mInputState == ePureAscii) and \
    self._escDetector.search(self._mLastChar + aBuf):
```

Führen Sie den Test nun erneut aus:

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: Can't convert 'bytes' object to str implicitly
```

Aha! Das Problem befindet sich nicht in der ersten Bedingung (`self._mInputState == ePureAscii`), sondern in der zweiten. Doch was könnte dort einen `TypeError` verursachen? Vielleicht denken Sie jetzt, dass die `search()`-Methode einen anderen Wertetyp erwartet, doch das hätte andere Traceback-Angaben zur Folge. Python-Funktionen können jeden Wert übernehmen; übergeben Sie die korrekte Anzahl an Argumenten, wird die Funktion ausgeführt. Sie könnte abstürzen, wenn Sie einen unerwarteten Wertetyp übergeben, doch dann würden die Traceback-Angaben auf das Innere der Funktion verweisen. Die vorliegenden Traceback-Angaben zeigen jedoch, dass die `search()`-Methode überhaupt nicht aufgerufen wurde. Das Problem muss also die +-Operation sein, die versucht, den Wert zu konstruieren, der schließlich an die `search()`-Methode übergeben wird.

Wir wissen bereits, dass `aBuf` ein Bytearray ist. Was ist also `self._mLastChar`? Es ist eine Instanzvariable, die in der Methode `reset()` definiert wird. `reset()` wiederum wird von der `__init__()`-Methode aufgerufen.

```
class UniversalDetector:
    def __init__(self):
        self._highBitDetector = re.compile(b'[\x80-\xFF]')
        self._escDetector = re.compile(b'(\033|~{)')
        self._mEscCharSetProber = None
        self._mCharSetProbers = []
        self.reset()

    def reset(self):
        self.result = {'encoding': None, 'confidence': 0.0}
        self.done = False
        self._mStart = True
        self._mGotData = False
        self._mInputState = ePureAscii
        self._mLastChar = ''
```

Das ist die Antwort. Sehen Sie sie? `self._mLastChar` ist ein String, `aBuf` dagegen ist ein Bytearray. Sie können ein Bytearray nicht mit einem String verketten – nicht einmal mit einem leeren String.

`self._mLastChar` befindet sich in der `feed()`-Methode, nur ein paar Zeilen hinter der Stelle der Traceback-Angaben.

```
if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
            self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

self._mLastChar = aBuf[-1]
```

Die aufrufende Funktion ruft diese `feed()`-Methode immer wieder mit einigen Bytes auf einmal auf. Die Methode verarbeitet die übergebenen Bytes (als `aBuf` übergeben) und speichert das letzte Byte in `self._mLastChar`, falls es beim nächsten Aufruf benötigt werden sollte. (Bei einer Multi-Byte-Codierung könnte die `feed()`-Methode zunächst mit der ersten Hälfte eines Zeichens und beim zweiten Aufruf mit der zweiten Hälfte des Zeichens aufgerufen werden.) Doch da `aBuf` ein Bytearray ist, muss auch `self._mLastChar` ein Bytearray sein. Daraus folgt:

```
def reset(self):
      .
      .
      .
-     self._mLastChar = ''
+     self._mLastChar = b''
```

Durchsuchen wir den gesamten Code des Moduls nach „`mLastChar`“, finden wir ein ähnliches Problem auch in der Datei `mbcharsetprober.py`. Statt jedoch das letzte Zeichen zu verfolgen, verfolgt es die letzten zwei Zeichen. Die Klasse `MultiByteCharSetProber` verwendet eine Liste von Strings aus nur einem Zeichen, um die letzten beiden Zeichen zu verfolgen. In Python 3 muss die Klasse eine Liste von Ganzzahlen nutzen, da sie keine Zeichen, sondern Bytes verfolgt. (Bytes sind lediglich Ganzzahlen von `0-255`.)

```
class MultiByteCharSetProber(CharSetProber):
      def __init__(self):
          CharSetProber.__init__(self)
          self._mDistributionAnalyzer = None
          self._mCodingSM = None
-         self._mLastChar = ['\x00', '\x00']
+         self._mLastChar = [0, 0]

      def reset(self):
          CharSetProber.reset(self)
          if self._mCodingSM:
              self._mCodingSM.reset()
          if self._mDistributionAnalyzer:
              self._mDistributionAnalyzer.reset()
-         self._mLastChar = ['\x00', '\x00']
+         self._mLastChar = [0, 0]
```

16.6.6 Nicht unterstützte Datentypen für Operand +: `'int'` und `'bytes'`

Ich habe eine gute und eine schlechte Nachricht. Die gute Nachricht: Wir machen Fortschritte.

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 101, in feed
    self._escDetector.search(self._mLastChar + aBuf):
TypeError: unsupported operand type(s) for +: 'int' and 'bytes'
```

Die schlechte Nachricht: Es kommt einem nicht immer vor wie Fortschritte.

Doch wir machen Fortschritte! Wirklich! Die Traceback-Angaben beziehen sich zwar auf dieselbe Codezeile, der Fehler ist aber diesmal ein anderer. Fortschritt! Was ist also nun das Problem? Bei der letzten Überprüfung hat diese Codezeile nicht versucht, ein `int` mit einem Bytearray (`bytes`) zu verketten. Wir haben gerade sehr viel Zeit damit verbracht, sicherzustellen, dass `self._mLastChar` ein Bytearray ist. Wie wurde es zu einem `int`?

Die Antwort finden wir nicht in den vorherigen Zeilen, sondern in den folgenden.

```
if self._mInputState == ePureAscii:
    if self._highBitDetector.search(aBuf):
        self._mInputState = eHighbyte
    elif (self._mInputState == ePureAscii) and \
            self._escDetector.search(self._mLastChar + aBuf):
        self._mInputState = eEscAscii

self._mLastChar = aBuf[-1]
```

Dieser Fehler tritt nicht beim ersten Aufruf der `feed()`-Methode auf; er tritt erst beim zweiten Aufruf auf, nachdem `self-mLastChar` auf das letzte Byte von `aBuf` gesetzt wurde. Wo liegt das Problem? Will man ein einzelnes Element eines Bytearrays erhalten, bekommt man eine Ganzzahl, kein Bytearray. Folgen Sie mir in die *interaktive Shell*, um den Unterschied zu sehen:

```
>>> aBuf = b'\xEF\xBB\xBF'          ①
>>> len(aBuf)
3
>>> mLastChar = aBuf[-1]
>>> mLastChar                       ②
191
>>> type(mLastChar)                 ③
<class 'int'>
>>> mLastChar + aBuf                ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'bytes'
>>> mLastChar = aBuf[-1:]           ⑤
>>> mLastChar
b'\xbf'
>>> mLastChar + aBuf                ⑥
b'\xbf\xef\xbb\xbf'
```

① Definieren Sie ein Bytearray mit der Länge 3.

② Das letzte Element des Bytearrays ist `191`.

③ Das ist eine Ganzzahl.

④ Es ist nicht möglich, eine Ganzzahl mit einem Bytearray zu verketten. Damit haben wir den gerade in `universaldetector.py` gefundenen Fehler reproduziert.

⑤ Wir können den Fehler beseitigen. Statt das letzte Element des Bytearrays herzunehmen, verwenden wir Listen-Slicing zur Erstellung eines neuen Bytearrays, das lediglich das letze Element enthält. Wir beginnen also mit dem letzten Element

und slicen solange, bis wir das Ende des Bytearrays erreicht haben. `mLastChar` ist nun ein Bytearray der Länge 1.

⑥ Verketten wir ein Bytearray der Länge 1 mit einem Bytearray der Länge 3, so erhalten wir ein Bytearray der Länge 4.

Um also sicherzustellen, dass die `feed()`-Methode auch bei mehrfachen Aufrufen funktioniert, müssen Sie `self._mLastChar` als Bytearray der Länge 0 initialisieren und *dafür sorgen, dass es ein Bytearray bleibt*.

```
                self._escDetector.search(self._mLastChar + aBuf):
            self._mInputState = eEscAscii

- self._mLastChar = aBuf[-1]
+ self._mLastChar = aBuf[-1:]
```

16.6.7 `ord()` erwartet String der Länge 1, `int` gefunden

Keine Lust mehr? Sie haben es fast geschafft.

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                      ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\utf8prober.py", line 53, in feed
    codingState = self._mCodingSM.next_state(c)
  File "C:\home\chardet\chardet\codingstatemachine.py", line 43, in next_state
    byteCls = self._mModel['classTable'][ord(c)]
TypeError: ord() expected string of length 1, but int found
```

`c` ist also ein `int`, doch die `ord()`-Funktion hat einen String aus einem Zeichen erwartet. Na gut. Wo ist `c` definiert?

```
# codingstatemachine.py
def next_state(self, c):
    # for each byte we get its class
    # if it is first byte, we also get byte length
    byteCls = self._mModel['classTable'][ord(c)]
```

Das hilft uns nicht weiter; es wird einfach der Funktion übergeben. Sehen wir uns woanders um.

```
# utf8prober.py
def feed(self, aBuf):
    for c in aBuf:
        codingState = self._mCodingSM.next_state(c)
```

Fällt Ihnen etwas auf? In Python 2 war `aBuf` ein String, also war `c` ein aus einem Zeichen bestehender String. (Dies erhalten Sie beim Durchlaufen eines Strings

– alle Zeichen, eins nach dem andern.) Doch nun ist `aBuf` ein Bytearray und `c` somit ein `int`. Das bedeutet, dass die `ord()`-Funktion überhaupt nicht aufgerufen werden muss, da `c` bereits ein `int` *ist!*

Somit ergibt sich:

```
 def next_state(self, c):
      # for each byte we get its class
      # if it is first byte, we also get byte length
-     byteCls = self._mModel['classTable'][ord(c)]
+     byteCls = self._mModel['classTable'][c]
```

Durchsuchen wir den kompletten Code des Moduls nach „`ord(c)`“, so finden wir ähnliche Probleme in `sbcharsetprober.py` …

```
# sbcharsetprober.py
def feed(self, aBuf):
    if not self._mModel['keepEnglishLetter']:
        aBuf = self.filter_without_english_letters(aBuf)
    aLen = len(aBuf)
    if not aLen:
        return self.get_state()
    for c in aBuf:
        order = self._mModel['charToOrderMap'][ord(c)]
```

… und in `latin1prober.py` …

```
# latin1prober.py
def feed(self, aBuf):
    aBuf = self.filter_with_english_letters(aBuf)
    for c in aBuf:
        charClass = Latin1_CharToClass[ord(c)]
```

`c` wird zum Durchlaufen von `aBuf` verwendet, ist also eine Ganzzahl, kein String. Die bereits bekannte Lösung: Schreiben Sie statt `ord(c)` einfach `c`.

```
 # sbcharsetprober.py
  def feed(self, aBuf):
      if not self._mModel['keepEnglishLetter']:
          aBuf = self.filter_without_english_letters(aBuf)
      aLen = len(aBuf)
      if not aLen:
          return self.get_state()
      for c in aBuf:
-         order = self._mModel['charToOrderMap'][ord(c)]
+         order = self._mModel['charToOrderMap'][c]

  # latin1prober.py
  def feed(self, aBuf):
      aBuf = self.filter_with_english_letters(aBuf)
      for c in aBuf:
-         charClass = Latin1_CharToClass[ord(c)]
+         charClass = Latin1_CharToClass[c]
```

16.6.8 Unsortierbare Datentypen: `int() >= str()`

Weiter geht's.

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 10, in <module>
    u.feed(line)
  File "C:\home\chardet\chardet\universaldetector.py", line 116, in feed
    if prober.feed(aBuf) == constants.eFoundIt:
  File "C:\home\chardet\chardet\charsetgroupprober.py", line 60, in feed
    st = prober.feed(aBuf)
  File "C:\home\chardet\chardet\sjisprober.py", line 68, in feed
    self._mContextAnalyzer.feed(self._mLastChar[2 - charLen :], charLen)
  File "C:\home\chardet\chardet\jpcntx.py", line 145, in feed
    order, charLen = self.get_order(aBuf[i:i+2])
  File "C:\home\chardet\chardet\jpcntx.py", line 176, in get_order
    if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
TypeError: unorderable types: int() >= str()
```

Worum geht's überhaupt? „Unsortierbare Datentypen"? Wieder einmal zeigt der Unterschied zwischen Bytearrays und Strings seine hässliche Fratze. Sehen Sie sich diesen Code an:

```
class SJISContextAnalysis(JapaneseContextAnalysis):
    def get_order(self, aStr):
        if not aStr: return -1, 1
        # find out current char's byte length
        if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
           ((aStr[0] >= '\xE0') and (aStr[0] <= '\xFC')):
            charLen = 2
        else:
            charLen = 1
```

Woher kommt denn `aStr`? Schauen wir doch mal:

```
def feed(self, aBuf, aLen):
    .
    .
    .
    i = self._mNeedToSkipCharNum
    while i < aLen:
        order, charLen = self.get_order(aBuf[i:i+2])
```

Oh, sehen Sie doch, es ist unser alter Freund: `aBuf`. Wie Sie sicher anhand aller anderen Probleme dieses Kapitels erraten haben, ist `aBuf` ein Bytearray. Die `feed()`-Methode übergibt es hier *nicht* im Ganzen; es wird gesliced. Wie Sie aber bereits gesehen haben, gibt das Slicen eines Bytearrays ein Bytearray zurück. Der der `get_order()`-Methode übergebene `aStr`-Parameter ist daher immer noch ein Bytearray.

Was will dieser Code denn mit `aStr` tun? Er nimmt das erste Element des Bytearrays und vergleicht es mit einem String der Länge 1. In Python 2 hat dies funktioniert, da `aStr` und `aBuf` Strings waren und auch `aStr[0]` ein String gewesen wäre. Strings kann man auf Ungleichheit überprüfen. In Python 3 dagegen sind `aStr` und `aBuf` Bytearrays und `aStr[0]` eine Ganzzahl. Ganzzahlen und Strings

kann man nicht auf Ungleichheit überprüfen, wenn man es nicht ausdrücklich angibt.

Im vorliegenden Fall müssen wir den Code nicht noch komplizierter machen, indem wir es ausdrücklich angeben. `aStr[0]` ergibt eine Ganzzahl; Sie vergleichen immer mit Konstanten. Ändern wir diese von Strings aus einem Zeichen zu Ganzzahlen. Und wo wir schon dabei sind, können wir auch gleich `aStr` in `aBuf` umbenennen, da es eigentlich kein String ist.

```
class SJISContextAnalysis(JapaneseContextAnalysis):
-     def get_order(self, aStr):
-      if not aStr: return -1, 1
+     def get_order(self, aBuf):
+      if not aBuf: return -1, 1
          # find out current char's byte length
-         if ((aStr[0] >= '\x81') and (aStr[0] <= '\x9F')) or \
-            ((aBuf[0] >= '\xE0') and (aBuf[0] <= '\xFC')):
+         if ((aBuf[0] >= 0x81) and (aBuf[0] <= 0x9F)) or \
+            ((aBuf[0] >= 0xE0) and (aBuf[0] <= 0xFC)):
              charLen = 2
          else:
              charLen = 1

          # return its order if it is hiragana
-      if len(aStr) > 1:
-             if (aStr[0] == '\202') and \
-                (aStr[1] >= '\x9F') and \
-                (aStr[1] <= '\xF1'):
-                return ord(aStr[1]) - 0x9F, charLen
+      if len(aBuf) > 1:
+             if (aBuf[0] == 202) and \
+                (aBuf[1] >= 0x9F) and \
+                (aBuf[1] <= 0xF1):
+                return aBuf[1] - 0x9F, charLen

          return -1, charLen

  class EUCJPContextAnalysis(JapaneseContextAnalysis):
-     def get_order(self, aStr):
-      if not aStr: return -1, 1
+     def get_order(self, aBuf):
+      if not aBuf: return -1, 1
          # find out current char's byte length
-         if (aStr[0] == '\x8E') or \
-           ((aStr[0] >= '\xA1') and (aStr[0] <= '\xFE')):
+         if (aBuf[0] == 0x8E) or \
+           ((aBuf[0] >= 0xA1) and (aBuf[0] <= 0xFE)):
              charLen = 2
-         elif aStr[0] == '\x8F':
+         elif aBuf[0] == 0x8F:
              charLen = 3
          else:
              charLen = 1

        # return its order if it is hiragana
-    if len(aStr) > 1:
-           if (aStr[0] == '\xA4') and \
-              (aStr[1] >= '\xA1') and \
-              (aStr[1] <= '\xF3'):
-                 return ord(aStr[1]) - 0xA1, charLen
+    if len(aBuf) > 1:
+           if (aBuf[0] == 0xA4) and \
+              (aBuf[1] >= 0xA1) and \
+              (aBuf[1] <= 0xF3):
+               return aBuf[1] - 0xA1, charLen

        return -1, charLen
```

Durchsuchen wir den kompletten Code nach „ord()“, so finden wir dasselbe Problem in chardistribution.py (insbesondere in den Klassen EUCTWDistributionAnalysis, EUCKRDistributionAnalysis, GB2312DistributionAnalysis, Big5DistributionAnalysis, SJISDistributionAnalysis und EUCJPDistributionAnalysis). In jedem dieser Fälle erfolgt die Lösung des Problems auf dieselbe Weise, wie wir sie bei den Klassen EUCJPContextAnalysis und SJISContextAnalysis in jpcntx.py durchgeführt haben.

16.6.9 Globaler Bezeichner 'reduce' ist nicht definiert

Begeben wir uns erneut in die Höhle des Löwen …

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml                      ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    u.close()
  File "C:\home\chardet\chardet\universaldetector.py", line 141, in close
    proberConfidence = prober.get_confidence()
  File "C:\home\chardet\chardet\latin1prober.py", line 126, in get_confidence
    total = reduce(operator.add, self._mFreqCounter)
NameError: global name 'reduce' is not defined
```

Gemäß des offiziellen *What's New In Python 3.0*-Dokuments (http://docs.python.org/3.0/whatsnew/3.0.html#builtins) wurde die reduce()-Funktion vom globalen Namensbereich in das Modul functools verschoben. Ich zitiere aus diesem Dokument: „Verwenden Sie functools.reduce(), wenn es unbedingt nötig ist; in 99% der Fälle ist eine for-Schleife jedoch lesbarer.“

```
def get_confidence(self):
    if self.get_state() == constants.eNotMe:
        return 0.01

    total = reduce(operator.add, self._mFreqCounter)
```

Die reduce()-Funktion übernimmt zwei Argumente – eine Funktion und eine Liste (*genauer*: ein beliebiges iterierbares Objekt) – und wendet die Funktion kumulativ auf jedes Element der Liste an. Im vorliegenden Fall werden also alle Listenelemente addiert und das Ergebnis zurückgegeben. Sehr umständlich, nicht wahr?

Dieses Ungetüm wurde so häufig eingesetzt, dass eine globale sum()-Funktion zu Python hinzugefügt wurde.

```
  def get_confidence(self):
      if self.get_state() == constants.eNotMe:
          return 0.01

-     total = reduce(operator.add, self._mFreqCounter)
+     total = sum(self._mFreqCounter)
```

Da wir nicht länger das operator-Modul verwenden, können wir die import-Anweisung am Anfang der Datei entfernen.

```
  from .charsetprober import CharSetProber
  from . import constants
- import operator
```

Wir sollten nun einen Test ausführen.

```
C:\home\chardet> python test.py tests\*\*
tests\ascii\howto.diveintomark.org.xml
ascii with confidence 1.0
tests\Big5\0804.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\blog.worren.net.xml
Big5 with confidence 0.99
tests\Big5\carbonxiv.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\catshadow.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\coolloud.org.tw.xml
Big5 with confidence 0.99
tests\Big5\digitalwall.com.xml
Big5 with confidence 0.99
tests\Big5\ebao.us.xml
Big5 with confidence 0.99
tests\Big5\fudesign.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\kafkatseng.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\ke207.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\leavesth.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\letterlego.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\linyijen.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\marilynwu.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\myblog.pchome.com.tw.xml
Big5 with confidence 0.99
tests\Big5\oui-design.com.xml
Big5 with confidence 0.99
tests\Big5\sanwenji.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\sinica.edu.tw.xml
Big5 with confidence 0.99
tests\Big5\sylvia1976.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\tlkkuo.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\tw.blog.xubg.com.xml
Big5 with confidence 0.99
tests\Big5\unoriginalblog.com.xml
Big5 with confidence 0.99
tests\Big5\upsaid.com.xml
Big5 with confidence 0.99
tests\Big5\willythecop.blogspot.com.xml
Big5 with confidence 0.99
tests\Big5\ytc.blogspot.com.xml
Big5 with confidence 0.99
tests\EUC-JP\aivy.co.jp.xml
EUC-JP with confidence 0.99
tests\EUC-JP\akaname.main.jp.xml
EUC-JP with confidence 0.99
```

```
tests\EUC-JP\arclamp.jp.xml
EUC-JP with confidence 0.99
.
.
.
316 tests
```

Wow, es funktioniert!

16.7 Zusammenfassung

Was haben wir in diesem Kapitel gelernt?

1. Es ist nicht einfach, umfangreichen Code von Python 2 zu Python 3 zu portieren. Man kommt nicht drum herum. Es ist wirklich schwierig.
2. Das `2to3`-Skript ist sehr hilfreich, solange es um einfache Dinge geht – Umbenennung von Funktionen und Modulen, Syntax-Änderungen. Es ist ein eindrucksvolles Skript, doch schlussendlich nicht mehr als ein intelligenter *Suchen-und-Ersetzen-Bot*.
3. Das Hauptproblem bei der vorliegenden Bibliothek war der Unterschied zwischen Strings und Bytes. Natürlich geht es bei der `chardet`-Bibliothek auch gerade darum – Bytes in Strings umwandeln. Doch Bytes kommen sehr viel häufiger vor, als Sie vielleicht glauben. Eine Datei im Binärmodus lesen? Sie erhalten Bytes. Eine Webseite abrufen? Eine Web-API aufrufen? Auch sie geben Bytes zurück.
4. Sie müssen Ihr Programm voll und ganz verstehen. Vorzugsweise, weil *Sie* es geschrieben haben. Zumindest sollten Sie jedoch seine Eigenarten und seine Ecken und Kanten kennen. Bugs lauern überall.
5. Testfälle sind unverzichtbar. Ohne sie sollten Sie niemals etwas portieren. Ich war allein deshalb überzeugt davon, dass `chardet` in Python 3 funktionieren würde, weil ich mit einer Test-Suite begonnen habe, die den Code geprüft hat. Haben Sie keine Tests, schreiben Sie welche, bevor Sie mit dem Portieren beginnen. Haben Sie ein paar Tests, schreiben Sie mehr. Haben Sie viele Tests, dann kann der Spaß beginnen.

Kapitel 17
Python-Bibliotheken packen

17.1 Los geht's

Sie möchten also ein Python-Skript, -Framework, eine Python-Bibliothek oder -Anwendung herausgeben? Ausgezeichnet. Die Welt braucht mehr Python-Code.

Python 3 enthält ein Pack-Framework namens *Distutils*. Distutils ist vieles: ein Erstellungs-Tool (für Sie), ein Installations-Tool (für Ihre Benutzer), ein Paket-Metadaten-Format (für Suchmaschinen) und mehr. Es fügt sich in den Python Package Index („PyPI" – `http://pypi.python.org/pypi`), ein zentrales Repository für Open-Source-Python-Bibliotheken, ein.

Im Mittelpunkt all dieser Facetten von Distutils steht das *Setup-Skript*, das traditionell `setup.py` genannt wird. In diesem Buch haben Sie bereits einige Distutils-Setup-Skripte gesehen. Sie haben Distutils im Kapitel *HTTP-Webdienste* verwendet, um `httplib2` zu installieren; ein weiteres Mal haben Sie es in unserer *Fallstudie zur Portierung von* `chardet` zur Installation des Moduls verwendet.

In diesem Kapitel werden Sie lernen, wie die Setup-Skripte von `chardet` und `httplib2` funktionieren und erfahren, wie Sie Ihre eigene Python-Software veröffentlichen können.

```
# setup.py von chardet
from distutils.core import setup
setup(
    name = "chardet",
    packages = ["chardet"],
    version = "1.0.2",
    description = "Universal encoding detector",
    author = "Mark Pilgrim",
    author_email = "mark@diveintomark.org",
    url = "http://chardet.feedparser.org/",
    download_url = "http://chardet.feedparser.org/download/python3-chardet-
1.0.1.tgz",
    keywords = ["encoding", "i18n", "xml"],
    classifiers = [
```

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_17, © Springer-Verlag Berlin Heidelberg 2010

```
        "Programming Language :: Python",
        "Programming Language :: Python :: 3",
        "Development Status :: 4 - Beta",
        "Environment :: Other Environment",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: GNU Library or Lesser General Public License
(LGPL)",
        "Operating System :: OS Independent",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Text Processing :: Linguistic",
        ],
    long_description = """\
Universal character encoding detector
-------------------------------------


Detects
 - ASCII, UTF-8, UTF-16 (2 variants), UTF-32 (4 variants)
 - Big5, GB2312, EUC-TW, HZ-GB-2312, ISO-2022-CN (Traditional and Simplified Chinese)
 - EUC-JP, SHIFT_JIS, ISO-2022-JP (Japanese)
 - EUC-KR, ISO-2022-KR (Korean)
 - KOI8-R, MacCyrillic, IBM855, IBM866, ISO-8859-5, windows-1251 (Cyrillic)
 - ISO-8859-2, windows-1250 (Hungarian)
 - ISO-8859-5, windows-1251 (Bulgarian)
 - windows-1252 (English)
 - ISO-8859-7, windows-1253 (Greek)
 - ISO-8859-8, windows-1255 (Visual and Logical Hebrew)
 - TIS-620 (Thai)

This version requires Python 3 or later; a Python 2 version is available separately.
"""
)
```

☞ `chardet` und `httplib2` stehen unter einer Open-Source-Lizenz. Sie sind jedoch nicht gezwungen, Ihre eigenen Python-Bibliotheken unter einer bestimmten Lizenz herauszugeben. Die Vorgehensweise dieses Kapitels setzt keine bestimmte Lizenz voraus.

17.2 Was kann Distutils nicht für Sie tun?

Das erste Python-Paket herauszugeben kann sehr entmutigend sein. (Beim zweiten wird es etwas einfacher.) Distutils versucht, soviel wie möglich zu automatisieren, doch es gibt einige Dinge, die Sie einfach selbst machen müssen.

- **Wählen Sie eine Lizenz.** Das ist ein kompliziertes Thema, voller Tücken. Möchten Sie Ihre Software als Open Source veröffentlichen, sollten Sie folgende fünf Tipps beherzigen:
  - Schreiben Sie keine eigene Lizenz.
  - Schreiben Sie keine eigene Lizenz.
  - Schreiben Sie keine eigene Lizenz.
  - Es muss nicht unbedingt die GPL sein, doch in jedem Fall kompatibel dazu.
  - Schreiben Sie keine eigene Lizenz.
- **Klassifizieren Sie Ihre Software** mithilfe des Klassifizierungssystems von PyPI. Das werde ich später noch näher erläutern.
- **Schreiben Sie eine „Lies Mich"-Datei.** Verzichten Sie nicht darauf. Die Datei sollte Ihren Benutzern zumindest einen Überblick über die Funktionen und den Installationsvorgang Ihres Programms verschaffen.

17.3 Verzeichnisstruktur

Vor dem Packen Ihrer Python-Software müssen Sie zuerst Ihre Dateien und Verzeichnisse ordnen. Das `httplib2`-Verzeichnis sieht wie folgt aus:

```
httplib2/                ①
|
+--README.txt            ②
|
+--setup.py              ③
|
+--httplib2/             ④
   |
   +--__init__.py
   |
   +--iri2uri.py
```

① Erstellen Sie ein Wurzelverzeichnis, das alles andere enthält. Benennen Sie es genauso wie Ihr Python-Modul.

② Ihre „Lies Mich"-Datei sollte die Dateierweiterung `.txt` haben und Zeilenumbrüche nach Windows-Art enthalten. Nur weil Sie einen ausgefallenen Texteditor benutzen, der auf der Kommandozeile läuft und seine eigene Makrosprache besitzt, heißt das nicht, dass Sie Ihren Benutzern das Leben schwer machen müssen. (Ihre Benutzer verwenden den Windows-Editor. Traurig, aber wahr.) Auch wenn

Sie unter Linux oder Mac OS X arbeiten, besitzt Ihr Texteditor sicher eine Option, mit der Sie Dateien mit Windows-Zeilenumbrüchen speichern können.

③ Ihr Distutils-Setup-Skript sollte `setup.py` heißen, es sei denn, Sie haben einen guten Grund für eine andere Bezeichnung. Sie haben keinen guten Grund.

④ Besteht Ihr Python-Programm aus einer einzigen `.py`-Datei, sollten Sie diese zusammen mit der „Lies Mich"-Datei und dem Setup-Skript im Wurzelverzeichnis ablegen. `httplib2` ist jedoch keine einzelne `.py`-Datei; es ist ein Modul aus mehreren Dateien. Das ist in Ordnung! Legen Sie das `httplib2`-Verzeichnis einfach im Wurzelverzeichnis ab, so dass sich die `__init__.py`-Datei im Verzeichnis `httplib2` befindet, welches sich wiederum im `httplib2`-Wurzelverzeichnis befindet. Das ist kein Problem; es erleichtert Ihnen sogar den Packvorgang.

Das `chardet`-Verzeichnis sieht ein klein wenig anders aus. Es ist, wie `httplib2`, ein Modul aus mehreren Dateien. Es gibt also ein `chardet`-Verzeichnis innerhalb des `chardet`-Wurzelverzeichnisses. Zusätzlich zur Datei `README.txt` enthält `chardet` innerhalb des Verzeichnisses `docs/` eine mit HTML formatierte Dokumentation. Das `docs`-Verzeichnis beinhaltet verschiedene `.html`- und `.css`-Dateien sowie ein Unterverzeichnis namens `images/`, welches `.png`- und `.gif`-Dateien beherbergt. (Dies wird später noch von Bedeutung sein.) Außerdem ist gemäß der Konvention für (L)GPL-lizenzierte Software noch eine Datei namens `COPYING.txt` vorhanden, die den kompletten Text der LGPL enthält.

```
chardet/
|
+--COPYING.txt
|
+--setup.py
|
+--README.txt
|
+--docs/
|  |
|  +--index.html
|  |
|  +--usage.html
|  |
|  +--images/ ...
|
+--chardet/
   |
   +--__init__.py
   |
   +--big5freq.py
   |
   +--...
```

17.4 Das Setup-Skript schreiben

Das Distutils-Setup-Skript ist ein Python-Skript. Theoretisch kann es alles tun, was mit Python möglich ist. Praktisch sollte es allerdings so wenig wie möglich machen. Setup-Skripte sollten langweilig sein. Je exotischer Ihr Installationsvorgang ist, desto exotischer werden auch die Fehlerberichte ausfallen.

Die erste Zeile eines Distutils-Setup-Skripts lautet immer wie folgt:

```
from distutils.core import setup
```

Dadurch wird die Funktion `setup()`, der Haupteinstiegspunkt in Distutils, importiert. 95% aller Distutils-Setup-Skripte bestehen nur aus einem einzigen Aufruf von `setup()`. (Diese Statistik habe ich gerade frei erfunden, doch sollte Ihr Setup-Skript mehr tun als die `setup()`-Funktion aufzurufen, sollten Sie einen guten Grund dazu haben. Haben Sie einen guten Grund? Ich gehe nicht davon aus.)

Die `setup()`-Funktion kann dutzende Parameter übernehmen. Sie müssen für jeden Parameter benannte Argumente verwenden. Dies ist keine einfache *Konvention*; es ist eine *Voraussetzung*. Wenn Sie die `setup()`-Funktion mit unbenannten Argumenten aufrufen, wird Ihr Setup-Skript abstürzen.

Die folgenden benannten Argumente werden vorausgesetzt:

- `name` – Der Name des Pakets
- `version` – Die Versionsnummer des Pakets
- `author` – Ihr vollständiger Name
- `author_email` – Ihre E-Mail-Adresse
- `url` – Die Homepage Ihres Projekts. Sollten Sie keine eigene Webseite für Ihr Projekt besitzen, kann dies auch die PyPI-Seite Ihres Pakets sein.

Auch wenn die folgenden Argumente nicht vorausgesetzt werden, empfehle ich doch deren Verwendung:

- `description` – Eine einzeilige Zusammenfassung des Projekts.
- `long_description` – Ein mehrzeiliger String im Format reStructuredText (siehe: `http://docutils.sourceforge.net/rst.html`). PyPI wandelt diesen String in HTML um und zeigt ihn auf der Seite Ihres Pakets an.
- `classifiers` – Eine Liste speziell formatierter Strings, die im nächsten Abschnitt näher beschrieben werden.

☞ Setup-Skript-Metadaten werden in *PEP 314* definiert.

Sehen wir uns nun das Setup-Skript von `chardet` an. Es besitzt alle vorausgesetzten und empfohlenen Parameter sowie einen bisher nicht erwähnten: `packages`.

```
from distutils.core import setup
setup(
    name = 'chardet',
    packages = ['chardet'],
    version = '1.0.2',
    description = 'Universal encoding detector',
    author='Mark Pilgrim',
    ...
)
```

Der Parameter `packages` hebt eine unglückliche Überschneidung in der Wortwahl des Veröffentlichungsvorgangs hervor. Wir reden schon die ganze Zeit über das „Paket“ („package“), das wir gerade erstellen (und möglicherweise im Python „Package“ Index eintragen). Das ist jedoch nicht das, worauf der `packages`-Parameter verweist. Er bezieht sich auf die Tatsache, dass das `chardet`-Modul ein Modul aus mehreren Dateien, auch bekannt als „Paket“, ist. Der `packages`-Parameter sorgt dafür, dass Distutils das `chardet`-Verzeichnis, seine `__init__.py`-Datei und alle anderen `.py`-Dateien des `chardet`-Moduls erfasst. Das ist ziemlich wichtig; das ganze Gerede über Dokumentation und Metadaten ist nutzlos, wenn Sie vergessen, den eigentlichen Code mit einzubeziehen.

17.5 Ihr Paket klassifizieren

Der Python Package Index („PyPI“) enthält tausende Python-Bibliotheken. Durch die Verwendung korrekter Metadaten zur Klassifizierung wird Ihr Paket einfacher gefunden. PyPI erlaubt die Paketsuche über *Klassifizierer*. Um Ihre Suche einzugrenzen, können Sie sogar mehrere Klassifizierer auswählen. Klassifizierer sind also keine unsichtbaren Metadaten, die Sie einfach ignorieren können.

Übergeben Sie der `setup()`-Funktion von Distutils den Parameter `classifiers`, um Ihre Software zu klassifizieren. Der `classifiers`-Parameter ist eine Liste von Strings. Diese Strings können Sie nicht frei wählen. Alle Strings zur Klassifizierung sollten aus der Liste von PyPI gewählt werden. (Diese Liste finden Sie hier: `http://pypi.python.org/pypi?:action=list_classifiers`.)

Klassifizierer sind optional. Sie können Distutils-Setup-Skripte auch komplett ohne Klassifizierer schreiben. Tun Sie das nicht. Sie sollten immer mindestens die folgenden Klassifizierer einfügen:

- **Programmiersprache.** Sie sollten sowohl `"Programming Language :: Python"` als auch `"Programming Language :: Python :: 3"` einfügen. Fügen Sie dies nicht ein, wird Ihr Paket nicht in der Liste der Python 3-kompatiblen Bibliotheken angezeigt, auf die von jeder Seite von `pypi.python.org` verlinkt wird.
- **Lizenz.** Nach dieser schaue ich als Allererstes, wenn ich Bibliotheken von Drittanbietern auswähle. Lassen Sie mich nicht nach dieser wichtigen Information suchen. Fügen Sie nur dann mehrere Lizenz-Klassifizierer ein, wenn Ihre Software

ausdrücklich unter mehreren Lizenzen verfügbar ist. (Geben Sie Software nicht unter mehreren Lizenzen heraus, es sei denn, Sie sind dazu gezwungen. Zwingen Sie andere nicht dazu. Lizenzierung an sich ist schon kompliziert genug; machen Sie es nicht noch schlimmer.)

- **Betriebssystem.** Ist Ihre Software nur unter Windows (oder Mac OS X oder Linux) lauffähig, möchte ich das möglichst schnell erfahren. Läuft Ihre Software ohne plattformabhängigen Code, verwenden Sie den Klassifizierer `"Operating System :: OS :: Independent"`. Mehrere Betriebssystem-Klassifzierer sind nur dann nötig, wenn Ihre Software für jede der Plattformen besondere Voraussetzungen erfordert. (Das ist unüblich.)

Ich empfehle außerdem die Verwendung der folgenden Klassifizierer:

- **Entwicklungsstatus.** Welchen Status hat Ihre Software? Beta? Alpha? Pre-Alpha? Entscheiden Sie sich. Seien Sie ehrlich.
- **Zielgruppe.** Wer wird Ihre Software herunterladen? Die Wahl fällt hier häufig auf `Developers` (Entwickler), `End Users/Desktop` (Endbenutzer), `Science/Research` (Wissenschaft/Forschung) und `System Administrators` (Systemadministratoren).
- **Framework.** Ist Ihre Software ein Plug-In für ein größeres Python-Framework wie Django oder Zope, fügen Sie den passenden Framework-Klassifizierer ein. Wenn nicht, lassen Sie ihn aus.
- **Themengebiet.** Die Auswahl der Themengebiete ist sehr groß; wählen Sie alle zu Ihrer Software passenden aus.

17.5.1 Beispiele guter Paket-Klassifizierer

Beispielhaft seien hier die Klassifizierer von Django dargestellt, einem sofort einsetzbaren, *plattformübergreifenden*, *BSD-lizensierten* Web-Framework, das auf einem Server läuft. (Django ist bisher nicht zu Python 3 kompatibel; der Klassifizierer `Programming Language :: Python :: 3` wird daher nicht angegeben.)

```
Programming Language :: Python
License :: OSI Approved :: BSD License
Operating System :: OS Independent
Development Status :: 5 - Production/Stable
Environment :: Web Environment
Framework :: Django
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Internet :: WWW/HTTP :: Dynamic Content
Topic :: Internet :: WWW/HTTP :: WSGI
Topic :: Software Development :: Libraries :: Python Modules
```

Hier nun die Klassifizierer von `chardet`, der Bibliothek zur Zeichencodierungserkennung, die ich Ihnen im vorherigen Kapitel vorgestellt habe. `chardet` ist im

Beta-Status, plattformübergreifend, Python 3-kompatibel, *LGPL-lizenziert* und für *Entwickler* gedacht, die sie in Ihre eigenen Produkte integrieren.

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: GNU Library or Lesser General Public
License (LGPL)
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Other Environment
Intended Audience :: Developers
Topic :: Text Processing :: Linguistic
Topic :: Software Development :: Libraries :: Python Modules
```

Sehen wir uns außerdem noch die Klassifizierer von `httplib2` an, dem Ihnen bereits bekannten HTTP-Modul. `httplib2` ist im *Beta-Status*, *plattformübergreifend*, *MIT-lizenziert* und auf *Python-Entwickler* zugeschnitten.

```
Programming Language :: Python
Programming Language :: Python :: 3
License :: OSI Approved :: MIT License
Operating System :: OS Independent
Development Status :: 4 - Beta
Environment :: Web Environment
Intended Audience :: Developers
Topic :: Internet :: WWW/HTTP
Topic :: Software Development :: Libraries :: Python Modules
```

17.6 Zusätzliche Dateien mit einem Manifest angeben

Per Voreinstellung fügt Distutils die folgenden Dateien in Ihr Paket ein:

- `README.txt`
- `setup.py`
- Die von den im `packages`-Parameter angegebenen Modulen aus mehreren Dateien benötigten `.py`-Dateien
- Die im `py_modules`-Parameter angegebenen `.py`-Dateien

Damit sind alle Dateien des `httplib2`-Projekts abgedeckt. Beim `chardet`-Projekt möchten wir dagegen auch noch die Lizenzdatei `COPYING.txt` sowie das komplette `docs/`-Verzeichnis, das Bilder und HTML-Dateien enthält, einfügen. Um Distutils dies mitzuteilen, benötigen wir eine *Manifest-Datei*.

Eine Manifest-Datei ist eine Textdatei mit der Bezeichnung `MANIFEST.in`. Legen Sie sie wie `README.txt` und `setup.py` im Wurzelverzeichnis des Projekts ab. Manifest-Dateien sind keine Python-Skripte, sondern Textdateien, die eine Reihe von „Befehlen“ in einem Distutils eigenen Format enthalten. Manifest-Befehle erlauben Ihnen das Ein- oder Ausschließen bestimmter Dateien und Verzeichnisse.

Hier sehen Sie die komplette Manifest-Datei des `chardet`-Projekts:

```
include COPYING.txt                                    ①
recursive-include docs *.html *.css *.png *.gif        ②
```

① Die erste Zeile ist selbsterklärend: Schließe die im Wurzelverzeichnis des Projekts liegende Datei `COPYING.txt` mit ein.

② Die zweite Zeile ist ein wenig komplizierter. Der Befehl `recursive-include` nimmt einen Verzeichnisnamen und einen oder mehrere Dateinamen entgegen. Die Dateinamen müssen nicht genau angegeben werden; sie können Platzhalter enthalten. Diese Zeile bedeutet: „Im Wurzelverzeichnis befindet sich ein Verzeichnis namens `docs/`. Gehe dort (rekursiv) hinein und schaue nach `.html`-, `.css`-, `.png`- und `.gif`-Dateien. Diese Dateien sollen alle in mein Paket."

Alle Manifest-Befehle erhalten die Verzeichnisstruktur, die Sie in Ihrem Projektverzeichnis angelegt haben. Der Befehl `recursive-include` legt die `.html`- und `.png`-Dateien *nicht* im Wurzelverzeichnis des Pakets ab. Er *erhält* das bestehende `docs`-Verzeichnis, legt darin jedoch nur die Dateien ab, die durch die Platzhalter erfasst werden. (Ich habe es bisher nicht erwähnt, doch die Dokumentation zu `chardet` ist eigentlich in XML geschrieben und wird durch ein Skript in HTML umgewandelt. Die XML-Dateien sollen im fertigen Paket nicht enthalten sein, sondern nur die HTML-Dateien und die Bilder.)

Noch einmal: Sie müssen nur dann eine Manifest-Datei erstellen, wenn Sie Dateien zu Ihrem Paket hinzufügen möchten, die Distutils per Voreinstellung nicht hinzufügt. Benötigen Sie eine Manifest-Datei, so sollte sie lediglich die Dateien und Verzeichnisse beinhalten, die Distutils andernfalls nicht finden würde.

17.7 Ihr Setup-Skript auf Fehler untersuchen

Sie müssen an vieles denken. Distutils besitzt einen integrierten Befehl zur Gültigkeitsprüfung, der überprüft, ob alle benötigten Metadaten in Ihrem Setup-Skript vorliegen. Vergessen Sie beispielsweise den `version`-Parameter, wird Distutils Sie darauf hinweisen.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check
running check
warning: check: missing required meta-data: version
```

Haben Sie dann den `version`-Parameter (und alle anderen benötigten Metadaten) eingefügt, sieht der `check`-Befehl wie folgt aus:

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py check
running check
```

17.8 Eine Quellcode-Distribution erstellen

Distutils unterstützt die Erstellung verschiedener Distributionen (also zur Veröffentlichung vorgesehene Pakete). Sie sollten zumindest eine „Quellcode-Distribution“ erstellen, die den Quellcode, das Setup-Skript, die „Lies Mich“-Datei sowie alle anderen zusätzlich gewünschten Dateien enthält. Um eine Quellcode-Distribution zu erstellen, übergeben Sie Ihrem Setup-Skript den Befehl `sdist`.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py sdist
running sdist
running check
reading manifest template 'MANIFEST.in'
writing manifest file 'MANIFEST'
creating chardet-1.0.2
creating chardet-1.0.2\chardet
creating chardet-1.0.2\docs
creating chardet-1.0.2\docs\images
copying files to chardet-1.0.2...
copying COPYING -> chardet-1.0.2
copying README.txt -> chardet-1.0.2
copying setup.py -> chardet-1.0.2
copying chardet\__init__.py -> chardet-1.0.2\chardet
copying chardet\big5freq.py -> chardet-1.0.2\chardet
...
copying chardet\universaldetector.py -> chardet-1.0.2\chardet
copying chardet\utf8prober.py -> chardet-1.0.2\chardet
copying docs\faq.html -> chardet-1.0.2\docs
copying docs\history.html -> chardet-1.0.2\docs
copying docs\how-it-works.html -> chardet-1.0.2\docs
copying docs\index.html -> chardet-1.0.2\docs
copying docs\license.html -> chardet-1.0.2\docs
copying docs\supported-encodings.html -> chardet-1.0.2\docs
copying docs\usage.html -> chardet-1.0.2\docs
copying docs\images\caution.png -> chardet-1.0.2\docs\images
copying docs\images\important.png -> chardet-1.0.2\docs\images
copying docs\images\note.png -> chardet-1.0.2\docs\images
copying docs\images\permalink.gif -> chardet-1.0.2\docs\images
copying docs\images\tip.png -> chardet-1.0.2\docs\images
copying docs\images\warning.png -> chardet-1.0.2\docs\images
creating dist
creating 'dist\chardet-1.0.2.zip' and adding 'chardet-1.0.2' to it
adding 'chardet-1.0.2\COPYING'
adding 'chardet-1.0.2\PKG-INFO'
adding 'chardet-1.0.2\README.txt'
```

```
adding 'chardet-1.0.2\setup.py'
adding 'chardet-1.0.2\chardet\big5freq.py'
adding 'chardet-1.0.2\chardet\big5prober.py'
...
adding 'chardet-1.0.2\chardet\universaldetector.py'
adding 'chardet-1.0.2\chardet\utf8prober.py'
adding 'chardet-1.0.2\chardet\__init__.py'
adding 'chardet-1.0.2\docs\faq.html'
adding 'chardet-1.0.2\docs\history.html'
adding 'chardet-1.0.2\docs\how-it-works.html'
adding 'chardet-1.0.2\docs\index.html'
adding 'chardet-1.0.2\docs\license.html'
adding 'chardet-1.0.2\docs\supported-encodings.html'
adding 'chardet-1.0.2\docs\usage.html'
adding 'chardet-1.0.2\docs\images\caution.png'
adding 'chardet-1.0.2\docs\images\important.png'
adding 'chardet-1.0.2\docs\images\note.png'
adding 'chardet-1.0.2\docs\images\permalink.gif'
adding 'chardet-1.0.2\docs\images\tip.png'
adding 'chardet-1.0.2\docs\images\warning.png'
removing 'chardet-1.0.2' (and everything under it)
```

Hier gibt es einiges zu beachten:

- Distutils hat die Manifest-Datei erkannt (`MANIFEST.in`).
- Distutils hat die Manifest-Datei erfolgreich verarbeitet und die zusätzlich gewünschten Dateien – `COPYING.txt` sowie die HTML- und Bilddateien im Verzeichnis `docs/` – hinzugefügt.
- Werfen Sie nun einen Blick in Ihr Projektverzeichnis, so finden Sie dort das von Distutils erzeugte Verzeichnis `dist/`. Innerhalb des `dist/`-Verzeichnisses befindet sich eine `.zip`-Datei, die Sie nun verteilen können.

```
c:\Users\pilgrim\chardet> dir dist
 Volume in drive C has no label.
 Volume Serial Number is DED5-B4F8

 Directory of c:\Users\pilgrim\chardet\dist

07/30/2009  06:29 PM    <DIR>          .
07/30/2009  06:29 PM    <DIR>          ..
07/30/2009  06:29 PM           206,440 chardet-1.0.2.zip
               1 File(s)        206,440 bytes
               2 Dir(s)  61,424,635,904 bytes free
```

17.9 Einen grafischen Installer erstellen

Meiner Meinung nach verdient jede Python-Bibliothek einen grafischen Installer für Windows-Benutzer. Es ist einfach, einen solchen zu erstellen (und das sogar dann, wenn Sie selbst gar kein Windows benutzen) und Windows-Benutzer freuen sich darüber.

Distutils erstellt einen grafischen Windows-Installer für Sie, wenn Sie Ihrem Setup-Skript den Befehl `bdist_wininst` übergeben.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py bdist_wininst
running bdist_wininst
running build
running build_py
creating build
creating build\lib
creating build\lib\chardet
copying chardet\big5freq.py -> build\lib\chardet
copying chardet\big5prober.py -> build\lib\chardet
...
copying chardet\universaldetector.py -> build\lib\chardet
copying chardet\utf8prober.py -> build\lib\chardet
copying chardet\__init__.py -> build\lib\chardet
installing to build\bdist.win32\wininst
running install_lib
creating build\bdist.win32
creating build\bdist.win32\wininst
creating build\bdist.win32\wininst\PURELIB
creating build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\big5freq.py ->
build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\big5prober.py ->
build\bdist.win32\wininst\PURELIB\chardet
...
copying build\lib\chardet\universaldetector.py ->
build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\utf8prober.py ->
build\bdist.win32\wininst\PURELIB\chardet
copying build\lib\chardet\__init__.py ->
build\bdist.win32\wininst\PURELIB\chardet
running install_egg_info
Writing build\bdist.win32\wininst\PURELIB\chardet-1.0.2-py3.1.egg-info
creating 'c:\users\pilgrim\appdata\local\temp\tmp2f4h7e.zip' and adding
'.' to it
adding 'PURELIB\chardet-1.0.2-py3.1.egg-info'
adding 'PURELIB\chardet\big5freq.py'
adding 'PURELIB\chardet\big5prober.py'
```

```
...
adding 'PURELIB\chardet\universaldetector.py'
adding 'PURELIB\chardet\utf8prober.py'
adding 'PURELIB\chardet\__init__.py'
removing 'build\bdist.win32\wininst' (and everything under it)
c:\Users\pilgrim\chardet> dir dist
c:\Users\pilgrim\chardet>dir dist
 Volume in drive C has no label.
 Volume Serial Number is AADE-E29F

 Directory of c:\Users\pilgrim\chardet\dist

07/30/2009  10:14 PM    <DIR>          .
07/30/2009  10:14 PM    <DIR>          ..
07/30/2009  10:14 PM           371,236 chardet-1.0.2.win32.exe
07/30/2009  06:29 PM           206,440 chardet-1.0.2.zip
               2 File(s)        577,676 bytes
               2 Dir(s)  61,424,070,656 bytes free
```

17.9.1 Installierbare Pakete für andere Betriebssysteme erzeugen

Distutils kann Ihnen dabei behilflich sein, installierbare Pakete für Linux-Benutzer zu erstellen. Meiner Meinung nach ist das Ihre Zeit nicht wert. Sollten Sie Ihre Software für Linux zur Verfügung stellen wollen, können Sie diese Zeit besser darin investieren, mit Community-Mitgliedern zusammenzuarbeiten, die sich auf das Packen von Software für die großen Linux-Distrubutionen spezialisiert haben.

Meine `chardet`-Bibliothek befindet sich zum Beispiel in den Debian GNU/Linux-Repositorys (und somit auch in den Ubuntu-Repositorys). Damit hatte ich nichts zu tun; die Pakete sind dort einfach irgendwann aufgetaucht. Die Debian-Community hat ihre eigenen Regeln zum Packen von Python-Bibliotheken, und das Debian-Paket `python-chardet` folgt diesen Konventionen. Da sich das Paket in den Debian-Repositorys befindet, erhalten die Debian-Benutzer je nach ihren Systemeinstellungen Sicherheitsaktualisierungen und/oder neue Versionen der Software.

Die von Distutils erzeugten Linux-Pakete bieten keinen dieser Vorteile. Ihre Zeit können Sie besser anders nutzen.

17.10 Ihre Software zum Python Package Index hinzufügen

Das Hinzufügen von Software zum Python Package Index erfolgt in drei Schritten.

1. Registrieren Sie sich selbst
2. Registrieren Sie Ihre Software

3. Laden Sie Ihre mit `setup.py` und `setup.py bdist_*` erstellten Pakete hoch.

Um sich selbst zu registrieren besuchen Sie die Seite zur Benutzerregistrierung (`http://pypi.python.org/pypi?:action=register_form`). Geben Sie Ihren gewünschten Benutzernamen und Ihr Passwort sowie eine gültige E-Mail-Adresse ein und klicken Sie auf die Schaltfläche `Register`. (Besitzen Sie einen PGP- oder GPG-Schlüssel, können Sie auch diesen eingeben. Besitzen Sie keinen oder wissen Sie nicht, was das heißt, kümmern Sie sich nicht darum.) Überprüfen Sie Ihr E-Mail-Postfach; innerhalb weniger Minuten sollten Sie eine Nachricht von PyPI erhalten, die einen Bestätigungslink enthält. Klicken Sie den Link an, um die Registrierung abzuschließen.

Nun müssen Sie Ihre Software bei PyPI registrieren und hochladen. Das geschieht in nur einem Schritt.

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist
bdist_wininst upload                                          ①
running register
We need to know who you are, so please choose either:
 1. use your existing login,
 2. register as a new user,
 3. have the server generate a new password for you (and email it to
you), or
 4. quit
Your selection [default 1]:  1                                ②
Username: MarkPilgrim                                         ③
Password:
Registering chardet to http://pypi.python.org/pypi            ④
Server response (200): OK
running sdist                                                 ⑤
... output trimmed for brevity ...
running bdist_wininst                                         ⑥
... output trimmed for brevity ...
running upload                                                ⑦
Submitting dist\chardet-1.0.2.zip to http://pypi.python.org/pypi
Server response (200): OK
Submitting dist\chardet-1.0.2.win32.exe to http://pypi.python.org/pypi
Server response (200): OK
I can store your PyPI login so future submissions will be faster.
(the login will be stored in c:\home\.pypirc)
Save your login (y/N)?n                                       ⑧
```

① Wenn Sie Ihr Projekt zum ersten Mal veröffentlichen, fügt Distutils Ihre Software zum Python Package Index hinzu und gibt ihr eine eigene URL. Jedes weitere Mal aktualisiert es die Metadaten des Projekts mit den Änderungen, die Sie an den Parametern von `setup.py` durchgeführt haben könnten. Danach erzeugt

es eine Quellcode-Distribution (`sdist`) sowie einen Windows-Installer (`bdist_wininst`) und lädt alles zu PyPI hoch (`upload`).

② Geben Sie `1` ein oder drücken Sie `EINGABE`, um „use your existing login“ auszuwählen. Damit verwenden Sie Ihren bereits bestehenden Account.

③ Geben Sie nun Ihren zuvor gewählten Benutzernamen und Ihr Passwort ein. Distutils wird Ihr Passwort nicht anzeigen; nicht einmal Sternchen werden ausgegeben. Tippen Sie einfach Ihr Passwort und drücken Sie `EINGABE`.

④ Distutils registriert Ihr Paket im Python Package Index …

⑤ … erzeugt die Quellcode-Distribution …

⑥ … erstellt den Windows-Installer …

⑦ … und lädt beides zum Python Package Index hoch.

⑧ Möchten Sie diesen Vorgang für spätere Veröffentlichungen automatisieren, müssen Sie Ihre Zugangsdaten in einer lokalen Datei speichern. Dies ist sehr unsicher und völlig optional.

Gratulation, Sie besitzen nun Ihre eigene Seite im Python Package Index! Die Adresse lautet `http://pypi.python.org/pypi/`*NAME*; `NAME` ist dabei der String, den Sie Ihrer Datei `setup.py` im Parameter `name` übergeben haben.

Möchten Sie eine neue Version veröffentlichen, geben Sie in `setup.py` einfach die neue Versionsnummer an und starten denselben Befehl noch einmal:

```
c:\Users\pilgrim\chardet> c:\python31\python.exe setup.py register sdist
bdist_wininst upload
```

17.11 Die Zukunft des Packens von Python-Software

Distutils ist nicht das Nonplusultra des Packens von Python-Software, doch während ich dies schreibe (August 2009), ist es das einzige Pack-Framework, das in Python 3 funktioniert. Es existieren noch einige andere Frameworks für Python 2; einige konzentrieren sich auf die Installation, andere auf das Testen und die Verteilung der Software. Einige oder alle dieser Frameworks werden vielleicht zukünftig zu Python 3 portiert.

Bei diesen Frameworks liegt der Schwerpunkt auf der Installation:

- Setuptools
- Pip
- Distribute

Bei den folgenden dreht sich alles um das Testen und die Verteilung:

- `virtualenv`
- `zc.buildout`
- Paver
- Fabric
- `py2exe`

Anhang A – Code mithilfe von `2to3` von Python 2 zu Python 3 portieren

A.1 Los geht's

Nahezu jedes Python 2-Programm benötigt Anpassungen, um korrekt in Python 3 zu laufen. Python 3 enthält ein Skript namens `2to3`, das Ihren Python 2-Code übernimmt und soviel wie möglich davon in Python 3-Code umwandelt. Das Kapitel *Fallstudie: `chardet` zu Python 3 konvertieren* zeigt, wie man das `2to3`-Skript ausführt und einige Probleme, die es *nicht* automatisch beheben kann. Dieser Anhang zeigt nun, was es automatisch machen *kann*.

A.2 `print`-Anweisung

In Python 2 war `print` eine Anweisung. Der auszugebende Text folgte einfach auf den `print`-Befehl. In Python 3 ist `print()` eine Funktion. Übergeben Sie den auszugebenden Text so an `print()`, wie Sie es bei jeder anderen Funktion auch tun würden.

Hinweis	Python 2	Python 3
①	`print`	`print()`
②	`print 1`	`print(1)`
③	`print 1, 2`	`print(1, 2)`
④	`print 1, 2,`	`print(1, 2, end=' ')`
⑤	`print >>sys.stderr, 1, 2, 3`	`print(1, 2, 3, file=sys.stderr)`

① Zur Ausgabe einer leeren Zeile, rufen Sie `print()` ohne Argumente auf.

② Um einen einzelnen Wert auszugeben, rufen Sie `print()` mit einem Argument auf.

③ Zur Ausgabe zweier durch ein Leerzeichen getrennter Werte, rufen Sie `print()` mit zwei Argumenten auf.

④ Das ist recht knifflig. In Python 2 führte ein Komma am Ende der `print`-Anweisung dazu, dass die Werte getrennt durch Leerzeichen ausgegeben wurden

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_18, © Springer-Verlag Berlin Heidelberg 2010

und darauf ein weiteres Leerzeichen, jedoch kein Zeilenumbruch, folgte. Um dies in Python 3 zu erreichen, müssen Sie der `print()`-Funktion das Argument `end= ' '` übergeben. Per Voreinstellung ist `end '\n'` (Wagenrücklauf); überschreiben wir dies, wird der Zeilenumbruch am Ende also unterdrückt.

⑤ In Python 2 konnten Sie die Ausgabe zu einer *Pipe* – wie `sys.stderr` – umleiten, indem Sie die Syntax `>>pipe_name` verwendeten. In Python 3 müssen Sie die Pipe dazu im Argument `file` übergeben. Das Argument `file` ist per Standardeinstellung `sys.stdout` (Standard-Ausgabe); überschreiben wir diese Einstellung, wird die Ausgabe an eine andere Pipe geleitet.

A.3 Unicode-Stringliterale

Python 2 besaß zwei Arten von Strings: Unicode-Strings und Nicht-Unicode-Strings. Python 3 besitzt nur einen Typ: Unicode-Strings.

Hinweis	Python 2	Python 3
①	`u'PapayaWhip'`	`'PapayaWhip'`
②	`ur'PapayaWhip\foo'`	`r'PapayaWhip\foo'`

① Unicode-Stringliterale werden einfach in Stringliterale umgewandelt, die in Python 3 immer Unicode sind.

② Unicode-Rohstrings (in welchen Python einen Backslash nicht als Beginn einer Escape-Sequenz ansieht) werden in Rohstrings ungewandelt. In Python 3 sind Rohstrings immer Unicode.

A.4 Globale `unicode()`-Funktion

Python 2 besaß zwei globale Funktionen mit denen man Objekte in Strings umwandeln konnte: `unicode()` zum Umwandeln in Unicode-Strings und `str()` zum Umwandeln in Nicht-Unicode-Strings. Da Python 3 nur einen Stringtyp besitzt, Unicode-Strings, benötigen Sie lediglich die `str()`-Funktion. (Die `unicode()`-Funktion gibt es nicht mehr.)

Python 2	Python 3
`unicode(irgendetwas)`	`str(irgendetwas)`

A.5 Datentyp `long`

Python 2 besaß für Ganzzahlen zwei verschiedene Datentypen: `int` und `long`. Ein `int` konnte nicht größer als `sys.maxint` sein; dieser Wert hing von der verwendeten Plattform ab. Werte vom Typ `long` wurden durch Anhängen eines `L`

ans Ende der Zahl definiert; sie konnten größer als int-Werte sein. In Python 3 existiert nur noch ein Datentyp für Ganzzahlen: int. Dieser verhält sich im Allgemeinen so, wie der Datentyp long in Python 2. Da es nun keine zwei verschiedenen Datentypen mehr gibt, benötigt man auch nicht länger eine spezielle Syntax zur Unterscheidung.

Hinweis	Python 2	Python 3
①	`x = 1000000000000L`	`x = 1000000000000`
②	`x = 0xFFFFFFFFFFFFL`	`x = 0xFFFFFFFFFFFF`
③	`long(x)`	`int(x)`
④	`type(x) is long`	`type(x) is int`
⑤	`isinstance(x, long)`	`isinstance(x, int)`

① Dezimale long-Literale werden zu dezimalen int-Literalen.

② Hexadezimale long-Literale werden zu hexadezimalen int-Literalen.

③ Die Funktion long() existiert in Python 3 nicht mehr, da der Datentyp long nicht mehr existiert. Verwenden Sie die Funktion int(), um eine Variable in eine Ganzzahl umzuwandeln.

④ Um zu überprüfen, ob eine Variable eine Ganzzahl ist, rufen Sie ihren Datentyp ab und vergleichen Sie diesen mit int, nicht mit long.

⑤ Sie können zur Datentyp-Überprüfung auch die Funktion isinstance() verwenden; benutzen Sie bei einer Ganzzahl-Prüfung int, nicht long.

A.6 <>-Vergleich

Python 2 unterstützte <> als Alternative zu !=, den Vergleichsoperator für Ungleichheit. Python 3 unterstützt den Operator !=, aber *nicht* <>.

Hinweis	Python 2	Python 3
①	`if x <> y:`	`if x != y:`
②	`if x <> y <> z:`	`if x != y != z:`

① Ein einfacher Vergleich.

② Ein etwas komplexerer Vergleich dreier Werte.

A.7 Dictionary-Methode has_key()

In Python 2 besaßen Dictionarys die Methode has_key(), mit der man prüfen konnte, ob das Dictionary einen bestimmten Schlüssel besitzt. Python 3 verzichtet auf diese Methode. Sie müssen stattdessen den Operator in verwenden.

Hinweis	Python 2	Python 3
①	`a_dictionary.has_key('PapayaWhip')`	`'PapayaWhip' in a_dictionary`
②	`a_dictionary.has_key(x) or a_dictionary.has_key(y)`	`x in a_dictionary or y in a_dictionary`
③	`a_dictionary.has_key(x or y)`	`(x or y) in a_dictionary`
④	`a_dictionary.has_key(x + y)`	`(x + y) in a_dictionary`
⑤	`x + a_dictionary.has_key(y)`	`x + (y in a_dictionary)`

① Die einfachste Form.

② Der `in`-Operator hat Vorrang vor dem Operator `or`; Sie müssen hier also keine Klammern um `x in a_dictionary` oder `y in a_dictionary` setzen.

③ Hier benötigen Sie allerdings Klammern um `x or y`, da `in` Vorrang vor `or` hat. (Beachten Sie: Dieser Code unterscheidet sich völlig von dem vorherigen. Zunächst verarbeitet Python `x or y`, was entweder `x` (sollte `x` in einem booleschen Kontext wahr sein) oder `y` ergibt. Danach wird dieser Wert genommen und überprüft, ob er ein Schlüssel von `a_dictionary` ist.)

④ Der Operator + hat Vorrang vor dem `in`-Operator, die Klammern um `x + y` sind hier also eigentlich überflüssig. `2to3` fügt sie dennoch ein.

⑤ Hier werden eindeutig Klammern um `y in a_dictionary` benötigt, da der +-Operator Vorrang vor dem `in`-Operator hat.

A.8 Dictionary-Methoden, die Listen zurückgeben

In Python 2 gaben viele Dictionary-Methoden Listen zurück. Die am häufigsten verwendeten Methoden waren `keys()`, `items()` und `values()`. In Python 3 geben all diese Methoden dynamische *Views* zurück. Manchmal ist das kein Problem. Wird der Rückgabewert der Methode sofort an eine andere Funktion übergeben, die die komplette Folge durchläuft, macht es keinen Unterschied, ob der Datentyp eine Liste oder eine View ist. Andere Male spielt es dagegen eine sehr große Rolle. Wenn Sie eine vollständige Liste adressierbarer Elemente erwartet haben, wird Ihr Code hängenbleiben, *da Views keine Indizierung unterstützen*.

Hinweis	Python 2	Python 3
①	`a_dictionary.keys()`	`list(a_dictionary.keys())`
②	`a_dictionary.items()`	`list(a_dictionary.items())`
③	`a_dictionary.iterkeys()`	`iter(a_dictionary.keys())`
④	`[i for i in a_dictionary.iterkeys()]`	`[i for i in a_dictionary.keys()]`
⑤	`min(a_dictionary.keys())`	*Keine Änderung*

① `2to3` geht auf Nummer sicher und wandelt den Rückgabewert von `keys()` mithilfe der `list()`-Funktion in eine statische Liste um. Dies funktioniert zwar immer, ist jedoch weniger effizient als die Verwendung einer View. Sie sollten den

konvertierten Code durchsehen und prüfen, ob eine Liste absolut notwendig ist, oder nicht auch eine View ausreichend wäre.

② Eine weitere Umwandlung einer View in eine Liste. Diesmal unter Verwendung der Methode `items()`. `2to3` liefert auch bei der Methode `values()` dasselbe Ergebnis.

③ Python 3 unterstützt nicht länger die Methode `iterkeys()`. Verwenden Sie `keys()` und wandeln Sie, sofern nötig, die View mithilfe der `iter()`-Funktion in einen Iterator um.

④ `2to3` erkennt die Verwendung der Methode `iterkeys()` innerhalb einer List Comprehension und wandelt sie in die Methode `keys()` um. Dies funktioniert, da Views iterierbar sind.

⑤ `2to3` erkennt, dass die Methode `keys()` sofort an eine Funktion übergeben wird, die eine komplette Folge durchläuft. Es besteht also kein Grund, den Rückgabewert in eine Liste umzuwandeln. Die Funktion `min()` durchläuft stattdessen fröhlich die View. Dies trifft auf `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()` und `all()` zu.

A.9 Umbenannte und umstrukturierte Module

Einige Module der Python-Standardbibliothek wurden umbenannt. Andere untereinander in Beziehung stehende Module wurden zusammengefasst oder so umstrukturiert, dass Ihre Verbindungen logischer werden.

A.9.1 `http`

In Python 3 wurden einige in Beziehung stehende HTTP-Module in einem Paket – http – zusammengefasst.

Hinweis	Python 2	Python 3
①	`import httplib`	`import http.client`
②	`import Cookie`	`import http.cookies`
③	`import cookielib`	`import http.cookiejar`
④	`import BaseHTTPServer`	`import http.server`
	`import SimpleHTTPServer`	
	`import CGIHttpServer`	

① Das Modul `http.client` implementiert eine Low-Level-Bibliothek, die HTTP-Ressourcen anfragen und HTTP-Antworten verarbeiten kann.

② Das Modul `http.cookies` stellt ein Python-Interface für im `http`-Header `Set-Cookie:` gesendete Browser-Cookies zur Verfügung.

③ Das Modul `http.cookiejar` dient der Bearbeitung der auf der Festplatte gespeicherten Dateien, in denen die bekannten Browser Cookies speichern.

④ Das Modul `http.server` stellt einen einfachen HTTP-Server bereit.

A.9.2 urllib

Python 2 besaß Unmengen sich überschneidender Module zum Parsen, Codieren und Abrufen von URLs. In Python 3 wurden sie alle umstrukturiert und in einem Paket, `urllib`, zusammengefasst.

Hinweis	Python 2	Python 3
①	`import urllib`	`import urllib.request,` `urllib.parse, urllib.error`
②	`import urllib2`	`import urllib.request, urllib.error`
③	`import urlparse`	`import urllib.parse`
④	`import robotparser`	`import urllib.robotparser`

⑤	`from urllib import FancyURLopener`
	`from urllib import urlencode`
		`from urllib.request import`
		`FancyURLopener`
		`from urllib.parse import urlencode`

⑥	`from urllib2 import Request` `from urllib2 import HTTPError`	`from urllib.request import Request` `from urllib.error import HTTPError`

① Das alte `urllib`-Modul besaß in Python 2 eine Vielzahl an Funktionen, wie `urlopen()` zum Abrufen von Daten und `splittype()`, `splithost()` und `splituser()` zum Aufteilen einer URL in ihre Bestandteile. Diese Funktionen wurden innerhalb des neuen `urllib`-Pakets logischer strukturiert. `2to3` passt alle Aufrufe dieser Funktionen an die neuen Bezeichnungen an.

② Das aus Python 2 bekannte Modul `urllib2` wurde in Python 3 mit dem `urllib`-Paket zusammengelegt. Die `build_opener()`-Methode, `Request`-Objekte und `HTTPBasicAuthHandler` sind nach wie vor vorhanden.

③ Das Python 3-Modul `urllib.parse` in Python 3 enthält all die Parsing-Funktionen des alten Python 2-Moduls `urlparse`.

④ Das Modul `urllib.robotparser` verarbeitet `robots.txt`-Dateien.

⑤ Die Klasse `FancyURLopener`, die HTTP-Weiterleitungen und andere Statuscodes verarbeitet, ist im neuen `urllib.request`-Modul immer noch vorhanden. Die Funktion `urlencode()` wurde nach `urllib.parse` verschoben.

⑥ Das `Request`-Objekt befindet sich in `urllib.request`. Konstanten wie `HTTPError` wurden jedoch nach `urllib.error` verschoben.

Habe ich schon erwähnt, dass `2to3` auch die Funktionsaufrufe anpasst? Importiert Ihr Python 2-Code z. B. das `urllib`-Modul und ruft `urllib.urlopen()` zum Abrufen von Daten auf, passt `2to3` sowohl die `import`-Anweisung als auch den Funktionsaufruf an.

Python 2	Python 3

`import urllib`
 	
`print`
`urllib.urlopen('http://diveintopython3.`
`org/').read()`
	`import urllib.request, urllib.parse,`
	`urllib.error`
	`print(urllib.request.urlopen('http://`
	`diveintopython3.org/').read())

A.9.3 dbm

All die verschiedenen DBM-Klone befinden sich nun in einem einzigen Paket: `dbm`. Benötigen Sie eine bestimmte Variante – wie GNU DBM –, so können Sie das entsprechende Modul innerhalb des `dbm`-Pakets importieren.

Python 2	Python 3
`import dbm`	`import dbm.ndbm`
`import gdbm`	`import dbm.gnu`
`import dbhash`	`import dbm.bsd`
`import dumbdbm`	`import dbm.dumb`
`import anydbm`	`import dbm`
`import whichdb`	

A.9.4 xmlrpc

XML-RPC ist eine effiziente Methode zur Ausführung entfernter RPC-Aufrufe über HTTP. Die XML-RPC-Client-Bibliothek und verschiedene XML-RPC-Server-Implementierungen sind nun in einem Paket, `xmlrpc`, vereint.

Python 2	Python 3
`import xmlrpclib`	`import xmlrpc.client`
`import DocXMLRPCServer`	`import xmlrpc.server`
`import SimpleXMLRPCServer`	

A.9.5 Weitere Module

Hinweis	Python 2	Python 3

①	`try:`
	`  import cStringIO as StringIO`
	`except ImportError:`
	`  import StringIO`
		`import io`

Hinweis	Python 2	Python 3

②	`try:`
	`    import cPickle as pickle`
	`except ImportError:`
	`    import pickle`
		`import pickle`

③	`import __builtin__`	`import builtins`
④	`import copy_reg`	`import copyreg`
⑤	`import Queue`	`import queue`
⑥	`import SocketServer`	`import socketserver`
⑦	`import ConfigParser`	`import configparser`
⑧	`import repr`	`import reprlib`
⑨	`import commands`	`import subprocess`

① In Python 2 war der Versuch `cStringIO` als `StringIO` zu importieren und – sollte dies fehlschlagen – das Importieren von `StringIO` ein häufig benutzter Ausdruck. Tun Sie dies in Python 3 nicht; das `io`-Modul erledigt es für Sie. Es sucht die schnellste Implementierung und verwendet sie automatisch.

② Ein ähnlicher Ausdruck wurde auch verwendet, um die schnellste `pickle`-Implementierung zu importieren. Tun Sie dies in Python 3 nicht; das `pickle`-Modul erledigt es für Sie.

③ Das Modul `builtins` enthält die globalen Funktionen, Klassen und Konstanten, die überall in Python benutzt werden. Das Redefinieren einer Funktion im `builtins`-Modul redefiniert die globale Funktion überall. Das ist so mächtig und erschreckend wie es klingt.

④ Das Modul `copyreg` fügt `pickle`-Unterstützung für eigene in C definierte Datentypen hinzu.

⑤ Das Modul `queue` implementiert eine Warteschlange.

⑥ Das Modul `socketserver` stellt eine generische Basisklasse zur Implementierung verschiedener Socket-Server bereit.

⑦ Das Modul `configparser` verarbeitet im `INI`-Format vorliegende Konfigurationsdateien.

⑧ Das Modul `reprlib` implementiert die integrierte Funktion `repr()` neu und fügt Möglichkeiten hinzu, die Länge der Repräsentationen zu bestimmen, bevor sie abgeschnitten werden.

⑨ Das Modul `subprocess` erlaubt das Erzeugen von Prozessen, das Verbinden zu ihren Pipes und das Abrufen ihrer Rückgabecodes.

A.10 Relative Importe innerhalb eines Pakets

Ein Paket ist eine Gruppe zusammengehöriger Module, die als eine Einheit fungieren. Haben Module innerhalb eines Pakets in Python 2 gegenseitig auf sich verwiesen, benutzten Sie `import foo` oder `from foo import Bar`. Der Python

2-Interpreter suchte dann zunächst innerhalb des aktuellen Pakets nach einer Datei namens `foo.py`; dann bewegte er sich weiter durch die anderen Verzeichnisse des Python-Suchpfads (`sys.path`). In Python 3 funktioniert dies ein wenig anders. Statt zunächst das aktuelle Paket zu durchsuchen, wird sofort der Python-Suchpfad zu Rate gezogen. Möchten Sie, dass ein Modul eines Pakets ein anderes Modul desselben Pakets importiert, müssen Sie den relativen Pfad zwischen beiden Modulen ausdrücklich angeben.

Nehmen wir an, wir hätten das folgende Paket mit mehreren Dateien:

```
chardet/
|
+--__init__.py
|
+--constants.py
|
+--mbcharsetprober.py
|
+--universaldetector.py
```

Nehmen wir außerdem an, dass `universaldetector.py` die komplette Datei `constants.py` sowie eine Klasse aus `mbcharsetprober.py` importieren muss. Wie gehen wir dazu vor?

Hinweis	Python 2	Python 3
①	`import constants`	`from . import constants`
②	`from mbcharsetprober`	`from .mbcharsetprober`
	`import MultiByteCharSetProber`	`import MultiByteCharsetProber`

① Müssen Sie ein komplettes Modul von anderswo in ihrem Paket importieren, verwenden Sie die Syntax `from . import`. Der Punkt stellt dabei einen relativen Pfad von dieser Datei (`universaldetector.py`) zur zu importierenden Datei (`constants.py`) dar. Da sich die Dateien in diesem Fall im selben Verzeichnis befinden, benötigen wir nur einen Punkt. Sie können außerdem aus dem übergeordneten Verzeichnis (`from .. import einweiteresmodul`) oder aus einem Unterverzeichnis importieren.

② Um eine bestimmte Klasse oder Funktion aus einem anderen Modul direkt in den Namensbereich Ihres Moduls zu importieren, müssen Sie einen relativen Pfad (ohne Schrägstrich) vor das gewünschte Modul setzen. Im vorliegenden Fall befindet sich `mbcharsetprober.py` im selben Verzeichnis wie `universaldetector.py`; der Pfad ist daher ein einzelner Punkt. Sie können auch aus dem übergeordneten Verzeichnis (`from ..einanderesmodul import EineWeitereKlasse`) oder aus einem Unterverzeichnis importieren.

A.11 Die Iteratormethode `next()`

In Python 2 besaßen Iteratoren eine `next()`-Methode, die das nächste Element der Folge zurückgab. Das gilt auch in Python 3 noch; doch nun existiert auch eine *globale* `next()`-Funktion, die einen Iterator als Argument übernimmt.

Hinweis	Python 2	Python 3
①	`anIterator.next()`	`next(anIterator)`
②	`a_function().next()`	`next(a_function())`

③	`class A:`
	`  def next(self):`
	`  pass`
		`class A:`
		`  def __next__(self):`
		`  pass`

④	`class A:`
	`  def next(self, x, y):`
	`  pass`
		Keine Änderung

⑤	`next = 42`
	`for an_iterator in`
	`a_sequence_of_iterators:`
	`  an_iterator.next()`
		`next = 42`
		`for an_iterator in`
		`a_sequence_of_iterators:`
		`  an_iterator.__next__()`

① Im einfachsten Fall rufen Sie nicht die `next()`-Methode des Iterators auf, sondern übergeben den Iterator selbst an die globale `next()`-Funktion.

② Liegt eine Funktion vor, die einen Iterator zurückgibt, rufen Sie die Funktion auf und übergeben Sie das Ergebnis der `next()`-Funktion. (Das `2to3`-Skript ist schlau genug, dies korrekt umzuwandeln.)

③ Definieren Sie Ihre eigene Klasse und beabsichtigen Sie diese als Iterator zu verwenden, müssen Sie die spezielle Methode `__next__()` definieren.

④ Definieren Sie eine eigene Klasse die zufällig eine Methode namens `next()` besitzt, welche ein oder mehrere Argumente übernimmt, rührt `2to3` diese Methode nicht an. Diese Klasse kann nicht als Iterator genutzt werden, da ihre `next()`-Methode Argumente übernimmt.

⑤ Dies ist etwas knifflig. Verwenden Sie eine lokale Variable namens `next`, so hat diese Vorrang vor der globalen `next()`-Funktion. In diesem Fall müssen Sie die `__next__()`-Methode des Iterators aufrufen, um das nächste Element der Folge zu erhalten. (Alternativ könnten Sie den Code auch so umgestalten, dass die lokale Variable nicht mehr `next` hieße; `2to3` wird eine solche Anpassung nicht automatisch durchführen.)

A.12 Die globale Funktion `filter()`

In Python 2 gab die `filter()`-Funktion eine Liste zurück; diese war das Ergebnis des Filterns einer Folge durch eine Funktion, die für jedes Element der Folge `True` oder `False` zurückgab. In Python 3 gibt die `filter()`-Funktion keine Liste, sondern einen Iterator zurück.

Hinweis	Python 2	Python 3
①	`filter(a_function, a_sequence)`	`list(filter(a_function, a_sequence))`
②	`list(filter(a_function, a_sequence))`	*Keine Änderung*
③	`filter(None, a_sequence)`	`[i for i in a_sequence if i]`
④	`for i in filter(None, a_sequence):`	*Keine Änderung*
⑤	`[i for i in filter(a_function, a_sequence)]`	*Keine Änderung*

① Im einfachsten Fall umschließt `2to3` den Aufruf von `filter()` mit einem Aufruf von `list()`. Das Argument wird einfach durchlaufen und eine echte Liste zurückgegeben.

② Wird der Aufruf von `filter()` dagegen schon von `list()` umschlossen, tut `2to3` gar nichts, da die Tatsache, dass `filter()` einen Iterator zurückgibt ohne Bedeutung ist.

③ Die spezielle Syntax `filter(None, …)` wird von `2to3` in eine semantisch identische List Comprehension umgewandelt.

④ In Kontexten wie `for`-Schleifen, die sowieso die komplette Folge durchlaufen, sind keine Änderungen nötig.

⑤ Auch hier sind keine Änderungen nötig, da die List Comprehension die komplette Folge durchläuft und es keinen Unterschied macht, ob `filter()` einen Iterator oder eine Liste zurückgibt.

A.13 Die globale Funktion `map()`

Die `map()`-Funktion gibt nun – wie `filter()` – einen Iterator zurück. (In Python 2 gab `map()` eine Liste zurück.)

Hinweis	Python 2	Python 3
①	`map(a_function, 'PapayaWhip')`	`list(map(a_function, 'PapayaWhip'))`
②	`map(None, 'PapayaWhip')`	`list('PapayaWhip')`
③	`map(lambda x: x+1, range(42))`	`[x+1 for x in range(42)]`
④	`for i in map(a_function, a_sequence):`	*Keine Änderung*
⑤	`[i for i in map(a_function, a_sequence)]`	*Keine Änderung*

① Wie schon bei `filter()` umschließt `2to3` im einfachsten Fall einen Aufruf der `map()`-Funktion mit einem Aufruf von `list()`.

② Die spezielle Syntax `map(None, …)` wird von `2to3` in einen gleichwertigen Aufruf von `list()` umgewandelt.

③ Ist das erste Argument von `map()` eine Lambda-Funktion, wandelt `2to3` dies in eine identische List Comprehension um.

④ In Kontexten wie `for`-Schleifen, die sowieso die komplette Folge durchlaufen, sind keine Änderungen nötig.

⑤ Auch hier sind keine Änderungen nötig, da die List Comprehension die komplette Folge durchläuft und es keinen Unterschied macht, ob `map()` einen Iterator oder eine Liste zurückgibt.

A.14 Die globale Funktion `reduce()`

In Python 3 wurde die `reduce()`-Funktion aus dem globalen Namensbereich entfernt und ins Modul `functools` verschoben.

Python 2	Python 3
`reduce(a, b, c)`	`from functools import reduce` `reduce(a, b, c)`

A.15 Die globale Funktion `apply()`

Python 2 besaß eine globale Funktion namens `apply()`, die eine Funktion `f` und eine Liste `[a, b, c]` übernahm und `f(a, b, c)` zurückgab. Dasselbe Ergebnis erzielen Sie auch, wenn Sie die Funktion direkt aufrufen und ihr die Liste von Argumenten mit vorangestelltem Sternchen (`*`) übergeben. In Python 3 existiert die `apply()`-Funktion nicht mehr; Sie müssen stattdessen die Sternchen-Notation verwenden.

Hinweis	Python 2	Python 3
①	`apply(a_function, a_list_of_args)`	`a_function(*a_list_of_args)`
②	`apply(a_function, a_list_of_args,` `a_dictionary_of_named_args)`	`a_function(*a_list_of_args,` `**a_dictionary_of_named_args)`
③	`apply(a_function, a_list_of_args + z)`	`a_function(*a_list_of_args + z)`
④	`apply(aModule.a_` `function, a_list_of_args)`	`aModule.a_` `function(*a_list_of_args)`

① Die einfachste Möglichkeit besteht darin, eine Funktion mit einer Liste von Argumenten (eine echte Liste wie `[a, b, c]`) mit vorangestelltem Sternchen aufzurufen. Dies entspricht genau der alten `apply()`-Funktion.

② In Python 2 konnte die `apply()`-Funktion tatsächlich sogar drei Parameter übernehmen: eine Funktion, eine Liste von Argumenten und ein Dictionary von benannten Argumenten. In Python 3 können Sie dasselbe erreichen, indem Sie der Liste von Argumenten ein Sternchen (`*`) und dem Dictionary von benannten Argumenten zwei Sternchen (`**`) voranstellen.

③ Der +-Operator, der hier zur Listenverkettung verwendet wird, hat Vorrang vor dem `*`-Operator; zusätzliche Klammern um `a_list_of_args + z` werden daher nicht benötigt.

④ Das 2to3-Skript ist klug genug, auch komplizierte apply()-Aufrufe umzuwandeln, darunter auch das Aufrufen von Funktionen innerhalb importierter Module.

A.16 Die globale Funktion intern()

In Python 2 konnten Sie die intern()-Funktion mit einem String aufrufen, um diesen zur Optimierung der Performance festzusetzen. In Python 3 wurde die intern()-Funktion ins Modul sys verschoben.

Python 2	Python 3
`intern(aString)`	`sys.intern(aString)`

A.17 exec-Anweisung

Die exec-Anweisung ist, genau wie die print-Anweisung, in Python 3 eine Funktion. Die exec()-Funktion übernimmt einen String aus beliebigem Python-Code und führt diesen aus, als wäre es eine weitere Anweisung oder ein weiterer Ausdruck. exec() ist wie eval(), aber *noch* mächtiger und böser. Die eval()-Funktion kann lediglich einen einzelnen Ausdruck auswerten, exec() dagegen kann mehrere Anweisungen, Importe, Funktionsdeklarationen – im Grunde ein vollständiges Python-Programm in einem String – ausführen.

Hinweis	Python 2	Python 3
①	`exec codeString`	`exec(codeString)`
②	`exec codeString in a_global_namespace`	`exec(codeString, a_global_namespace)`
③	`exec codeString in a_global_namespace, a_local_namespace`	`exec(codeString, a_global_namespace, a_local_namespace)`

① Im einfachsten Fall umschließt das 2to3-Skript den *Code-als-String* mit Klammern, da exec() nun eine Funktion und keine Anweisung ist.

② Die alte exec-Anweisung konnte einen globalen Namensbereich übernehmen, in dem der *Code-als-String* ausgeführt wurde. In Python 3 ist dies ebenfalls möglich; übergeben Sie den Namensbereich dazu einfach als zweites Argument an die exec()-Funktion.

③ Die alte exec-Anweisung konnte außerdem auch einen lokalen Namensbereich (wie die innerhalb einer Funktion definierten Variablen) übernehmen. In Python 3 ist auch dies mit der exec()-Funktion möglich.

A.18 execfile-Anweisung

Genau wie die alte exec-Anweisung führt auch die alte execfile-Anweisung Strings aus als wäre es Python-Code. Während exec einen String übernahm, übernahm execfile einen Dateinamen. In Python 3 gibt es execfile nicht mehr. Sollten Sie unbedingt den Code einer Python-Datei ausführen müssen (und wollen Sie diese Datei nicht einfach importieren), so können Sie dasselbe Ergebnis erhalten, indem Sie die Datei öffnen, ihren Inhalt auslesen, die globale Funktion compile() aufrufen (veranlasst den Python-Interpreter zum Kompilieren des Codes) und dann die neue exec()-Funktion aufrufen.

Python 2	Python 3
`execfile('a_filename')`	`exec(compile(open('a_filename').read(), 'a_filename', 'exec'))`

A.19 repr-Literale (Backticks)

In Python 2 gab es eine spezielle Syntax, mit der man eine Repräsentation eines beliebigen Objekts erhalten konnte. Dazu verwendete man sogenannte *Backticks* (rückwärtige Anführungszeichen) (wie `` `x` ``). In Python 3 hat man diese Möglichkeit immer noch, jedoch muss man dazu nun die globale Funktion repr() nutzen.

Hinweis	Python 2	Python 3
①	`` `x` ``	`repr(x)`
②	`` `'PapayaWhip' + `2`` ``	`repr('PapayaWhip' + repr(2))`

① x kann alles sein – eine Klasse, eine Funktion, ein Modul, ein einfacher Datentyp etc. Die repr()-Funktion kommt mit allem zurecht.

② In Python 2 konnten Backticks verschachtelt werden, wodurch solch verwirrende (aber gültige) Ausdrücke möglich waren. Das 2to3-Skript ist schlau genug, daraus verschachtelte repr()-Aufrufe zu machen.

A.20 try…except-Anweisung

Die Syntax zum Abfangen von Ausnahmen hat sich gegenüber Python 2 leicht verändert.

Hinweis	Python 2	Python 3

①	`try:`
	`  import mymodule`
	`except ImportError, e`
	`  pass`
		`try:`
		`  import mymodule`
		`except ImportError as e:`
		`  pass`

②	`try:`
	`  import mymodule`
	`except (RuntimeError,`
	`ImportError), e`
	`  pass`
		`try:`
		`  import mymodule`
		`except (RuntimeError, ImportError) as e:`
		`  pass`

③	`try:`
	`  import mymodule`
	`except ImportError:`
	`  pass`
		Keine Änderung

④	`try:`
	`  import mymodule`
	`except:`
	`  pass`
		Keine Änderung

① Statt eines auf den Ausnahmetyp folgenden Kommas verwendet Python 3 ein neues Keyword: `as`.

② `as` funktioniert auch dann, wenn man mehrere Ausnahmetypen auf einmal abfängt.

③ Wollen Sie lediglich die Ausnahme abfangen, interessieren sich aber nicht für den Zugriff auf das Ausnahmeobjekt selbst, so stimmt die Syntax von Python 3 mit der von Python 2 überein.

④ Wenn Sie eine Methode nutzen, bei der alle Ausnahmen abgefangen werden, ist die Syntax ebenfalls identisch.

☞ Wenn Sie Module importieren, sollten Sie niemals alle Ausnahmen abfangen (und auch sonst ist dies ratsam). Tun Sie es dennoch, werden auch Dinge wie `KeyboardInterrupt` (wenn der Benutzer `Strg-c` zum Abbrechen des Programms drückt) abgefangen, wodurch der Debugging-Prozess erschwert wird.

A.21 `raise`-Anweisung

Die Syntax zum Auslösen Ihrer eigenen Ausnahmen hat sich geringfügig verändert.

Hinweis	Python 2	Python 3
①	`raise MyException`	*Keine Änderung*
②	`raise MyException, 'error message'`	`raise MyException('error message')`
③	`raise MyException,` `'error message', a_traceback`	`raise MyException('error mes-` `sage').with_traceback(a_traceback)`
④	`raise 'error message'`	*Nicht unterstützt*

① Im einfachsten Fall, dem Auslösen einer Ausnahme ohne eigene Fehlermitteilung, bleibt die Syntax unverändert.

② Die Veränderung wird sichtbar, wenn Sie eine Ausnahme auslösen und eine eigene Fehlermitteilung ausgeben möchten. In Python 2 wurden Ausnahmeklasse und Mitteilung durch ein Komma getrennt; in Python 3 wird die Mitteilung als Parameter übergeben.

③ Python 2 unterstützte eigene Traceback-Angaben beim Auslösen einer Ausnahme. In Python 3 ist dies ebenfalls möglich, die Syntax hat sich jedoch verändert.

④ In Python 2 konnten Sie eine Ausnahme ohne Angabe einer Ausnahmeklasse auslösen. Es wurde dann lediglich eine Fehlermitteilung ausgegeben. In Python 3 ist dies nicht mehr möglich. `2to3` teilt Ihnen in diesem Fall mit, dass dieses Problem nicht automatisch behoben werden konnte.

A.22 `throw`-Methode bei Generatoren

In Python 2 haben Generatoren eine `throw()`-Methode. Der Aufruf von `a_generator.throw()` löst eine Ausnahme an der Stelle aus, an der der Generator angehalten wurde. Danach gibt die Methode den nächsten von der Generator-Funktion erzeugten Wert zurück. In Python 3 ist dies auch noch möglich, die Syntax hat sich allerdings geringfügig geändert.

Hinweis	Python 2	Python 3
①	`a_generator.throw(MyException)`	*Keine Änderung*
②	`a_generator.throw(MyException, 'error message')`	`a_generator.throw(MyException('error message'))`
③	`a_generator.throw('error message')`	*Nicht unterstützt*

① Im einfachsten Fall löst ein Generator eine Ausnahme ohne eigene Fehlermitteilung aus. Die Syntax hat sich nicht geändert.

② Löst der Generator eine Ausnahme mit einer eigenen Fehlermitteilung aus, müssen Sie der Ausnahme den Fehlerstring beim Erstellen übergeben.

③ In Python 2 war es ebenfalls möglich eine Ausnahme nur mit einer eigenen Fehlermitteilung auszulösen. In Python 3 ist dies nicht mehr möglich. Das `2to3`-Skript teilt Ihnen dann mit, dass es das Problem nicht automatisch beheben konnte.

A.23 Die globale Funktion `xrange()`

In Python 2 konnte man auf zwei Arten einen Zahlenbereich erhalten: `range()` gab eine Liste zurück, `xrange()` einen Iterator. In Python 3 gibt `range()` einen Iterator zurück; `xrange()` existiert nicht mehr.

Hinweis	Python 2	Python 3
①	`xrange(10)`	`range(10)`
②	`a_list = range(10)`	`a_list = list(range(10))`
③	`[i for i in xrange(10)]`	`[i for i in range(10)]`
④	`for i in range(10):`	*Keine Änderung*
⑤	`sum(range(10))`	*Keine Änderung*

① Im einfachsten Fall wandelt das `2to3`-Skript `xrange()` einfach in `range()` um.

② Benutzte Ihr Python 2-Code `range()`, so kann das `2to3`-Skript nicht wissen, ob Sie eine Liste benötigen oder ob es auch ein Iterator tun würde. Sicherheitshalber legt es den Rückgabewert mithilfe der `list()`-Funktion in einer Liste ab.

③ Befand sich die `xrange()`-Funktion innerhalb einer List Comprehension muss das Ergebnis nicht in einer Liste abgelegt werden, da die List Comprehension auch mit einem Iterator einwandfrei funktioniert.

④ Auch eine `for`-Schleife hat keinerlei Probleme mit einem Iterator. Eine Änderung ist also auch hier nicht nötig.

⑤ Die `sum()`-Funktion funktioniert ebenfalls mit einem Iterator, `2to3` führt also keine Änderungen durch. Wie bei den Dictionary-Methoden, die Views statt Listen zurückgeben, trifft dies auf `min()`, `max()`, `sum()`, `list()`, `tuple()`, `set()`, `sorted()`, `any()` und `all()` zu.

A.24 Die globalen Funktionen `raw_input()` und `input()`

Python 2 besaß zwei Funktionen, die den Benutzer dazu aufforderten Eingaben auf der Kommandozeile zu machen. Die erste, `input()`, erwartete, dass der Benutzer einen Python-Ausdruck eingab (und gab das Ergebnis zurück). Die zweite, `raw_input()`, gab genau das zurück, was der Benutzer eingegeben hatte. Dies war für Einsteiger sehr verwirrend und wurde von vielen als Ärgernis empfunden. Python 3 hat dieses Ärgernis beseitigt und `raw_input()` in `input()` umbenannt, so dass die Funktion das tut, was man intuitiv von ihr erwartet.

Hinweis	Python 2	Python 3
①	`raw_input()`	`input()`
②	`raw_input('prompt')`	`input('prompt')`
③	`input()`	`eval(input())`

① Im einfachsten Fall wird `raw_input()` in `input()` umgewandelt.

② In Python 2 konnte die `raw_input()`-Funktion einen Eingabeaufforderungstext übernehmen. Dies wurde in Python 3 beibehalten.

③ Müssen Sie den Benutzer auffordern, einen auszuwertenden Python-Ausdruck einzugeben, benutzen Sie die `input()`-Funktion und übergeben Sie das Ergebnis an `eval()`.

A.25 func_*-Funktionsattribute

In Python 2 konnte der Code innerhalb von Funktionen auf spezielle Attribute der Funktion selbst zugreifen. In Python 3 wurden diese Funktionsattribute aus Gründen der Einheitlichkeit umbenannt.

Hinweis	Python 2	Python 3
①	`a_function.func_name`	`a_function.__name__`
②	`a_function.func_doc`	`a_function.__doc__`
③	`a_function.func_defaults`	`a_function.__defaults__`
④	`a_function.func_dict`	`a_function.__dict__`
⑤	`a_function.func_closure`	`a_function.__closure__`
⑥	`a_function.func_globals`	`a_function.__globals__`
⑦	`a_function.func_code`	`a_function.__code__`

① Das Attribut `__name__` (ehemals `func_name`) enthält den Funktionsnamen.

② Das Attribut `__doc__` (ehemals `func_doc`) enthält den im Quellcode der Funktion definierten Docstring.

③ Das Attribut `__defaults__` (ehemals `func_defaults`) ist ein Tupel, das die voreingestellten Argumentwerte enthält, sofern solche vorhanden sind.

④ Das Attribut `__dict__` (ehemals `func_dict`) ist der Namensbereich, der beliebige Funktionsattribute unterstützt.

⑤ Das Attribut `__closure__` (ehemals `func_closure`) ist ein Tupel, das Bindungen für die freien Variablen der Funktion enthält.

⑥ Das Attribut `__globals__` (ehemals `func_globals`) ist eine Referenz auf den globalen Namensbereich des Moduls, in dem die Funktion definiert wurde.

Das Attribut `__code__` (ehemals `func_code`) ist ein Code-Objekt, das den kompilierten Funktionskörper repräsentiert.

A.26 Die Ein-/Ausgabemethode xreadlines()

Dateiobjekte besaßen in Python 2 eine Methode namens `xreadlines()`, die einen Iterator zurückgab, der die Datei zeilenweise auslas. Dies war unter anderem in `for`-Schleifen sehr nützlich. Tatsächlich war es sogar so nützlich, dass spätere Versionen von Python 2 diese Fähigkeit zu den Dateiobjekten selbst hinzufügten.

In Python 3 ist die `xreadlines()`-Methode nicht mehr vorhanden. `2to3` kann zwar in einfachen Fällen die Anpassungen automatisch vornehmen, bei schwierigeren Fällen muss man allerdings manuell eingreifen.

Hinweis	Python 2	Python 3
①	`for line in a_file.xreadlines():`	`for line in a_file:`
②	`for line in a_file.xreadlines(5):`	*Keine Änderung (funktioniert nicht)*

① Haben Sie `xreadlines()` ohne Argumente aufgerufen, wird `2to3` dies einfach in ein Dateiobjekt umwandeln. In Python 3 wird so dasselbe erreicht: Die Datei wird Zeile für Zeile gelesen und der Körper der `for`-Schleife ausgeführt.

② Haben Sie `xreadlines()` dagegen mit Argumenten aufgerufen (die Anzahl der zu lesenden Zeilen), so wird `2to3` keine Anpassungen durchführen und Ihr Code wird mit einem `AttributeError: '_io.TextIOWrapper' object has no attribute 'xreadlines'` fehlschlagen. Sie können `xreadlines()` jedoch manuell in `readlines()` umändern, so dass es auch unter Python 3 korrekt funktioniert. (Die `readlines()`-Methode gibt nun einen Iterator zurück, ist also genauso effizient wie `xreadlines()` in Python 2.)

A.27 `lambda`-Funktionen, die ein Tupel anstatt mehrerer Parameter übernehmen

In Python 2 konnten Sie namenlose `lambda`-Funktionen definieren, die mehrere Parameter übernehmen konnten, indem die Funktion so definiert wurde, als würde sie ein Tupel mit einer bestimmten Anzahl an Elementen übernehmen. Python 2 „entpackte“ dieses Tupel dann in benannte Argumente, auf die Sie innerhalb der `lambda`-Funktion (per Name) verweisen konnten. Auch in Python 3 können Sie einer `lambda`-Funktion ein Tupel übergeben; der Python-Interpreter entpackt dieses Tupel aber nicht in benannte Argumente. Stattdessen müssen Sie jedes Argument über seinen Index ansprechen.

Hinweis	Python 2	Python 3
①	`lambda (x,): x + f(x)`	`lambda x1: x1[0] + f(x1[0])`
②	`lambda (x, y): x + f(y)`	`lambda x_y: x_y[0] + f(x_y[1])`
③	`lambda (x, (y, z)): x + y + z`	`lambda x_y_z: x_y_z[0] + x_y_z[1][0] + x_y_z[1][1]`
④	`lambda x, y, z: x + y + z`	*Keine Änderung*

① Hätten Sie eine `lambda`-Funktion definiert, die ein Tupel aus einem Element übernähme, würde dies in Python 3 zu einer `lambda`-Funktion mit Verweisen auf `x1[0]`. Der Bezeichner `x1` wird vom `2to3`-Skript, basierend auf den benannten Argumenten des ursprünglichen Tupels, automatisch erzeugt.

② Eine `lambda`-Funktion mit einem Tupel aus zwei Elementen `(x, y)` wird zu `x_y` mit den Argumenten `x_y[0]` und `x_y[1]`.

③ Das `2to3`-Skript kommt sogar mit verschachtelten Tupeln benannter Argumente zurecht. Der sich daraus ergebende Python 3-Code ist zwar recht unlesbar, funktioniert jedoch genauso wie der alte Code in Python 2.

④ Sie können lambda-Funktionen definieren, die mehrere Argumente übernehmen. Ohne Klammern um die Argumente, behandelt Python 2 dies als eine einfache lambda-Funktion mit mehreren Argumenten; innerhalb der lambda-Funktion können Sie, wie bei jeder anderen Funktion, über die Namen auf die Argumente zugreifen. Diese Syntax funktioniert auch in Python 3 noch.

A.28 Besondere Methodenattribute

In Python 2 können Klassenmethoden sowohl auf das Klassenobjekt in dem sie definiert sind als auch auf das Methodenobjekt selbst verweisen. im_self ist die Klasseninstanz; im_func ist das Funktionsobjekt; im_class ist die Klasse von im_self. In Python 3 wurden diese besonderen Methodenattribute aus Gründen der Konsistenz umbenannt.

Python 2	Python 3
aClassInstance.aClassMethod.*im_func*	aClassInstance.aClassMethod.__func__
aClassInstance.aClassMethod.*im_self*	aClassInstance.aClassMethod.__self__
aClassInstance.aClassMethod.*im_class*	aClassInstance.aClassMethod.__self__.__class__

A.29 Die spezielle Methode __nonzero__

In Python 2 konnten Sie Ihre eigenen in einem booleschen Kontext verwendbaren Klassen erstellen. So konnten Sie z. B. eine Instanz einer Klasse erstellen und diese Instanz in einer if-Anweisung verwenden. Um dies zu erreichen, definierten Sie die spezielle Methode __nonzero__(), die True oder False zurückgab und immer dann aufgerufen wurde, wenn die Instanz in einem booleschen Kontext verwendet wurde. In Python 3 steht Ihnen diese Funktion auch zu Verfügung, jedoch unter dem Namen __bool__().

Hinweis	Python 2	Python 3

①	class A:
	  def *__nonzero__*(self):
	  pass
		class A:
		  def *__bool__*(self):
		  pass

②	class A:
	  def __nonzero__(self, x, y):
	  pass
		Keine Änderung

① Statt __nonzero__() ruft Python 3 beim Auswerten einer in einem booleschen Kontext verwendeten Instanz __bool__() auf.

② Haben Sie allerdings eine __nonzero__()-Methode definiert, die Argumente übernimmt, geht das 2to3-Skript davon aus, dass die Methode in einem anderen Zusammenhang verwendet wird und führt keine Änderungen durch.

A.30 Oktale Literale

Die Syntax zur Definition von auf der Basis 8 basierenden Zahlen (oktale Zahlen) hat sich geringfügig verändert.

Python 2	Python 3
`x = 0755`	`x = 0o755`

A.31 sys.maxint

Aufgrund der Zusammenlegung der Datentypen `long` und `int`, ist die Konstante `sys.maxint` nicht mehr korrekt. Da der Wert aber bei der Bestimmung plattformspezifischer Möglichkeiten nach wie vor nützlich sein kann, wurde er unter der Bezeichnung `sys.maxsize` beibehalten.

Hinweis	Python 2	Python 3
①	`from sys import maxint`	`from sys import maxsize`
②	`a_function(sys.maxint)`	`a_function(sys.maxsize)`

① `maxint` wird zu `maxsize`.
② Jedes Vorkommen von `sys.maxint` wird zu `sys.maxsize`.

A.32 Die globale Funktion callable()

In Python 2 konnten Sie mithilfe der globalen Funktion `callable()` überprüfen, ob ein Objekt aufrufbar war oder nicht. In Python 3 existiert diese globale Funktion nicht mehr. Um zu überprüfen, ob ein Objekt aufrufbar ist, müssen Sie prüfen, ob die spezielle Methode `__call__()` vorhanden ist.

Hinweis	Python 2	Python 3
①	`callable(irgendwas)`	`hasattr(irgendwas, '__call__')`

A.33 Die globale Funktion zip()

In Python 2 übernahm die globale Funktion `zip()` eine beliebige Zahl an Folgen und gab eine Liste von Tupeln zurück. Das erste Tupel enthielt das erste Element jeder Folge; das zweite Tupel enthielt das zweite Element jeder Folge usw. In Python 3 gibt `zip()` einen Iterator statt einer Liste zurück.

Hinweis	Python 2	Python 3
①	`zip(a, b, c)`	`list(zip(a, b, c))`
②	`d.join(zip(a, b, c))`	*Keine Änderung*

① Im einfachsten Fall erhalten Sie die alte Funktionsweise der `zip()`-Funktion, indem Sie den Rückgabewert mit einem Aufruf von `list()` umschließen; so wird der von `zip()` zurückgegebene Iterator durchlaufen und eine echte Liste der Ergebnisse zurückgegeben.

② Wird `zip()` in einem Kontext verwendet, der bereits alle Elemente einer Folge durchläuft (wie bei diesem Aufruf der `join()`-Methode), funktioniert der von `zip()` zurückgegebene Iterator völlig problemlos. Das `2to3`-Skript ist klug genug, solche Fälle zu erkennen und führt dann keinerlei Änderungen am Code durch.

A.34 Die Ausnahme `StandardError`

In Python 2 bildete `StandardError` die Basisklasse aller integrierten Ausnahmen außer `StopIteration`, `GeneratorExit`, `KeyboardInterrupt` und `SystemExit`. In Python 3 ist `StandardError` nicht mehr vorhanden; verwenden Sie anstelle dessen `Exception`.

Python 2	Python 3
`x = StandardError()`	`x = Exception()`
`x = StandardError(a, b, c)`	`x = Exception(a, b, c)`

A.35 Konstanten des Moduls `types`

Das Modul `types` enthält verschiedene Konstanten, die Ihnen beim bestimmen des Typs eines Objekts helfen. In Python 2 enthielt das Modul Konstanten für alle einfachen Datentypen, wie `dict` und `int`. In Python 3 wurden diese Konstanten entfernt; benutzen Sie stattdessen einfach den Namen des Datentyps.

Python 2	Python 3
`types.UnicodeType`	`str`
`types.StringType`	`bytes`
`types.DictType`	`dict`
`types.IntType`	`int`
`types.LongType`	`int`
`types.ListType`	`list`
`types.NoneType`	`type(None)`
`types.BooleanType`	`bool`
`types.BufferType`	`memoryview`
`types.ClassType`	`type`

Python 2	Python 3
types.ComplexType	complex
types.EllipsisType	type(Ellipsis)
types.FloatType	float
types.ObjectType	object
types.NotImplementedType	type(NotImplemented)
types.SliceType	slice
types.TupleType	tuple
types.TypeType	type
types.XRangeType	range

☞ types.StringType wird zu bytes statt str, weil ein Python 2-„String“ (kein Unicode-String, sondern ein einfacher String) in Wirklichkeit eine Bytefolge in einer bestimmten Zeichencodierung ist.

A.36 Die globale Funktion isinstance()

Die Funktion isinstance() überprüft, ob ein Objekt eine Instanz einer bestimmten Klasse oder eines bestimmten Datentyps ist. In Python 2 konnten Sie dazu ein Tupel von Datentypen übergeben, woraufhin isinstance() True zurückgab, falls das Objekt einem dieser Typen entsprach. In Python 3 ist dies immer noch möglich, jedoch ist das wiederholte Angeben eines Datentyps nicht mehr erlaubt.

Python 2	Python 3
isinstance(x, (int, float, int))	isinstance(x, (int, float))

A.37 Der Datentyp basestring

Python 2 besaß zwei Arten von Strings: Unicode-Strings und Nicht-Unicode-Strings. Doch außerdem gab es noch einen weiteren Typ: basestring. Dies war ein abstrakter Typ, eine Überklasse für die Typen str und unicode. basestring konnte weder direkt aufgerufen noch instanziiert werden; doch es war möglich, basestring an die globale Funktion isinstance() zu übergeben, um so zu überprüfen, ob ein Objekt ein Unicode-String oder ein Nicht-Unicode-String war. Da es in Python 3 nur noch einen Stringtyp gibt, existiert basestring nicht mehr.

Python 2	Python 3
isinstance(x, basestring)	isinstance(x, str)

A.38 Das Modul itertools

Mit Python 2.3 wurde das Modul itertools eingeführt, das Varianten der globalen Funktionen zip(), map() und filter() definierte, die Iteratoren statt Listen zurückgaben. In Python 3 geben die entsprechenden globalen Funktionen bereits Iteratoren zurück, weshalb diese Funktionen aus dem itertools-Modul entfernt wurden. (Das itertools-Modul enthält aber immer noch sehr viele nützliche Funktionen.)

Hinweis	Python 2	Python 3
①	`itertools.izip(a, b)`	`zip(a, b)`
②	`itertools.imap(a, b)`	`map(a, b)`
③	`itertools.ifilter(a, b)`	`filter(a, b)`
④	`from itertools import imap, izip, foo`	`from itertools import foo`

① Nutzen Sie statt itertools.izip() einfach die globale zip()-Funktion.
② Statt itertools.imap() nutzen Sie einfach map().
③ Aus itertools.ifilter() wird filter().
④ Das itertools-Modul ist in Python 3 nach wie vor vorhanden; lediglich die zum globalen Namensbereich migrierten Funktionen sind nicht mehr darin enthalten. Das 2to3-Skript ist klug genug, die nicht mehr vorhandenen Importe zu entfernen, die anderen aber intakt zu lassen.

A.39 sys.exc_type, sys.exc_value, sys.exc_traceback

Python 2 besaß im sys-Modul drei Variablen, auf die Sie zugreifen konnten, während eine Ausnahme behandelt wurde: sys.exc_type, sys.exc_value und sys.exc_traceback. (Tatsächlich waren diese Variablen schon in Python 1 vorhanden.) Seit Python 1.5 waren diese Variablen veraltet und man nutzte nun sys.exc_info, ein Tupel, das alle drei Werte enthält. In Python 3 sind die drei einzelnen Variablen nicht mehr vorhanden; Sie *müssen* nun sys.exc_info nutzen.

Python 2	Python 3
`sys.exc_type`	`sys.exc_info()[0]`
`sys.exc_value`	`sys.exc_info()[1]`
`sys.exc_traceback`	`sys.exc_info()[2]`

A.40 Tupeldurchlaufende List Comprehensions

Wollten Sie in Python 2 eine List Comprehension programmieren, die ein Tupel durchlief, mussten Sie *keine* Klammern um die Werte des Tupels setzen. In Python 3 sind Klammern dagegen *ausdrücklich* nötig.

Python 2	Python 3
`[i for i in 1, 2]`	`[i for i in (1, 2)]`

A.41 Die Funktion `os.getcwdu()`

Python 2 besaß eine Funktion namens `os.getcwd()`, die das aktuelle Arbeitsverzeichnis als (nicht-Unicode) String zurückgab. Da moderne Dateisysteme mit Verzeichnisnamen in beliebigen Zeichencodierungen umgehen können, wurde mit Python 2.3 die Funktion `os.getcwdu()` eingeführt. Diese Funktion gab das aktuelle Arbeitsverzeichnis als Unicode-String zurück. Da es in Python 3 nur noch einen Stringtyp (Unicode) gibt, benötigen Sie lediglich `os.getcwd()`.

Python 2	Python 3
`os.getcwdu()`	`os.getcwd()`

A.42 Metaklassen

In Python 2 konnten Sie eine Metaklasse entweder durch die Definition des Arguments `metaclass` innerhalb der Klassendeklaration oder durch die Definition eines speziellen Attributs auf Klassenebene – `__metaclass__` – erstellen. In Python 3 ist das Attribut auf Klassenebene *nicht* länger vorhanden.

Hinweis	Python 2	Python 3
①	`class C(metaclass=PapayaMeta):` `  pass`	*Keine Änderung*
②	`class Whip:` `  __metaclass__ = PapayaMeta`	`class Whip(metaclass=PapayaMeta):` `  pass`

③	`class C(Whipper, Beater):`
	`  __metaclass__ = PapayaMeta`
		`class C(Whipper, Beater, metaclass=`
		`PapayaMeta):`
		`  pass`

① Die Deklaration der Metaklasse innerhalb der Klassendeklaration funktionierte in Python 2, und ebenso auch noch in Python 3.

② Die Deklaration der Metaklasse durch ein Klassenattribut funktionierte in Python 2; in Python 3 jedoch nicht mehr.

③ Das `2to3`-Skript ist schlau genug, selbst dann eine gültige Klassendeklaration aufzubauen, wenn die Klasse von einer oder mehreren Basisklassen abgeleitet ist.

A.43 Stilfragen

Die restlichen hier genannten „Problembehebungen“ sind eigentlich keine Behebungen. Es geht dabei vielmehr um Stilfragen als ums Wesentliche. Sie funktionieren in Python 3 genauso gut wie in Python 2. Doch da die Entwickler von Python ein Interesse an der Einheitlichkeit des Python-Codes haben, gibt es einen offiziellen „Python Style Guide“, der – mit entsetzlicher Genauigkeit – darlegt, wie man seinen Code gestalten sollte. Die Entwickler von Python haben sich aufgrund der hervorragenden Möglichkeiten des 2to3-Skripts dazu entschieden, einige optionale Funktionen zur Verbesserung der Lesbarkeit Ihrer Python-Programme einzufügen.

A.43.1 `set()`-Literale (ausdrücklich)

In Python 2 bestand die einzige Möglichkeit zur Definition eines literalen Sets darin, `set(eine_folge)` aufzurufen. Diese Vorgehensweise funktioniert zwar auch in Python 3, jedoch besteht nun eine verständlichere Möglichkeit dies zu erreichen: geschweifte Klammern. Dies funktioniert immer, außer bei einem leeren Set, da auch Dictionarys geschweifte Klammen verwenden. `{}` ist daher ein leeres Dictionary, kein leeres Set.

☞ Das 2to3-Skript führt diese Anpassung nicht automatisch durch. Um sie einzuschalten, müssen Sie beim Aufruf von 2to3 auf der Kommandozeile zusätzlich `-f set_literal` eingeben.

Vorher	Nachher
`set([1, 2, 3])`	`{1, 2, 3}`
`set((1, 2, 3))`	`{1, 2, 3}`
`set([i for i in a_sequence])`	`{i for i in a_sequence}`

A.43.2 Die globale Funktion `buffer()` (ausdrücklich)

In C implementierte Python-Objekte können ein „Buffer-Interface“ exportieren, das anderem Python-Code das direkte Lesen und Schreiben eines Speicherblocks erlaubt. (Das ist in der Tat so mächtig und gefährlich wie es klingt.) In Python 3 wurde `buffer()` in `memoryview()` umbenannt. (In Wirklichkeit ist es noch etwas komplizierter, aber die Unterschiede kann man fast immer ignorieren.)

☞ Das 2to3-Skript führt diese Anpassung nicht automatisch durch. Um sie einzuschalten, müssen Sie beim Aufruf von 2to3 auf der Kommandozeile zusätzlich `-f buffer` eingeben.

Python 2	Python 3
`x = buffer(y)`	`x = memoryview(y)`

A.43.3 Whitespace bei Kommas (ausdrücklich)

Auch wenn Python bei Whitespace zum Ein- und Ausrücken sehr rigoros ist, so ist es bei Whitespace an anderen Stellen recht tolerant. Innerhalb von Listen, Tupeln, Sets und Dictionarys kann vor und nach Kommas ohne Probleme Whitespace vorkommen. Gemäß dem *Python Style Guide* soll vor einem Komma kein Leerzeichen stehen; hinter einem Komma soll dagegen ein Leerzeichen stehen. Auch wenn dies nur einen ästhetischen Hintergrund hat (der Code funktioniert so oder so), kann das `2to3`-Skript diese Anpassung für Sie vornehmen.

☞ Das `2to3`-Skript führt diese Anpassung nicht automatisch durch. Um sie einzuschalten, müssen Sie beim Aufruf von `2to3` auf der Kommandozeile zusätzlich `-f wscomma` eingeben.

Python 2	Python 3
`a ,b`	`a, b`
`{a :b}`	`{a: b}`

A.43.4 Geläufige Ausdrücke (ausdrücklich)

Innerhalb der Python-Community haben sich einige geläufige Ausdrücke herausgebildet. Manche davon, darunter die `while 1:`-Schleife, gehen zurück auf Python 1. (Python hatte bis zur Version 2.3 keinen booleschen Datentyp, also haben die Entwickler stattdessen `1` und `0` genommen.) Zeitgemäße Python-Programmierer sollten sich die modernen Versionen dieser Ausdrücke aneignen.

☞ Das `2to3`-Skript führt diese Anpassung nicht automatisch durch. Um sie einzuschalten, müssen Sie beim Aufruf von `2to3` auf der Kommandozeile zusätzlich `-f idioms` eingeben.

Python 2	Python 3
`while 1:` `  do_stuff()`	`while True:` `  do_stuff()`
`type(x) == T`	`isinstance(x, T)`
`type(x) is T`	`isinstance(x, T)`
`a_list = list(a_sequence)`	`a_list = sorted(a_sequence)`
`a_list.sort()` `do_stuff(a_list)`	`do_stuff(a_list)`

Anhang B – Spezielle Methoden

B.1 Los geht's

An anderer Stelle des Buches haben wir uns bereits mit einigen speziellen Methoden beschäftigt – „magische" Methoden, die Python aufruft, wenn Sie eine bestimmte Syntax verwenden. Mithilfe spezieller Methoden können Ihre Klassen sich wie Folgen, Dictionarys, Funktionen, Iteratoren oder sogar Zahlen verhalten. Dieser Anhang dient sowohl als Referenz zu den bereits bekannten speziellen Methoden als auch als kurze Einführung zu den seltsameren Methoden.

B.2 Grundlegendes

Wenn Sie die Einführung in Klassen gelesen haben, kennen Sie die am häufigsten anzutreffende spezielle Methode bereits: `__init__()`. Die meisten meiner Klassen benötigen schlussendlich eine Initialisierung. Außerdem gibt es noch weitere grundlegende spezielle Methoden, die vor allem beim Debuggen Ihrer eigenen Klassen nützlich sein können.

Hinweis	Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
①	Eine Instanz initialisieren	`x = MyClass()`	`x.__init__()`
②	Die „formelle" String-Darstellung	`repr(x)`	`x.__repr__()`
③	Den „formlosen" Wert als String	`str(x)`	`x.__str__()`
④	Den „formlosen" Wert als Bytearray	`bytes(x)`	`x.__bytes__()`
⑤	Den Wert als formatierten String	`format(x, format_spec)`	`x.__format__(format_spec)`

M. Pilgrim, *Python 3 – Intensivkurs,*
DOI 10.1007/978-3-642-04377-2_19, © Springer-Verlag Berlin Heidelberg 2010

① Die `__init__()`-Methode wird nach dem Erstellen der Instanz aufgerufen. Wenn Sie Einfluss auf den Erstellungsprozess selbst haben wollen, müssen Sie die Methode `__new__()` verwenden.

② Gemäß Konvention soll die `__repr__()`-Methode einen String zurückgeben, der ein gültiger Python-Ausdruck ist.

③ Die `__str__()`-Methode wird auch beim Aufruf von `print(x)` aufgerufen.

④ Neu in Python 3, da der `bytes`-Typ eingeführt wurde.

⑤ Gemäß Konvention soll `format_spec` mit der *Format Specification Mini-Language* übereinstimmen. `decimal.py` in der Python-Standardbibliothek stellt eine eigene `__format__()`-Methode bereit.

B.3 Klassen, die sich wie Iteratoren verhalten

Im Kapitel zu Iteratoren haben Sie bereits gesehen, wie Sie mithilfe der Methoden `__iter__()` und `__next__()` einen Iterator von Grund auf erstellen können.

Hinweis	Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
①	Eine Folge (`seq`) durchlaufen	`iter(seq)`	`seq.__iter__()`
②	Den nächsten Wert eines Iterators abrufen	`next(seq)`	`seq.__next__()`
③	Einen Iterator in umgekehrter Reihenfolge erstellen	`reversed(seq)`	`seq.__reversed__()`

① Die `__iter__()`-Methode wird immer beim Erstellen eines neuen Iterators aufgerufen. Sie ist der ideale Ort, den Iterator mit Anfangswerten zu initialisieren.

② Die `__next__()`-Methode wird beim Abfragen des nächsten Wertes des Iterators aufgerufen.

③ Die `__reversed__()`-Methode ist ungewöhnlich. Sie übernimmt eine bestehende Folge und gibt einen Iterator zurück, der die Elemente der Folge in umgekehrter Reihenfolge, vom Letzten zum Ersten, ausgibt.

Wie Sie im Kapitel zu Iteratoren erfahren haben, kann eine `for`-Schleife einen Iterator beeinflussen. In dieser Schleife …

```
for x in seq:
    print(x)
```

… ruft Python 3 zum Erstellen eines Iterators `seq.__iter__()` auf. Anschließend wird die `__next__()`-Methode dieses Iterators aufgerufen, um jeden Wert

von `x` zu erhalten. Sobald die `__next__()`-Methode eine `StopIteration`-Ausnahme auslöst, wird die `for`-Schleife anstandslos beendet.

B.4 Berechnete Attribute

Hinweis	Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
①	Ein berechnetes Attribut erhalten (unbedingt)	`x.my_property`	`x.__getattribute__ ('my_property')`
②	Ein berechnetes Attribut erhalten (ersatzweise)	`x.my_property`	`x.__getattr__('my_ property')`
③	Ein Attribut setzen	`x.my_property = value`	`x.__setattr__('my_ property', value)`
④	Ein Attribut löschen	`del x.my_property`	`x.__delattr__('my_ property')`
⑤	Alle Attribute und Methoden auflisten	`dir(x)`	`x.__dir__()`

① Definiert Ihre Klasse eine `__getattribute__()`-Methode, ruft Python diese für jede Referenz auf jedes Attribut und jede Methode auf (außer spezielle Methoden, da dies eine Endlosschleife verursachen würde).

② Definiert Ihre Klasse eine `__getattr__()`-Methode, ruft Python diese erst auf, sobald das Attribut an allen gewöhnlichen Stellen gesucht wurde. Definiert eine Instanz `x` ein Attribut `color`, wird durch `x.color` nicht `x.__getattr__ ('color')` aufgerufen, sondern einfach der bereits definierte Wert von `x.color` zurückgegeben.

③ Die `__setattr__()`-Methode wird aufgerufen, wenn Sie einem Attribut einen Wert zuweisen.

④ Die `__delattr__()`-Methode wird aufgerufen, wenn Sie ein Attribut löschen.

⑤ Die `__dir__()`-Methode ist dann nützlich, wenn Sie eine `__getattr__ ()`- oder `__getattribute__()`-Methode definieren. Normalerweise würde der Aufruf von `dir(x)` nur die regulären Attribute und Methoden auflisten. Würde Ihre `__getattr__()`-Methode das Attribut `color` dynamisch verarbeiten, dann würde `dir(x) color` nicht als eines der verfügbaren Attribute anzeigen. Das Überschreiben der `__dir__()`-Methode erlaubt es Ihnen, auch das Attribut `color` als verfügbar anzuzeigen; dies ist dann besonders hilfreich, wenn andere Programmierer Ihre Klasse benutzen wollen, ohne sich erst mit den inneren Strukturen auseinanderzusetzen.

Die Unterscheidung zwischen `__getattr__()` und `__getattribute__ ()` ist wichtig. Lassen Sie es mich an zwei Beispielen veranschaulichen:

```
class Dynamo:
    def __getattr__(self, key):
        if key == 'color':          ①
            return 'PapayaWhip'
        else:
            raise AttributeError    ②

>>> dyn = Dynamo()
>>> dyn.color                       ③
'PapayaWhip'
>>> dyn.color = 'LemonChiffon'
>>> dyn.color                       ④
'LemonChiffon'
```

① Der Attributname wird der `__getattr__()`-Methode als String übergeben. Ist der Name `'color'`, gibt die Methode einen Wert zurück. (Im vorliegenden Fall ist der String hardcodiert; normalerweise würden Sie irgendeine Art von Berechnung durchführen und das Ergebnis zurückgeben.)

② Ist der Attributname unbekannt, muss Ihre `__getattr__()`-Methode eine `AttributeError`-Ausnahme auslösen, da Ihr Code sonst beim Zugriff auf undefinierte Attribute ohne Hinweis fehlschlagen würde. (Tatsächlich gibt die Methode `None` zurück, wenn keine Ausnahme ausgelöst oder ausdrücklich ein Wert zurückgegeben wird. Das bedeutet, dass alle nicht ausdrücklich definierten Attribute den Wert `None` erhalten; dies ist ziemlich sicher nicht das, was Sie wollen.)

③ Die Instanz `dyn` enthält kein Attribut namens `color`; es wird also die `__getattr__()`-Methode aufgerufen, die einen berechneten Wert bereitstellt.

④ Nachdem `dyn.color` ausdrücklich ein Wert zugewiesen wurde, wird die `__getattr__()`-Methode nicht länger zur Bereitstellung eines Wertes aufgerufen, da `dyn.color` bereits definiert ist.

Der von der `__getattribute__()`-Methode zurückgegebene Wert ist hingegen immer gültig.

```
class SuperDynamo:
    def __getattribute__(self, key):
        if key == 'color':
            return 'PapayaWhip'
        else:
            raise AttributeError

>>> dyn = SuperDynamo()
>>> dyn.color                       ①
'PapayaWhip'
>>> dyn.color = 'LemonChiffon'
>>> dyn.color                       ②
'PapayaWhip'
```

① Die `__getattribute__()`-Methode wird aufgerufen, um einen Wert für `dyn.color` zur Verfügung zu stellen.

② Selbst nachdem Sie `dyn.color` ausdrücklich einen Wert zugewiesen haben, wird immer noch die `__getattribute__()`-Methode aufgerufen, die einen Wert für `dyn.color` bereitstellt. Ist die `__getattribute__()`-Methode vorhanden, wird sie ohne Wenn und Aber für jedes Attribut und jede Methode aufgerufen, sogar für Attribute, denen Sie nach dem Erstellen einer Instanz ausdrücklich Werte zugewiesen haben.

☞ Definiert Ihre Klasse eine `__getattribute__()`-Methode, möchten Sie vielleicht auch eine `__setattr__()`-Methode definieren, und beide verwenden, um die Attributwerte zu verfolgen. Andernfalls werden alle nach dem Erstellen der Instanz gesetzten Attribute im Nirwana verschwinden.

Mit der `__getattribute__()`-Methode müssen Sie besonders vorsichtig sein, da sie auch aufgerufen wird, wenn Python einen Methodennamen Ihrer Klasse nachschlägt.

```
class Rastan:
    def __getattribute__(self, key):
        raise AttributeError           ①
    def swim(self):
        pass

>>> hero = Rastan()
>>> hero.swim()                        ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __getattribute__
AttributeError
```

① Diese Klasse definiert eine `__getattribute__()`-Methode, die immer eine `AttributeException`-Ausnahme auslöst. Der Aufruf von Attributen und Methoden wird *niemals* erfolgreich verlaufen.

② Rufen Sie `hero.swim()` auf, sucht Python in der Klasse `Rastan` nach einer `swim()`-Methode. Dieser Aufruf wird von der `__getattribute__()`-Methode verarbeitet, da *alle* Attribut- und Methodenaufrufe *immer* die `__getattribute__()`-Methode durchlaufen. Im vorliegenden Fall löst die `__getattribute__()`-Methode eine `AttributeError`-Ausnahme aus, weshalb der Aufruf der `swim()`-Methode fehlschlägt.

B.5 Klassen, die sich wie Funktionen verhalten

Die Instanz einer Klasse machen Sie aufrufbar – genau wie eine Funktion aufrufbar ist – indem Sie die Methode `__call__()` definieren.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Eine Instanz wie eine Funktion „aufrufen“	`my_instance()`	`my_instance.__call__()`

Das Modul `zipfile` nutzt diese Methode zur Definition einer Klasse, die eine verschlüsselte ZIP-Datei mithilfe eines vorhandenen Passworts entschlüsseln kann. Der Entschlüsselungs-Algorithmus erfordert während des Entschlüsselungsvorgangs die Speicherung des Status. Die Definition des Decodierers als Klasse erlaubt Ihnen die Verwaltung dieses Status innerhalb einer einzelnen Instanz dieser Klasse. Der Status wird innerhalb der `__init__()`-Methode initialisiert und während des Entschlüsselns der Datei aktualisiert. Da die Klasse aber auch, wie eine Funktion, „aufrufbar“ ist, können Sie die Instanz als erstes Argument der `map()`-Funktion übergeben. Sehen Sie es sich an:

```
# Ausschnitt aus zipfile.py
class _ZipDecrypter:
.
.
.
    def __init__(self, pwd):
        self.key0 = 305419896                    ①
        self.key1 = 591751049
        self.key2 = 878082192
        for p in pwd:
            self._UpdateKeys(p)

    def __call__(self, c):                       ②
        assert isinstance(c, int)
        k = self.key2 | 2
        c = c ^ (((k * (k^1)) >> 8) & 255)
        self._UpdateKeys(c)
        return c
.
.
.
zd = _ZipDecrypter(pwd)                          ③
bytes = zef_file.read(12)
h = list(map(zd, bytes[0:12]))                   ④
```

① Die `_ZipDecrypter`-Klasse verwaltet den Status in Form von drei rotierenden Schlüsseln, die später innerhalb der Methode `_UpdateKeys()` aktualisiert werden (hier nicht gezeigt).

② Die Klasse definiert eine `__call__()`-Methode, die Klasseninstanzen aufrufbar macht. Im vorliegenden Beispiel entschlüsselt die `__call__()`-Methode ein einzelnes Byte der ZIP-Datei und aktualisiert dann die rotierenden Schlüssel basierend auf dem entschlüsselten Byte.

③ `zd` ist eine Instanz der `_ZipDecrypter`-Klasse. Die Variable `pwd` wird an die `__init__()`-Methode übergeben, in der sie gespeichert und später zum erstmaligen Aktualisieren der Schlüssel verwendet wird.

④ Die ersten 12 B einer ZIP-Datei werden entschlüsselt, indem sie `zd` zugeordnet werden; dadurch wird `zd` zwölfmal „aufgerufen" und damit die `__call__()`-Methode, die ihren internen Status aktualisiert und zwölfmal ein Byte als Ergebnis zurückgibt.

B.6 Klassen, die sich wie Folgen verhalten

Dient Ihre Klasse als Behälter für einen Wertesatz und ist es daher sinnvoll herauszufinden, ob Ihre Klasse einen Wert „enthält", sollte die Klasse möglicherweise die folgenden speziellen Methoden definieren, die dafür sorgen, dass sie sich wie eine Folge verhält.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Die Länge einer Folge (`seq`) abrufen	`len(seq)`	`seq.__len__()`
Erfahren, ob eine Folge einen bestimmten Wert enthält	`x in seq`	`seq.__contains__(x)`

Das `cgi`-Modul nutzt diese Methoden in seiner Klasse `FieldStorage`, die alle zu einer dynamischen Webseite gesendeten Formularfelder oder Abfrageparameter repräsentiert.

```
# Ein Skript, das auf http://example.com/search?q=cgi antwortet
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:                                   ①
  do_search()

# Ein Ausschnitt aus cgi.py, der zeigt, wie dies funktioniert
class FieldStorage:
.
.
.
    def __contains__(self, key):                ②
        if self.list is None:
            raise TypeError('not indexable')
        return any(item.name == key for item in self.list)  ③

    def __len__(self):                          ④
        return len(self.keys())                 ⑤
```

① Haben Sie einmal eine Instanz der `cgi.FieldStorage`-Klasse erstellt, können Sie den `in`-Operator zum Überprüfen des Vorhandenseins eines bestimmten Parameters im Abfragestring nutzen.

② Das ist keine Zauberei, sondern wird von der Methode `__contains__()` ermöglicht.

③ Wenn Sie schreiben `if 'q' in fs`, sucht Python nach der `__contains__()`-Methode des `fs`-Objekts, welches in `cgi.py` definiert ist. Der Wert `'q'` wird der `__contains__()`-Methode als Argument `key` übergeben.

④ Die `FieldStorage`-Klasse unterstützt außerdem die Rückgabe ihrer Länge; Sie können also `len(fs)` aufrufen, womit die `__len__()`-Methode der `FieldStorage`-Klasse aufgerufen wird, die die Anzahl der erkannten Abfrageparameter zurückgibt.

⑤ Die Methode `self.keys()` prüft, ob `self.list` den Wert `None` besitzt; die `__len__()`-Methode muss diese Fehlerprüfung also nicht noch einmal durchführen.

B.7 Klassen, die sich wie Dictionarys verhalten

Sie können nicht nur Klassen definieren, die auf den `in`-Operator und die `len()`-Funktion reagieren, sondern auch solche, die sich in vollem Umfang wie Dictionarys verhalten, Werte also basierend auf Schlüsseln zurückgeben.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Einen Wert über seinen Schlüssel erhalten	`x[key]`	`x.__getitem__(key)`
Einen Wert über seinen Schlüssel setzen	`x[key] = value`	`x.__setitem__(key, value)`
Ein Schlüssel-Wert-Paar löschen	`del x[key]`	`x.__delitem__(key)`
Einen Standardwert für fehlende Schlüssel bereitstellen	`x[nonexistent_key]`	`x.__missing__(nonexistent_key)`

Die `FieldStorage`-Klasse des `cgi`-Moduls definiert auch diese speziellen Methoden. Folgendes ist daher möglich:

```
# Ein Skript, das auf http://example.com/search?q=cgi antwortet
import cgi
fs = cgi.FieldStorage()
if 'q' in fs:
  do_search(fs['q'])                    ①

# Ein Ausschnitt aus cgi.py, der zeigt, wie dies funktioniert
class FieldStorage:
```

```
.
.
.
    def __getitem__(self, key):    ②
        if self.list is None:
            raise TypeError('not indexable')
        found = []
        for item in self.list:
            if item.name == key: found.append(item)
        if not found:
            raise KeyError(key)
        if len(found) == 1:
            return found[0]
        else:
            return found
```

① Das `fs`-Objekt ist eine Instanz von `cgi.FieldStorage`, kann jedoch Ausdrücke wie `fs['q']` auswerten.

② `fs['q']` ruft die `__getitem__()`-Methode mit dem auf `'q'` gesetzten Parameter `key` auf. Innerhalb der intern verwalteten Liste der Abfrageparameter (`self.list`) sucht die Methode dann nach einem Element, dessen `.name` dem angegebenen Schlüssel (`key`) entspricht.

B.8 Klassen, die sich wie Zahlen verhalten

Unter Verwendung der passenden speziellen Methoden können Sie eigene Klassen definieren, die sich wie Zahlen verhalten. Sie können sie addieren, subtrahieren und andere mathematische Operationen mit ihnen ausführen. So sind auch Brüche implementiert – die Klasse `Fraction` implementiert diese speziellen Methoden, was Ihnen z. B. Folgendes erlaubt:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> x / 3
Fraction(1, 9)
```

Hier ist die umfassende Liste aller spezieller Methoden, die zum Implementieren einer als Zahl verwendeten Klasse benötigt werden.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Addition	`x + y`	`x.__add__(y)`
Subtraktion	`x - y`	`x.__sub__(y)`
Multiplikation	`x * y`	`x.__mul__(y)`
Division	`x / y`	`x.__truediv__(y)`
„Abrundende“ Division	`x // y`	`x.__floordiv__(y)`

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Modulo (Rest)	`x % y`	`x.__mod__(y)`
„Abrundende“ Division und Modulo	`divmod(x, y)`	`x.__divmod__(y)`
Potenzieren	`x ** y`	`x.__pow__(y)`
Bitverschiebung nach links	`x << y`	`x.__lshift__(y)`
Bitverschiebung nach rechts	`x >> y`	`x.__rshift__(y)`
Bitweises `and`	`x & y`	`x.__and__(y)`
Bitweises `xor`	`x ^ y`	`x.__xor__(y)`
Bitweises `or`	`x | y`	`x.__or__(y)`

Das ist ja schön und gut, wenn `x` eine Instanz einer Klasse ist, die diese Methoden implementiert. Doch was, wenn die Klasse keine dieser Methoden implementiert? Oder noch schlimmer: Was, wenn sie sie zwar implementiert, aber mit bestimmten Argumenten nicht umgehen kann? Ein Beispiel:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)
>>> 1 / x
Fraction(3, 1)
```

Hier wird ein Bruch nicht durch eine Ganzzahl geteilt (wie im vorherigen Beispiel). Das vorherige Beispiel war völlig unproblematisch: `x/3` ruft `x.__truediv__(3)` auf, woraufhin die `__truediv__()`-Methode der `Fraction`-Klasse sich um die Mathematik kümmert. Ganzzahlen dagegen „wissen“ gar nicht, wie sie arithmetische Operationen mit Brüchen ausführen sollen. Wieso funktioniert dieses Beispiel also?

Es existiert ein weiterer Satz spezieller arithmetischer Methoden mit *reflektierten Operanden*. Eine arithmetische Operation unter Verwendung zweier Operanden (z. B. `x/y`) vorausgesetzt, gibt es zwei Möglichkeiten, diese auszuführen:

1. Man teilt `x` mit, sich selbst durch `y` zu dividieren, oder
2. Man teilt `y` mit, `x` durch sich selbst (`y`) zu dividieren.

Die oben gezeigten speziellen Methoden nutzen die erste Vorgehensweise: `x/y` vorausgesetzt, geben sie `x` die Möglichkeit zu sagen „Ich weiß, wie ich mich selbst durch `y` dividieren kann.“ Die folgenden speziellen Methoden nutzen die zweite Vorgehensweise: Sie geben `y` die Möglichkeit zu sagen „Ich weiß, wie ich der Nenner sein kann und `x` durch mich selbst teilen kann.“

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Addition	`x + y`	`x.__radd__(x)`
Subtraktion	`x - y`	`x.__rsub__(x)`
Multiplikation	`x * y`	`x.__rmul__(x)`
Division	`x / y`	`x.__rtruediv_(x)`
„Abrundende“ Division	`x // y`	`x.__rfloordiv__(x)`
Modulo (Rest)	`x % y`	`x.__rmod__(x)`

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
„Abrundende“ Division und Modulo	`divmod(x, y)`	`x.__rdivmod__(x)`
Potenzieren	`x ** y`	`x.__rpow__(x)`
Bitverschiebung nach links	`x << y`	`x.__rlshift__(x)`
Bitverschiebung nach rechts	`x >> y`	`x.__rrshift__(x)`
Bitweises `and`	`x & y`	`x.__rand__(x)`
Bitweises `xor`	`x ^ y`	`x.__rxor__(x)`
Bitweises `or`	`x | y`	`x.__ror__(x)`

Warten Sie! Ich habe noch mehr zu bieten! Nutzen Sie verkürzte Operationen, wie `x/=3`, existieren noch mehr spezielle Methoden, die Sie definieren können.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Addition (Kurzschreibweise)	`x += y`	`x.__iadd__(y)`
Subtraktion (Kurzschreibweise)	`x -= y`	`x.__isub__(y)`
Multiplikation (Kurzschreibweise)	`x *= y`	`x.__imul__(y)`
Division (Kurzschreibweise)	`x /= y`	`x.__itruediv__(y)`
„Abrundende“ Division (Kurzschreibweise)	`x //= y`	`x.__ifloordiv__(y)`
Modulo (Rest) (Kurzschreibweise)	`x %= y`	`x.__imod__(y)`
Potenzieren (Kurzschreibweise)	`x **= y`	`x.__ipow__(y)`
Bitverschiebung nach links (Kurzschreibweise)	`x <<= y`	`x.__ilshift__(y)`
Bitverschiebung nach rechts (Kurzschreibweise)	`x >>= y`	`x.__irshift__(y)`
Bitweises `and` (Kurzschreibweise)	`x &= y`	`x.__iand__(y)`
Bitweises `xor` (Kurzschreibweise)	`x ^= y`	`x.__ixor__(y)`
Bitweises `or` (Kurzschreibweise)	`x |= y`	`x.__ior__(y)`

Beachten Sie: Meist werden die Methoden der verkürzten Operationen nicht benötigt. Definieren Sie für eine bestimmte Operation keine verkürzte Methode, probiert Python die Methoden aus. Um beispielsweise den Ausdruck `x /= y` auszuführen, versucht Python:

1. `x.__itruediv__(y)` aufzurufen. Ist diese Methode definiert und gibt sie einen anderen Wert als `NotImplemented` zurück, sind wir schon fertig.
2. `x.__truediv__(y)` aufzurufen. Ist diese Methode definiert und gibt sie einen anderen Wert als `NotImplemented` zurück, wird der alte Wert von `x` verworfen und mit dem Rückgabewert ersetzt; das ist so, als hätten Sie `x = x / y` aufgerufen.

3. `x.__rtruediv__(x)` aufzurufen. Ist diese Methode definiert und gibt sie einen anderen Wert als `NotImplemented` zurück, wird der alte Wert von `x` verworfen und mit dem Rückgabewert ersetzt.

Sie müssen verkürzte Methoden wie `__itruediv__()` daher nur dann definieren, wenn Sie besondere Optimierungen durchführen möchten. Andernfalls wird Python den verkürzten Operand (z. B. +=) durch einen regulären Operand + eine Variable ersetzen.

Außerdem existieren einige unäre mathematische Operationen, die lediglich mit einem Objekt selbst ausgeführt werden.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Negative Zahl	`-x`	`x.__neg__()`
Positive Zahl	`+x`	`x.__pos__()`
Betrag	`abs(x)`	`x.__abs__()`
Kehrwert	`~x`	`x.__invert__()`
Komplexe Zahl	`complex(x)`	`x.__complex__()`
Ganzzahl	`int(x)`	`x.__int__()`
Fließkommazahl	`float(x)`	`x.__float__()`
Auf nächste Ganzzahl gerundete Zahl	`round(x)`	`x.__round__()`
Auf `n` Stellen gerundete Zahl	`round(x, n)`	`x.__round__(n)`
Kleinste Ganzzahl >= `x`	`math.ceil(x)`	`x.__ceil__()`
Größte Ganzzahl <= `x`	`math.floor(x)`	`x.__floor__()`
`x` auf nächste Ganzzahl in Richtung `0` abschneiden	`math.trunc(x)`	`x.__trunc__()`
Zahl als Listenindex	`a_list[x]`	`a_list[x.__index__()]`

B.9 Vergleichbare Klassen

Ich habe diesen Abschnitt vom vorherigen getrennt, da man Vergleiche nicht nur mit Zahlen durchführen kann. Viele Datentypen lassen sich vergleichen – Strings, Listen, sogar Dictionarys. Erstellen Sie Ihre eigene Klasse, bei der es sinnvoll ist, Ihre Objekte mit anderen Objekten zu vergleichen, können Sie die folgenden speziellen Methode zur Implementierung von Vergleichen verwenden.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Gleichheit	`x == y`	`x.__eq__(y)`
Ungleichheit	`x != y`	`x.__ne__(y)`
Kleiner als	`x < y`	`x.__lt__(y)`
Kleiner oder gleich	`x <= y`	`x.__le__(y)`
Größer als	`x > y`	`x.__gt__(y)`
Größer oder gleich	`x >= y`	`x.__ge__(y)`
Wahrheitswert in einem booleschen Kontext	`if x:`	`x.__bool__()`

☞ Definieren Sie eine `__lt__()`-Methode, aber keine `__gt__()`-Methode, verwendet Python die `__lt__()`-Methode mit vertauschten Werten. Python verknüpft die Methoden jedoch nicht; haben Sie z. B. eine `__lt__()`-Methode und eine `__eq__()`-Methode definiert, wird Python beim Auftreten von `x <= y` nicht nacheinander `__lt__()` und `__eq__()` aufrufen, sondern ausschließlich die `__le__()`-Methode.

B.10 Serialisierbare Klassen

Python unterstützt die Serialisierung und Deserialisierung beliebiger Objekte (auch „Pickling“ und „Unpickling“ genannt). Dies ist nützlich zum Speichern und Laden eines Status *in* bzw. *aus* einer Datei. Alle nativen Datentypen unterstützen Pickling von Haus aus. Erstellen Sie eine eigene Klasse, die Pickling unterstützen soll, können Sie die folgenden speziellen Methoden verwenden:

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Eine individuelle Objekt-Kopie	`copy.copy(x)`	`x.__copy__()`
Eine individuelle *Deep Copy* eines Objekt	`copy.deepcopy(x)`	`x.__deepcopy__()`
Den Status eines Objekts vor dem „Pickling“ erhalten	`pickle.dump(x, file)`	`x.__getstate__()`
Ein Objekt serialisieren	`pickle.dump(x, file)`	`x.__reduce__()`
Ein Objekt serialisieren (neues Pickle-Protokoll)	`pickle.dump(x, file, protocol_version)`	`x.__reduce_ex__(protocol_version)`
Steuern, wie ein Objekt beim „Unpickling“ erstellt wird	`x = pickle.load(file)`	`x.__getnewargs__()`
Den Status eines Objekts nach dem „Unpickling“ wiederherstellen	`x = pickle.load(file)`	`x.__setstate__()`

Zum Wiederherstellen eines serialisierten Objekts muss Python ein neues Objekt erstellen, das wie das serialisierte Objekt aussieht, und dem neuen Objekt alle Attributwerte des serialisierten Objekts zuweisen. Die Methode `__getnewargs__()` regelt die Erstellung des neuen Objekts, während sie `__setstate__()`-Methode die Wiederherstellung der Attributwerte steuert.

B.11 Klassen, die innerhalb eines `with`-Blocks verwendet werden können

Ein `with`-Block definiert einen Laufzeitkontext; Sie „betreten“ den Kontext, indem Sie die `with`-Anweisung ausführen. Sie „verlassen“ den Kontext, nachdem die letzte Anweisung des Blocks ausgeführt wurde.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Etwas tun, wenn ein `with`-Block betreten wird	`with x:`	`x.__enter__()`
Etwas tun, wenn ein `with`-Block verlassen wird	`with x:`	`x.__exit__(exc_type,exc_value, traceback)`

```
# Ausschnitt aus io.py:
def _checkClosed(self, msg=None):
    '''Internal: raise an ValueError if file is closed
    '''
    if self.closed:
        raise ValueError('I/O operation on closed file.'
                         if msg is None else msg)

def __enter__(self):
    '''Context management protocol.  Returns self.'''
    self._checkClosed()                                    ①
    return self                                            ②

def __exit__(self, *args):
    '''Context management protocol.  Calls close()'''
    self.close()                                           ③
```

① Das Dateiobjekt definiert sowohl eine `__enter__()`- als auch eine `__exit__()`-Methode. Die `__enter__()`-Methode prüft, ob die Datei geöffnet ist; wenn nicht, löst die Methode `_checkClosed()` eine Ausnahme aus.

② Die `__enter__()`-Methode sollte fast immer `self` zurückgeben – dies ist das Objekt, unter dessen Verwendung der `with`-Block Eigenschaften und Methoden sendet.

③ Nach dem `with`-Block wird das Dateiobjekt automatisch geschlossen. Wie geht das? In der `__exit__()`-Methode wird `self.close()` aufgerufen.

☞ Die `__exit__()`-Methode wird immer aufgerufen, auch wenn innerhalb des `with`-Blocks eine Ausnahme ausgelöst wurde. Tatsächlich werden beim Auslösen einer Ausnahme die Informationen dazu an die `__exit__()`-Methode übergeben.

B.12 Wirklich seltsames Zeug

Wenn Sie wissen was Sie tun, können Sie nahezu vollständig bestimmen, wie Klassen verglichen, Attribute definiert und welche Art von Klassen als Unterklassen Ihrer Klasse angesehen werden.

Sie möchten …	Also schreiben Sie …	Und Python ruft auf …
Einen Klassenkonstruktor	`x = MyClass()`	`x.__new__()`
Einen Klassendestruktor	`del x`	`x.__del__()`
Dass nur bestimmte Attribute definiert werden		`x.__slots__()`
Einen individuellen Hashwert	`hash(x)`	`x.__hash__()`
Den Wert einer Eigenschaft abrufen	`x.color`	`type(x).__dict__['color'].__get__(x, type(x))`
Den Wert einer Eigenschaft setzen	`x.color = 'PapayaWhip'`	`type(x).__dict__['color'].__set__(x, 'PapayaWhip')`
Eine Eigenschaft löschen	`del x.color`	`type(x).__dict__['color'].__del__(x)`
Steuern, ob ein Objekt eine Instanz Ihrer Klasse ist	`isinstance(x, MyClass)`	`MyClass.__instancecheck__(x)`
Steuern, ob eine Klasse eine Unterklasse Ihrer Klasse ist	`issubclass(C, MyClass)`	`MyClass.__subclasscheck__(C)`
Steuern, ob eine Klasse eine Unterklasse Ihrer abstrakten Basisklasse ist	`issubclass(C, MyABC)`	`MyABC.__subclasshook__(C)`

Sachverzeichnis