Python für alle

Python für alle

Einführung in die Datenanalyse mit Python 3

Charles R. Severance

Deutsche Ausgabe

Fabian Eberts
Heiner Giefers

Originaltitel: *Python for Everybody. Exploring Data Using Python 3.*

Übersetzung und Bearbeitung: Fabian Eberts, Heiner Giefers
Redaktionelle Unterstützung: Elliott Hauser, Sue Blumenberg
Covergestaltung: Aimee Andrion

ISBN 979-8-42547-509-1

1. Auflage 2022

Autorisierte deutsche Übersetzung und Bearbeitung der englischen Ausgabe von *Python for Everybody. Exploring Data Using Python 3.*

Vorwort

Vorwort zur deutschen Ausgabe

Die Deutsche Nationalbibliothek listet über 500 Titel zum Thema Programmierung mit Python, digitale Medien eingeschlossen sind es sogar über 1000. Man kann sich also die Frage stellen, ob das *1001. Buch über Python* wirklich notwendig ist. Wir finden *ja*, denn der Ansatz dieses Buches unterscheidet sich von den meisten anderen zum Thema. *Python für alle* ist Teil eines offenen Curriculums zu Python, das bereits in verschiedene Sprachen übersetzt wurde und in diversen Hochschulen sowie in Onlinekursen Anwendung findet.

Mein großer Dank gilt Charles Severance, der diese Materialsammlung aufgebaut hat und weltweit Menschen dabei unterstützt, seine Inhalte zu verwenden und weiterzuentwickeln. Genau wie der Autor stand auch ich vor der Entscheidung, für einen Kurs zur Einführung in die Programmierung mit Python Lehrmaterialien selbst zu entwickeln oder ein bestehendes Lehrbuch zu verwenden. Viele der vielen Lehrbücher zu Python waren prinzipiell geeignet, aber keines passte genau auf die Anforderungen der Veranstaltung. Der pragmatische Ansatz von *Python for Everybody* eignet sich sehr für Programmieranfänger. Es gibt keinen Anspruch auf Vollständigkeit; vielmehr zielt das Buch darauf ab, die Programmierung mit Python in logisch aufeinander aufbauenden Kapiteln *von Grund auf* zu vermitteln. Dies und die Möglichkeit, das Buch inhaltlich zu erweitern, haben schließlich zum Entschluss geführt, *Python for Everybody* ins Deutsche zu übersetzen.

Die vorliegende Übersetzung orientiert sich weitgehend am englischsprachigen Original. Die Einleitung des Themas in Kapitel 1 wurde etwas verkürzt, in den Kapiteln 2–16 wurden einige Abschnitte aktualisiert und ergänzt, allerdings ohne dabei den Grundaufbau zu verändern. Bei den Programmbeispielen sind die Ausgaben größtenteils ins Deutsche übersetzt worden, Eingabedaten und Bezeichner wurden überwiegend wie in der Originalausgabe belassen.

Neben Charles Severance und allen anderen, die an der Entwicklung der Inhalte dieses Buches beteiligt waren, möchte ich mich auch bei denjenigen bedanken, die so tatkräftig bei der Übersetzung des Buches mitgewirkt haben. Hier sind vor allem Fabian Eberts, Julia Warnke und Sebastian Schmidt zu nennen. Ohne ihren Einsatz hätte sich das Buch nicht in nur wenigen Wochen übersetzen lassen.

Heiner Giefers

Neubearbeitung eines Open-Books

Es ist ganz normal, dass Akademiker, die ständig „publish or perish“ hören, ihre Arbeiten immer von Grund auf neu schaffen wollen. Dieses Buch dagegen versucht, eben nicht bei null anzufangen, sondern stattdessen das Buch *Think Python: How to Think Like a Computer Scientist* von Allen B. Downey, Jeff Elkner und anderen neu zu bearbeiten.

Im Dezember 2009 war ich gerade dabei, mich darauf vorzubereiten, das fünfte Semester in Folge *Networked Programming* an der University of Michigan zu

unterrichten, und beschloss, dass es an der Zeit war, ein Python-Lehrbuch zu schreiben, das sich auf die Analyse von Daten konzentriert, anstatt auf das Vermitteln von Algorithmen und Abstraktionen. Mein Ziel in *Networked Programming* ist es, den Lesern Fähigkeiten im Umgang mit Daten mittels mit Python zu vermitteln. Nur wenige meiner Studenten hatten vor, professionelle Computerprogrammierer zu werden. Stattdessen wollten sie Bibliothekare, Manager, Anwälte, Biologen, Wirtschaftswissenschaftler usw. werden, die in ihrem jeweiligen Fachgebiet die Technologie geschickt einsetzen wollten.

Ich konnte nie das perfekte datenorientierte Python-Buch für meinen Kurs finden, also habe ich mich daran gemacht, ein solches Buch zu schreiben. Glücklicherweise zeigte mir Dr. Atul Prakash bei einer Fakultätssitzung drei Wochen bevor ich in den Ferien mit meinem neuen Buch beginnen wollte das Buch *Think Python*, das er in diesem Semester für seinen Python-Kurs verwendet hatte. Es ist ein gut geschriebenes Informatikbuch mit dem Schwerpunkt auf kurzen, direkten Erklärungen und leichter Erlernbarkeit.

Die Gesamtstruktur des Buches wurde geändert, um so schnell wie möglich zu den Problemen der Datenanalyse zu gelangen und von Anfang an eine Reihe von Beispielen und Übungen zur Datenanalyse anzubieten.

Die Kapitel 2–10 ähneln dem Buch *Think Python*, aber es gibt wichtige Änderungen. Zahlenorientierte Beispiele und Übungen sind durch datenorientierte Übungen ersetzt worden. Die Themen werden in der Reihenfolge präsentiert, die für die Erstellung von zunehmend anspruchsvolleren Datenanalyselösungen erforderlich ist. Einige Themen wie `try` und `except` werden vorgezogen und als Teil des Kapitels über Kontrollstrukturen vorgestellt. Funktionen werden nur sehr oberflächlich behandelt, bis sie zur Bewältigung der Programmkomplexität benötigt werden, aber nicht als frühe Lektion in Abstraktion eingeführt. Fast alle benutzerdefinierten Funktionen wurden aus dem Beispielcode und den Übungen außerhalb von Kapitel 4 entfernt. Das Wort *Rekursion*[1] kommt in dem Buch überhaupt nicht vor.

In den Kapiteln 1 und 11–16 ist das gesamte Material brandneu und konzentriert sich auf reale Anwendungen und einfache Beispiele von Python für die Datenanalyse einschließlich regulärer Ausdrücke für die Suche und das Parsing, die Automatisierung von Aufgaben auf Ihrem Computer, das Abrufen von Daten über das Netzwerk, das Scraping von Webseiten nach Daten, objektorientierte Programmierung, die Verwendung von Webdiensten, das Parsing von XML- und JSON-Daten, die Erstellung und Verwendung von Datenbanken mit der Structured Query Language und die Visualisierung von Daten.

Das ultimative Ziel all dieser Änderungen ist es, den Schwerpunkt von der Informatik auf die Datenverarbeitung und -analyse zu verlagern und nur noch Themen aufzunehmen, die auch dann nützlich sein können, wenn man sich entscheidet, kein professioneller Programmierer zu werden.

Studierende, die dieses Buch interessant finden und weiter vertiefen wollen, sollten sich das Buch *Think Python* von Allen B. Downey ansehen. Da es viele Überschneidungen zwischen den beiden Büchern gibt, werden die Studierenden schnell Fähigkeiten in den zusätzlichen Bereichen der technischen Programmierung und des algorithmischen Denkens erwerben, die in *Think Python* behandelt werden. Und da

[1] außer natürlich in dieser Zeile!

die Bücher einen ähnlichen Schreibstil haben, sollten sie in der Lage sein, *Think Python* mit einem Minimum an Aufwand schnell durchzuarbeiten.

Als Inhaber des Copyrights von *Think Python* hat mir Allen B. Downey die Erlaubnis erteilt, die Lizenz für das Material aus seinem Buch, das in diesem Buch enthalten ist, von der GNU Free Documentation License auf die neuere Creative Commons Attribution-Share Alike Lizenz zu ändern. Dies folgt dem aktuellen Trend der Verschiebung der Lizenzen für offene Dokumentation von der GFDL zur CC-BY-SA (z. B. Wikipedia). Durch die Verwendung der CC-BY-SA-Lizenz wird die starke Copyleft-Tradition des Buches beibehalten, während es für neue Autoren noch einfacher wird, dieses Material nach eigenem Ermessen weiterzuverwenden.

Ich bin der Meinung, dass dieses Buch ein Beispiel dafür ist, warum offene Materialien so wichtig für die Zukunft der Bildung sind, und ich möchte Allen B. Downey und Cambridge University Press für ihre zukunftsweisende Entscheidung danken, das Buch unter einem offenen Copyright zur Verfügung zu stellen. Ich hoffe, dass sie mit dem Ergebnis meiner Bemühungen zufrieden sind und ich hoffe, dass Sie, die Leser, mit *unseren* gemeinsamen Bemühungen zufrieden sind.

Ich möchte Allen B. Downey und Lauren Cowles für ihre Hilfe, Geduld und Beratung bei der Klärung von Urheberrechtsfragen im Zusammenhang mit diesem Buch danken.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Charles Severance ist Professor an der University of Michigan School of Information.

Inhaltsverzeichnis

Kapitel 1

Warum sollte man Programmieren lernen?

Das Schreiben von Programmen (oder Programmieren) ist eine sehr kreative und lohnende Tätigkeit. Wir können Programme aus vielen Gründen schreiben, angefangen mit dem Ziel, damit den Lebensunterhalt zu verdienen, ein schwieriges Datenanalyseproblem zu lösen, Spaß zu haben oder um jemand anderem bei der Lösung eines Problems zu helfen. Dieses Buch geht davon aus, dass *jeder* wissen sollte, wie man programmiert, und dass man, sobald man die Programmierung beherrscht, herausfindet, was man mit den neugewonnenen Fähigkeiten machen kann.

Wir sind in unserem täglichen Leben von Computern umgeben, von Laptops bis hin zu Handys. Wir nehmen diese Computer als unsere „persönlichen Assistenten“ war, die viele Dinge für uns erledigen können. Die Hardware in unseren heutigen Computern ist im Wesentlichen so gebaut, dass sie uns ständig die Frage stellt: „Was soll ich als Nächstes tun?“

Programmierer fügen der Hardware ein Betriebssystem und eine Reihe von Anwendungen hinzu, und schon haben wir einen persönlichen digitalen Assistenten, der uns bei vielen Problemen des Alltags nützlich sein kann. Unsere Computer sind schnell und haben riesige Mengen an Speicher und könnten uns sehr hilfreich sein, wenn wir nur die Sprache beherrschen würden, um dem Computer zu erklären, was er als Nächstes tun soll. Wenn wir diese Sprache kennen, könnten wir dem Computer sagen, dass er in unserem Namen Aufgaben erledigen soll, die sich wiederholen. Interessanterweise sind die Dinge, die Computer am besten können oft die Dinge, die wir Menschen langweilig und stumpfsinnig finden.

Schauen wir uns zum Beispiel die ersten drei Absätze dieses Kapitels an und finden heraus, welches Wort am häufigsten verwendet wird und wie oft es vorkommt. Während wir in der Lage waren, die Wörter in wenigen Sekunden zu lesen und zu verstehen, ist das Zählen der Wörter fast schmerzhaft, weil es nicht die Art von Problem ist, die der menschliche Verstand einfach lösen kann. Für einen Computer ist das Gegenteil der Fall: Das Lesen und Verstehen von Text auf einem Blatt Papier ist für einen Computer schwer, aber die Wörter zu zählen und Ihnen zu

sagen, wie oft das am häufigsten verwendete Wort verwendet wurde, ist für den Computer sehr einfach.

```
python words.py
Enter file:words.txt
Das Wort "die" kommt 9-mal vor
```

Unser Programm sagt uns schnell, dass das Wort „die“ 9 mal im oberen Teil dieses Kapitels verwendet wurde. Genau diese Tatsache, dass Computer Dinge gut können, die Menschen eher nicht gut bzw. schnell können, ist der Grund, warum man die Programmiersprachen beherrschen sollten. Sobald man eine Programmiersprache gelernt hat, kann man viele alltägliche Aufgaben durch den Computer erledigen lassen. So bleibt einem mehr Zeit für die diejenigen Aufgaben, für die wir Menschen einzigartig geeignet sind, nämlich Kreativität, Intuition und Ideenreichtum.

Kreativität und Motivation

Wenn wir gerade mit dem Programmieren beginnen, werden wir noch einige Erfahrungen sammeln müssen, um professionelle Programme (auch *Software* genannt) entwickeln zu können. Professionelles Programmieren ist allerdings sowohl finanziell als auch persönlich eine sehr lohnende Aufgabe. Nützliche, elegante und clevere Programme zu erstellen, ist eine sehr kreative Aktivität, die durchaus Spaß machen kann. Lukrativ ist die Programmierung vor allem dann, wenn unseren Kunden, bzw. den Nutzern das Programm gefällt und uns einen Nutzen bringt. Dabei stehen wir in Konkurrenz zu anderen Entwicklern. Man sollte also versuchen, dass das eigene Programm besser funktioniert, einen höheren Funktionsumfang hat, sich besser bedienen lässt oder einfach schöner aussieht.

Im Moment besteht unsere Hauptmotivation nicht darin, Geld zu verdienen oder den Endnutzern zu gefallen. Wir möchten unsere eigenen Arbeitsabläufe automatisieren und produktiver mit den Daten und Informationen umgehen, die uns in unserem Leben begegnen. Bei unseren Programmieranfängen sind wir sowohl der Programmierer als auch der Endnutzer unserer Programme. Je mehr Erfahrung wir sammeln und je umfangreicher unsere Programme werden, desto mehr werden wir befähigt, auch Programme für andere zu entwickeln.

Der Aufbau eines Computers

Bevor wir anfangen, die Sprachen zu lernen, mit der wir Computer *programmieren* können, sollten wir uns ein wenig damit beschäftigen, wie Computer aufgebaut sind. Wenn wir unseren Computer oder unser Handy auseinandernehmen und tief ins Innere schauen würden, würden wir die folgenden Teile finden:

Die wichtigsten Definitionen dieser Teile lauten wie folgt:

- Die *Central Processing Unit* (oder CPU) ist der Teil des Computers, der so gebaut ist, dass er von der Frage „Was kommt als Nächstes?“ besessen ist.

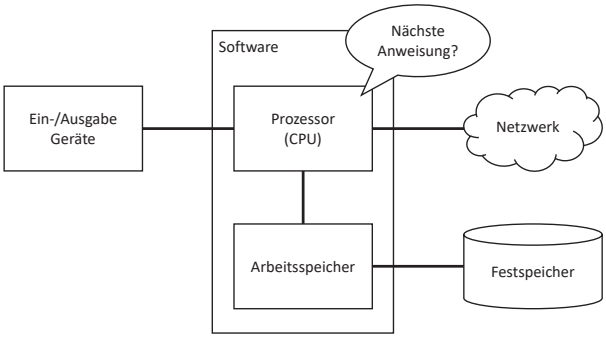

Abbildung 1.1: Aufbau eines Computers

Wenn der Computer auf 3,0 Gigahertz eingestellt ist, bedeutet das, dass die CPU drei Milliarden Mal pro Sekunde fragt: „Was kommt als Nächstes?“. Wir müssten lernen, schnell zu sprechen, um mit der CPU Schritt halten zu können.

- Der *Hauptspeicher* wird zum Speichern von Informationen verwendet, die die CPU schnell benötigt. Der Hauptspeicher ist fast so schnell wie die CPU. Aber die im Hauptspeicher gespeicherten Informationen verschwinden, wenn der Computer ausgeschaltet wird.

- Der *Sekundärspeicher* (oder auch *Festspeicher*) wird ebenfalls zum Speichern von Informationen verwendet, ist aber viel langsamer als der Hauptspeicher. Der Vorteil des Sekundärspeichers ist, dass er Informationen auch dann speichern kann, wenn der Computer nicht mit Strom versorgt wird. Beispiele für Sekundärspeicher sind Festplattenlaufwerke oder Flash-Speicher (typischerweise in USB-Sticks und tragbaren Musikplayern zu finden).

- Die *Eingabe- und Ausgabegeräte* sind unser Bildschirm, unsere Tastatur, Maus, Mikrofon, Lautsprecher oder Touchpad und dienen der Interaktion mit dem Computer.

- Heutzutage haben die meisten Computer auch eine *Netzwerkverbindung*, um Informationen über ein Netzwerk abzurufen. Wir können uns das Netzwerk als einen sehr langsamen Ort vorstellen, an dem Daten gespeichert und abgerufen werden, die nicht immer „verfügbar“ sind. In gewissem Sinne ist das Netzwerk also eine langsamere und manchmal unzuverlässige Form des *Sekundärspeichers*.

Die meisten Details über die Funktionsweise dieser Komponenten überlässt man am besten den Computerbauern, aber es ist hilfreich, eine Terminologie zu haben, damit wir beim Schreiben unserer Programme über diese verschiedenen Teile sprechen können.

Als Programmierer ist es unsere Aufgabe, jede dieser Ressourcen zu nutzen und zu koordinieren, um das Problem zu lösen, das wir lösen müssen, und die Daten zu analysieren, die wir aus der Lösung erhalten. Als Programmierer werden wir hauptsächlich mit der CPU „reden“ und ihr sagen, was sie als Nächstes tun soll.

Manchmal werden wir der CPU sagen, dass sie den Hauptspeicher, den sekundären Speicher, das Netzwerk oder die Eingabe-/Ausgabegeräte verwenden soll.

Wir müssen jeweils die Person sein, die der CPU die Frage „Was nun?“ beantwortet. Allerdings wäre es sehr ineffizient, wenn wir den Dialog mit der CPU *Live* führen würden. Die CPU kann drei Milliarden Mal pro Sekunde einen Befehl ausführen, wir wären aber lange nicht in der Lage, mit diesem Tempo mitzuhalten. Stattdessen müssen wir unsere Anweisungen im Voraus aufschreiben. Wir nennen diese gespeicherten Anweisungen ein *Programm* und den Akt des Aufschreibens dieser Anweisungen und die korrekte Ausführung der Anweisungen *Programmierung*.

Programmierung verstehen

Im weiteren Verlauf dieses Buches werden wir versuchen, aus uns Personen zu machen, die die Kunst des Programmierens beherrschen. Am Ende werden wir echte *Programmierer* sein – vielleicht keine professionellen Programmierer, aber zumindest werden wir die Fähigkeit besitzen, ein Daten-/Informationsanalyseproblem zu betrachten und ein Programm zur Lösung des Problems zu entwickeln.

In gewissem Sinne braucht man zwei Fähigkeiten, um ein Programmierer zu sein:

- Erstens müssen wir die Programmiersprache (Python) kennen – wir müssen das Vokabular und die Grammatik kennen. Wir müssen in der Lage sein, die Wörter in dieser neuen Sprache richtig zu schreiben und wissen, wie man wohlgeformte „Sätze“ in dieser neuen Sprache konstruiert.
- Zweitens müssen wir „eine Geschichte erzählen“ können. Beim Schreiben einer Geschichte kombinieren wir Wörter und Sätze, um dem Leser eine Idee zu vermitteln. Es ist eine Kunst, eine Geschichte zu konstruieren, und die Fähigkeit, eine Geschichte zu schreiben, wird verbessert, indem man etwas schreibt und Feedback erhält. Beim Programmieren ist unser Programm die „Geschichte“ und das Problem, das wir zu lösen versuchen, ist die „Idee“.

Wenn man einmal eine Programmiersprache wie Python gelernt hat, wird es einem viel leichter fallen, eine zweite Programmiersprache wie JavaScript oder C++ zu lernen. Die neue Programmiersprache hat einen ganz anderen Wortschatz und eine andere Grammatik, aber die Problemlösungsfähigkeiten sind in allen Programmiersprachen gleich.

Wir werden das „Vokabular“ und die „Sätze“ von Python ziemlich schnell lernen. Es wird länger dauern, bis man in der Lage ist, ein zusammenhängendes Programm zu schreiben, um ein brandneues Problem zu lösen. Wir lehren das Programmieren ähnlich wie das Schreiben. Wir beginnen damit, Programme zu lesen und zu erklären. Dann schreiben wir einfache Programme und mit der Zeit immer komplexere Programme. Durch das wiederholte Schreiben von Programmen schleift sich eine Routine ein und man beginnt bei neuen Problemstellungen geeignete Lösungsmuster von selbst zu erkennen. Wenn man diesen Punkt erreicht hat, wird das Programmieren zu einem sehr angenehmen und kreativen Prozess.

Wenn Sie nun mit dem Erlernen des Vokabulars und der Struktur von (Python-) Programmen beginnen, seien Sie geduldig und bleiben Sie motiviert, auch einfache

Beispiele nachzuvollziehen und zu variieren. Denken Sie vielleicht daran wie es war, Lesen und Schreiben zu lernen. Auch dies ist am Anfang mühsam gewesen, hat Ihnen aber schlussendlich das Tor geöffnet, um Wissen zu erlangen und weiterzuentwickeln.

Wörter und Sätze

Im Gegensatz zu menschlichen Sprachen ist der Wortschatz von Python ziemlich klein. Wir nennen diesen „Wortschatz“ die „reservierten Wörter“. Das sind Wörter, die für Python eine ganz besondere Bedeutung haben. Wenn Python diese Wörter in einem Python-Programm sieht, haben sie eine (und nur eine) Bedeutung für Python. Später, wenn wir Programme schreiben, werden wir unsere eigenen Wörter erfinden, die für uns eine Bedeutung haben und *Bezeichner* genannt werden. Bei der Wahl der Namen für unsere Bezeichner haben wir einen großen Spielraum, aber wir können keines der reservierten Wörter von Python als Namen für eigene Zwecke verwenden.

Zu den reservierten Wörtern in der Sprache Python gehören die folgenden:

```
and      as        assert  break   class   continue  def
del      elif      else    except  False   finally   for
from     global    if      import  in      is        lambda
None     nonlocal  not     or      pass    raise     return
True     try       while   with    yield
```

Wir werden diese reservierten Wörter und ihre Verwendung zu gegebener Zeit lernen, aber jetzt konzentrieren wir uns erst einmal darauf, wie wir unser Python-Programm mit uns *sprechen* lassen können. Da Programme üblicherweise nicht in Form von gesprochener Sprache, sondern eher durch das Anzeigen von Texten und Bildern „reden“, heißt das Kommando `print` also *Drucke*:

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

Damit haben wir unseren ersten syntaktisch korrekten Python-Satz geschrieben. Der Satz beginnt mit der Funktion `print`, gefolgt von einer Zeichenfolge unserer Wahl, die in einfachen Anführungszeichen steht. Die Zeichenketten in den `print`-Anweisungen sind in Anführungszeichen eingeschlossen. Einfache Anführungszeichen und doppelte Anführungszeichen haben die gleiche Funktion; die meisten Leute verwenden einfache Anführungszeichen, außer in den Fällen, wo ein einfaches Anführungszeichen (ein Apostroph) in der Zeichenkette selbst erscheint.

Wie Sie der Tabelle oben entnehmen können, ist *print* kein reserviertes Wort, sondern ein *Bezeichner*. In diesem Fall haben aber nicht wir den Bezeichner (also den Namen) eingeführt, sondern es gibt ihn bereits *in Python*. Bezeichner, auf denen im Programm direkt eine öffnende Klammer folgt bezeichnen i. d. R. Funktionen, also soetwas wie Unterprogramme, die eine bestimmte Teilaufgabe erledigen. Python bietet eine Vielzahl von solchen Funktionen, die Ihnen das Leben als Programmierer sehr erleichtern. Wir werden im Verlauf des Buches noch viele dieser Funktionen kennen lernen.

Konversation mit Python

Nachdem wir nun ein Wort und einen einfachen Satz in Python kennen, müssen wir wissen, wie wir eine Unterhaltung mit Python beginnen können, um unsere neuen Sprachkenntnisse zu testen.

Bevor wir uns mit Python unterhalten können, müssen wir zunächst die Python-Software auf unserem Computer installieren und lernen, wie man Python auf diesem startet. Sie fragen sich nun vielleicht, warum Sie für das Ausführen Ihrer Python-Programme ein anderes Programm auf Ihrem Computer installieren müssen. Das liegt daran, dass Ihre CPU die Sprache Python nicht direkt versteht. Ihr Python-Programm muss also vor – oder besser gesagt *bei* der Ausführung – von der Sprache Python in die Sprache der CPU *übersetzt* werden. Passiert dieses Übersetzten vor dem Starten des Programms (und damit in der Regel einmalig), nennt man den Vorgang *Kompilieren*. Werden Programme unmittelbar bei der Ausführung, und damit jedes Mal erneut übersetzt, so nennt man das *Interpretieren*.

Letzteres ist bei Python der Fall und daher müssen wir einen *Python-Interpreter* auf unserem PC oder Notebook installieren. Leider gibt es hierzu nicht „die eine Anleitung“. Python ist eine *offene* Programmiersprache. Das bedeutet, dass die Regeln, wie Python-Programme geschrieben werden müssen und wie die Anweisungen der Sprache funktionieren, auf einem gemeinschaftsbasiertes Entwicklungsmodell beruht und vollkommen offengelegt ist. Jeder kann also, nach den vorgegebenen Regeln, einen Python-Interpreter entwickeln und anbieten. Es gibt allerdings eine Standardversion, die man über die Homepage von Python (www.python.org) herunterladen kann. Hier gibt es auch verschiedene Versionen für Windows, MacOS und Linux.

Im Gegensatz zu vielen anderen Programmiersprachen kann man in Python nicht nur ganze Programme starten, sondern man kann auch interaktiv arbeiten und dem Computer einem Befehl nach dem anderen geben. Um dies zu tun, müssen wir auf unserem Computer ein Kommandozeilenfenster öffnen und `python` eingeben. Ist Python korrekt installiert, wird durch diesen Aufruf der *Python-Interpreter* im *interaktiven Modus* gestartet. Das Fenster sollte dann in etwa so aussehen:

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25)
[MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Die Eingabeaufforderung `>>>` ist die Art und Weise, wie der Python-Interpreter fragt: „Was soll ich als Nächstes tun?“ Python ist bereit, ein Gespräch mit uns zu führen. Alles, was wir wissen müssen, ist, wie man die Sprache Python spricht.

Nehmen wir an, wir kennen nicht einmal die einfachsten Wörter oder Sätze in Python. Wir könnten den Standardsatz verwenden, den Astronauten verwenden, wenn sie auf einem fernen Planeten landen und versuchen, mit den Bewohnern des Planeten zu sprechen:

```
>>> I come in peace, please take me to your leader
File "<stdin>", line 1
  I come in peace, please take me to your leader
       ^
SyntaxError: invalid syntax
>>>
```

Das läuft nicht so gut. Wenn uns nicht schnell etwas einfällt, werden die Bewohner des Planeten uns wahrscheinlich nicht sonderlich ernst nehmen. Probieren wir doch lieber mal einen Python-Satz, von dem wir bereits wissen, dass er korrekt ist:

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

Das sieht schon viel besser aus, also versuchen wir, noch etwas mehr zu kommunizieren:

```
>>> print('You must be the legendary god that comes from the sky')
You must be the legendary god that comes from the sky
>>> print('We have been waiting for you for a long time')
We have been waiting for you for a long time
>>> print('Our legend says you will be very tasty with mustard')
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
File "<stdin>", line 1
  print 'We will have a feast tonight unless you say
                                                    ^
SyntaxError: Missing parentheses in call to 'print'
>>>
```

Das Gespräch lief eine Zeit lang sehr gut, und dann haben wir den kleinsten Fehler bei der Verwendung der Sprache Python gemacht, und Python hält uns diesen Fehler gnadenlos vor. An diesem Punkt kann man bereits erkennen, dass Python erstaunlich komplex und mächtig ist und dabei sehr wählerisch ist, was die Syntax angeht, die wir zur Programmierung verwenden. Gleichwohl ist Python aber *nicht* intelligent. Auch wenn es für uns klar ist, was die Anweisung tun soll, wird Python nicht arbeiten können, solange es einen Fehler bei der Syntax gibt.

Bevor wir nun unser erstes Gespräch mit dem Python-Interpreter beenden, sollten wir noch wissen, wie man sich korrekt von Python „verabschiedet“:

```
>>> good-bye
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined
>>> if you don't mind, I need to leave
File "<stdin>", line 1
  if you don't mind, I need to leave
           ^
```

```
SyntaxError: invalid syntax
>>> quit()
```

Man kann feststellen, dass der Fehler bei den ersten beiden Fehlversuchen anders ist. Der zweite Fehler ist anders, weil `if` ein reserviertes Wort ist und Python das reservierte Wort sah und dachte, wir wollten etwas sagen, aber die Syntax des Satzes war falsch.

Der richtige Weg, sich von Python zu verabschieden, ist die Eingabe von `quit()` am interaktiven `>>>`-Prompt. Wir hätten wahrscheinlich eine ganze Weile gebraucht, um das zu erraten, also wird es sich als hilfreich erweisen, ein Buch zur Hand zu haben.

Interpreter und Compiler

Python ist eine *Hochsprache*, die für Menschen relativ einfach zu lesen und zu schreiben und für Computer zu lesen und zu verarbeiten ist. Andere Hochsprachen sind Java, C++, PHP, Ruby, Basic, Perl, JavaScript und viele mehr. Die eigentliche Hardware in der Central Processing Unit (CPU) versteht keine dieser Hochsprachen.

Die CPU versteht eine Sprache, die wir *Maschinensprache* nennen. Maschinensprache ist sehr einfach und ehrlich gesagt sehr mühsam zu schreiben, weil sie nur aus Nullen und Einsen besteht:

```
00101000111010010010101000000001111
11100110000011101010010101101101
...
```

Die Maschinensprache scheint auf den ersten Blick recht einfach zu sein, da es nur Nullen und Einsen gibt, aber ihre Syntax ist noch komplexer und weitaus komplizierter als Python. Daher schreiben nur sehr wenige Programmierer jemals Maschinensprache. Stattdessen entwickeln wir verschiedene Übersetzer, die es Programmierern ermöglichen, Hochsprachen wie Python oder JavaScript zu schreiben, und diese Übersetzer wandeln die Programme in Maschinensprache um, die dann von der CPU ausgeführt wird.

Da die Maschinensprache an die Computerhardware gebunden ist, ist die Maschinensprache nicht *portabel* über verschiedene Arten von Hardware. In Hochsprachen geschriebene Programme können zwischen verschiedenen Computern übertragen werden, indem ein anderer Interpreter auf dem neuen Computer verwendet wird oder der Code neu kompiliert wird, um eine Maschinensprachversion des Programms für die neue Maschine zu erhalten.

Wie schon im oberen Teil dieses Kapitels angedeutet, lassen sich Programmiersprachenübersetzer in zwei allgemeine Kategorien einteilen: (1) Interpreter und (2) Compiler.

Ein *Interpreter* liest den Quellcode des Programms, wie er vom Programmierer geschrieben wurde, analysiert den Quellcode und interpretiert die Befehle im laufenden Betrieb. Python ist ein Interpreter, und wenn wir Python interaktiv

ausführen, können wir eine Zeile (einen Satz) in Python eingeben und Python verarbeitet sie sofort und ist bereit für die Eingabe einer weiteren Python-Zeile.

Bei den Anweisungen in Programmiersprachen kommt es häufig vor, dass wir uns einen Wert für eine spätere Aufgabe merken wollen. In Python können wir das ganz einfach erledigen, indem wir uns einen Namen ausdenken und den Wert mittels eines Gleichheitszeichens dem Namen *zuweisen*.

Wir kennen das Prinzip aus der Mathematik, wo man spätestens in der 5. oder 6. Schulklasse das Rechnen mit *Variablen* erlernt. Höhere Mathematik ohne die Verwendung von Variablen ist praktisch nicht möglich, denn sie sind *das* zentrale Mittel, um Regeln oder Aussagen zu verallgemeinern. Auch in der Programmierung nennt man einen Platzhalter für Werte i. d. R. *Variable*. Dass Python eigentlich keine Variablen verwendet, sondern ausschließlich Namen ist ein technisches Detail. Weil der Begriff der *Variablen* aber so verbreitet ist – in den allermeisten Python Büchern wird von Variablen gesprochen – verwenden wir ihn auch in diesem Buch.

```
>>> x = 6
>>> print(x)
6
>>> y = x * 7
>>> print(y)
42
>>>
```

In diesem Beispiel bitten wir Python, sich den Wert `6` zu merken und dafür den Namen `x` zu verwenden. Wir überprüfen, ob Python sich den Wert tatsächlich gemerkt hat, indem wir `print` verwenden. Dann bitten wir Python, den Wert `x` abzurufen, mit sieben zu multiplizieren und den neu berechneten Wert unter dem Namen `y` zu speichern. Dann möchten wir uns den Wert anzeigen lassen, der sich gerade hinter dem Namen `y` befindet.

Auch wenn wir diese Befehle Zeile für Zeile in Python eingeben, behandelt Python sie als eine geordnete Folge von Anweisungen, wobei spätere Anweisungen Daten abrufen können, die in früheren Anweisungen erstellt wurden. Wir schreiben unseren ersten einfachen Absatz mit vier Sätzen in einer logischen und sinnvollen Reihenfolge.

Es liegt in der Natur eines *Interpreters*, dass er in der Lage ist ein interaktives Gespräch zu führen, wie oben gezeigt. Ein *Compiler* muss das *gesamte* Programm in einer oder mehreren Dateien erhalten. Dann führt er einen Prozess aus, um den High-Level-Quellcode in Maschinensprache zu übersetzen. Danach stellt der Compiler die resultierende Maschinensprache in einer Datei zur späteren Ausführung zur Verfügung.

Wenn wir ein Windows-System haben, haben diese ausführbaren Maschinensprache-Programme die Endungen `.exe` oder `.dll`, welche für „ausführbar“ bzw. „dynamisch gelinkte Bibliothek“ stehen. Unter Linux und Macintosh gibt es kein Suffix, das eine Datei eindeutig als ausführbar kennzeichnet.

Wenn wir eine ausführbare Datei in einem Texteditor öffnen würden, sähe sie völlig verrückt aus und wäre unlesbar:

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

Es ist nicht einfach, Maschinensprache zu lesen oder zu schreiben, daher ist es gut, dass wir *Interpreter* und *Compiler* haben, die es uns ermöglichen, in Hochsprachen wie Python oder C zu schreiben.

An diesem Punkt in unserer Diskussion über Compiler und Interpreter, sollte man sich ein wenig über den Python-Interpreter selbst Gedanken machen. In welcher Sprache ist er geschrieben? Ist er in einer kompilierten Sprache geschrieben? Wenn wir `python` eintippen, was genau passiert dann?

Der (Standard-) Python-Interpreter ist in einer Hochsprache namens „C“ geschrieben. Wir können uns den eigentlichen Quellcode des Python-Interpreters ansehen, indem wir www.python.org aufrufen und uns zum Quellcode durcharbeiten. Python ist also selbst ein Programm und wird in Maschinencode kompiliert. Als wir Python auf unserem Computer installiert haben, haben wir eine Maschinencode-Kopie des übersetzten Python-Programms auf unser System geladen. Unter Windows befindet sich der ausführbare Maschinencode für Python wahrscheinlich in einer Datei mit einem Namen wie:

```
C:\Python35\python.exe
```

Das ist mehr, als man wissen muss, um ein Python-Programmierer zu werden, aber manchmal lohnt es sich, diese kleinen, nervigen Fragen gleich zu Beginn zu beantworten.

Ein Programm schreiben

Das Eingeben von Befehlen in den Python-Interpreter ist ein guter Weg, um mit den Funktionen von Python zu experimentieren, jedoch ist es nicht empfehlenswert für die Lösung komplexer Probleme.

Wenn wir ein Programm schreiben wollen, verwenden wir einen Texteditor, um die Python-Anweisungen in eine Datei zu schreiben, die *Skript* genannt wird. Konventionell haben Python-Skripte die Endung `.py`.

Um das Skript auszuführen, müssen wir dem Python-Interpreter den Namen der Datei mitteilen. In einem Befehlsfenster würden wir `python hallo.py` wie folgt eingeben:

```
$ cat hello.py
print('Hello world!')
$ python hello.py
Hello world!
```

Das `\$` ist die Eingabeaufforderung des Betriebssystems, und das `cat hello.py` zeigt zeigt uns, dass die Datei `hello.py` ein einzeiliges Python-Programm enthält, das eine Zeichenkette druckt.

Wir rufen den Python-Interpreter auf und sagen ihm, dass er den Quellcode aus der Datei `hello.py` lesen soll, anstatt uns interaktiv nach Python-Codezeilen zu fragen.

Man kann feststellen, dass es am Ende des Python-Programms keine Notwendigkeit für `quit()` gibt. Wenn Python den Quellcode aus einer Datei liest, weiß es, dass es aufhören muss, wenn das Ende der Datei erreicht wurde.

Was ist ein Programm?

Die Definition eines *Programms* ist im Grunde genommen eine Abfolge von Python-Anweisungen, die so gestaltet sind, dass sie etwas tun. Selbst unser einfaches Skript `hello.py` ist ein Programm. Es ist ein einzeiliges Programm, das nicht besonders nützlich ist, aber nach der strengsten Definition ist es ein Python-Programm.

Es ist vielleicht am einfachsten zu verstehen, was ein Programm ist, wenn man an ein Problem denkt, für dessen Lösung ein Programm erstellt werden könnte, und dann ein Programm betrachtet, dass dieses Problem lösen würde.

Nehmen wir an, wir forschen im Bereich Social-Computing über Facebook-Posts und interessieren uns für das am häufigsten verwendete Wort in einer Reihe von Beiträgen. Wir könnten den Datenstrom der Facebook-Posts ausgeben und den Text nach dem häufigsten Wort durchsuchen, aber das würde sehr viel Zeit in Anspruch nehmen und wäre sehr fehleranfällig. Es wäre klug, ein Python-Programm zu schreiben, dass diese Aufgabe schnell und genau erledigt, damit wir das Wochenende mit etwas schönerem verbringen können.

Betrachten wir zum Beispiel den folgenden Text über einen Clown und ein Auto. Sehen wir uns den Text an und finden heraus, welches Wort am häufigsten vorkommt und wie oft es vorkommt.

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

Dann stellen wir uns vor, dass wir diese Aufgabe erledigen, indem wir uns Millionen von Zeilen von Text anschauen. Offen gesagt wäre es schneller, Python zu lernen und ein Python-Programm zu schreiben, um die Wörter zu zählen, als sie manuell durchzusehen.

Die gute Nachricht ist, dass bereits ein einfaches Programm entwickelt wurde, um das häufigste Wort in einer Textdatei zu finden. Es wurde schon geschrieben und getestet, damit wir etwas Zeit sparen können.

```
name = input('Welche Datei?:')
handle = open(name, 'r')
counts = dict()
```

```
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in list(counts.items()):
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(f'Das Wort "{bigword}" kommt {bigcount}-mal vor')

# Code: https://tiny.one/py4de/code3/words.py
```

Wir müssen nicht einmal Python beherrschen können, um dieses Programm zu nutzen. Als Endanwender benutzen wir einfach das Programm und freuen uns darüber, wie viel manuelle Arbeit wir gespart haben.

Dies ist ein gutes Beispiel dafür, wie Python und die Python-Sprache als Vermittler zwischen uns (dem Endbenutzer) und den Programmierern fungieren. Python bietet uns die Möglichkeit, nützliche Befehlssequenzen bzw. Programme in einer gemeinsamen Sprache auszutauschen, die von jedem verwendet werden kann, der Python auf seinem Computer installiert. Keiner von uns spricht also *mit Python*, sondern wir kommunizieren miteinander *durch* Python.

Die Bausteine von Programmen

In den nächsten Kapiteln, werden wir mehr über das Vokabular, die Satzstruktur, Absatzstruktur und Erzählstruktur von Python lernen. Uns werden die mächtigen Fähigkeiten von Python näher gebracht werden und wir werden uns aneignen, wie man diese Fähigkeiten zusammensetzen kann, um nützliche Programme zu erstellen.

Es gibt einige konzeptionelle Muster auf niedriger Ebene, die wir zum Erstellen von Programmen benutzen. Diese Konstrukte sind nicht nur für Python-Programme geeignet, sie sind Teil jeder Programmiersprache, von der Maschinensprache bis hin zu den Hochsprachen. Das obige Wortzählprogramm verwendet alle bis auf eines dieser Muster.

Eingabe Abrufen von Daten aus der „Außenwelt“. Dies kann das Lesen von Daten aus einer Datei sein oder sogar aus Sensoren wie einem Mikrofon oder GPS. In unseren ersten Programmen wird die Eingabe durch den Benutzer erfolgen, der Daten über die Tastatur eingibt.

Ausgabe Anzeigen der Ergebnisse des Programms auf einem Bildschirm oder Speichern in einer Datei.

Sequentielle Ausführung Die Ausführung von Anweisungen nacheinander in der Reihenfolge wie sie im Skript vorkommen.

Bedingte Ausführung Prüfung auf bestimmte Bedingungen und anschließende Ausführung oder Überspringen einer Folge von Anweisungen.

Wiederholte Ausführung Wiederholtes Ausführen einer Reihe von Anweisungen, normalerweise mit einer gewissen Variation.

Wiederverwenden Reihe von Anweisungen einmal schreiben und ihnen einen Namen geben. Danach verwenden wir dann diese Anweisungen je nach Bedarf in unserem Programm wieder.

Dass alle Programme fast ausschließlich aus diesen Mustern bestehen, klingt fast zu einfach, um wahr zu sein. Und natürlich ist es nicht ganz so einfach, ein neues Programm zu schreiben. Um einen Vergleich anzustellen, könnte man sagen, dass Gehen einfach „einen Fuß vor den anderen setzen“ bedeutet; und trotzdem braucht ein Kleinkind sehr viel Übung um richtig Laufen zu können. Beim Programmieren ist das Erlernen der Grundmuster recht einfach. Die „Kunst“ ein Programm zu schreiben besteht aber darin, diese Grundelemente immer wieder neu zusammenzusetzen und zu verweben, um eine nützliche Lösung für ein gegebenes Problem zu schaffen.

Was kann schon schief gehen?

Wie wir in unseren ersten „Gesprächen“ mit Python gesehen haben, müssen wir sehr genau sein, wenn wir Python-Code schreiben. Die kleinste Abweichung oder der kleinste Fehler führen dazu, dass Python unser Programm nicht ausführen kann.

```
C:\Python> python.exe
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit
(AMD64)]
Type "help", "copyright", "credits" or "license" for more
information.
>>> primt 'Hello World'
  File "<stdin>", line 1
    primt 'Hello World'
          ^
SyntaxError: invalid syntax
>>> primt('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'primt' is not defined
>>> Come on, Python
  File "<stdin>", line 1
    Come on, Python
         ^
SyntaxError: invalid syntax
>>>
```

Es ist aussichtslos, mit Python zu verhandeln, doch bitte den Code zu verstehen. Der Interpreter kann nur gültige Python Anweisungen ausführen. Beim kleinsten

Fehler wird er abbrechen und ausgeben, an welcher Stelle er beim „Verstehen“ nicht weiter gekommen ist. Wenn Python `SyntaxError: invalid syntax` sagt, bedeutet das: „Ich verstehe einfach nicht was du gemeint hast, aber bitte rede weiter mit mir (`>>>`).“ Ein `NameError` bedeutet, „du scheinst etwas verwenden zu möchten, was ich nicht kenne.“

Wenn die Programme immer anspruchsvoller werden, wird man auf drei allgemeine Arten von Fehlern stoßen:

Syntaxfehler Dies sind die ersten Fehler, die man machen wird, und die am einfachsten zu beheben sind. Ein Syntaxfehler bedeutet, dass wir die „Grammatikregeln“ von Python verletzt haben. Python tut sein Bestes, um genau auf die Zeile und das Zeichen zu zeigen wo es bemerkt hat, dass es verwirrt war. Das einzig tückische an Syntaxfehlern ist, dass der Fehler, der behoben werden muss, manchmal eigentlich früher im Programm liegt als an der Stelle, an der Python *verwirrt* war. Also die Zeile und das Zeichen die Python bei einem Syntaxfehler anzeigt, sind zunächst nur ein Ausgangspunkt für unsere Untersuchungen.

Semantische Fehler Ein semantischer Fehler liegt vor, wenn die Syntax, also die Grammatik des Quellcodes eigentlich korrekt ist, das Programm *so* aber nicht funktioniert. Der `NameError` oben ist ein Beispiel dafür: Es *könnte* eine Funktion namens `primt` existieren und wenn es sie gäbe, wäre das Programm vermutlich korrekt. Es gibt diese Funktion aber im vorliegenden Programm nicht und daher tritt hier bei der Ausführung der Anweisung ein Fehler auf.

Logikfehler Logikfehler sind besonders tückisch, denn sie können von Python so gut wie gar nicht erkannt werden. Es handelt sich um Fehler in der Logik des Programms. Das Programm ist vollkommen korrekt, aber es tut nicht, was wir *beabsichtigt* haben. Ein einfaches Beispiel wäre, wenn wir einer Person den Weg zu einem Restaurant erklären und bei der Wegbeschreibung einmal Links und Rechts verwechseln. Die Person kann daraufhin die Beschreibung nachvollziehen, biegt aber an einer Kreuzung in die völlig falsche Richtung ab und kommt höchstwahrscheinlich niemals an dem Restaurant an.

Bei allen drei Arten von Fehlern versucht Python lediglich genau das zu tun, was wir verlangt haben. Dass Python die Ausführung abbricht, sobald es einen Fehler erkennt, und nicht einfach versucht zu verstehen, was wir mit der fehlerhaften Anweisung *gemeint* haben, wird sich noch als sehr nützlich herausstellen. Sie werden mit jedem Fehler lernen, wie man korrekten Quellcode schreibt. Gleichzeitig können Sie immer davon ausgehen, dass Python genau das tut, was Sie programmiert haben. Es gibt keinen Interpretationsspielraum beim Abarbeiten der Anweisungen; wenn das Programm nicht funktioniert liegt das immer am Programm selbst und nicht daran, dass Python etwas falsch verstanden hat.

Debugging

Wenn Python einen Fehler ausspuckt oder sogar ein Ergebnis liefert, dass sich von dem unterscheidet, was wir beabsichtigt hatten, dann beginnt die Suche nach der Fehlerursache. Beim Debugging geht es darum, die Ursache des Fehlers in unserem

Code zu finden. Bei der Fehlersuche in einem Programm, insbesondere wenn wir an einem schwerwiegenden Fehler arbeiten, gibt es vier Dinge, die wir versuchen sollten:

Lesen Nochmal den Code durch erneutes Lesen prüfen. Haben wir alles so gesagt, wie wir es sagen wollten?

Ausführen Man sollte einfach nochmal experimentieren, indem man nachverfolgbare Änderungen vornimmt und verschiedene Versionen ausführt. Häufig wir das Problem offensichtlich, wenn wir es an der richtigen Stelle im Programm anzeigen lassen. Manchmal muss man etwas Zeit aufwenden, um ein gutes Gerüst zu bauen.

Nachdenken Man sollte sich Zeit zum Nachdenken nehmen. Was für ein Fehler ist es? Handelt es sich um einen Syntaxfehler, einen Laufzeitfehler oder einen semantischen Fehler? Welche Informationen können wir aus den Fehlermeldungen bzw. der Ausgabe des Programms entnehmen? Welche Art von Fehler könnte die Ursache sein für das Problem, das man sehen kann? Was haben wir zuletzt geändert, bevor das Problem auftrat?

Rückzug Irgendwann ist es am besten sich zurückzuziehen und die letzten Änderungen rückgängig zu machen, bis man wieder ein Programm hat, das funktioniert und das man verstehen kann. Dann kann man mit dem Neuaufbau beginnen.

Programmieranfänger bleiben manchmal bei einer dieser Aktivitäten stecken und vergessen die anderen. Um einen Fehler zu finden, muss man lesen, experimentieren, grübeln und sich manchmal zurückziehen. Wenn man bei einer dieser Aktivitäten nicht weiter kommt, versucht man die anderen. Jede Aktivität hat ihren eigenen Fehlermodus.

Das Lesen des Codes kann zum Beispiel helfen, wenn das Problem ein Tippfehler ist, aber nicht, wenn es sich um ein konzeptionelles Missverständnis handelt. Wenn man nicht versteht, was das Programm tut, kann man es 100 Mal lesen, ohne den Fehler zu sehen.

Experimentieren kann hilfreich sein, vor allem, wenn man kleine, einfache Tests durchführt. Wenn man jedoch Experimente durchführt, ohne dabei nachzudenken oder den Code genau zu lesen, könnte man in ein Muster verfallen, dass man als „Zufallsprogrammierung“ bezeichnen könnte. Dabei handelt es sich um einen Prozess, bei dem zufällige Änderungen vorgenommen werden, bis das Programm (vielleicht) irgendwann das Richtige tut. Es ist unnötig zu erwähnen, dass diese Art der Programmierung sehr lange dauern kann.

Man muss sich Zeit zum Nachdenken nehmen. Fehlersuche ist wie eine experimentelle Wissenschaft. Man sollte mindestens eine Hypothese darüber haben, was das Problem ist. Wenn es zwei oder mehr Möglichkeiten gibt, versucht man, einen Test zu finden, der eine von ihnen ausschließt.

Eine Pause hilft beim Nachdenken. Das gilt auch für das Reden. Wenn man das Problem einer anderen Person (oder sogar sich selbst) erklärt, findet man manchmal die Antwort, bevor man die Frage zu Ende gestellt hat.

Aber selbst die besten Debugging-Techniken versagen, wenn es zu viele Fehler gibt, oder wenn der Code, den wir zu beheben versuchen, zu groß und kompliziert ist.

Manchmal ist es am besten, sich zurückzuziehen und das Programm zu vereinfachen, bis man zu einem Ergebnis gelangt, welches funktioniert und man versteht.

Programmieranfänger zögern oft, sich zurückzuziehen, weil sie es nicht ertragen können eine Codezeile zu löschen (selbst wenn sie falsch ist). Wenn man sich dadurch besser fühlt, sollte man einfach das Programm in eine andere Datei sichern, bevor man es zerlegt. Dann kann man die Teile nach und nach wieder Stück für Stück einfügen.

Dass Sie größere Teile Ihres Programms löschen müssen, können Sie meist auch ganz gut verhindern, indem Sie Ihren Lösungsweg für die gesamte Aufgabe in kleinere Teilaufgaben einteilen. Überlegen Sie sich bevor Sie mit dem Schreiben des Codes anfangen, welche logischen Teilfunktionen in Ihrer Aufgabe stecken. Und dann beginnen Sie, die erste Teilaufgabe zu lösen und – ganz wichtig – zu testen. Wenn Sie sich einigermaßen sicher sind, dass die erste Teilaufgabe gelöst ist, nehmen Sie sich die zweite vor, usw. Diese Vorgehensweise hilft sehr, mögliche Fehlerursachen einzukreisen. Wenn Sie stattdessen zu viel „auf einen Schlag“ programmieren, können mehr Fehler auftreten und die Fehlersuche sowie die Beseitigung dauern viel länger.

Der Lernprozess

Im weiteren Verlauf des Buches sollte man keine Angst haben, wenn die Konzepte beim ersten Mal nicht gut zusammenzupassen scheinen. Als wir sprechen gelernt haben, war es in den ersten Jahren kein Problem, dass man nur niedliche Gluckgeräusche gemacht hat. Es war in Ordnung, wenn es sechs Monate dauerte, um von einfachen Vokabeln zu einfachen Sätzen zu kommen, und 5–6 Jahre brauchte, um von Sätzen zu Absätzen zu kommen, und noch ein paar Jahre, um in der Lage zu sein, selbständig eine interessante, vollständige Kurzgeschichte zu schreiben.

Wir wollen, dass wir Python viel schneller lernen, deshalb lernen wir in den nächsten Kapiteln alles gleichzeitig. Aber es ist wie beim Erlernen einer neuen Sprache, die man sich erst aneignen und verstehen muss, bevor sie sich natürlich anfühlt. Das führt zu einiger Verwirrung, wenn wir Themen lernen und wieder aufgreifen, um das große Ganze zu sehen, während wir die winzigen Fragmente definieren, die dieses große Bild ausmachen. Das Buch ist zwar linear geschrieben und wenn man einen Kurs belegt, wird er auch linear verlaufen. Man sollte jedoch nicht zögern, sich dem Stoff sehr unlinear zu nähern. Egal ob vorwärts und rückwärts oder quer gelesen. Durch Überfliegen von fortgeschrittenem Material, ohne die Details vollständig zu verstehen, erhält man ein besseres Verständnis für das „Warum?“ der Programmierung. Durch Wiederholung früherer Inhalte und sogar Wiederholung früherer Übungen werden wir feststellen, dass wir tatsächlich viel gelernt haben, auch wenn das Material, welches wir gerade anstarren, ein wenig undurchdringlich erscheint.

Wenn wir unsere erste Programmiersprache lernen, gibt es normalerweise ein paar wunderbare „Aha!“-Momente. Dinge, für die Sie vorher Stunden oder Tage benötigt hätten, können automatisiert in Bruchteilen von Sekunden ausgeführt werden. Dazu gehört aber auch, dass man manchmal beim Lernen der Programmiersprache Aufgaben zu lösen hat, die nicht wirklich ein Problem für einen selbst darstellen.

Nehmen Sie solche Aufgaben als „Fingerübungen“ hin, die Ihnen helfen, Routine beim Programmieren zu entwickeln.

Glossar

Bug Ein Fehler im Programmcode.

Central Processing Unit Das Herz eines jeden Computers. Auf ihm läuft die Software, die wir schreiben; es wird auch *CPU* oder *Prozessor* genannt.

Kompilieren Übersetzung eines in einer Hochsprache geschriebenen Programms in eine niedrigere Sprache, um es später ausführen zu können.

Hochsprache Eine Programmiersprache wie Python, die so konzipiert wurde, dass sie für Menschen leicht zu lesen und zu schreiben ist.

Interaktiver Modus Eine Methode zur Verwendung des Python-Interpreters durch Eingabe von Befehlen und Ausdrücken in der Eingabeaufforderung.

Interpretieren Ausführen eines Programms in einer Hochsprache durch zeilenweises Übersetzen.

Low-Level-Sprache Eine Programmiersprache, die so konzipiert ist, dass sie von einem Computer leicht ausgeführt werden kann. Auch *Maschinencode* oder *Assemblersprache* genannt.

Maschinencode Die niedrigste Sprache für Software, also die Sprache, die direkt von der zentralen Recheneinheit (CPU) ausgeführt wird.

Hauptspeicher Speichert Programme und Daten. Der Hauptspeicher verliert seine Informationen, wenn der Strom ausgeschaltet wird.

Parsen Ein Programm untersuchen und die syntaktische Struktur analysieren.

Portabilität Eine Eigenschaft eines Programms, das auf mehr als einer Art von Endgerät laufen kann.

Print-Funktion Eine Anweisung, die den Python-Interpreter veranlasst, einen Wert auf dem Bildschirm anzuzeigen.

Problemlösungsprozess Der Prozess ein Problem zu formulieren, eine Lösung zu finden und die Lösung umzusetzen.

Programm Ein Satz von Anweisungen, der eine Berechnung vorgibt.

Eingabeaufforderung Wenn ein Programm eine Meldung anzeigt und eine Pause macht, damit der Benutzer eine Eingabe im Programm machen kann.

Sekundärspeicher Speichert Programme und Daten und behält die Informationen auch dann bei, wenn der Strom abgeschaltet wird. Im Allgemeinen langsamer als der Hauptspeicher. Beispiele für Sekundärspeicher sind z. B. Festplattenlaufwerke und Flash-Speicher in USB-Sticks.

Semantik Die Bedeutung eines Programms.

Semantischer Fehler Ein Fehler in einem Programm, der dazu führt, dass es etwas anderes tut, als der Programmierer beabsichtigt hat.

Quellcode Der Programmcode eines in einer Hochsprache geschriebenen Programms.

Übungen

Übung 1: Welche Funktion hat der Sekundärspeicher in einem Computer?

a) Alle Berechnungen und die Logik des Programms ausführen
b) Abruf von Webseiten über das Internet
c) Informationen langfristig zu speichern, auch über einen Stromausfall hinaus
d) Eingaben des Benutzers entgegennehmen

Übung 2: Was ist ein Programm?

Übung 3: Was ist der Unterschied zwischen einem Compiler und einem Interpreter?

Übung 4: Was enthält Maschinencode?

a) Der Python-Interpreter
b) Eine Python-Quelldatei
c) Ein Textdokument

Übung 5: Was ist falsch an folgendem Code?

```
>>> primt 'Hello world!'
File "<stdin>", line 1
  primt 'Hello world!'
                ^
SyntaxError: invalid syntax
>>>
```

Übung 6: Wo im Computer wird eine Variable wie `x` gespeichert, nachdem die folgende Python-Zeile beendet ist?

```
x = 123
```

a) CPU
b) Hauptspeicher
c) Sekundärspeicher
d) Eingabegeräte
e) Ausgabegeräte

Übung 7: Was wird das folgende Programm ausgeben:

```
x = 43
x = x + 1
print(x)
```

a) 43
b) 44
c) x + 1
d) Fehler, denn x = x + 1 ist mathematisch nicht möglich

Übung 8: Erläutern Sie die folgenden Punkte anhand eines Beispiels für eine menschliche Fähigkeit: (1) CPU, (2) Hauptspeicher, (3) Sekundärspeicher, (4) Eingabegerät und (5) Ausgabegerät. Zum Beispiel: „Was ist das menschliche Äquivalent zu einer CPU?“

Übung 9: Wie behebt man einen Syntaxfehler?

Kapitel 2

Bezeichner, Ausdrücke und Anweisungen

In diesem Kapitel schauen wir uns an, wie man *Werte* im Programm verwenden kann. Wir werden sehen, dass alles, was wir in Python zu einem Wert *auswerten* können, ein Ausdruck ist. Dazu zählen fest im Programm angegebene Werte, die sogenannten *Konstanten*, aber auch komplexere Ausdrücke, bei denen mehrere Teile durch *Operatoren* verknüpft sind. Wir werden sehen, dass wir mit einer *Zuweisung* einzelnen Werten einen Namen geben können und diese Werte im Programm dann über Ihren *Bezeichner* verwenden können.

Werte und Datentypen

Ein *Wert* ist eines der grundlegenden Elemente, mit denen ein Programm arbeitet, wie ein Buchstabe oder eine Zahl. Die Werte, die wir bis jetzt gesehen haben, sind `1`, `2` und `'Hallo Welt!'`.

Diese Werte gehören zu unterschiedlichen *Datentypen*: `2` ist eine Ganzzahl, und `'Hallo Welt!'` ist eine Zeichenkette (englisch *String*), so genannt, weil sie eine „Kette“ von Buchstaben enthält. Wir (und der Python-Interpreter) können Zeichenketten erkennen, weil sie in Anführungszeichen eingeschlossen sind.

Die Anweisung `print` funktioniert auch für Ganzzahlen. Wir verwenden den Befehl `python`, um den Interpreter zu starten.

```
python
>>> print(4)
4
```

Wenn man nicht sicher sind, welchen Typ ein Wert hat, kann der Interpreter es einem sagen.

```
>>> type('Hello, World!')
<class 'str'>
>>> type(17)
<class 'int'>
```

Es überrascht nicht, dass Zeichenketten zum Typ `str` und Ganzzahlen zum Typ `int` gehören. Weniger offensichtlich gehören Zahlen mit einem Dezimalpunkt zu einem Typ namens `float`, da diese Zahlen in einem Format namens *Fließkomma* (oder *Gleitkommazahl*, englisch floating point number) dargestellt werden.

Beim *Fließkomma* müssen Sie beachten, einen *Dezimalpunkt* zu verwenden. Python orientiert sich hier wie die meisten Programmiersprachen an dem Dezimaltrennzeichen, das in den meisten englischsprachigen Ländern verwendet wird; und das ist eben der Punkt und nicht das Komma, so wie wir es in Deutschland verwenden.

```
>>> type(3.2)
<class 'float'>
```

Was ist mit Werten wie `17` und `3.2`? Sie sehen aus wie Zahlen, stehen aber in Anführungszeichen wie Zeichenketten.

```
>>> type('17')
<class 'str'>
>>> type('3.2')
<class 'str'>
```

Es sind Zeichenketten.

Wenn man eine große Ganzzahl eingibt, könnte man versucht sein, Tausender-Trennzeichen zu verwenden. In Ländern, die den Dezimalpunkt verwenden, werden zum Abgrenzen der Tausenderstellen Kommata benutzt. Wenn wir dies in Python tun, also etwa eine Million als `1,000,000` schreiben ist das keine gültige Ganzzahl mehr. Allerdings ist `1,000,000` ein gültiger Python-Ausdruck, wie wir in folgendem Beispiel sehen:

```
>>> print(1,000,000)
1 0 0
```

Nun, das ist überhaupt nicht das, was wir erwartet haben! Python interpretiert `1,000,000` als eine durch Kommata getrennte Folge von ganzen Zahlen, die es mit Leerzeichen dazwischen ausgibt.

Dies ist ein Beispiel für einen *semantischen Fehler*: Der Code wird ohne eine Fehlermeldung zu erzeugen ausgeführt, aber er tut nicht das „Richtige“. Wenn wir die `1,000,000` als *eine* Zahl verwenden würden, würde der Interpreter auch einen Fehler erzeugen, da wir hier keine einzelne, sondern eine Folge von Zahlen haben.

Leider, oder glücklicherweise – das liegt ganz im Auge des Betrachters – ist Python sehr „großzügig“ bei der Interpretation Ihres Codes. Wenn Sie `1,000,000` hinschreiben, wird Python annehmen, dass Sie genau eine solche Folge meinen. Wenn Sie `a = 1,000,000` hinschreiben, wird Python dieser Folge den Namen `a` geben. Dass das möglicherweise nicht das ist, was Sie meinten, kann Python nicht ahnen. Ein weiteres Beispiel dafür, dass Sie beim Programmieren *genau* sein müssen.

Werte benennen

Die zentrale Eigenschaft einer Programmiersprache ist die Fähigkeit, *Daten* zu manipulieren. In Python sind alle Daten als sogenannte *Objekte* abgespeichert. Übrigens sind auch die einzelnen Funktionen *Objekte* und Objekte selbst sind *Objekte*, weswegen der Leitsatz gilt: *Alles in Python ist ein Objekt.* Zu verwirrend? Ich kann Sie gut verstehen, wir werden uns dieses Thema später noch einmal vornehmen, dann wird sicher klarer, was hier mit *Objekt* gemeint ist.

Unsere Objekte, also z. B. Ganzzahlen oder Zeichenketten sind im Hauptspeicher abgelegt. Um Sie sinnvoll verwenden zu können, geben wir Ihnen Namen. Ein solches Paar, also einen Wert (bzw. ein Objekt) und einen dazugehörigen Namen nennt man auch *Variable*. Wir werden im Folgenden den Begriff *Variable* synonym für den Namen verwenden. Lassen Sie sich davon nicht verwirren. Wenn wir von „Variable“ sprechen, ist immer der Name eines Wert-Objekts gemeint.

Die Bindung eines Namens an einen Wert erfolgt durch eine *Zuweisung*. Existiert der Name vorher noch nicht, wird er im Zuge der Zuweisung neu erzeugt:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

In diesem Beispiel werden drei Zuweisungen vorgenommen. Die erste weist eine Zeichenkette einer neuen Variablen namens `message` zu; die zweite weist die ganze Zahl `17` der Variablen `n` zu; die dritte weist den (ungefähren) Wert von π `pi` zu.

Um den Wert einer Variablen anzuzeigen, können Sie eine `print`-Anweisung verwenden:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

Der Typ einer Variablen ist der Typ des Wertes, auf den sie sich bezieht.

```
>>> type(message)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

Bezeichner und Schlüsselwörter

Programmierer wählen für ihre Variablen in der Regel Namen, die aussagekräftig sind und dokumentieren, wofür die Variable verwendet wird.

Variablennamen können beliebig lang sein. Sie können sowohl Buchstaben als auch Zahlen enthalten, aber sie dürfen nicht mit einer Zahl beginnen. Es ist erlaubt, Großbuchstaben zu verwenden, aber es ist eine gute Idee, Variablennamen mit einem Kleinbuchstaben zu beginnen (Wir werden später sehen, warum).

Der Unterstrich (`_`) kann in einem Variablennamen vorkommen. Er wird häufig in Variablen mit mehreren Wörtern verwendet, z. B. `my_name` oder `speed_of_light`. Variablennamen können mit einem Unterstrich beginnen, aber wir vermeiden dies im Allgemeinen. Variablen die mit einem oder mit zwei Unterstrichen beginnen, haben eine spezielle Bedeutung. Wenn wir also „normale“ Variablen mit einem Unterstrich beginnen lassen, würde dies Personen verwirren, die unseren Code lesen und nachvollziehen wollen.

Wenn wir einer Variablen einen unzulässigen Namen geben, erhalten wir einen Syntaxfehler:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

Der Variablenname `76trombones` ist unzulässig, weil er mit einer Zahl beginnt. `more@` ist unzulässig, weil es ein unerlaubtes Zeichen (`@`) enthält. Aber was ist falsch an `class`?

Es stellt sich heraus, dass `class` eines von Pythons *Schlüsselwörtern* (englisch *Keyword*) ist. Der Interpreter verwendet Schlüsselwörter, um die Struktur des Programms zu erkennen, und sie können nicht als Variablennamen verwendet werden.

Python reserviert 35 Schlüsselwörter:

```
and         as          assert      async       await       break
class       continue    def         del         elif        else
except      False       finally     for         from        global
if          import      in          is          lambda      None
nonlocal    not         or          pass        raise       return
True        try         while       with        yield
```

Man sollte diese Liste griffbereit halten. Wenn sich der Interpreter über einen der Variablennamen beschwert und wir nicht wissen, warum, sehen wir nach, ob er auf dieser Liste steht.

Anweisungen

Eine *Anweisung* ist eine Einheit von Code, die der Python-Interpreter ausführen kann. Wir haben zwei Arten von Anweisungen gesehen: Ausdrücke wie `print(n)` und Zuweisungen wie `n = 17`.

Wenn wir eine Anweisung im interaktiven Modus eingeben, führt der Interpreter sie aus und zeigt das Ergebnis an, falls es eines gibt.

Ein Skript enthält normalerweise eine Folge von Anweisungen. Wenn es mehr als eine Anweisung gibt, erscheinen die Ergebnisse nacheinander, während die Anweisungen ausgeführt werden.

Zum Beispiel erzeugt das Skript

```
print(1)
x = 2
print(x)
```

folgende Ausgabe:

```
1
2
```

Die Zuweisung erzeugt keine Ausgabe.

Operatoren und Operanden

Operatoren sind spezielle Symbole, die Berechnungen wie Addition und Multiplikation darstellen. Die Werte, auf die der Operator angewendet wird, werden *Operanden* genannt.

Die Operatoren `+`, `-`, `*`, `/` und `**` führen Addition, Subtraktion, Multiplikation, Division und Potenzierung aus, wie in den folgenden Beispielen:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

Später werden wir sehen, dass diese Operatoren noch zu ganz anderen Zwecken verwendet werden können. Solange man die *mathematischen Operatoren* allerdings auf Ganz- oder Fließkommazahlen (bzw. Variablen, die auf solche Werte verweisen) anwendet, werden auch die normalen mathematischen Operationen berechnet.

Im Normalfall kommt bei einer mathematischen Operation auf Ganz- oder Fließkommazahlen wieder der gleiche Typ heraus. Die Summe zweier Ganzzahlen ist wieder eine ganze Zahl, die Multiplikation zweier Fließkommazahlen ist wieder eine Fließkommazahl, usw. Eine Ausnahme bildet hier die Division. Teilt man eine ganze Zahl durch eine weitere ganze Zahl, kann natürlich ein Bruch herauskommen, den wir im Computer als Fließkommazahl speichern. Es ändert sich also der Typ des Ergebnisses.

Wir können die Division aber auch als *ganzzahlig* betrachten, also nur den ganzzahligen Teil des Ergebnisses übernehmen. Auch diese Operation braucht man beim Schreiben von Programme recht häufig. Wenn es aber zwei verschiedenen Arten zu dividieren gibt, stellt sich die Frage, was nun der Divisionsoperator `/` eigentlich bewirkt:

```
>>> minute = 59
>>> minute/60
0.9833333333333333
```

Sie können sich einfach merken, dass der normale Divisionsoperator `/` eine Fließkommazahl, also das genaue Ergebnis liefert. Allerdings ist dies nur die „halbe Wahrheit“ und an dieser Stelle wird es etwas hakelig: Python gibt es nämlich in verschiedenen Versionen. Momentan ist die Version 3.9 aktuell und wenn Sie heute mit dem Python-Programmieren beginnen, werden Sie sicher mit *Python 3* (also einer Version 3.x) arbeiten. In der vorherigen Version *Python 2* (also mit einer Versionsnummer 2.x) war der Großteil der Sprache identisch. Mit dem Sprung von 2 auf 3 hat man aber einige Eigenschaften und Regeln geändert, hauptsächlich um die Sprache klarer zu machen und Ungereimtheiten zu beseitigen. Eine dieser Änderungen betrifft den Divisionsoperator.

Die obige Aussage, dass die Division mit `/` eine Fließkommazahl liefert, ist also erst seit Python 3.0 korrekt. Der Divisionsoperator in Python 2.0 würde zwei Ganzzahlen dividieren und als Ergebnis den Anteil vor dem Komma verwenden:

```
>>> minute = 59
>>> minute/60
0
```

Um das gleiche Ergebnis in Python 3.0 zu erhalten, verwenden wir die ganzzahlige Division (`//` Ganzzahl).

```
>>> minute = 59
>>> minute//60
0
```

In Python 3.0 funktioniert die ganzzahlige Division also eher so, wie wir es erwarten würden, wenn wir den Ausdruck in einen Taschenrechner eingeben würden.

Es gibt noch einige weitere Unterschiede zwischen Python 2.0 und 3.0. Wenn man mit dem Programmieren beginnt, ist es allerdings müßig, sich gleich zu Beginn diese Unterschiede im Detail anzusehen. Nehmen Sie also aus den vorherigen Beispielen mit, dass sich eine Programmiersprache von Version zu Version weiterentwickeln kann und sich dabei auch die Bedeutung von bestimmten Anweisungen oder Ausdrücken ändern kann. Alles, was Sie in diesem Buch lernen, bezieht sich auf die Version 3 von Python. Nachdem Sie Python 3 gelernt haben, werden Sie aber auch Programme, die in Python 2 geschrieben wurden, ohne weiteres verstehen können. Die Unterschiede sind wirklich marginal.

Ausdrücke

Ein *Ausdruck* ist eine Kombination aus Werten, Variablen und Operatoren, die wiederum zu einem Wert abgeleitet werden kann. Ein Wert allein wird als Ausdruck betrachtet, ebenso wie eine Variable, sodass die folgenden Ausdrücke alle zulässig sind (unter der Annahme, dass der Variablen `x` ein Wert zugewiesen wurde):

```
17
x
x + 17
```

Wenn wir einen Ausdruck im interaktiven Modus eingeben, wird er vom Interpreter *ausgewertet* und das Ergebnis angezeigt:

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

Aber in einem Skript bewirkt ein Ausdruck für sich allein gar nichts! Dies ist eine häufige Ursache für Verwirrung bei Programmieranfängern.

Übung 1: Geben Sie die folgenden Anweisungen in den Python-Interpreter ein, um zu sehen, was sie bewirken:

```
5
x = 5
x + 1
```

Reihenfolge der Auswertung

Wenn mehr als ein Operator in einem Ausdruck vorkommt, hängt die Reihenfolge der Auswertung von den *Vorrangregeln* ab. Für mathematische Operatoren folgt Python der mathematischen Konvention:

- **Klammern** haben den höchsten Vorrang und können verwendet werden, um die Auswertung eines Ausdrucks in der gewünschten Reihenfolge zu erzwingen. Da Ausdrücke in Klammern zuerst ausgewertet werden, ist `2 * (3-1)` gleich 4 und `(1+1)**(5-2)` gleich 8. Wir können Klammern auch verwenden, um einen Ausdruck leichter lesbar zu machen, wie in `(Minute * 100) / 60`, auch wenn dies das Ergebnis nicht ändert.
- **Potenzrechnung** hat die nächsthöhere Priorität, also ist `2**1+1` gleich 3, nicht 4, und `3*1**3` ist 3, nicht 27.
- **Multiplikation** und **Division** haben den gleichen Vorrang, der höher ist als der von **Addition** und **Subtraktion**, die ebenfalls den gleichen Vorrang haben. Also ist `2*3-1` gleich 5, nicht 4, und `6+4/2` ist 8, nicht 5.

- Operatoren mit der gleichen Rangfolge werden normalerweise **von links nach rechts** ausgewertet. Der Ausdruck `5-3-1` ist also 1, nicht 3, weil zuerst `5-3` ausgeführt wird und dann `1` von 2 subtrahiert wird. Eine Ausnahme ist die Potenzrechnung. Der Ausdruck 4^{3^2} ist in Python übersetzt `4**3**2` und ergibt 262144 (und nicht 4096). Der Potenz-Operator `**` wird also **von rechts nach links** ausgewertet.

Wenn man sich nicht sicher ist, in welcher Reihenfolge ein Ausdruck ausgewertet wird, ist es sehr empfehlenswert, Klammern zu setzen. Überflüssige Klammern verfälschen den Wert des Ausdrucks nicht, helfen Ihnen aber dabei, genau die Berechnungsreihenfolge festzulegen, die Sie anwenden möchten.

Division mit Rest

Wir haben oben gesehen, dass man in Python eine ganzzahlige Division mit dem Operator `//` durchführen kann. Häufig ist man aber bei einer ganzzahligen Division gar nicht an dem eigentlichen Wert interessiert, sondern eher an dem *Rest*, der sich bei einer solchen Division ergibt. Auch hierfür gibt es einen Operator in Python.

Der *Modulo-Operator* arbeitet mit ganzen Zahlen und liefert den Rest, wenn der erste Operand durch den zweiten geteilt wird. In Python ist der Modulo-Operator ein Prozentzeichen (`%`). Die Syntax ist die gleiche wie bei den anderen Operatoren:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

Also 7 geteilt durch 3 ist 2 und der Rest 1 bleibt.

Der Modulo-Operator erweist sich als erstaunlich nützlich. Man kann zum Beispiel prüfen, ob eine Zahl durch eine andere teilbar ist: Wenn `x % y` Null ist, dann ist `x` durch `y` teilbar.

Man kann die Modulo-Operation auch verwenden, um auf die einzelnen Ziffern einer Zahl zuzugreifen. Nehmen wir an, in unserem Programm gibt es eine (größere) Zahl `x`. Die Berechnung `x % 10` liefert uns die äußerste rechte Ziffer von `x`. In gleicher Weise ergibt `x % 100` die letzten beiden Ziffern.

```
>>> 4**3**2
262144
>>> x = 1234
>>> x % 10
4
>>> x = x // 10
>>> x % 10
3
```

Operationen mit Zeichenketten

Bisher haben wir alle Operatoren lediglich auf Zahlen angewendet oder auf Variablen, unter denen wir Zahlen abgelegt haben. Erstaunlicherweise lassen sich viele dieser Operatoren aber auch auf Objekte anderer Datentypen anwenden. Der Operator `+` zum Beispiel hat auch eine Bedeutung für Zeichenketten, aber natürlich ist es dann keine Addition im mathematischen Sinne mehr. Stattdessen führt er eine *Konkatenation* (Verkettung) durch, d. h. er verbindet die Zeichenketten, indem er sie Ende an Ende verknüpft. Zum Beispiel:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
>>> second = '150'
>>> print(first + second)
100150
>>> print('Hallo' + 'Welt')
HalloWelt
```

Sogar der Operator `*` arbeitet mit Zeichenketten. Mit ihm kann der Inhalt einer Zeichenkette mithilfe einer ganzen Zahl vervielfältigt werden. Zum Beispiel:

```
>>> first = 'Test '
>>> second = 3
>>> print(first * second)
Test Test Test
```

Zuweisungen

Zuweisungen an Namen haben wir ja bereits oben kennengelernt. Dabei haben wir aber meist nur einen konkreten Wert (eine sogenannte *Konstante*) an den Namen gebunden. Häufig verwendet man aber auf der rechten Seite des Gleichheitszeichens (auf der linken steht ja der Name, den wir verwenden möchten) ein zusammengesetzter Ausdruck. Man kann sich einfach merken, dass dieser Ausdruck auf der rechten Seite *vor* der Zuweisung an den Namen komplett ausgewertet wird. Warum ist es wichtig, das zu wissen? Schauen wir uns folgendes Beispiel an:

```
>>> x = 42
>>> x = x + 1
>>> x
43
```

Wir sehen, dass auf der rechten und linken Seite des Gleichheitszeichens hier ein `x` steht. Da Python zuerst die rechte Seite betrachtet, wird aus dem `x + 1` ein `42 + 1`, was im Folgenden zu `43` ausgewertet wird. Dieser Wert wird dann an den Namen `x` gebunden. Nach dieser Zuweisung bezeichnet `x` also den Wert 43.

Typen

Wir haben bis hierher schon mehrfach erwähnt, dass Werte – bzw. allgemeiner gesehen *Objekte* – verschiedene Datentypen besitzen können. Eine `42` ist eine Ganzzahl (Typ `int` für *Integer*), `3.1415` ist eine Fließkommazahl (Typ `float`) und `Hallo` ist eine Zeichenkette (Typ `str` für *String*).

Bisher haben wir Werte dieser Datentypen verwendet, ohne jemals genau festzulegen, dass unsere Werte diese Typen besitzen sollen. Python hat die Typen der Werte immer selbst erkannt; dieses Prinzip nennt man auch *Typinferenz* (englisch *Type Inference*). Nicht alle Programmiersprachen machen das so. In C oder Java (vor Version 10) muss der Datentyp einer neuen Variablen immer explizit angegeben werden.

Wenn man in Python einen Namen an einen Wert bindet, analysiert der Python-Interpreter den Wert des Ausdrucks. Im folgenden Beispiel addieren wir eine ganze Zahl und eine Fließkommazahl. Python muss nun einen Typ für das Ergebnis festlegen und entscheidet sich in diesem Fall für `float`. Das ist auch sicher eine gute Wahl, denn damit wird das Ergebnis *genau* erfasst.

```
>>> a = 42
>>> b = 3.1415
>>> c = a + b
>>> type(c)
<class 'float'>
```

Benutzereingaben

Manchmal möchten wir den Wert für eine Variable über eine Tastatureingabe einlesen. Python bietet eine eingebaute Funktion namens `input`, die Eingaben von der Tastatur entgegennimmt. Wenn diese Funktion aufgerufen wird, hält das Programm an und wartet darauf, dass der Benutzer etwas eingibt. Wenn der Benutzer die Enter-Taste drückt, wird das Programm fortgesetzt und `input` gibt zurück, was der Benutzer als Zeichenkette getippt hat.

```
>>> inp = input()
Was willst du von mir?
>>> print(inp)
Was willst du von mir?
```

Bevor man eine Eingabe vom Benutzer erhält, ist es eine gute Idee, eine Eingabeaufforderung (Prompt) auszugeben, die dem Benutzer mitteilt, was er eingeben soll. Man kann eine Zeichenkette an `input` übergeben, die dem Benutzer angezeigt wird, bevor er zur Eingabe aufgefordert wird:

```
>>> name = input('Wie heisst du?\n')
Wie heisst du?
```

```
Heiner
>>> print(name)
Heiner
```

Die Sequenz `\n` am Ende des Prompt stellt ein *Newline* dar, ein Sonderzeichen, das einen Zeilenumbruch bewirkt. Deshalb erscheint die Eingabe des Benutzers unterhalb des Prompts. Ohne dieses Sonderzeichen würde die Benutzereingabe direkt nach dem `?` beginnen.

Wir sehen, dass `input(...)` ein Ausdruck ist, der ausgewertet wird und demnach auch einen Wert zurückliefert. Da die Eingabe von der Tastatur des Benutzers kommt, wird sie immer als Zeichenkette aufgefasst. Nun stellt sich die Frage, was passiert, wenn wir eine Zahl eingeben. Wie im ersten Kapitel erwähnt, wird Python niemals *raten*, was wir bei der Eingabe meinen. Die Zeichenfolge `123` bleibt eine Zeichenfolge, auch wenn man sie als Zahl interpretieren könnte.

Als Programmierer wissen wir aber meistens, welche Art von Eingabe wir erwarten. Wenn wir also den Benutzer bitten, eine Zahl einzugeben, können wir die resultierende Zeichenkette als ganze Zahl interpretieren. Das geht mit der Typumwandlungsfunktion `int()`, welche die Zeichenkette in einen `int` umwandelt:

```
>>> prompt = 'Was ist die Fluggeschwindigkeit einer Schwalbe?\n'
>>> speed = input(prompt)
Was ist die Fluggeschwindigkeit einer Schwalbe?
17
>>> int(speed)
17
```

Wenn der Benutzer jedoch etwas anderes als eine Ziffernfolge eingibt, erhalten wir einen Fehler:

```
>>> speed = input(prompt)
Was ist die Fluggeschwindigkeit einer Schwalbe?
Meinst du eine europäische oder eine afrikanische Schwalbe?
>>> int(speed)
ValueError: invalid literal for int() with base 10: ...
```

Auch wenn das Beispiel hier ein wenig schräg erscheint, sind fehlerhafte Benutzereingaben keine Seltenheit. Eingaben von der Tastatur, aus Dateien oder auch aus dem Internet sind häufig nicht genau so, wie Ihr Programm es vielleicht erwartet. Und in solchen Fällen bricht Python, ohne dass Sie den Fehler vorhersehen und in irgendeiner Form *behandeln*, die Abarbeitung des Programms ab. Wir werden später sehen, wie diese Art von *Fehlerbehandlung* zu realisieren ist.

Übrigens: Einige der im Buch verwendeten Beispiele, so wie das mit der Schwalbe oben, sind an Dialoge oder Szenen aus Filmen und Sketchen der Komikergruppe *Monty Python* angelehnt. Der Erfinder von Python, Guido Van Rossum, ist ein Fan dieser Gruppe und hat sich bei der Namensfindung für seine Programmiersprache von ihnen inspirieren lassen. Python hat also nichts mit Schlangen zu tun.

Kommentare

Je größer und komplizierter Programme werden, desto schwieriger werden sie zu lesen. Formale Sprachen sind komplex, und es ist oft schwierig, einen Teil des Codes zu betrachten und herauszufinden, was er tut oder warum. Selbst bei den kleinen Programmbeispielen in diesem Kapitel wäre es praktisch gewesen, wenn wir Erklärungen für einzelne Zeilen des Quellcodes direkt an Ort und Stelle in den Code geschrieben hätten. Ohne Weiteres hätte das aber nicht funktioniert, denn Erklärungen, also „normaler Text“, sind kein gültiger Python Code. Der Interpreter hätte sich also über unseren Text beschwert.

Um dennoch die Möglichkeit zu haben, Notizen zu unseren Programmen hinzuzufügen, um in natürlicher Sprache zu erklären, was das Programm tut, verwenden wir *Kommentare*. In Python werden Kommentare mit dem Symbol `#` eingeleitet. Alles, was nach dem `#`-Zeichen (und bis zum Zeilenende) steht, wird vom Python-Interpreter einfach ignoriert und kann folglich auch nicht zu einem Fehler führen.

```
# Berechne die in dieser Stunde abgelaufene Zeit in Prozent
percentage = (minute * 100) / 60
```

In diesem Fall erscheint der Kommentar in einer eigenen Zeile. Wir können Kommentare auch als Zeilenendkommentar an das Ende einer Zeile setzen:

```
percentage = (minute * 100) / 60 # Abgelaufene Minuten in Prozent
```

Kommentare sind am nützlichsten, wenn sie nicht offensichtliche Merkmale des Codes dokumentieren. Es ist vernünftig anzunehmen, dass der Leser herausfinden kann, *was* der Code tut; es ist viel nützlicher, zu erklären *warum*.

Dieser Kommentar ist im Zusammenhang mit dem Code redundant und nutzlos:

```
v = 5      # Weise v den Wert 5 zu
```

Dieser Kommentar dagegen enthält nützliche Informationen, die nicht im Code enthalten sind:

```
v = 5      # Geschwindigkeit in m/s.
```

Gute Variablennamen können die Notwendigkeit von Kommentaren reduzieren, aber zu lange Namen können komplexe Ausdrücke schwer lesbar machen. Daher sollten geeignete Variablennamen gewählt werden.

Wählen sprechender Variablennamen

Solange wir die einfachen Regeln für die Benennung von Variablen befolgen und reservierte Wörter vermeiden, haben wir bei der Benennung unserer Variablen einen

großen Spielraum. Am Anfang kann diese Auswahl verwirrend sein, sowohl wenn wir ein Programm lesen, als auch wenn wir unsere eigenen Programme schreiben. Die folgenden drei Programme sind zum Beispiel identisch in Bezug auf das, was sie leisten, aber sehr unterschiedlich, wenn wir sie lesen und versuchen, sie zu verstehen.

```
a = 35.0
b = 12.50
c = a * b
print(c)
```

```
stunden = 35.0
rate = 12.50
kosten = stunden * rate
print(kosten)
```

```
x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print(x1q3p9afd)
```

Der Python-Interpreter interpretiert alle drei Programme als *gleichwertig*, aber Menschen sehen und verstehen diese Programme ganz anders. Menschen werden die *Intention* des zweiten Programms am schnellsten verstehen, weil der Programmierer Variablennamen gewählt hat, die seine Absicht bezüglich der Daten, die in jeder Variablen gespeichert werden, widerspiegeln. Wir nennen diese sinnvoll gewählten Variablennamen „sprechende“ Variablennamen.

Obwohl das alles toll klingt und es eine sehr gute Idee ist, sprechende Variablennamen zu verwenden, können sprechende Variablennamen den Programmieranfänger bei der Analyse von Code verwirren. Das liegt daran, dass sich Programmieranfänger die reservierten Schlüsselwörter noch nicht eingeprägt haben (es gibt nur 35 davon). So kann es vorkommen, das ein Anfänger einen sprechenden Variablennamen fälschlicherweise als Teil der Programmiersprache auffasst.

Werfen wir einen kurzen Blick auf den folgenden Python-Beispielcode, der mithilfe einer Schleife durch einen Datensatz iteriert. Wir werden Schleifen bald behandeln, aber versuchen wir jetzt erst einmal, uns vorzustellen, was hier passiert:

```
for word in words:
    print(word)
```

Was ist hier los? Welche der Token (for, word, in, etc.) sind reservierte Wörter und welche sind nur Variablennamen? Versteht Python auf einer fundamentalen Ebene den Begriff der Wörter? Programmieranfänger haben Schwierigkeiten zu unterscheiden, welche Teile des Codes so sein *müssen* wie sie es sind und welche Teile des Codes freie Entscheidungen des Programmierers sind.

Der folgende Code ist äquivalent zum obigen Code:

```
for slice in pizza:
    print(slice)
```

Für den Programmieranfänger ist es einfacher, sich diesen Code anzusehen und zu wissen, welche Teile reservierte Wörter sind, die von Python definiert wurden, und welche Teile einfach vom Programmierer gewählte Variablennamen sind. Es ist ziemlich klar, dass Python kein grundlegendes Verständnis von Pizza und Pizzastücken hat und der Tatsache, dass eine Pizza aus einer Menge von einer oder mehreren Pizzastücken besteht.

Aber wenn es in unserem Programm wirklich darum geht, Daten zu lesen und nach Wörtern in den Daten zu suchen, sind `pizza` und `slice` wenig sprechende Variablennamen. Sie als Variablennamen zu wählen, lenkt vom Sinn des Programms ab.

Nach recht kurzer Zeit wird man die häufigsten reservierten Wörter kennen, und man wird anfangen, die reservierten Wörter zu sehen, die einem entgegenspringen:

Die Teile des Codes, die von Python definiert sind (`for`, `in`, `print` und `:`), sind fett gedruckt, die vom Programmierer gewählten Variablen (`word` und `words`) sind nicht fett gedruckt. Viele Texteditoren sind sich der Python-Syntax bewusst und färben reservierte Wörter anders ein, um Hinweise zu geben, die Variablen und reservierten Wörter getrennt zu halten. Nach einer Weile wird man anfangen, Python zu lesen und schnell feststellen, was eine Variable und was ein reserviertes Wort ist.

Debugging

An dieser Stelle ist der Syntaxfehler, den wir am ehesten machen, ein illegaler Variablenname, wie `class` und `yield`, die Schlüsselwörter sind, oder `odd~job` und `US$`, die unzulässige Zeichen enthalten.

Wenn wir ein Leerzeichen in einen Variablennamen setzen, denkt Python, dass es sich um zwei Operanden ohne einen Operator handelt:

```
>>> bad name = 5
SyntaxError: invalid syntax

>>> month = 09
  File "<stdin>", line 1
    month = 09
              ^
SyntaxError: invalid token
```

Bei Syntaxfehlern sind die Fehlermeldungen nicht sehr hilfreich. Die häufigsten Meldungen sind `SyntaxError: invalid syntax` und `SyntaxError: invalid token`, die beide nicht sehr informativ sind.

Im Beispiel oben besteht übrigens das Problem, dass Zahlen nicht mit führenden Nullen beginnen dürfen. `09` ist also kein gültiges *Wort* in Python. Wenn Sie das Buch bisher sorgfältig gelesen haben, sollten Sie nun stutzig werden. Wir hatten ja ein Beispiel, in dem der Code `1,000,000` als Folge der Zahlen `1` und `0` gedeutet wurde. Dabei gibt es doch auch führende Nullen, oder? Nicht ganz! Folgen von beliebig vielen Nullen interpretiert Python als `0`. Dies ist eines von vielen kleinen

Details, die manchmals sonderbar erscheinen und die man nicht unbedingt verstehen muss.

Ein Laufzeitfehler, den man am Anfang gelegentlich verursacht, ist ein „use before definition“, d. h. der Versuch, eine Variable zu verwenden, bevor man ihr einen Wert zugewiesen hat. Dies kann z. B. passieren, wenn man einen Variablennamen falsch schreibt:

```
>>> kapital = 327.68
>>> zinsen = kapitel * zinssatz
NameError: name 'kapitel' is not defined
```

Bei Variablennamen wird zwischen Groß- und Kleinschreibung unterschieden (englisch *case-sensitivity*), daher ist `LaTeX` nicht dasselbe wie `latex`.

An diesem Punkt ist die wahrscheinlichste Ursache für einen logischen Fehler die Reihenfolge der Operationen. Um z. B. $\frac{1}{2\pi}$ auszuwerten, könnte man versucht sein, zu schreiben

```
>>> 1.0 / 2.0 * pi
```

aber die Division geschieht zuerst, also würden wir $\pi/2$ erhalten, was nicht dasselbe ist! Es gibt keine Möglichkeit für Python zu wissen, was wir eigentlich schreiben wollten, also bekommen wir in diesem Fall keine Fehlermeldung; Wir bekommen einfach die falsche Antwort.

Glossar

Zuweisung Eine Anweisung, die einer Variablen einen Wert zuweist.

konkatenieren Verkettung zweier Operanden.

Kommentar Informationen in einem Programm, die für andere Programmierer (oder jeden, der den Quellcode liest) bestimmt sind und keinen Einfluss auf die Ausführung des Programms haben.

auswerten Einen Ausdruck vereinfachen, indem die Operationen so ausgeführt werden, dass ein einziger Wert entsteht.

Ausdruck Eine Kombination aus Variablen, Operatoren und Werten, die einen einzelnen Ergebniswert darstellt.

Gleitkommazahl Ein Datentyp, der Zahlen mit Nachkommastellen darstellt.

Ganzzahl Ein Datentyp, der ganze Zahlen darstellt.

Schlüsselwort Ein reserviertes Wort, das vom Compiler verwendet wird, um ein Programm zu analysieren; man kann Schlüsselwörter wie `if`, `def` und `while` nicht als Variablennamen verwenden.

Modulo-Operator Ein Operator, der mit einem Prozentzeichen (`%`) gekennzeichnet ist, der mit ganzen Zahlen arbeitet und den Rest ergibt, wenn eine Zahl durch eine andere dividiert wird.

Operand Einer der Werte, mit dem ein Operator arbeitet.

Operator Ein spezielles Symbol, das eine einfache Berechnung wie Addition, Multiplikation oder String-Verkettung darstellt.

Vorrangregeln Die Reihenfolge, in der Ausdrücke mit mehreren Operatoren und Operanden ausgewertet werden.

Anweisung Ein Abschnitt des Codes, der einen Befehl oder eine Aktion darstellt. Die Anweisungen, die wir bisher gesehen haben, sind Zuweisungen und Ausgabe-Anweisungen.

Zeichenkette Ein Datentyp, der Zeichenfolgen (englisch *string*) repräsentiert.

Datentyp Eine Kategorie von Werten. Die Typen, die wir bisher gesehen haben, sind Ganzzahlen (Typ `int`), Gleitkommazahlen (Typ `float`) und Zeichenketten (Typ `str`).

Wert Eine der grundlegenden Dateneinheiten, wie z. B. eine Zahl oder eine Zeichenkette, die von einem Programm manipuliert wird.

Variable Ein Bezeichner, der sich auf einen Wert bezieht.

Übungen

Übung 2: Schreiben Sie ein Programm, das mit `input` einen Benutzer zur Eingabe seines Namens auffordert und ihn dann begrüßt.

```
Gib deinen Namen ein: Heiner
Hallo Heiner
```

Übung 3: Schreiben Sie ein Programm, das den Benutzer nach Arbeitsstunden und Stundensatz fragt, um den Bruttolohn zu berechnen.

```
Gib die Arbeitsstunden an: 35
Gib den Stundensatz an: 2.75
Zahlung: 96.25
```

Wir werden uns erst einmal nicht darum kümmern, dass unsere Auszahlung genau zwei Stellen nach dem Komma hat. Wenn Sie möchten, können Sie mit der eingebauten Python-Funktion `round` experimentieren, um den resultierenden Lohn auf zwei Dezimalstellen zu runden.

Übung 4: Angenommen, wir führen die folgenden Zuweisungen aus:

```
breite = 17
hoehe = 12.0
```

Schreiben Sie für jeden der folgenden Ausdrücke den Wert des Ausdrucks und den Datentyp (des Wertes des Ausdrucks).

1. `breite//2`
2. `breite/2.0`
3. `hoehe/3`
4. `1 + 2 * 5`

Verwenden Sie den Python-Interpreter, um Ihre Antworten zu überprüfen.

Übung 5: Schreiben Sie ein Programm, das den Benutzer zur Eingabe einer Celsius-Temperatur auffordert, die Temperatur in Fahrenheit konvertiert und die konvertierte Temperatur ausgibt.

Die Umrechnung von Celsius zu Fahrenheit lautet: °F = °C * 1,8 + 32

```
Gib die Temperatur in Celsius ein: 37
Fahrenheit in 98.6
```

Kapitel 3

Bedingte Ausführung

In den vorherigen Kapiteln haben wir den Python-Interpreter eher wie einen Taschenrechner benutzt. Wir haben Werte definiert oder eingegeben, den Werten Namen gegeben und Berechnungen mit diesen Werten angestellt. Vieles, was Computerprogramme ausmacht, kann man damit nicht erreichen.

Ein wichtiges Konzept, das uns fehlt, ist der sogenannte *Kontrollfluss*. Dabei handelt es sich, grob gesagt, um das Steuern von Abläufen im Programm. Nehmen wir an, wir fragen den Benutzer nach der Eingabe eines Zeichens und nur wenn das Zeichen kein `q` (für *quit*) ist, soll das Programm weiter laufen.

Wie wir die Eingabe erledigen, haben wir bereits kennengelernt, nicht aber wie wir feststellen können, ob das gelesene Zeichen ein `q` ist. Außerdem wissen wir noch nicht, wie man die Ausführung bestimmter Anweisungen von dem Ergebnis einer solchen Überprüfung abhängig macht. Glücklicherweise brauchen wir dazu nicht viel Neues und nach dem Lesen der folgenden Seiten wird uns klar sein, wie man solche Aufgaben mit Python erledigt.

Boolesche Ausdrücke

Um eine bedingte Ausführung programmieren zu können, müssen wir zunächst Bedingungen überprüfen. Dazu benötigen wir eine spezielle Form von Ausdrücken, die sich nicht, wie bisher gesehen, zu einem allgemeinen Wert (Zahl, Zeichenkette, etc.) ableiten lassen, sondern zu einem *Wahrheitswert*.

Einen solchen Ausdruck, der im Ergebnis entweder wahr oder falsch ist, nennt man *booleschen Ausdruck*. Die folgenden Beispiele verwenden den Operator `==`, der zwei Werte (die sogenannten *Operanden*) vergleicht und `True` erzeugt, wenn sie gleich sind, und sonst `False`:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` und `False` sind spezielle Werte, die zur Klasse `bool` gehören; sie sind keine Zeichenketten, sondern die beiden logischen Werte *wahr* und *falsch*:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Der Operator `==` ist ein *Vergleichsoperator*. Wenn man mit einem solchen Operator zwei Operanden vergleicht, erhält man einen Wahrheitswert zurück. Neben dem Vergleich auf *Gleichheit* gibt es noch einige weitere *Vergleichsoperatoren*. Diese sind:

```
x != y               # x ist ungleich y
x > y                # x ist groesser als y
x < y                # x ist kleiner als y
x >= y               # x ist groesser oder gleich y
x <= y               # x ist kleiner oder gleich y
x is y               # x bezeichnet das selbe Objekt wie y
x is not y           # x bezeichnet nicht das selbe Objekt wie y
```

Obwohl Ihnen diese Operationen wahrscheinlich bekannt sind, unterscheiden sich die Python-Symbole von den mathematischen Symbolen für dieselben Operationen. Ein häufiger Fehler ist die Verwendung eines einfachen Gleichheitszeichens (`=`) anstelle eines doppelten Gleichheitszeichens (`==`). Wir müssen daran denken, dass `=` ein Zuweisungsoperator und `==` ein Vergleichsoperator ist. So etwas wie `=<` oder `=>` gibt es nicht.

Etwas unklar mag Ihnen zum jetzigen Zeitpunkt der Unterschied zwischen `==` und `is` erscheinen. Um das zu erklären, muss man etwas mehr über die Funktionsweise der Programmiersprache selbst verstehen. Kurz gesagt, vergleicht `==` Werte und `is` Objekte. Was bedeutet das? Nun, unterschiedliche Objekte, also z. B. Daten im Speicher können gleiche Werte haben. Ganzzahl Objekte mit einem bestimmten Wert, sagen wir `1234`, kann es beispielsweise mehrfach unter verschiedenen Namen im Programm geben. Wenn einer dieser Werte als `a`, der andere als `b` benannt ist, wäre `a is b` falsch, denn wir haben es mit unterschiedlichen Objekten zu tun. `a == b` ist hingegen wahr, schließlich tragen `a` und `b` den gleichen Wert.

Zu Beginn werden wir häufiger `==`, aber fast nie `is` verwenden. Der Vergleich auf Objektidentität ist eher nur für einige „Spezialfälle“ wichtig. In den meisten Fällen wollen wir Werte vergleichen.

Logische Operatoren

Es gibt drei *logische Operatoren*: `and`, `or`, und `not`. Die Semantik (Bedeutung) dieser Operatoren ist ähnlich wie ihre Bedeutung im Englischen. Zum Beispiel,

`x > 0 and x < 10`

ist nur wahr, wenn `x` größer als 0 *und* kleiner als 10 ist.

`n%2 == 0 or n%3 == 0` ist wahr, wenn *eine* der beiden Bedingungen wahr ist, d. h. wenn die Zahl durch 2 *oder* 3 teilbar ist.

Schließlich negiert der Operator `not` einen booleschen Ausdruck, sodass `not (x > y)` wahr ist, wenn `x > y` falsch ist; das heißt, wenn `x` kleiner oder gleich `y` ist.

Streng genommen sollten die Operanden der logischen Operatoren boolesche Ausdrücke sein, aber Python ist nicht sehr streng. Jede Zahl ungleich Null wird als „wahr“ interpretiert.

```
>>> 17 and True
True
```

Diese Flexibilität kann nützlich sein, aber es gibt einige Feinheiten, die verwirrend sein können. Wir sollten sie vielleicht vermeiden, bis wir sicher sind, dass wir wissen, was wir tun.

Bedingte Ausführung

Um sinnvolle Programme zu schreiben, brauchen wir fast immer die Möglichkeit, Bedingungen zu prüfen und das Verhalten des Programms entsprechend zu ändern. *Bedingte Anweisungen* geben uns diese Fähigkeit. Die einfachste Form ist die `if`-Anweisung:

```
if x > 0:
    print('x is positive')
```

Der boolesche Ausdruck nach der `if`-Anweisung wird als *Bedingung* bezeichnet. Wir beenden die `if`-Anweisung mit einem Doppelpunkt (:) und die Zeile(n) nach der `if`-Anweisung werden eingerückt.

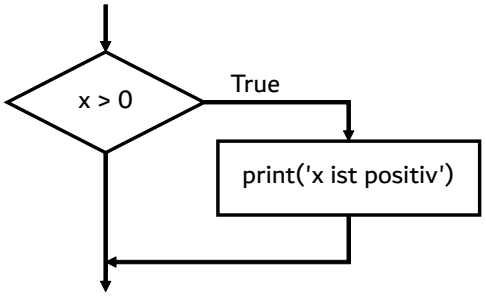

Abbildung 3.1: Ablaufdiagramm einer If-Anweisung

Wenn die logische Bedingung wahr ist, dann wird die eingerückte Anweisung ausgeführt. Wenn die logische Bedingung falsch ist, wird die eingerückte Anweisung übersprungen. Die *Einrückung* erfüllt hier **zwei** wesentliche Aufgaben im Programm. Erstens: Sie macht das Programm besser lesbar. Man erkennt sofort, dass die Anweisung `print` „unterhalb“ der Überprüfung mit `if` steht. Sie stehet also nicht

auf gleicher *Ebene* wie das `if` selbst, sondern gehört sichtbar nur zu dem „Wahr-Teil“ der Überprüfung. Zweitens: Statt der einen `print`-Anweisung könnten an der gleichen Stelle mehrere Anweisungen, und sogar wieder neue `if`-Anweisungen stehen. Alle Anweisungen mit der gleichen Einrückungstiefe gehören zur gleichen *Ebene* des Programms.

Die Verwendung von Einrückungen führt schnell zu Syntaxfehlern. Wenn Sie einmal nicht genau aufpassen und ein Leerzeichen zu viel oder zu wenig, oder auch ein Tab-Zeichen, statt mehrerer Leerzeichen verwenden, wird Python sich beschweren. Allerdings lassen sich diese Fehler sehr schnell finden. Außerdem helfen Ihnen die Editoren und Werkzeuge, die Sie zum Programmieren verwenden, die Einrückungstiefe beizubehalten.

Viele andere Programmiersprachen verwenden Klammern, um Blöcke von Anweisungen zusammenzufassen. Dies erscheint auf den ersten Blick logischer, als mit Leerzeichen und Zeilen zu arbeiten. Klammern haben aber zwei große Nachteile. Zum einen benötigt man zusätzliche Zeichen, die den Programmcode verlängern und unübersichtlicher machen. Zum anderen ist man, wenn man Klammern verwendet, nicht dazu gezwungen, eine gute optische Aufteilung des Codes in Ebenen einzuhalten. Gerade Programmieranfänger halten sich oft nicht an Formatierungsregeln und leider führt das oft zu logischen Fehlern, die nur sehr schwierig aufzuspüren sind.

Der Aufbau von `if`-Anweisungen ähnelt dem von Funktionsdefinitionen oder `for`-Schleifen[1]. Die Anweisung besteht aus einer Kopfzeile, die mit einem Doppelpunkt (`:`) endet, gefolgt von einem eingerückten Block. Anweisungen wie diese werden *zusammengesetzte Anweisungen* genannt, weil sie sich über mehr als eine Zeile erstrecken.

Es gibt keine Begrenzung für die Anzahl der Anweisungen, die im Rumpf erscheinen können, aber es muss mindestens eine geben. Gelegentlich ist es nützlich, einen Block ohne Anweisungen zu haben (normalerweise als Platzhalter für Code, den wir noch nicht geschrieben haben). In diesem Fall können wir die Anweisung `pass` verwenden, die nichts tut.

```
if x < 0:
    pass          # need to handle negative values!
```

Wenn wir im Python-Interpreter eine `if`-Anweisung eingeben, ändert sich die Eingabeaufforderung von drei Größer-Zeichen zu drei Punkten, um anzuzeigen, dass wir uns in der Mitte eines Anweisungsblocks befinden, wie unten gezeigt:

```
>>> x = 3
>>> if x < 10:
...     print('x ist klein')
...
x ist klein
>>>
```

Wenn wir den Python-Interpreter verwenden, müssen wir am Ende eines Blocks eine Leerzeile stehen lassen, sonst gibt Python einen Fehler zurück:

[1] Funktionen werden wir in Kapitel 4 und Schleifen in Kapitel 5 kennenlernen.

```
>>> x = 3
>>> if x < 10:
...     print('x ist klein')
... print('Fertig')
  File "<stdin>", line 3
    print('Fertig')
        ^
SyntaxError: invalid syntax
```

Eine Leerzeile am Ende eines Anweisungsblocks ist beim Schreiben und Ausführen eines Scripts nicht notwendig, kann aber die Lesbarkeit unseres Codes verbessern.

Alternative Ausführung

Eine zweite Form der `if`-Anweisung ist die *alternative Ausführung*, bei der es zwei Möglichkeiten gibt und die Bedingung bestimmt, welche davon ausgeführt wird. Die Syntax sieht wie folgt aus:

```
if x%2 == 0:
    print('x ist gerade')
else:
    print('x ist ungerade')
```

Wenn der Rest bei der Division von `x` durch 2 gleich 0 ist, dann wissen wir, dass `x` gerade ist, und das Programm gibt eine entsprechende Meldung aus. Wenn die Bedingung falsch ist, wird der zweite Block von Anweisungen ausgeführt.

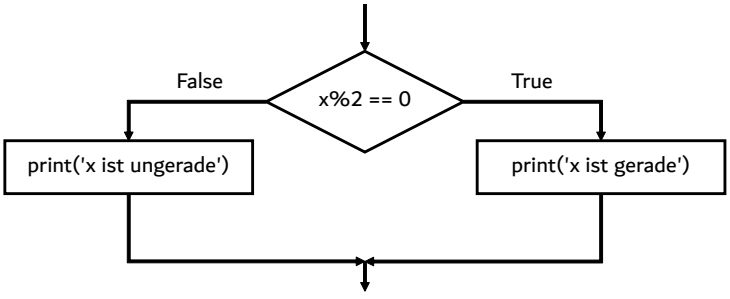

Abbildung 3.2: Ablaufdiagramm einer If-Else-Anweisung

Da die Bedingung entweder wahr oder falsch sein muss, wird genau eine der Alternativen ausgeführt. Die Alternativen werden *Verzweigungen* genannt, weil sie Verzweigungen im Ablauf der Ausführung sind.

Verkettete Bedingungen

Manchmal gibt es mehr als zwei Möglichkeiten und wir brauchen mehr als zwei Verzweigungen. Eine Art, dies zu erreichen, sind *verkettete Bedingungen*:

```
if x < y:
    print('x ist kleiner als y')
elif x > y:
    print('x ist groesser als y')
else:
    print('x und y sind gleich')
```

`elif` ist eine Abkürzung für „else if“. Auch hier wird genau eine Verzweigung ausgeführt.

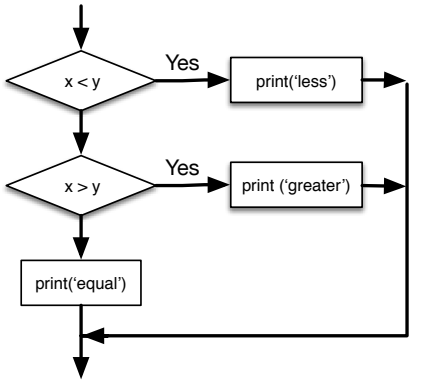

Abbildung 3.3: If-Then-ElseIf-Logik

Es gibt keine Begrenzung für die Anzahl der `elif`-Anweisungen. Wenn es eine `else`-Klausel gibt, muss sie am Ende stehen, aber es muss nicht unbedingt eine geben.

```
if choice == 'a':
    print('Nicht richtig!')
elif choice == 'b':
    print('Richtig!')
elif choice == 'c':
    print('Fast, aber nicht richtig!')
```

Jede Bedingung wird der Reihe nach geprüft. Wenn die erste falsch ist, wird die nächste geprüft, und so weiter. Wenn eine von ihnen wahr ist, wird der entsprechende Zweig ausgeführt und die Anweisung endet. Auch wenn mehr als eine Bedingung wahr ist, wird nur der erste wahre Zweig ausgeführt.

Verschachtelte Bedingungen

Eine Bedingung kann auch in eine andere verschachtelt werden. Wir hätten das Beispiel mit den drei Verzweigungen auch so schreiben können:

```
if x == y:
    print('x und y sind gleich')
```

```
else:
    if x < y:
        print('x it kleiner als y')
    else:
        print('x ist groesser als y')
```

Die äußere Bedingung enthält zwei Verzweigungen. Der erste Zweig enthält eine einfache Anweisung. Der zweite Zweig enthält eine weitere `if`-Anweisung, die ihrerseits zwei Zweige hat. Diese beiden Zweige sind beide einfache Anweisungen, obwohl sie auch bedingte Anweisungen hätten sein können.

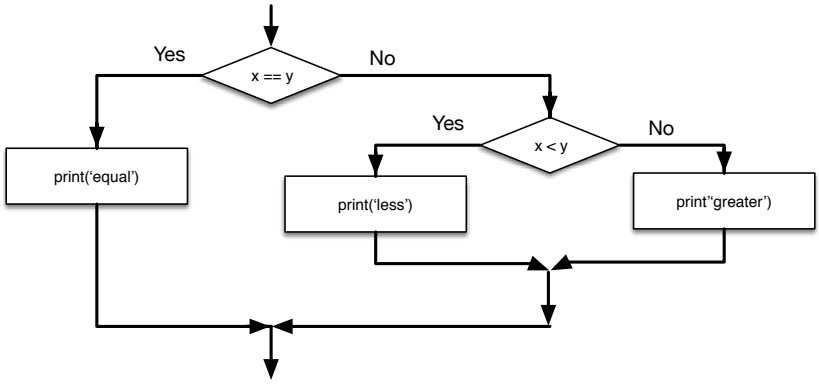

Abbildung 3.4: Verschachtelte If-Anweisungen

Obwohl die Einrückung der Anweisungen die Struktur deutlich macht, werden *verschachtelte Bedingungen* sehr schnell unübersichtlich. Im Allgemeinen ist es eine gute Idee, sie wenn möglich zu vermeiden.

Logische Operatoren bieten oft eine Möglichkeit, verschachtelte bedingte Anweisungen zu vereinfachen. Zum Beispiel können wir den folgenden Code mit einer einzigen Bedingung umschreiben:

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

Die `print`-Anweisung wird nur ausgeführt, wenn wir es an beiden Bedingungen vorbeischaffen, also können wir den gleichen Effekt mit dem `and`-Operator erzielen:

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

Abfangen von Ausnahmen mit `try` und `except`

Im vorherigen Kapitel haben wir ein Codesegment gesehen, in dem wir die Funktionen `input` und `int` verwendet haben, um eine vom Benutzer eingegebene Ganzzahl zu lesen und zu analysieren. Wir haben auch gesehen, wie tückisch dies sein kann:

```
>>> speed = input(prompt)
Was ist die Fluggeschwindigkeit einer unbeladenen Schwalbe?
Meinst du eine europäische oder eine afrikanische Schwalbe?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
```

Wenn wir diese Anweisungen im Python-Interpreter ausführen, erhalten wir eine neue Eingabeaufforderung vom Interpreter, denken „ups“ und machen mit der nächsten Anweisung weiter.

Wenn wir diesen Code jedoch in ein Python-Skript einfügen und dieser Fehler auftritt, bleibt unser Skript sofort mit einem *Traceback* stehen. Es führt die folgende Anweisung nicht aus.

Hier ist ein Beispielprogramm zur Umrechnung einer Fahrenheit-Temperatur in eine Celsius-Temperatur:

```
inp = input('Gib die Temperatur in Fahrenheit ein: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Code: https://tiny.one/py4de/code3/fahren.py
```

Probieren wir diesen Code einmal mit einer korrekten Eingabe, z. B. mit 99 °F:

```
python fahren.py
Gib die Temperatur in Fahrenheit ein: 99
37.22222222222222
```

Das sieht korrekt aus. Wenn wir aber nun diesen Code erneut ausführen und ihm eine ungültige Eingabe geben, bricht er einfach mit einer unfreundlichen Fehlermeldung ab:

```
python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

Es gibt eine in Python eingebaute Struktur für die bedingte Ausführung, um diese Arten von erwarteten und unerwarteten Fehlern zu behandeln, die „try/except“ genannt wird. Die Idee von `try` und `except` ist, dass wir wissen, dass eine Folge von Anweisungen ein Problem haben könnte und wir einige Anweisungen hinzufügen möchten, die ausgeführt werden, wenn ein Fehler auftritt. Diese zusätzlichen Anweisungen (der *except-Block*) werden ignoriert, wenn kein Fehler auftritt.

Man kann sich die `try`- und `except`-Funktion in Python als eine „Versicherungspolice“ für eine Folge von Anweisungen vorstellen.

Wir können unseren Temperaturwandler wie folgt umschreiben:

```
inp = input('Gib die Temperatur in Fahrenheit ein: ')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Bitte gib eine Zahl ein!')

# Code: https://tiny.one/py4de/code3/fahren2.py
```

Python beginnt mit der Ausführung der Anweisungsfolge im `try`-Block. Wenn alles gut geht, überspringt es den `except`-Block und fährt fort. Wenn im `try`-Block eine *Ausnahme* (englisch *Exception*) auftritt, springt Python aus dem `try`-Block heraus und führt die Folge von Anweisungen im `except`-Block aus.

```
python fahren2.py
Gib die Temperatur in Fahrenheit ein: 99
37.22222222222222
```

```
python fahren2.py
Gib die Temperatur in Fahrenheit ein: fred
Bitte gib eine Zahl ein!
```

Die Behandlung einer Ausnahme mit einer `try`-Anweisung wird als *Fangen* einer Ausnahme bezeichnet. In diesem Beispiel gibt die `except`-Klausel eine Fehlermeldung aus. Im Allgemeinen gibt Ihnen das Abfangen einer Ausnahme die Möglichkeit, das Problem zu beheben, es erneut zu versuchen oder zumindest das Programm ordnungsgemäß zu beenden.

Verkürzte Auswertung logischer Ausdrücke

Wenn Python einen logischen Ausdruck wie `x >= 2 and (x/y) > 2` verarbeitet, wertet es den Ausdruck von links nach rechts aus. Aufgrund der Definition von `and` ist, wenn `x` kleiner als 2 ist, der Ausdruck `x >= 2` logisch falsch (also `False`) und somit ist der gesamte Ausdruck `False`, unabhängig davon, ob `(x/y) > 2` als `True` oder `False` ausgewertet wird.

Wenn Python feststellt, dass durch die Auswertung des restlichen Teils eines logischen Ausdrucks sich am Endergebnis nichts mehr ändern wird, bricht es die Auswertung ab und führt die Berechnungen im restlichen Teil des logischen Ausdrucks nicht aus. Wenn die Auswertung eines logischen Ausdrucks stoppt, weil der Gesamtwert bereits bekannt ist, bezeichnet man dies als *verkürzte Auswertung* (auch *Kurzschlussauswertung* oder *bedingte Auswertung*, englisch *short-circuit evaluation*).

Auch wenn dies wie eine Kleinigkeit erscheinen mag, führt die abgekürzte Auswertung zu einer cleveren Technik, die als *Wächter-Muster* (englisch *guardian pattern*) bezeichnet wird. Betrachten wir die folgende Codesequenz im Python-Interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Die dritte Berechnung schlug fehl, weil Python `(x/y)` auswertete und `y` null war, was einen Laufzeitfehler (`division by zero`) verursacht. Eine Division durch null ist mathematisch nicht definiert, Python *kann* die Berechnung also nicht durchführen (und Sie wissen ja, dass Python nicht „rät“). Allerdings schlug das zweite Beispiel *nicht* fehl, obwohl hier auch `(x/y) > 2` stand und `y` zu dem Zeitpunkt den Wert `0` hatte. Dass es zu keinem Fehler kam, liegt daran, dass der erste Teil dieses Ausdrucks `x >= 2` zu `False` ausgewertet wurde, sodass `(x/y)` aufgrund der verkürzten Auswertung nie ausgeführt wurde und es damit keinen Fehler gab.

Wir können den logischen Ausdruck so konstruieren, dass wir strategisch eine *Schutz*-Auswertung direkt vor der Auswertung platzieren, die einen Fehler verursachen könnte, wie folgt:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>>
```

Im ersten logischen Ausdruck ist `x >= 2` `False`, also stoppt die Auswertung am ersten `and`. Im zweiten logischen Ausdruck ist `x >= 2` `True`, aber `y != 0` ist `False`, also erreichen wir nie `(x/y)`. Die Bedingung `y != 0` dient hier also als *Schutz*, um sicherzustellen, dass wir `(x/y)` nur ausführen, wenn `y` ungleich Null ist.

Unbedingt beachten müssen wir allerdings die Auswertung von links nach rechts. Im folgenden Beispiel folgt der Schutz-Klausel `y != 0` *nach* der `(x/y)` Berechnung, sodass der Ausdruck mit einem Fehler fehlschlägt.

```
>>> x = 6
>>> y = 0
```

```
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Debugging

Der *Traceback*, den Python beim Auftreten eines Fehlers anzeigt, enthält eine Menge Informationen, die aber überwältigend sein können. Die nützlichsten Teile sind normalerweise:

- welche Art von Fehler es war, und
- wo er aufgetreten ist.

Syntaxfehler sind in der Regel leicht zu finden, aber es gibt ein paar Tücken. Whitespace-Fehler können schwierig sein, weil Leerzeichen und Tabulatoren unsichtbar sind und wir gewohnt sind, sie zu ignorieren.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

In diesem Beispiel besteht das Problem darin, dass die zweite Zeile um ein Leerzeichen eingerückt ist. Aber die Fehlermeldung zeigt auf `y`, was irreführend ist. Im Allgemeinen zeigen Fehlermeldungen an, wo das Problem entdeckt wurde, aber der tatsächliche Fehler kann früher im Code liegen, manchmal in einer vorhergehenden Zeile.

Im Allgemeinen wird in den Fehlermeldungen angegeben, wo das Problem entdeckt wurde, aber das ist oft nicht die Ursache des Problems.

Glossar

Block Die Folge von Anweisungen innerhalb einer zusammengesetzten Anweisung.

boolescher Ausdruck Ein Ausdruck, dessen Wert entweder `True` oder `False` ist.

Verzweigung Eine der alternativen Anweisungsfolgen in einer bedingten Anweisung.

verkettete Bedingungen Eine bedingte Anweisung mit einer Reihe von alternativen Verzweigungen.

Vergleichsoperator Einer der Operatoren, der seine Operanden vergleicht: `==`, `!=`, `>`, `<`, `>=` und `<=`.

bedingte Anweisung Eine Anweisung, die den Ablauf der Ausführung in Abhängigkeit von einer Bedingung steuert.

Bedingung Der boolesche Ausdruck in einer bedingten Anweisung, der bestimmt, welcher Zweig ausgeführt wird.

zusammengesetzte Anweisung Eine Anweisung, die aus einer Kopfzeile und einem Rumpf besteht. Die Kopfzeile endet mit einem Doppelpunkt (`:`). Der Rumpf wird relativ zur Kopfzeile eingerückt.

logischer Operator Einer der Operatoren, der boolesche Ausdrücke kombiniert: `and`, `or`, und `not`.

verschachtelte Bedingung Eine bedingte Anweisung, die in einem der Zweige einer anderen bedingten Anweisung erscheint.

Traceback Eine Auflistung aller ausgeführten Funktionen, die beim Auftreten einer Ausnahme ausgegeben wird.

abgekürzte Auswertung Wenn Python die Auswertung eines logischen Ausdrucks teilweise durchläuft und die Auswertung anhält, weil Python den Endwert für den Ausdruck kennt, ohne den Rest des Ausdrucks auswerten zu müssen.

Übungen

Übung 1: Schreiben Sie Ihr Programm zur Lohnberechnung so um, dass ein Mitarbeiter das 1,5-fache des Stundensatzes für Arbeitsstunden über 40 Stunden erhält.

```
Anzahl Arbeitsstunden: 45
Stundenlohn: 10
Monatsgehalt: 475.0
```

Übung 2: Schreiben Sie Ihr Programm zur Lohnberechnung unter Verwendung von `try` und `except` so um, dass es nichtnumerische Eingaben elegant behandelt, indem es eine Meldung ausgibt und das Programm beendet. Im Folgenden sehen Sie zwei beispielhafte Ausführungen des Programms:

```
Anzahl Arbeitsstunden: 20
Stundenlohn: neun
Fehler, bitte gib eine Zahl ein
```

```
Anzahl Arbeitsstunden: vierzig
Fehler, bitte gib eine Zahl ein
```

Übung 3: Schreiben Sie ein Programm, das nach einem Wert zwischen 0,0 und 1,0 fragt. Wenn die Punktzahl außerhalb des Bereichs liegt, geben Sie eine Fehlermeldung aus. Wenn die Punktzahl zwischen 0,0 und 1,0 liegt, geben Sie eine Note anhand der folgenden Tabelle aus:

```
Punkte   Note
>= 0.9     1
>= 0.8     2
>= 0.7     3
>= 0.6     4
 < 0.6     5
```

Führen Sie das Programm wiederholt wie nachfolgend gezeigt aus, um die verschiedenen Werte für die Eingabe zu testen.

```
Punkte eingeben: 0.95
1


Punkte eingeben: perfekt
Falsche Punktanzahl


Punkte eingeben: 10.0
Falsche Punktanzahl


Punkte eingeben: 0.75
3


Punkte eingeben: 0.5
5
```

Kapitel 4

Funktionen

Eine Kernidee der Programmierung ist es, die unterschiedlichen Teilaufgaben, die in einem Programm verwendet werden, in einzelne logische Einheiten aufzuteilen. Der große Vorteil dieses Vorgehens ist, dass unsere Programme besser strukturiert sind und die einzelnen Teile im gleichen oder auch in anderen Programmen wiederverwendet werden können. Jedes Python-Programm verwendet, selbst wenn Sie das nicht einmal sehen, Funktionen und auch wir haben schon einige *eingebaute Funktionen* von Python kennengelernt, darunter `print` und `input`. Funktionen kann man aber nicht nur verwenden, sondern auch selbst definieren. Dies und vieles Weitere zu Funktionen lernen Sie in diesem Kapitel kennen.

Funktionsaufrufe

Im Zusammenhang mit der Programmierung ist eine *Funktion* eine benannte Folge von Anweisungen, die eine Berechnung durchführt. Wenn wir eine Funktion definieren, geben wir den Namen und die Abfolge der Anweisungen an. Später können wir die Funktion mit dem Namen „aufrufen“. Wir haben bereits ein Beispiel für einen *Funktionsaufruf* gesehen:

```
>>> type(32)
<class 'int'>
```

Der Name der Funktion ist `type`. Der Ausdruck in Klammern wird das *Argument* der Funktion genannt. Das Argument ist ein Wert oder eine Variable, die wir in die Funktion als Eingabe für die Funktion übergeben. Das Ergebnis ist der Datentyp des Arguments.

Es ist üblich zu sagen, dass eine Funktion ein Argument „nimmt“ oder „akzeptiert“ und ein Ergebnis „zurückgibt“. Das Ergebnis wird als *Rückgabewert* bezeichnet.

Built-in-Funktionen

Python bietet eine Reihe von wichtigen eingebauten Funktionen[1] (sogenannte *Built-in-Funktionen*), die wir verwenden können, ohne die Funktionsdefinition angeben zu müssen. Die Schöpfer von Python haben eine Reihe von Funktionen geschrieben, um häufige Probleme zu lösen, und sie in Python integriert, damit wir sie verwenden können.

```
>>> abs(-42)
42
>>> float(21)
21.0
>>>
```

Funktionen können dabei nicht nur ein Argument haben, sondern auch mehrere. Die Built-in-Funktionen `max` und `min` beispielsweise, geben den größten bzw. kleinsten Wert aus einer Folge von Attributen zurück:

```
>>> max(2,-1.2,5.7,4)
5.7
>>> min(2.2,-1,5.7,4)
-1
>>>
```

Die Funktion `max` liefert uns den „größten Wert“ der Argumente, die Funktion `min` den kleinsten Wert. Viele Funktionen sind dabei sehr flexibel programmiert, d. h. man kann sie mit unterschiedlichen Argumenten aufrufen.

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

Im obigen Beispiel sehen wir die Funktionen `max` und `min` erneut, aber diesmal aufgerufen mit der Zeichenkette `'Hello world'`. Nun gibt uns die Funktion `max` das „größte“ Zeichen in der Zeichenkette (was sich als der Buchstabe `'w'` herausstellt) zurück und die Funktion `min` liefert uns das „kleinste“ Zeichen (was sich als ein Leerzeichen herausstellt).

Eine weitere sehr gebräuchliche eingebaute Funktion ist die Funktion `len`, die uns sagt, wie viele Elemente in unserem Argument enthalten sind. Wenn das Argument von `len` eine Zeichenkette ist, gibt sie die Anzahl der Zeichen in der Zeichenkette zurück.

```
>>> len('Hello world')
11
>>>
```

[1] https://docs.python.org/3/library/functions.html#built-in-functions

Wie viele weitere Funktionen, die uns Python bereitstellt, ist auch `len` nicht auf die Verarbeitung genau eines Datentyps (wie z. B. hier von Zeichenketten) beschränkt. Wir können mit jeder Menge von Werten arbeiten, wie wir in späteren Kapiteln sehen werden.

Funktionen zur Typumwandlung

Python bietet auch eingebaute Funktionen, die Werte von einem Typ in einen anderen konvertieren. Die Funktion `int` nimmt einen beliebigen Wert und konvertiert ihn in eine Ganzzahl, wenn sie es kann, oder beschwert sich andernfalls:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

`int` kann Fließkommawerte in Ganzzahlen umwandeln, rundet aber nicht ab, sondern schneidet den Nachkommateil ab:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` konvertiert Ganzzahlen und Zeichenketten in Fließkommazahlen:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Schließlich wandelt `str` das Argument in eine Zeichenkette um:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Übrigens sollten Sie die Namen der eingebauten Funktionen wie reservierte Wörter behandeln. Anders als bei den *Schlüsselwörtern* sind die Namen von eingebauten Funktionen bzw. von Funktionen ganz allgemein nicht vom Interpreter geschützt. Python verhindert es also nicht, wenn wir `max` als Namen für eine eigene Variable oder Funktion verwenden. Wenn man dies tut, führt das manchmal zu kuriosen, meist aber zu ärgerlichen und schwer zu findenden Fehlern.

Die Standardbibliothek

Neben den Built-in-Funktionen werden wir noch viele weitere verwenden, die uns Python bereitstellt. Python verfolgt eine *Batteries Included* Philosophie, was bedeutet, dass viele Funktionen (und Datenstrukturen), die zum produktiven Programmieren nützlich sind, bereits in der Sprache enthalten sind. Dieser Aspekt hat wesentlich zur Verbreitung von Python beigetragen, denn es erlaubt ein Programmieren auf „höherer Ebene“.

Sehr viele Funktionen stellt Python über die sogenannte *Standardbibliothek* bereit, einige Teile der Standardbibliothek werden wir uns in den kommenden Abschnitten genauer ansehen. In der Programmierung bezeichnet der Begriff „Bibliothek“ eine Sammlung von wiederverwendbaren Funktionen und Datenstrukturen. Es gibt Bibliotheken für den Zugriff auf Internetseiten, zum Verarbeiten von großen Datenmengen, für spezielle mathematische Aufgaben, für Audio- oder Bildverarbeitung und für viele weitere Anwendungsgebiete.

Wenn wir für unsere Problemstellung eine geeignete Bibliothek zur Verfügung haben, müssen wir nicht „das Rad neu erfinden“, sondern können die bestehenden Funktionen nutzen, um unsere konkrete Aufgabenstellung damit zu lösen. Wie bei einem gut gefüllten Werkzeugkasten geht es dann auch bei der Programmierung darum, das „passende“ Werkzeug zu finden und es richtig anzuwenden. Gelingt uns das, können wir unser Projekt in deutlich kürzerer Zeit und meist auch mit besserem Ergebnis abschließen.

Wenn Sie selbst Programme schreiben, werden Sie bei Ihren Recherchen in der Python Dokumentation, in anderen Lehrbüchern oder im Internet immer wieder auf Module oder komplette Bibliotheken stoßen, die Ihnen für die Lösung Ihres Problems nützlich sein können. Als erfolgreicher Entwickler sollten Sie lernen, bestehende Lösungen (die Sie lizenzrechtlich verwenden dürfen) einzusetzen. Beim Lernen der Sprache werden wir zwar – zu Trainingszwecken – auch solche Aufgaben „per Hand“ lösen, für die es eigentlich schon vorgefertigte Lösungen. Im Berufsalltag kommt es aber in aller Regel auf Produktivität an und dabei helfen Ihnen die Standardbibliothek oder auch Programmpakete, die andere bereits vor Ihnen entwickelt und frei zur Verfügung gestellt haben.

Da eine Bibliothek sehr umfangreich sein kann, ist es ratsam, die einzelnen Funktionalitäten noch weiter aufzuteilen. Wir werden in diesem Zusammenhang noch öfter von *Paket* oder auch *Modul* sprechen. Bei diesen Begriffen kann man folgende Unterscheidung vornehmen:

- Ein **Modul** ist eine Datei mit der Endung *.py*, die ausführbaren Python Code enthält, darunter in der Regel verschiedene Funktionen und Variablen.

- Ein **Paket** ist eine Sammlung von Modulen, die unter einem gemeinsamen Namensraum zusammengefasst sind und die für einen speziellen Anwendungsbereich entwickelt wurden. Unter den Modulen muss eine Datei namens `__init__.py` sein, damit der Python-Interpreter sie als Paket erkennt. Die `__init__.py` kann eine leere Datei sein, es ist nur wichtig, dass sie im Ordner mit den Modulen existiert.

- **Bibliothek** ist eher ein Oberbegriff, der in der Programmierung ganz allgemein eine Sammlung von Programmfunktionen beschreibt. Eine Bibliothek kann hunderte von Modulen oder auch Paketen enthalten, so wie die Python Standardbibliothek. Im Unterschied zu einem Paket können in einer Bibliothek Funktionalitäten für ganz unterschiedliche Problembereiche zusammengefasst sein. Auch dies ist bei der Standardbibliothek von Python der Fall.

Mathematische Funktionen

Ein häufig verwendetes Modul aus der Standardbibliothek ist `math`. Darin werden die am häufigsten verwendeten mathematischen Funktionen bereitstellt. Bevor wir das Modul verwenden können, müssen wir es importieren:

```
>>> import math
```

Diese Anweisung erzeugt ein *Modulobjekt* namens `math`. Wenn wir das Modulobjekt ausgeben, erhalten wir einige Informationen über das Objekt:

```
>>> print(math)
<module 'math' (built-in)>
```

Das Modulobjekt enthält die im Modul definierten Funktionen und Variablen. Um auf eine der Funktionen zuzugreifen, müssen wir den Namen des Moduls und den Namen der Funktion angeben, getrennt durch einen Punkt. Dieses Format wird als *Punktnotation* (englisch *dot notation*) bezeichnet.

```
>>> snr = signal / rauschen
>>> dezibel = 10 * math.log10(snr)

>>> rad = 0.7
>>> hoehe = math.sin(rad)
```

Im ersten Beispiel wird der Logarithmus zur Basis 10 des Signal-Rausch-Verhältnisses (englisch *Signal to Noise Ratio*, SNR) berechnet. Das Mathematikmodul bietet auch eine Funktion namens `log`, die Logarithmen zur Basis e berechnet.

Das zweite Beispiel ermittelt den Sinus von `rad`. Der Name der Variablen ist ein Hinweis darauf, dass `sin` und die anderen trigonometrischen Funktionen (`cos`, `tan`, etc.) Argumente im Bogenmaß (englisch *radian*) annehmen. Um von Grad in Bogenmaß umzurechnen, dividieren wir durch 360 und multiplizieren mit 2π:

```
>>> alpha = 45
>>> rad = alpha / 360.0 * 2 * math.pi
>>> math.sin(rad)
0.7071067811865476
```

Der Ausdruck `math.pi` holt die Variable `pi` aus dem Mathematikmodul. Der Wert dieser Variablen ist eine Näherung von π, die auf etwa 15 Stellen genau ist.

Wenn man sich in Trigonometrie auskennt, kann man das vorherige Ergebnis überprüfen, indem man es mit der Quadratwurzel aus zwei geteilt durch zwei vergleicht:

```
>>> math.sqrt(2) / 2.0
0.7071067811865476
```

Bevor wir mit anderen Modulen der Standardbibliothek weiter machen, sollten wir nochmal einen kurzen Blick auf die `import` Anweisung richten, die wir zum Einbinden eines Moduls benötigen. Wenn Sie gerade mit der Programmierung beginnen, werden Sie sich vielleicht wundern, warum man, nachdem ein Modul importiert wurde, die Funktionen nicht „einfach so“ aufrufen kann, sondern immer den Modulnamen (bzw. das Modulobjekt) voranstellen muss? Was hier umständlich aussieht, hat einen guten Grund: Es kann gut sein, dass Sie in Ihrem Programm viele unterschiedliche Pakete und Module verwenden. Da diese Module normalerweise unabhängig voneinander entwickelt wurden, können verschiedene Module die gleichen Bezeichner verwenden. In diesem Fall würde es Namenskonflikte geben, bei einem Aufruf könnte es schlicht dazu kommen, dass Python nicht erkennen kann, welche Funktion Sie meinen. Daher Arbeiten Module mit *Namensräumen*. Beim Importieren definieren Sie das Modulobjekt, über dessen Namen Sie das Modul eindeutig erreichen. Wenn wir später *objektorientiert* programmieren, fällt das explizite Angeben des Modulobjekts meistens weg. Wir können dann Funktionen *auf Objekten* aufrufen und, da Objekte „ihre“ Funktionen kennen, wird Python die passende Funktion auswählen. Aber dazu später mehr.

Zufallszahlen

Bei gleichen Eingaben erzeugen die meisten Computerprogramme jedes Mal die gleichen Ausgaben, weshalb sie als *deterministisch* bezeichnet werden. Determinismus ist normalerweise eine gute Sache, da wir erwarten, dass die gleiche Berechnung das gleiche Ergebnis liefert. Für einige Anwendungen wollen wir jedoch, dass der Computer unvorhersehbar ist. Spiele sind ein offensichtliches Beispiel, aber es gibt noch mehr.

Ein Programm wirklich nicht-deterministisch zu machen, erweist sich als nicht so einfach, aber es gibt Möglichkeiten, es zumindest nicht-deterministisch erscheinen zu lassen. Eine davon ist, *Algorithmen* zu verwenden, die *Pseudozufallszahlen* erzeugen. Pseudozufallszahlen sind nicht wirklich zufällig, da sie durch eine deterministische Berechnung erzeugt werden, aber allein durch das Betrachten der Zahlen ist es nahezu unmöglich, sie von Zufallszahlen zu unterscheiden.

Das Modul `random` stellt Funktionen zur Verfügung, die Pseudozufallszahlen erzeugen (die von hier an einfach als „zufällig“ bezeichnet werde).

Die Funktion `random` liefert eine zufällige Fließkommazahl zwischen 0,0 und 1,0 (einschließlich 0,0, aber ausschließlich 1,0). Jedes Mal, wenn wir `random` aufrufen,

erhalten wir die nächste Zahl langen Zufallszahlenfolge. Um ein Beispiel zu sehen, führen wir diese Schleife aus:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

Dieses Programm erzeugt die folgende Liste von 10 Zufallszahlen zwischen 0,0 und bis zu, aber nicht einschließlich 1,0.

```
0.11132867921152356
0.5950949227890241
0.04820265884996877
0.841003109276478
0.997914947094958
0.04842330803368111
0.7416295948208405
0.510535245390327
0.27447040171978143
0.028511805472785867
```

Übung 1: Führen Sie das Programm auf Ihrem System aus und schauen Sie, welche Zahlen Sie erhalten. Führen Sie das Programm mehr als einmal aus und sehen Sie, welche Zahlen Sie erhalten.

Die Funktion `random` ist nur eine von vielen Funktionen, die mit Zufallszahlen umgehen. Die Funktion `randint` nimmt die Parameter `low` und `high` und gibt eine ganze Zahl zwischen `low` und `high` (einschließlich) zurück.

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Um ein Element aus einer Sequenz zufällig auszuwählen, können Sie `choice` verwenden:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Das Modul `random` bietet auch Funktionen zur Erzeugung von Zufallswerten aus kontinuierlichen Verteilungen wie Gauß, exponentiell, Gamma und ein paar mehr.

Definition neuer Funktionen

Bisher haben wir nur die Funktionen verwendet, die mit Python geliefert werden, aber es ist auch möglich, neue Funktionen hinzuzufügen. Eine *Funktionsdefinition* gibt den Namen einer neuen Funktion und die Reihenfolge der Anweisungen an, die ausgeführt werden, wenn die Funktion aufgerufen wird. Sobald wir eine Funktion definiert haben, können wir die Funktion immer wieder in unserem Programm verwenden. Hier ein einfaches Beispiel:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')
```

`def` ist ein Schlüsselwort, das anzeigt, dass es sich um eine Funktionsdefinition handelt. Der Name der Funktion ist `print_lyrics`. Die Regeln für Funktionsnamen sind die gleichen wie für Variablennamen: Buchstaben, Zahlen und der Unterstrich sind zulässig, aber das erste Zeichen darf keine Zahl sein. Wir können kein Schlüsselwort als Funktionsnamen verwenden, und wir sollten vermeiden, dass eine Variable und eine Funktion denselben Namen haben.

Die leeren Klammern hinter dem Namen zeigen an, dass diese Funktion keine Argumente annimmt. Später werden wir Funktionen bauen, die Argumente als Eingaben entgegennehmen.

Die erste Zeile der Funktionsdefinition wird *Funktionskopf* (englisch *Header*) genannt; der Rest wird *Funktionsrumpf* (englisch *Body*) genannt. Der Funktionskopf muss mit einem Doppelpunkt enden und der Funktionsrumpf muss eingerückt sein. Die Konvention besagt, dass die Einrückung immer vier Leerzeichen beträgt. Der Funktionsrumpf kann eine beliebige Anzahl von Anweisungen enthalten.

Wenn wir eine Funktionsdefinition im interaktiven Modus eingeben, gibt der Interpreter eine Ellipse `...` (Auslassungspunkte) aus, um uns darauf hinzuweisen, dass die Definition nicht vollständig ist:

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print('I sleep all night and I work all day.')
...
```

Um die Funktion zu beenden, müssen wir im interaktiven Modus eine Leerzeile eingeben. Wenn Sie ein Python Skript (also Programmcode in eine *.py*-Datei) schreiben, ist die Leerzeile zwar nicht notwendig, führt aber zu übersichtlicherem Code und sollte daher auch gesetzt werden.

Wie bei einer Variablen ist in Python auch der Bezeichner einer Funktion einfach nur ein Name. Nur, dass über diesen Namen kein Wert-Objekt (wie bei einer Variablen) erreicht wird, sondern ein *Funktionsobjekt*.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

Der **Wert** von `print_lyrics` ist ein *Funktionsobjekt*, das den Typ `function` hat. Sie könnten der Funktion im Nachhinein auch andere Namen geben. Solange diese dasselbe Funktionsobjekt benennen, kann die Funktion mit allen diesen Namen verwendet werden.

Die Syntax für den Aufruf der neuen Funktion ist die gleiche wie für die von Python bereitgestellten Funktionen, die wir bereits verwendet haben:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Sobald wir eine Funktion definiert haben, können wir sie innerhalb einer anderen Funktion verwenden. Um zum Beispiel den vorherigen Refrain zu wiederholen, könnten wir eine Funktion namens `repeat_lyrics` schreiben:

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

Und dann rufen wir `repeat_lyrics` auf:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Aber so geht der Song nicht wirklich.

Definitionen und deren Verwendung

Wenn wir die Codefragmente aus dem vorherigen Abschnitt zusammenfassen, sieht das gesamte Programm wie folgt aus:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print('I sleep all night and I work all day.')


def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()

# Code: https://tiny.one/py4de/code3/lyrics.py
```

Dieses Programm enthält zwei Funktionsdefinitionen: `print_lyrics` und `repeat_lyrics`. Funktionsdefinitionen werden genau wie andere Anweisungen ausgeführt, aber der Effekt ist, dass Funktionsobjekte erstellt werden. Die Anweisungen innerhalb der Funktion werden erst ausgeführt, wenn die Funktion aufgerufen wird. Die Funktionsdefinition selbst erzeugt also keine Ausgabe.

Wie man sich vielleicht schon denken kann, muss man eine Funktion erst definieren, bevor man sie ausführen kann. Mit anderen Worten: Die Funktionsdefinition muss vor dem ersten Aufruf erfolgen.

Übung 2: Versetzen Sie die letzte Zeile dieses Programms an den Anfang, sodass der Funktionsaufruf vor den Definitionen erscheint. Führen Sie das Programm aus und beobachten Sie, welche Fehlermeldung Sie erhalten.

Übung 3: Verschieben Sie den Funktionsaufruf wieder nach unten und verschieben Sie die Definition von `print_lyrics` hinter die Definition von `repeat_lyrics`. Was passiert, wenn Sie dieses Programm ausführen?

Programmablauf

Um sicherzustellen, dass eine Funktion vor ihrer ersten Verwendung definiert ist, müssen wir die Reihenfolge kennen, in der Anweisungen ausgeführt werden, was als *Programmablauf* bezeichnet wird. Die Ausführung beginnt immer mit der ersten Anweisung des Programms. Die Anweisungen werden nacheinander in der Reihenfolge von oben nach unten ausgeführt. Funktionsdefinitionen verändern den Ablauf des Programms nicht, aber behalten wir im Kopf, dass Anweisungen innerhalb der Funktion erst ausgeführt werden, wenn die Funktion aufgerufen wird.

Ein Funktionsaufruf ist wie ein Umweg im Programmablauf. Anstatt zur nächsten Anweisung zu gehen, springt die Ausführung zum Rumpf der Funktion, führt alle Anweisungen dort aus und kommt dann zurück, um dort weiterzumachen, wo sie aufgehört hat. Das erscheint solange simpel, bis man sich daran erinnert, dass eine Funktion eine andere aufrufen kann. Während sich das Programm mitten in einer Funktion befindet, muss es möglicherweise die Anweisungen einer anderen Funktion ausführen. Aber während diese neue Funktion ausgeführt wird, muss das Programm möglicherweise noch eine andere Funktion ausführen!

Glücklicherweise ist Python gut darin, den Überblick zu behalten, wo es sich befindet. Jedes Mal, wenn eine Funktion abgeschlossen ist, macht das Programm dort weiter, wo es in der Funktion, die es aufgerufen hat, aufgehört hat.

So springt der Programmablauf von Funktion zu Funktion und erst, wenn bei diesem Ablauf die letzte Anweisung im (Haupt-) Programm beendet ist, sind wir schließlich am Ende angekommen und das Programm *terminiert*. Was ist die Moral der Geschichte? Wenn wir ein Programm lesen, sollten wir nicht immer von oben nach unten lesen. Manchmal macht es mehr Sinn, wenn man dem Programmablauf folgt.

Parameter und Argumente

Einige der eingebauten Funktionen, die wir gesehen haben, benötigen Argumente. Wenn wir zum Beispiel `math.sin` aufrufen, übergeben wir eine Zahl als Argument. Einige Funktionen benötigen mehr als ein Argument: `math.pow` benötigt zwei, die Basis und den Exponenten.

Innerhalb der Funktion werden die Argumente Variablen zugewiesen, die *Parameter* genannt werden. Hier ist ein Beispiel für eine benutzerdefinierte Funktion, die ein Argument annimmt:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Diese Funktion weist das Argument einem Parameter namens `bruce` zu. Wenn die Funktion aufgerufen wird, gibt sie den Wert des Parameters (welcher auch immer das ist) zweimal aus.

Diese Funktion funktioniert mit jedem Wert, der mit der `print`-Funktion ausgegeben werden kann.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> import math
>>> print_twice(math.pi)
3.141592653589793
3.141592653589793
```

Die gleichen Kompositionsregeln, die für integrierte Funktionen gelten, gelten auch für benutzerdefinierte Funktionen, sodass wir jede Art von Ausdruck als Argument für `print_twice` verwenden können:

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

Das Argument wird ausgewertet, bevor die Funktion aufgerufen wird, sodass in den Beispielen die Ausdrücke `'Spam '*4` und `math.cos(math.pi)` nur einmal ausgewertet werden.

Wir können auch eine Variable als Argument verwenden:

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

Der Name der Variablen, die wir als Argument übergeben (`michael`), hat nichts mit dem Namen des Parameters (`bruce`) zu tun. Es spielt keine Rolle, wie der Wert in der aufrufenden Umgebung genannt wurde; hier in `print_twice` heißt der Parameter immer `bruce`.

Funktionen mit und ohne Rückgabewert

Einige der Funktionen, die wir verwenden, wie z. B. die mathematischen Funktionen, liefern Ergebnisse; daher nennen wir sie *Funktionen mit Rückgabewert*. Andere Funktionen, wie `print_twice`, führen eine Aktion aus, geben aber keinen Wert zurück. Sie werden auch *void-Funktionen* genannt.

Wenn wir eine Funktion mit Rückgabewert aufrufen, wollen wir fast immer etwas mit dem Ergebnis machen; zum Beispiel können wir es einer Variablen zuweisen oder als Teil eines Ausdrucks verwenden:

```
x = math.cos(rad)
phi = (math.sqrt(5) + 1) / 2
```

Wenn wir eine Funktion im interaktiven Modus aufrufen, zeigt Python das Ergebnis an:

```
>>> math.sqrt(5)
2.23606797749979
```

Aber wenn wir in einem Skript eine Funktion mit Rückgabewert aufrufen und das Ergebnis der Funktion nicht in einer Variablen speichern, ist der Rückgabewert verloren!

```
math.sqrt(5)
```

Dieses Skript berechnet die Quadratwurzel aus 5, aber da es das Ergebnis nicht in einer Variablen speichert oder das Ergebnis anzeigt, ist es nicht sehr nützlich.

Void-Funktionen können etwas auf dem Bildschirm anzeigen oder einen anderen Effekt haben, aber sie haben keinen Rückgabewert. Wenn wir versuchen, das Ergebnis einer Variablen zuzuweisen, erhalten wir einen speziellen Wert namens `None`.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

Der Wert `None` ist nicht dasselbe wie die Zeichenkette `'None'`. Er ist ein spezieller Wert, der einen eigenen Typ hat:

```
>>> print(type(None))
<class 'NoneType'>
```

Es mag Ihnen seltsam erscheinen, dass Python einen eignen Typ besitzt, der nur einen Wert `None`, also nur *das Nichts* kennt. Tatsächlich ist das aber sehr praktisch und wird von vielen Programmiersprachen so oder in ähnlicher Form verwendet. Wir können damit nämlich an verschiedensten Stellen ausdrücken, dass etwas *nicht oder noch nicht* existiert. Z. B. können wir einen Variablennamen anlegen, der aber noch keinen Wert hat. Und wir können in einem booleschen Ausdruck überprüfen, ob ein Wert existiert oder nicht.

Um ein Ergebnis aus einer Funktion heraus zurückzugeben, verwenden wir die Anweisung `return`. Wir könnten zum Beispiel eine sehr einfache Funktion namens `addtwo` erstellen, die zwei Zahlen addiert und ein Ergebnis zurückgibt.

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)

# Code: https://tiny.one/py4de/code3/addtwo.py
```

Wenn dieses Skript ausgeführt wird, wird `8` ausgeben, weil die Funktion `addtwo` mit 3 und 5 als Argumente aufgerufen wurde. Innerhalb der Funktion waren die Parameter `a` und `b` jeweils 3 und 5. Die Funktion berechnete die Summe der beiden Zahlen und legte sie in der lokalen Funktionsvariablen namens `added` ab. Dann gab sie den berechneten Wert mit der Anweisung `return` als Funktionsergebnis an die aufrufende Umgebung zurück, die der Variablen `x` zugewiesen und ausgegeben wurde.

Wozu Funktionen?

Es ist vielleicht nicht sofort klar, warum es die Mühe wert ist, ein Programm in Funktionen zu unterteilen. Aber dafür gibt es mehrere Gründe:

- Das Erstellen einer neuen Funktion gibt einem die Möglichkeit, eine Gruppe von Anweisungen zu benennen, wodurch ein Programm leichter zu lesen, zu verstehen und zu debuggen ist.
- Funktionen können ein Programm kleiner machen, indem sie sich wiederholenden Code eliminieren. Wenn wir später eine Änderung vornehmen, müssen wir diese nur an einer Stelle vornehmen.

- Wenn wir ein langes Programm in Funktionen aufteilen, können wir die Teile einzeln debuggen und sie dann zu einem funktionierenden Ganzen zusammensetzen.
- Gut durchdachte Funktionen sind oft für viele Programme nützlich. Wenn wir einmal eine geschrieben und debuggt haben, können wir sie wiederverwenden.

Im weiteren Verlauf des Buchs werden wir oft eine Funktionsdefinition verwenden, um ein Konzept zu erklären. Ein Teil der Fertigkeit beim Erstellen und Verwenden von Funktionen besteht darin, dass eine Funktion eine Idee wie „finde den kleinsten Wert in einer Liste von Werten“ richtig erfasst. Wir werden uns überlegen, welche Daten wir der Funktion übergeben müssen, wie das Problem von der Funktion gelöst werden kann und in welcher Form sie mögliche Ergebnisse zurückgeben soll.

Debugging

Wenn wir einen Texteditor zum Schreiben unserer Skripte verwenden, kann es zu Problemen mit Leerzeichen und Tabulatoren kommen. Der beste Weg, diese Probleme zu vermeiden, ist, ausschließlich Leerzeichen zu verwenden (keine Tabulatoren). Die meisten Texteditoren, die sich mit Python auskennen, tun dies standardmäßig, manche aber auch nicht.

Tabulatoren und Leerzeichen sind normalerweise unsichtbar, was die Fehlersuche erschwert. Man sollte daher versuchen, einen Editor zu finden, der die Einrückung für einen automatisch vornimmt.

Man sollte auch nicht vergessen, das Programm zu speichern, bevor man es ausführt. Manche Entwicklungsumgebungen tun dies automatisch, manche aber auch nicht. In diesem Fall ist das Programm, das man sich im Texteditor ansieht, nicht dasselbe wie das Programm, dass man ausführt. Die Fehlersuche kann sehr lange dauern, wenn man das gleiche fehlerhafte Programm immer und immer wieder ausführt! Man sollte sich vergewissern, dass der Code, den man sich ansieht, auch der Code ist, den man ausführt. Wenn man sich nicht sicher sind, fügt man etwas wie `print("hello")` an den Anfang des Programms und führt es erneut aus. Wenn man kein `hello` sehen kann, führt man nicht das richtige Programm aus!

Glossar

Algorithmus Ein allgemeines Verfahren zum Lösen einer Kategorie von Problemen.

Argument Ein Wert, der einer Funktion zur Verfügung gestellt wird, wenn die Funktion aufgerufen wird. Dieser Wert wird dem entsprechenden Parameter in der Funktion zugewiesen.

Funktionsrumpf Die Folge von Anweisungen innerhalb einer Funktionsdefinition.

deterministisch Bezieht sich auf ein Programm, das bei jedem Durchlauf das Gleiche tut, wenn es die gleichen Eingaben hat.

Punktnotation Die Syntax für den Aufruf einer Funktion in einem anderen Modul durch Angabe des Modulnamens, gefolgt von einem Punkt und dem Funktionsnamen.

Programmablauf Die Reihenfolge, in der Anweisungen während eines Programmlaufs ausgeführt werden.

Funktion Eine benannte Folge von Anweisungen, die eine nützliche Operation ausführt. Funktionen können Argumente annehmen oder nicht und können ein Ergebnis erzeugen oder nicht.

Funktionsaufruf Eine Anweisung, die eine Funktion ausführt. Sie besteht aus dem Funktionsnamen, gefolgt von einer Argumentliste.

Funktionsdefinition Eine Anweisung, die eine neue Funktion erstellt und ihren Namen, ihre Parameter und die Anweisungen, die sie ausführt, angibt.

Funktionsobjekt Ein Wert, der durch eine Funktionsdefinition erzeugt wird. Der Name der Funktion ist eine Variable, die auf ein Funktionsobjekt verweist.

Funktionskopf Die erste Zeile einer Funktionsdefinition.

Importanweisung Eine Anweisung, die eine Moduldatei liest und ein Modulobjekt erstellt.

Modulobjekt Ein durch eine `import`-Anweisung erzeugter Wert, der den Zugriff auf die in einem Modul definierten Daten und den Code ermöglicht.

Parameter Ein Name, der innerhalb einer Funktion verwendet wird, um auf den als Argument übergebenen Wert zu verweisen.

pseudozufällig Bezieht sich auf eine Folge von Zahlen, die nur scheinbar zufällig sind, aber von einem deterministischen Programm erzeugt werden.

Rückgabewert Das Ergebnis einer Funktion. Wenn ein Funktionsaufruf als Ausdruck verwendet wird, ist der Rückgabewert der Wert des Ausdrucks.

void, Funktion ohne Rückgabewert Eine Funktion, die keinen Wert zurückgibt.

Übungen

Übung 4: Was ist der Zweck des Schlüsselworts `def` in Python?

a) Es ist Slang und bedeutet „der folgende Code ist wirklich cool“
b) Es zeigt den Beginn einer Funktion an
c) Es zeigt an, dass der folgende eingerückte Codeabschnitt für später gespeichert werden soll
d) b und c sind beide wahr
e) Keiner der oben genannten Punkte

Übung 5: Was wird das folgende Python-Programm ausgeben?

```
def fred():
   print("Zap")

def jane():
   print("ABC")

jane()
```

```
fred()
jane()
```

a) Zap ABC jane fred jane
b) Zap ABC Zap
c) ABC Zap jane
d) ABC Zap ABC
e) Zap Zap Zap

Übung 6: Schreiben Sie ihr Programm zur Lohnberechnung (mit 1,4-fachem Stundenlohn bei Überstunden) um. Ergänzen Sie eine Funktion `lohnberechnung`, welche die beiden Parameter `arbeitsstunden` und `stundenlohn` entgegennimmt.

```
Anzahl Arbeitsstunden: 45
Stundenlohn: 10
Monatsgehalt: 475.0
```

Übung 7: Schreiben Sie das Benotungsprogramm aus dem vorigen Kapitel neu, indem Sie eine Funktion namens `notenberechnung` verwenden, die eine Punktzahl als Parameter annimmt und eine Note zurückgibt.

```
Punkte   Note
>= 0.9     1
>= 0.8     2
>= 0.7     3
>= 0.6     4
 < 0.6     5
```

Führen Sie das Programm wiederholt aus, um die verschiedenen Werte für die Eingabe zu testen:

```
Punkte eingeben: 0.95
1
```

```
Punkte eingeben: perfekt
Falsche Punktanzahl
```

```
Punkte eingeben: 10.0
Falsche Punktanzahl
```

```
Punkte eingeben: 0.75
3
```

```
Punkte eingeben: 0.5
5
```

Kapitel 5

Iteration

Immer und immer wieder die gleichen Arbeitsschritte zu tun, ist für uns Menschen eine missliebige Vorstellung. Computer hingegen sind wahre Meister der Wiederholung. Ein Großteil Ihrer Laufzeit verbringen Programme damit, bestimmte Aufgabenschritte zu wiederholen. In diesem Kapitel geht es genau um solche *Programmschleifen*, mit denen man Python anweist, bestimmte Anweisungen zu wiederholen. Wir werden zwei unterschiedliche Arten solcher Schleifen kennenlernen. Die eine Wiederholt einen Anweisungsblock, solange eine vom Programmierer angegebene Bedingung erfüllt ist. Das ist ein wenig wie beim Zähneputzen; solange die drei Minuten noch nicht um sind, putzen Sie weiter. Die andere Art von Schleifen ist dazu da, Folgen von Elementen abzuarbeiten. Ein möglicher Vergleich hier ist eine Einkaufsliste, die Sie von oben nach unten Abarbeiten; wenn Sie den letzten Artikel auf der Liste in den Einkaufswagen gelegt haben, sind Sie fertig.

Aktualisieren von Variablen

Wenn wir bestimmte Dinge wiederholen wollen, müssen wir meist mitzählen. Auch in Programmen wird häufig gezählt, und zwar mithilfe von Variablen. Wenn wir mit einer Variable zählen wollen, müssen wir ihren Wert aktualisieren, wobei der neue Wert vom alten abhängt.

```
x = x + 1
```

Das bedeutet: „Ermittle den aktuellen Wert von `x`, addiere 1 und aktualisiere dann `x` mit diesem neuen Wert.“ Bereits in Kapitel 2 haben wir gesehen, dass es kein Problem ist, wenn hier `x` auf beiden Seiten des Gleichheitszeichens auftritt. Wir werten zuerst die rechte Seite aus, erhalten so einen Wert und weisen diesen dann auf den Namen `x` zu.

Wenn man versucht, eine Variable zu aktualisieren, die nicht existiert, erhält man einen Fehler, da Python versucht, die rechte Seite des Ausdrucks auszuwerten *bevor* `x` überhaupt einen Wert hat:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Bevor man eine Variable aktualisieren kann, muss man sie also *initialisieren*, normalerweise mit einer einfachen Zuweisung:

```
>>> x = 0
>>> x = x + 1
```

Das Aktualisieren einer Variablen durch Addieren von 1 wird als *Inkrementieren* bezeichnet; das Subtrahieren von 1 wird als *Dekrementieren* bezeichnet.

Die `while`-Schleife

Computer werden oft eingesetzt, um sich wiederholende Aufgaben zu automatisieren. Gleiche oder ähnliche Aufgaben fehlerfrei zu wiederholen, ist etwas, das Computer gut und Menschen schlecht können. Weil Iteration so häufig vorkommt, bietet Python mehrere Sprachfunktionen, um sie zu erleichtern.

Eine Form der Iteration in Python ist die `while`-Schleife. Hier ist ein einfaches Programm, das von fünf herunterzählt und dann `'Blastoff!'` sagt.

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

Wir können die `while`-Schleife fast wörtlich lesen. Sie bedeutet: „Solange `n` größer als 0 ist, zeige den Wert von `n` an und dekrementiere dann den Wert von `n` um 1. Wenn du zu 0 kommst, verlasse die `while`-Schleife und gebe `'Blastoff!'` aus.“

Formaler ausgedrückt, ist das hier der Ablauf der Ausführung einer `while`-Schleife:

1. Werte die Bedingung aus und liefere `True` oder `False`.
2. Wenn die Bedingung falsch ist, wird die `while`-Schleife verlassen und die Ausführung bei der nächsten Anweisung fortgesetzt.
3. Wenn die Bedingung wahr ist, führe den Schleifenrumpf aus und gehen dann zurück zu Schritt 1.

Diese Art von Ablauf wird als *Schleife* bezeichnet, weil der dritte Schritt eine Schleife zurück zum Anfang bildet. Jedes Mal, wenn der Schleifenrumpf ausgeführt wird, nennen wir das eine *Iteration*. Für die obige Schleife würden wir sagen: „Sie hatte fünf Iterationen“, was bedeutet, dass der Rumpf der Schleife fünfmal ausgeführt wurde.

Der Schleifenrumpf soll den Wert einer oder mehrerer Variablen so verändern, dass die Bedingung schließlich falsch wird und die Schleife beendet wird. Wir nennen die Variable, die sich bei jeder Ausführung der Schleife ändert und steuert, wann die Schleife beendet ist, die *Iterationsvariable* beziehungsweise den *Schleifenzähler*. Wenn es keinen Schleifenzähler gibt, wird die Schleife gegebenenfalls ewig wiederholt, was zu einer *Endlosschleife* führt.

Im Fall von `n` können wir beweisen, dass die Schleife endet, weil wir wissen, dass der Wert von `n` endlich ist, und wir können sehen, dass der Wert von `n` jedes Mal, wenn wir die Schleife durchlaufen, kleiner wird, sodass wir schließlich bei 0 ankommen müssen. In vielen anderen Fällen ist eine Schleife häufig unendlich, zum Beispiel weil sie überhaupt keinem Schleifenzähler hat.

Manchmal möchte man erreichen, dass eine Schleife mitten im Schleifenrumpf abgebrochen wird. In diesem Fall können wir absichtlich eine Endlosschleife schreiben (zum Beispiel mittels `True` als einziger Schleifenbedingung) und dann die Anweisung `break` verwenden, um *sofort* aus der Schleife herauszuspringen.

Diese Schleife ist offensichtlich eine *Endlosschleife*, weil der logische Ausdruck der `while`-Schleife die Konstante `True` ist:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

Wenn wir den Fehler machen und diesen Code ausführen, werden wir schnell lernen, wie wir einen durchgebrannten Python-Prozess auf unserem System stoppen oder herausfinden können, wo der Ausschaltknopf unseres Computers ist. Dieses Programm wird ewig laufen oder bis die Batterie leer ist, weil der logische Ausdruck am Anfang der Schleife immer wahr ist, weil der Ausdruck der konstante Wert `True` ist.

Obwohl es sich hierbei um eine dysfunktionale Endlosschleife handelt, können wir dieses Muster dennoch verwenden, um nützliche Schleifen zu konstruieren, solange wir sorgfältig Code in den Körper der Schleife einfügen, um die Schleife explizit mit `break` zu verlassen, wenn wir die Abbruchbedingung erreicht haben.

Nehmen wir zum Beispiel an, dass wir Eingaben des Benutzers entgegennehmen wollen, bis man `done` eingibt. Wir könnten schreiben:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

```
# Code: https://tiny.one/py4de/code3/copytildone1.py
```

Die Schleifenbedingung ist `True`, was immer wahr ist, sodass die Schleife wiederholt durchlaufen wird, bis sie auf die `break`-Anweisung trifft.

Bei jedem Durchlauf fordert das Programm den Benutzer mit einer spitzen Klammer zur Eingabe auf. Wenn der Benutzer `done` eingibt, wird die Schleife mit der Anweisung `break` verlassen. Andernfalls gibt das Programm aus, was auch immer der Benutzer zuvor eingegeben hat, und kehrt an den Anfang der Schleife zurück:

```
> hello there
hello there
> finished
finished
> done
Done!
```

Diese Art, `while`-Schleifen zu schreiben, ist üblich, weil wir die Bedingung an jeder Stelle der Schleife überprüfen können (nicht nur am Anfang) und weil wir die Abbruchbedingung aktiv („stoppe, wenn dies passiert“) statt passiv („mache weiter, bis jenes passiert.“) ausdrücken können.

Abbrechen einer Iteration mit `continue`

Manchmal befinden wir uns in einer Iteration einer Schleife und möchten die aktuelle Iteration beenden und sofort zur nächsten Iteration springen. In diesem Fall können wir die Anweisung `continue` verwenden, um zur nächsten Iteration zu springen, ohne den Rest des Schleifenrumpfes in der aktuellen Iteration durchlaufen zu müssen.

Hier ist ein Beispiel für eine Schleife, die eine Benutzereingabe sofort wieder ausgibt, bis der Benutzer `done` eingibt. Zeilen, die mit einer Raute beginnen, werden jedoch ignoriert.

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: https://tiny.one/py4de/code3/copytildone2.py
```

Hier ist ein Beispiellauf dieses neuen Programms mit hinzugefügtem `continue`.

```
> hello there
hello there
> # don't print this
```

```
> print this!
print this!
> done
Done!
```

Alle Zeilen werden ausgegeben, außer diejenigen, die mit dem Rautezeichen beginnt, denn wenn die Anweisung `continue` ausgeführt wird, bricht die aktuelle Iteration ab und die Programmausführung springt zurück zur Anweisung `while`, um die nächste Iteration zu starten, und überspringt damit die Anweisung `print`.

for-Schleifen

Manchmal wollen wir eine *Folge* von Dingen in einer Schleife durchlaufen, z. B. eine Liste von Wörtern, die Zeilen in einer Datei oder eine Liste von Zahlen. Wenn wir eine Liste von Dingen haben, die in einer Schleife durchlaufen werden sollen, können wir eine *bereichsbasierte* Schleife nutzen. Die `for`-Schleife durchläuft eine bekannte Folge von Elementen mit so vielen Iterationen, wie es Elemente in der Folge gibt.

Die Syntax einer `for`-Schleife ähnelt der `while`-Schleife insofern, als es eine `for`-Anweisung und einen Schleifenrumpf gibt:

```
friends = ['Anna', 'Ben', 'Carla']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

In Python ausgedrückt, ist die Variable `friends` eine Liste[1] von drei Zeichenketten und die `for`-Schleife durchläuft die Liste und führt den Rumpf einmal für jede der drei Zeichenketten in der Liste aus, was zu dieser Ausgabe führt:

```
Happy New Year: Anna
Happy New Year: Ben
Happy New Year: Carla
Done!
```

Die Übersetzung dieser `for`-Schleife in unsere Sprache ist nicht so direkt wie die der `while`-Schleife, aber wenn man sich `friends` als eine *Folge* oder als eine *Menge* vorstellt, könnte es etwa so lauten: „Führe die Anweisungen im Rumpf der Schleife einmal für jeden Freund aus der Folge aller Freunde aus."

Wenn man sich die `for`-Schleife ansieht, sind *for* und *in* reservierte Python-Schlüsselwörter, und `friend` und `friends` sind Variablen.

```
for friend in friends:
    print('Happy New Year:', friend)
```

[1] Wir werden Listen in einem späteren Kapitel genauer untersuchen.

Insbesondere ist `friend` die *Iterationsvariable* für die `for`-Schleife. Die Variable `friend` ändert sich bei jeder Iteration der Schleife und steuert, wann die `for`-Schleife beendet ist. Die *Iterationsvariable* geht nacheinander durch die drei Zeichenketten, die in der Variablen `friends` gespeichert sind.

Die Unterscheidung, ob es sich um eine *Folge* oder um eine *Menge* handelt, macht beim Durchlaufen der Daten einen Unterschied. Bei einer Folge sind die einzelnen Elemente geordnet und die Reihenfolge der Elemente in der Folge wird beim Durchlaufen beibehalten. In einer Menge sind die Elemente nicht geordnet. Die `for`-Schleife betrachtet also jedes Element der Menge genau einmal, in welcher Reihenfolge die Elemente ausgewählt werden, ist dabei nicht genau festgelegt.

Typische Anwendungen von Schleifen

Oft verwenden wir eine `for`- oder `while`-Schleife, um eine Liste von Elementen oder den Inhalt einer Datei zu durchlaufen, und wir suchen nach etwas wie dem größten oder kleinsten Wert einer Liste, die wir durchlaufen.

Diese Schleifen werden in der Regel so konstruiert:

- Initialisierung einer oder mehrerer Variablen vor Beginn der Schleife
- Durchführen von Berechnungen an jedem Element, wobei möglicherweise die Variablen im Schleifenkörper geändert werden
- Betrachten der resultierenden Variablen, wenn die Schleife beendet ist

Wir werden eine Liste von Zahlen verwenden, um die typischen Anwendungen von Schleifen zu demonstrieren.

Zählen und Summieren

Um zum Beispiel die Anzahl der Elemente in einer Liste zu zählen, würden wir die folgende `for`-Schleife schreiben:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

Wir setzen die Variable `count` auf 0, bevor die Schleife beginnt, dann schreiben wir eine `for`-Schleife, die die Liste der Zahlen durchläuft. Unsere *Iterations*-Variable heißt `itervar` und obwohl wir `itervar` nicht in der Schleife verwenden, steuert sie die Schleife und bewirkt, dass der Schleifenkörper für jeden der Werte in der Liste einmal ausgeführt wird.

Im Schleifenkörper addieren wir für jeden der Werte in der Liste 1 zum aktuellen Wert von `count`. Während die Schleife ausgeführt wird, entspricht der Wert von `count` der Anzahl der Werte, die wir bisher gesehen haben.

Sobald die Schleife abgeschlossen ist, ist der Wert von `count` die Gesamtzahl der Elemente. Die Gesamtzahl fällt uns am Ende der Schleife „in den Schoß“. Wir konstruieren die Schleife so, dass wir das haben, was wir wollen, wenn die Schleife beendet ist.

Eine weitere ähnliche Schleife, die die Summe einer Menge von Zahlen berechnet, sieht wie folgt aus:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Summe: ', total)
```

In dieser Schleife benutzen wir die *Iterationsvariable* tatsächlich. Anstatt wie in der vorherigen Schleife einfach eine 1 zu `count` zu addieren, fügen wir bei jeder Schleifeniteration die aktuelle Zahl (3, 41, 12, usw.) zur laufenden Summe hinzu. Wenn wir uns die Variable `total` vorstellen, enthält sie die laufende Summe der bisherigen Werte. Bevor die Schleife beginnt, ist `total` also 0, weil wir noch keine Werte gesehen haben, während der Schleife ist `total` die laufende Summe, und am Ende der Schleife ist `total` die Gesamtsumme aller Werte in der Liste.

Während die Schleife ausgeführt wird, akkumuliert `total` die Summe der Elemente; eine Variable, die auf diese Weise verwendet wird, nennt man manchmal einen *Akkumulator*.

Weder die Zählschleife noch die Summierschleife sind in der Praxis besonders nützlich, da es eingebaute Funktionen `len()` und `sum()` gibt, die die Anzahl der Elemente in einer Liste bzw. die Summe der Elemente in der Liste berechnen.

Maximum und Minimum ermitteln

Um den größten Wert in einer Liste oder Folge zu finden, konstruieren wir die folgende Schleife:

```
largest = None
print('Maximum zu Beginn:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest:
        largest = itervar
    print('Betrachte Wert:', itervar, 'Aktuelles Maximum:', largest)
print('Das Maximum ist:', largest)
```

Wenn das Programm ausgeführt wird, sieht die Ausgabe wie folgt aus:

```
Maximum zu Beginn: None
Betrachte Wert: 3 Aktuelles Maximum: 3
Betrachte Wert: 41 Aktuelles Maximum: 41
Betrachte Wert: 12 Aktuelles Maximum: 41
Betrachte Wert: 9 Aktuelles Maximum: 41
Betrachte Wert: 74 Aktuelles Maximum: 74
Betrachte Wert: 15 Aktuelles Maximum: 74
Das Maximum ist: 74
```

Die Variable `largest` kann man sich am besten als den „größten Wert, den wir bisher gesehen haben“ vorstellen. Vor der Schleife setzen wir `largest` auf die Konstante `None`. Die Konstante `None` ist ein spezieller konstanter Wert, den wir in einer Variablen speichern können, um die Variable als „leer“ zu markieren.

Bevor die Schleife beginnt, ist der größte Wert, den wir bisher gesehen haben, `None`, da wir noch keine Werte gesehen haben. Wenn während der Ausführung der Schleife `largest None` ist, nehmen wir den ersten Wert, den wir sehen, als den bisher größten. Man kann sehen, dass in der ersten Iteration, wenn der Wert von `itervar` 3 ist, wir sofort `largest` auf 3 setzen (weil `largest None` ist).

Nach der ersten Iteration ist `largest` nicht mehr `None`, sodass der zweite Teil des zusammengesetzten logischen Ausdrucks, der `itervar > largest` prüft, nur ausgelöst wird, wenn wir einen Wert sehen, der größer als der „bisher größte“ ist. Wenn wir einen neuen „noch größeren“ Wert sehen, weisen wir diesen neuen Wert `largest` zu. Man kann in der Programmausgabe sehen, dass `largest` von 3 über 41 bis 74 fortschreitet.

Am Ende der Schleife haben wir alle Werte überprüft und die Variable `largest` enthält nun den größten Wert in der Liste.

Um die kleinste Zahl zu berechnen, ist der Code sehr ähnlich, mit einer kleinen Änderung:

```
smallest = None
print('Before:', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest:', smallest)
```

Auch hier ist `smallest` der „bisher kleinste“ Wert vor, während und nach der Ausführung der Schleife. Wenn die Schleife beendet ist, enthält `smallest` den kleinsten Wert in der Liste.

Wiederum wie beim Zählen und Summieren machen die eingebauten Funktionen `max()` und `min()` das Schreiben dieser exakten Schleifen überflüssig.

Das Folgende Codefragment ist eine einfache Version der in Python eingebauten Funktion `min()`:

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

In dieser Funktion haben wir alle `print`-Anweisungen entfernt, damit sie äquivalent zur Funktion `min` ist, die bereits in Python eingebaut ist.

Debugging

Wenn wir anfangen, größere Programme zu schreiben, werden wir möglicherweise mehr Zeit mit der Fehlersuche verbringen. Mehr Code bedeutet mehr Möglichkeiten, einen Fehler zu machen und mehr Stellen, an denen sich Bugs verstecken können.

Eine Möglichkeit, die Debugging-Zeit zu verkürzen, ist das „Debugging durch Bisektion“ (*Halbierung*). Wenn das Programm z. B. 100 Zeilen enthält und man diese nacheinander überprüft, würde dies 100 Schritte erfordern.

Versuchen wir stattdessen, das Problem in zwei Hälften zu teilen. Suchen wir in der Mitte des Programms oder in dessen Nähe nach einem Zwischenwert, den Sie überprüfen können. Wir fügen eine `print`-Anweisung (oder etwas anderes, das eine überprüfbare Wirkung hat) hinzu und führen das Programm aus.

Wenn die Überprüfung an dieser Stelle falsch ist, muss das Problem in der ersten Hälfte des Programms liegen. Wenn sie korrekt ist, liegt das Problem in der zweiten Hälfte.

Jedes Mal, wenn wir eine solche Prüfung durchführen, halbieren wir die Anzahl der Zeilen, die wir durchsuchen müssen. Nach sechs Schritten (was viel weniger als 100 ist), wären wir auf eine oder zwei Codezeilen runter, zumindest theoretisch.

In der Praxis ist es nicht immer klar, was die „Mitte des Programms“ ist und nicht immer möglich, dies zu überprüfen. Es macht keinen Sinn, Zeilen zu zählen und die genaue Mitte zu finden. Überlegen wir stattdessen, an welchen Stellen im Programm es Fehler geben könnte und an welchen Stellen es einfach ist, eine Prüfung durchzuführen. Wir wählen dann eine Stelle, bei der wir denken, dass die Chancen etwa gleich groß sind, dass der Fehler vor oder nach der Prüfung liegt.

Glossar

Akkumulator Eine Variable, die in einer Schleife zum Aufaddieren oder Akkumulieren eines Ergebnisses verwendet wird.

Schleifenzähler Eine Variable, die in einer Schleife verwendet wird, um die Anzahl der Male zu zählen, die etwas passiert ist. Wir initialisieren einen Zähler auf Null und inkrementieren den Zähler dann jedes Mal, wenn wir etwas „zählen“ wollen.

Dekrementieren Eine Aktualisierung, die den Wert einer Variablen verringert.

Initialisierung Eine Zuweisung, die einer Variablen einen Anfangswert zuweist (Anfangswertzuweisung).

Inkrementieren Eine Aktualisierung, die den Wert einer Variablen erhöht (häufig um eins).

Endlosschleife Eine Schleife, in der die Abbruchbedingung nie erfüllt ist oder für die es keine Abbruchbedingung gibt.

Iteration Wiederholte Ausführung einer Reihe von Anweisungen unter Verwendung einer Funktion, die sich selbst aufruft, oder einer Schleife.

Übungen

Übung 1: Schreiben Sie ein Programm, das wiederholt Zahlen einliest, bis der Benutzer den Befehl `done` eingibt. Sobald `done` eingegeben wurde, geben Sie die Summe, die Anzahl und den Durchschnitt der Zahlen aus. Wenn der Benutzer etwas anderes als eine Zahl eingibt, erkennen Sie seinen Fehler mit `try` und `except` und geben eine Fehlermeldung aus und springen zur nächsten Zahl.

```
Bitte eine Zahl eingeben: 4
Bitte eine Zahl eingeben: 5
Bitte eine Zahl eingeben: sechs
Ungueltige Eingabe
Bitte eine Zahl eingeben: 7
Bitte eine Zahl eingeben: done
16 3 5.333333333333333
```

Übung 2: Schreiben Sie ein weiteres Programm, das eine Liste von Zahlen wie oben abfragt und am Ende sowohl das Maximum als auch das Minimum der Zahlen anstelle des Durchschnitts ausgibt.

Kapitel 6

Zeichenketten

Bei typischen Informatik-Begriffen wie „Datenverarbeitung“ oder auch „Rechner“ denkt man womöglich als Erstes an das Rechnen mit Zahlen. Tatsächlich machen aber, quantitativ gesehen, numerische Daten nur einen kleinen Teil der Daten aus, die wir mit Computern verarbeiten können. Viele Informationen liegen in Form von geschrieben Texten vor, die von unseren Programmen ausgewertet werden. In diesem Kapitel geht es um Text-Daten, die sogenannten Zeichenketten. Wir werden sehen, wie man in Python Zeichenketten anlegt und vor allem, wie man mit ihnen umgehen kann.

Noch einen kleinen Tipp vorab: vieles von dem, was man mit Zeichenketten machen kann, funktioniert genau so mit Folgen von völlig andren Elementen. Wenn Sie in diesem Kapitel genau hinsehen, kommt Ihnen das also in den folgenden umso mehr zugute.

Was ist eine Zeichenkette?

Eine Zeichenkette (englisch *String*) ist eine Folge einzelner Zeichen (englisch *Character*). Wir können die Folge von Zeichen einfach „wie Text“ im Programm angeben. Wichtig ist nur, dass sie in Anführungszeichen gesetzt wird, denn sonst würde Python die Zeichen als Bezeichner (also z. B. als Variable) deuten.

Python lässt übrigens sowohl einfache (`'...'`), als auch die im Deutschen verwendeten doppelten (`"..."`) Anführungszeichen zu. Es macht also keinen Unterschied, ob Sie die Zeichenkette als `'Hallo Welt'` oder als `"Hallo Welt"` angeben.

Auf den ersten Blick erscheint es überflüssig zu sein, zwei Schreibweisen für den gleichen Zweck zu haben. Es gibt aber mindestens ein gutes Gegenargument: Stellen Sie sich vor, sie möchten ein Anführungszeichen (egal ob doppelt oder einfach) in einer Zeichenkette verwenden. Python erlaubt es nun, dass Sie dasjenige Anführungszeichen zum Einfassen der Zeichen verwenden, dass Sie *in* der Zeichenkette nicht benötigen.

```
>>> text1 = 'Sie rief ihnen "Hallo" zu.'
```

```
>>> text2 = "Hello it's me"
>>> text3 = 'What\'s up?'
```

`text1` verwendet einfache Anführungszeichen, um im Satz die wörtliche Rede „Hallo“ in doppelten Anführungszeichen darstellen zu können. In `text2` verwenden wir doppelten Anführungszeichen, da wir im Text ein Apostroph setzen wollen. `text3` verwendet nur einfach Anführungszeichen, obwohl im Text ein Apostroph vorkommt. Wir müssen dieses Apostrophzeichen mit einem sogenannten *Maskierungszeichen* versehen. Python verwendet hier, wie die meisten Programmiersprachen den Backslash \. Dieses Sonderzeichen verhindert, dass das nachfolgende Zeichen vom ausführenden Programm als Funktionszeichen gesehen wird. Der Apostroph soll hier also nicht den String schließen, sondern als wirklicher Apostroph in der Zeichenkette erscheinen.

Es steht Ihnen also frei, einfache oder doppelten Anführungszeichen zu verwenden. Die Konvention, die sich bei vielen Python-Programmierern eingebürgert hat, ist einfache Anführungszeichen für einzelne Worte oder Begriffe zu verwenden und doppelte Anführungszeichen für ganze Sätze.

Da eine Zeichenkette eine Folge (einzelner) Zeichen ist, können wir auch auf die einzelnen Elemente dieser Folge zugreifen. Dies funktioniert mit dem indexbasierten Zugriffsoperator (auch *Klammeroperator* genannt):

```
>>> frucht = 'Banane'
>>> zeichen = frucht[1]
```

Die zweite Anweisung extrahiert das Zeichen an Indexposition 1 aus der Variablen `frucht` und weist es der Variablen `zeichen` zu.

Der Ausdruck in eckigen Klammern wird als *Index* bezeichnet. Der Index gibt an, auf welches Zeichen in der Sequenz zugegriffen werden soll. Aber wir bekommen vielleicht nicht das, was wir erwarten:

```
>>> print(zeichen)
a
```

Für die meisten Menschen ist der erste Buchstabe von „Banane“ eigentlich ein „B“, aber nicht „a“. Aber in Python ist der Index ein Offset vom Anfang der Zeichenkette, und der Offset des ersten Buchstabens ist 0.

```
>>> zeichen = frucht[0]
>>> print(zeichen)
B
```

So ist „B“ der nullte Buchstabe von „Banane“, „a“ der erste und „n“ der zweite Buchstabe.

Man kann einen beliebigen Ausdruck, einschließlich Variablen und Operatoren, als Index verwenden, aber der Wert des Index muss eine ganze Zahl sein. Sonst erhält man:

```
>>> zeichen = frucht[1.5]
TypeError: string indices must be integers
```

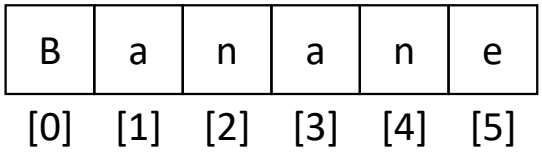

Abbildung 6.1: Indizes im String

Länge einer Zeichenkette

`len` ist eine eingebaute Funktion, die die Anzahl der Zeichen in einer Zeichenkette zurückgibt:

```
>>> frucht = 'Banane'
>>> len(frucht)
6
```

Um den letzten Buchstaben einer Zeichenkette zu erhalten, könnte man versucht sein, etwas wie das hier zu tun:

```
>>> length = len(frucht)
>>> last = frucht[length]
IndexError: string index out of range
```

Der Grund für den `IndexError` ist, dass es keinen Buchstaben in „Banane“ mit dem Index 6 gibt. Da wir bei null angefangen haben zu zählen, sind die sechs Buchstaben von 0 bis 5 nummeriert. Um das letzte Zeichen zu erhalten, müssen wir 1 von `length` subtrahieren:

```
>>> last = frucht[length-1]
>>> print(last)
e
```

Alternativ können wir auch negative Indizes verwenden, die vom Ende der Zeichenkette rückwärts zählen. Der Ausdruck `frucht[-1]` ergibt den letzten Buchstaben, `frucht[-2]` den vorletzten und so weiter.

Traversieren einer Zeichenkette

Bei vielen Berechnungen wird eine Zeichenkette Zeichen für Zeichen verarbeitet. Oft beginnen sie am Anfang, wählen jedes Zeichen der Reihe nach aus, machen etwas damit und fahren bis zum Ende fort. Dieses Verarbeitungsmuster wird als *Traversierung* bezeichnet. Eine Möglichkeit, eine solche Traversierung zu implementieren, ist eine `while`-Schleife:

```
index = 0
while index < len(frucht):
    zeichen = frucht[index]
    print(zeichen)
    index = index + 1
```

Diese Schleife durchläuft die Zeichenkette und zeigt jeden Buchstaben einzeln in einer Zeile an. Die Schleifenbedingung ist `index < len(frucht)`, wenn also `index` gleich der Länge der Zeichenkette ist, ist die Bedingung falsch und der Schleifenrumpf wird nicht mehr ausgeführt. Das letzte Zeichen, auf das zugegriffen wird, ist dasjenige mit dem Index `len(frucht)-1`, also das letzte Zeichen in der Zeichenkette.

Übung 1: Schreiben Sie eine `while`-Schleife, die beim letzten Zeichen in der Zeichenkette beginnt und sich rückwärts bis zum ersten Zeichen in der Zeichenkette vorarbeitet, wobei jeder Buchstabe in einer eigenen Zeile ausgegeben wird, natürlich rückwärts.

Eine andere Möglichkeit, einen Traversierung zu schreiben, ist mit einer `for`-Schleife:

```
for char in frucht:
    print(char)
```

Jedes Mal, wenn die Schleife durchlaufen wird, wird das nächste Zeichen in der Zeichenkette der Variablen `char` zugewiesen. Die Schleife wird fortgesetzt, bis keine Zeichen mehr übrig sind.

Der slice-Operator

Ein Segment einer Zeichenkette wird im Englischen als *Slice* oder *Substring* (deutsch Teilzeichenkette) bezeichnet. Die Auswahl einer Teilzeichenkette funktioniert ähnlich wie die Auswahl eines Zeichens:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

Der Operator gibt den Teil der Zeichenkette vom n-ten Zeichen bis zum m-ten Zeichen zurück, einschließlich des ersten, aber *ausschließlich* des letzten Zeichens. Die erste `print`-Anweisung im obigen Beispiel druckt also die Zeichen 0 bis 4 des Strings, das zweite `print` druckt die Zeichen 6 bis 11.

Wenn man den ersten Index (vor dem Doppelpunkt) weglässt, beginnt die Teilzeichenkette am Anfang der Zeichenkette. Wenn man den zweiten Index ebenfalls weglässt, geht die Teilzeichenkette bis zum Ende der Zeichenkette:

```
>>> frucht = 'Banane'
>>> frucht[:3]
'Ban'
>>> frucht[3:]
'ane'
```

Wenn der erste Index größer oder gleich dem zweiten ist, ist das Ergebnis eine *leere Zeichenkette*, dargestellt durch zwei Anführungszeichen:

```
>>> frucht = 'Banane'
>>> frucht[3:3]
''
```

Eine leere Zeichenkette enthält keine Zeichen und hat die Länge 0, aber ansonsten verhält sie sich genauso wie jede andere Zeichenkette.

Übung 2: Wenn `frucht` eine Zeichenkette ist, was bewirkt dann `frucht[:]`?

Zeichenketten sind unveränderlich

Es ist verlockend, den Operator auf der linken Seite einer Zuweisung zu verwenden, mit der Absicht, ein Zeichen in einer Zeichenkette zu ändern. Zum Beispiel:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

Das „Objekt“ ist in diesem Fall die Zeichenkette und das „Element“ ist das Zeichen, das wir versucht haben zuzuweisen. Im Moment ist ein *Objekt* dasselbe wie ein Wert, aber wir werden diese Definition später verfeinern. Ein *Element* ist einer der Werte in einer Sequenz.

Der Grund für den Fehler ist, dass Zeichenketten *unveränderlich* sind. Dies bedeutet, dass wir eine vorhandene Zeichenkette nicht ändern können. Das Beste, was wir tun können, ist, eine neue Zeichenfolge zu erstellen, die eine Variation der ursprünglichen Zeichenfolge ist:

```
>>> greeting = "Hello, world!"
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

In diesem Beispiel wird ein neuer Anfangsbuchstabe an ein Stück von `greeting` angehängt. Es hat keine Auswirkung auf die ursprüngliche Zeichenfolge.

Zählen mit Schleifen

Das folgende Programm zählt, wie oft der Buchstabe „a“ in einer Zeichenkette vorkommt:

```
word = 'Banane'
count = 0
for zeichen in word:
    if zeichen == 'a':
        count = count + 1
print(count)
```

Dieses Programm demonstriert ein Vorgehen, das man als *Zähler* bezeichnet könnte. Die Variable `count` wird mit 0 initialisiert und dann jedes Mal inkrementiert, wenn ein `'a'` gefunden wird. Wenn die Schleife beendet wird, enthält `count` das Ergebnis: die Gesamtzahl der „a“.

Übung 3: Lagern Sie diesen Code in eine Funktion namens `count` aus, und verallgemeinern Sie diese so, dass sie die Zeichenkette und den Buchstaben als Argumente akzeptiert.

Der `in`-Operator

Das Schlüsselwort `in` ist ein boolescher Operator, der zwei Zeichenketten annimmt und `True` zurückgibt, wenn die erste Zeichenkette als Teilzeichenkette in der zweiten erscheint:

```
>>> 'an' in 'Banane'
True
>>> 'anna' in 'Banane'
False
```

Vergleich von Zeichenketten

Die Vergleichsoperatoren arbeiten mit Zeichenketten. Um zu sehen, ob zwei Zeichenketten gleich sind:

```
if word == 'Banane':
    print('Genau, Banane!')
```

Andere Vergleichsoperationen sind nützlich, um Wörter in alphabetische Reihenfolge zu bringen:

```
if word < 'banane':
    print('Dein Wort,' + word + ', kommt vor banane.')
elif word > 'Banane':
    print('Dein Wort,' + word + ', kommt nach banane.')
else:
    print('Genau, banane!')
```

Python geht mit Groß- und Kleinbuchstaben nicht so um, wie es Menschen tun. Alle Großbuchstaben kommen vor allen Kleinbuchstaben, also:

```
Dein Wort,Traube, kommt vor banane.
```

Eine gängige Methode, dieses Problem zu beheben, besteht darin, Zeichenketten in ein Standardformat zu konvertieren, z. B. in Kleinbuchstaben, bevor der Vergleich durchgeführt wird.

Funktionen von Zeichenketten

Zeichenketten sind ein Beispiel für *Objekte* in Python. Ein Objekt enthält sowohl Daten (in diesem Beispiel die eigentliche Zeichenkette selbst) als auch sogenannte *Methoden*, also Funktionen, die in das Objekt eingebaut sind und jeder *Instanz* des Objekts zur Verfügung stehen.

Python hat eine Funktion namens `dir`, die die verfügbaren Methoden für ein Objekt auflistet. Die Funktion `type` zeigt den Typ eines Objekts und die Funktion `dir` seine Methoden.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

Im Gegensatz zur Funktion `dir`, die lediglich eine grobe Auflistung der Methoden anzeigt, liefert die Funktion `help` eine vollständige Hilfe. Besonders im interaktiven Modus ist `help` eine schnelle und gute Möglichkeit, um Informationen zu einer bestimmten Funktion zu erhalten. Noch übersichtlicher als `help` ist die Dokumentation der Standardbibliothek im Internet unter docs.python.org.

```
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
>>>
```

Der Aufruf einer *Methode* ist ähnlich wie der Aufruf einer Funktion (sie nimmt Argumente entgegen und gibt einen Wert zurück), aber die Syntax ist anders. Wir rufen eine Methode auf, indem wir den Methodennamen an den Variablennamen anhängen und den Punkt als Begrenzer verwenden.

Zum Beispiel nimmt die Methode `upper` eine Zeichenkette entgegen und gibt eine neue Zeichenkette zurück, die nur aus Großbuchstaben besteht:

Anstelle der Funktionssyntax `upper(word)` wird die Methodensyntax `word.upper()` verwendet.

```
>>> word = 'Banane'
>>> new_word = word.upper()
>>> print(new_word)
BANANE
```

Diese Form der Punktschreibweise gibt den Namen der Methode (`upper`) und den Namen der Zeichenkette (`word`), auf die die Methode angewendet werden soll an. Die leeren Klammern zeigen an, dass diese Methode kein Argument benötigt.

Ein Methodenaufruf wird als *Aufruf* bezeichnet; in diesem Fall würden wir sagen, dass wir `upper` auf `word` aufrufen. Die Aufrufkonvention mit der Punktnotation ist typisch für die *objektorientierte Programmierung*. Die Zeichenkette `word` ist in diesem Fall unser Objekt, und der Aufruf der Methode `upper` *auf* dem Objekt bewirkt, dass etwas mit dem Objekt passiert.

Es gibt weitere Zeichenketten-Methoden, beispielsweise die Methode `find`, die nach der Position einer Zeichenkette innerhalb einer anderen sucht:

```
>>> word = 'Banane'
>>> index = word.find('a')
>>> print(index)
1
```

In diesem Beispiel rufen wir `find` auf `word` auf und übergeben den gesuchten Buchstaben als Parameter.

Die Methode `find` kann sowohl Teilzeichenketten als auch Zeichen finden:

```
>>> word.find('na')
2
```

Sie kann als zweites Argument den Index annehmen, bei dem sie beginnen soll:

```
>>> word.find('a', 2)
3
```

Eine häufige Aufgabe ist das Entfernen von Whitespaces (Leerzeichen, Tabulatoren oder Zeilenumbrüche) am Anfang und Ende einer Zeichenkette mit der Methode `strip`:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Einige Methoden wie `startswith` geben boolesche Werte zurück.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

Man kann feststellen, dass `startswith` Groß- und Kleinschreibung unterscheidet, daher nehmen wir manchmal eine Zeile und wandeln mit `lower` alles in Kleinbuchstaben um, bevor wir eine Überprüfung mit der Methode `startswith` durchführen.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
>>> line.lower().startswith('h')
True
```

Im letzten Beispiel wird die Methode `lower` aufgerufen und dann wird mit `startswith` geprüft, ob die resultierende klein geschriebene Zeichenkette mit dem Buchstaben „h“ beginnt. Solange wir mit der Reihenfolge vorsichtig sind, können wir mehrere Methodenaufrufe in einem einzigen Ausdruck machen.

Übung 4: Es gibt eine String-Methode namens `count`, die der Funktion in der vorherigen Übung ähnlich ist. Lesen Sie die Dokumentation zu dieser Methode unter:

https://docs.python.org/library/stdtypes.html#string-methods

Schreiben Sie ein Programm, das die Anzahl der Vorkommen des Buchstabens „a“ in „Banane“ mithilfe der count-Methode zählt.

Parsen von Zeichenketten

Oft wollen wir in eine Zeichenkette schauen und eine Teilzeichenkette finden. Wenn wir zum Beispiel eine Liste von Zeilen erhalten, die wie folgt formatiert sind:

`From giefers.heiner@fh-swf.de Mon, 30 Aug 2021 16:20:09`

Wollten wir nur die *Domäne* der e-Mail Adresse (d. h. `fh-swf.de`) aus jeder Zeile extrahieren, können wir dies mit der Methode `find` und String-Slicing erreichen.

Zuerst wird die Position des at-Zeichens in der Zeichenkette ermittelt. Dann werden wir die Position des ersten Leerzeichens *nach* dem at-Zeichen finden. Und dann verwenden wir String-Slicing, um den Teil der Zeichenkette zu extrahieren, den wir suchen.

```
>>> data = 'From giefers.heiner@fh-swf.de Mon, 30 Aug 2021 16:20:09'
>>> atpos = data.find('@')
>>> print(atpos)
19
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
29
>>> host = data[atpos+1:sppos]
>>> print(host)
fh-swf.de
>>>
```

Wir verwenden eine Version der Methode `find`, die es uns erlaubt, eine Position in der Zeichenkette anzugeben, an der `find` mit der Suche beginnen soll. Dann extrahieren wir die Teilzeichenkette beginnen nach dem at-Zeichen bis hin (aber nicht einschließlich) zum Leerzeichen.

Die Dokumentation für die Methode `find` ist verfügbar unter

https://docs.python.org/library/stdtypes.html#string-methods.

Formatierte Zeichenketten

Wir haben nun schon mehrfach `print`-Anweisungen verwendet, bei denen wir sowohl Zeichenketten als auch die Werte bestimmter Variablen ausgegeben haben. Also z. B. so:

```
>>> count = 2
>>> zeichen = 'a'
>>> print('Buchstabe:', zeichen, 'Anzahl:', count)
Buchstabe: a Anzahl: 2
```

Das funktioniert, weil die `print`-Funktion eine Liste von Argumenten akzeptiert. Die Schreibweise mit den Kommas ist aber nicht gerade praktisch und auch nicht sehr übersichtlich.

Eine Abhilfe schaffen hier formatierte Zeichenketten, in Python auch *Format Strings* (oder *f-Strings*) genannt. Eine „normale“ Zeichenkette wird zu einem f-String, indem wir Ihr ein `f` direkt vor dem öffnenden Anführungszeichen voranstellen. In einem f-String können wir dann Werte einbetten, indem wir die Variablennamen in geschweifte Klammern direkt in den String schreiben.

```
>>> count = 2
>>> zeichen = 'a'
>>> print(f'Buchstabe: {zeichen} Anzahl: {count}')
Buchstabe: a Anzahl: 2
```

In Format Strings erkennt man die eingebetteten Variablen sehr schnell durch die geschweiften Klammern. Gleichzeitig benötigt man keine zusätzlichen Kommas oder schließenden und öffnenden Anführungszeichen.

Ein weiterer Vorteil von Format Strings ist, dass man die Darstellungsweise der Variablen beeinflussen kann. Das ist z. B. nützlich, wenn man Fließkommazahlen ausgeben möchte. Im folgenden Beispiel berechnen wir `10/3` was in einen Wert mit unendlich vielen Nachkommastellen resultiert[1].

```
>>> anteil = 10/3
>>> print("Ihr Anteil ist", anteil)
Ihr Anteil ist 3.3333333333333335
>>> print(f"Ihr Anteil ist {anteil:.2f}")
Ihr Anteil ist 3.33
```

Bei der Ausgabe wirkt die Darstellung so vieler Nachkommastellen häufig störend. Daher verwenden wir in der zweiten `print`-Anweisung des Beispiels einen f-String. Wir betten die Variable `anteil` ein, fügen aber direkt hinter dem Variablennamen noch ein Doppelpunkt ein. Danach steht ein Formatierungscode, in diesem Fall `.2f`. Das bedeutet, „stelle den Wert als Fließkommazahl mit zwei Nachkommastellen dar“.

Es gibt noch weitere Codes, mit denen die Ausgabe von Werten beeinflusst werden kann.Mit dem Formatierungszeichen `d` gibt man an, dass ein wert als Dezimalzahl (also „ganz normal“) formatiert werden soll. Ein `o` (`o` steht für „octal“) bewirkt eine Formatierung im Okatalsystem (also zur Basis 8), ein `x` (`x` steht für „hexadecimal“) stellte einen Wert im Hexadedezimalsystem (also zur Basis 16) dar.

```
>>> wert = 42
>>> print(f"Der Wert {wert:d} zur Basis 8: {wert:o} \
    und zur Basis 16: {wert:x}")
Der Wert 42 zur Basis 8: 52 und zur Basis 16: 2a
```

[1] Natürlich kann kein Computer dieser Welt reelle Zahlen *genau* abspeichern. Intern werden reelle Werte mit einer bestimmten Genauigkeit gespeichert, die auch in Berechnungen immer wieder zu *Rundungsfehlern* führen. Mit diesen Fragestellungen beschäftigt sich die *Computer-Arithmetik*, wir werden das Thema hier nicht genauer beleuchten. Allerdings sieht man die Auswirkungen auch im täglichen Umgang mit Python. Achten Sie mal auf die letzte Ziffer bei der Darstellung von 10/3 als Kommazahl.

Die Möglichkeiten, Darstellungsweisen mit Formatierungcodes zu verändern, sind sehr vielfältig und etwas kompliziert. Es ist also ratsam, sich eine Hilfe-Seite zur Hand zu nehmen, wenn man eine bestimmte Ausgabe erreichen möchte. Weiters zu dem Thema finden sie unter

https://docs.python.org/3/reference/lexical_analysis.html#f-strings

Debugging

Eine Fähigkeit, die man beim Programmieren kultivieren sollte, ist, sich immer zu fragen: „Was könnte hier schiefgehen?“ oder alternativ: „Welche verrückte Sache könnte unser Benutzer tun, um unser (scheinbar) perfektes Programm zum Absturz zu bringen?“.

Schauen wir uns zum Beispiel das Programm an, das wir zur Demonstration der `while`-Schleife im Kapitel über Iteration verwendet haben:

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: https://tiny.one/py4de/code3/copytildone2.py
```

Beobachten wir, was passiert, wenn der Benutzer eine leere Eingabezeile eingibt:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

Der Programm funktioniert gut, bis ihm eine leere Zeile präsentiert wird. Dann gibt es kein Zeichen an der nullten Indexposition, also erhalten wir einen Traceback. Es gibt zwei Lösungen, um Zeile drei „sicher“ zu machen, auch wenn die Zeichenkette leer ist.

Eine Möglichkeit ist, einfach die Methode `startswith` zu verwenden, die `False` zurückgibt, wenn die Zeichenkette leer ist.

```
if line.startswith('#'):
```

Eine andere Möglichkeit ist, die `if`-Anweisung mit einem „Wächter“ abzusichern und sicherzustellen, dass der zweite logische Ausdruck nur ausgewertet wird, wenn mindestens ein Zeichen in der Zeichenkette vorhanden ist:

```
if len(line) > 0 and line[0] == '#':
```

Glossar

Schleifenzähler Eine Variable, die zum Zählen von etwas verwendet wird und normalerweise mit 0 initialisiert ist und dann inkrementiert wird.

leere Zeichenkette Eine Zeichenkette ohne Zeichen und mit der Länge 0, dargestellt durch zwei Anführungszeichen.

Formatierungsoperator Der Operator `%`, der einen Format-String und ein Tupel entgegennimmt und eine Zeichenkette erzeugt, die die Elemente des Tupels enthält, die gemäß des Format-Strings formatiert sind.

Formatierungszeichen Ein Zeichen in einem Format-String, z. B. `%d`, das angibt, wie ein Wert formatiert werden soll.

Format-String Eine Zeichenkette, die mit dem Formatierungsoperator verwendet wird und Formatierungszeichen enthält.

Flag Eine boolesche Variable, die anzeigt, ob eine Bedingung wahr oder falsch ist.

Aufruf einer Methode Eine Anweisung, die eine Methode aufruft.

Unveränderlichkeit Die Eigenschaft einer Sequenz, deren Elemente nicht verändert werden können.

Index Ein ganzzahliger Wert, der verwendet wird, um ein Element in einer Sequenz auszuwählen, z. B. ein Zeichen in einer Zeichenkette.

Element Einer der Werte in einer Sequenz.

Methode Eine Funktion, die mit einem Objekt verknüpft ist und in Punktschreibweise aufgerufen wird.

Objekt Etwas, auf das sich eine Variable beziehen kann. Im Moment können wir „Objekt“ und „Wert“ noch austauschbar verwenden.

Suche Ein Muster, das beim Traversieren durch eine Zeichenkette dafür sorgt, dass die Suche beendet wird, sobald die gesuchte Teilzeichenkette gefunden wurde.

Folge Eine geordnete Menge, d. h. eine Menge von Werten, bei der jeder Wert durch einen ganzzahligen Index gekennzeichnet ist.

Teilzeichenkette Ein Teil einer Zeichenkette, der durch einen Bereich von Indizes angegeben wird.

Traversieren Durch die Elemente einer Sequenz iterieren und für jedes Element eine ähnliche Operation ausführen.

Übungen

Übung 5: Nehmen Sie den folgenden Python-Code, der eine Zeichenkette speichert:

```
str = 'X-DSPAM-Confidence: 0.8475'
```

Verwenden Sie `find` und String-Slicing, um den Teil der Zeichenkette nach dem Doppelpunkt zu extrahieren, und verwenden Sie dann die Funktion `float`, um die extrahierte Teilzeichenkette in eine Fließkommazahl zu konvertieren.

Übung 6: Lesen Sie die Dokumentation der String-Methoden unter

https://docs.python.org/library/stdtypes.html#string-methods

Vielleicht möchten Sie mit einigen von ihnen experimentieren, um sicherzustellen, dass Sie verstehen, wie sie funktionieren. `strip` und `replace` sind besonders nützlich.

Die Dokumentation verwendet eine Syntax, die verwirrend sein kann. Zum Beispiel in `find(sub[, start[, end]])` zeigen die Klammern optionale Argumente an. Also ist `sub` erforderlich, aber `start` ist optional, und wenn Sie `start` verwenden, dann ist `end` wiederum optional.

Kapitel 7

Dateien

Bisher haben wir gelernt, wie man Programme schreibt und der *CPU* unsere Absichten mit Hilfe von bedingter Ausführung, Funktionen und Iterationen mitteilt. Wir haben gelernt, wie wir Datenstrukturen im *Hauptspeicher* erstellen und verwenden. Die CPU und der Speicher sind der Ort, an dem unsere Software arbeitet und läuft. Hier findet das gesamte „Denken“ statt.

Aber wenn wir uns an unsere Diskussionen über die Hardware-Architektur erinnert, stellen wir fest, dass alles, was in der CPU oder im Hauptspeicher gespeichert ist, verloren geht, sobald der Strom abgeschaltet wird. Bis jetzt waren unsere Programme also nur flüchtige Spaßübungen, um Python zu lernen.

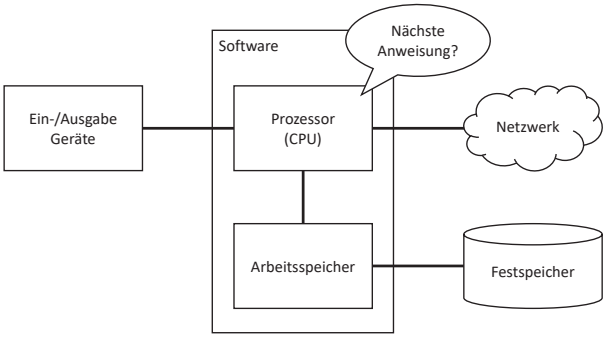

Abbildung 7.1: Arbeitsspeicher und Sekundärspeicher (Festspeicher)

In diesem Kapitel beginnen wir mit dem *Sekundär-* oder auch *Festspeicher* (also den Dateien) zu arbeiten. Der Sekundärspeicher wird nicht gelöscht, wenn der Strom abgeschaltet wird. Oder im Fall eines USB-Sticks können die Daten, die wir von unseren Programmen schreiben, aus dem System entfernt und zu einem anderen System transportiert werden.

Wir werden uns in erster Linie auf das Lesen und Schreiben von Textdateien konzentrieren, wie wir sie in einem Texteditor erstellen. Später werden wir sehen, wie man mit Datenbankdateien arbeitet, bei denen es sich um Binärdateien handelt,

die speziell für das Lesen und Schreiben durch Datenbanksoftware entwickelt wurden.

Öffnen von Dateien

Wenn wir eine Datei (z. B. auf unserer Festplatte) lesen oder schreiben wollen, müssen wir die Datei zuerst *öffnen*. Die `open`-Funktion kommuniziert mit unserem Betriebssystem, das weiß, wo die Daten für jede Datei gespeichert sind. Wenn wir eine Datei öffnen, bitten wir das Betriebssystem, die Datei anhand ihres Namens zu suchen und sicherzustellen, dass sie existiert. In diesem Beispiel öffnen wir die Datei `mbox.txt`, die in demselben Verzeichnis gespeichert sein sollte, in dem wir uns befinden, wenn wir Python starten. Man kann diese Datei hier herunterladen: tiny.one/py4de/code3/mbox.txt

```
>>> fhand = open('mbox.txt')
>>> print(fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='cp1252'>
```

Wenn das `open` erfolgreich ist, gibt uns das Betriebssystem ein *Dateihandle* zurück. Dieses Handle beeinhaltet nicht die eigentlichen Daten der Datei, sondern er sagt dem Python-Programm, *wo* das Betriebssystem die Datei abgelegt hat. Über dieses Handle können dann alle weiteren Operation für die Datei (mithilfe des Betriebssystems) ausgeführt werden.

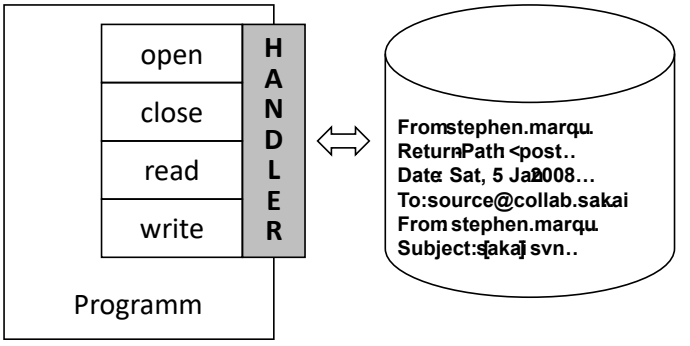

Abbildung 7.2: Ein Dateihandler

Wir erhalten ein Handle nur, wenn die angeforderte Datei existiert und wir die richtigen Berechtigungen zum Lesen oder schreiben der Datei haben. Wenn die Datei nicht existiert, schlägt `open` mit einem Traceback fehl und wir erhalten keinen Handle, um auf den Inhalt der Datei zuzugreifen:

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'stuff.txt'
```

Später werden wir `try` und `except` verwenden, um eleganter mit der Situation umzugehen, in der wir versuchen, eine Datei zu öffnen, die nicht existiert.

Textdateien

Eine Textdatei kann als eine Folge von Zeilen betrachtet werden, ähnlich wie eine Python-Zeichenkette als eine Folge von Zeichen betrachtet werden kann. Dies ist ein Beispiel für eine Textdatei, die die E-Mail-Aktivitäten verschiedener Personen in einem Open-Source-Projektentwicklungsteam aufzeichnet:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

Die gesamte Datei der Mail-Interaktionen ist verfügbar unter:

tiny.one/py4de/code3/mbox.txt

Eine gekürzte Version der Datei ist hier zu finden:

tiny.one/py4de/code3/mbox-short.txt

Diese Dateien haben ein Standardformat für eine Datei, die mehrere E-Mail-Nachrichten enthält. Die Zeilen, die mit `From` beginnen, trennen die einzelnen Nachrichten voneinander. Die Zeilen, die mit `From:` beginnen, sind dagegen Teil der Nachrichten. Weitere Informationen über das mbox-Format findet man unter https://en.wikipedia.org/wiki/Mbox.

Um die Datei in Zeilen zu unterteilen, gibt es ein spezielles Zeichen, das das Zeilenende darstellt, das *Newline*-Zeichen.

In Python stellen wir das Zeichen *Newline* (`\n`) als Rückstrich gefolgt von einem „n“ in Zeichenketten dar. Auch wenn dies wie zwei Zeichen aussieht, ist es tatsächlich ein einzelnes Zeichen. Wenn wir uns die Variable ansehen, indem wir `stuff` in den Interpreter eingeben, zeigt er uns das `\n` in der Zeichenkette, aber wenn wir `print` verwenden, um die Zeichenkette anzuzeigen, sehen wir die Zeichenkette durch das Newline-Zeichen in zwei Zeilen unterbrochen.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
```

```
X
Y
>>> len(stuff)
3
```

Man kann zusätzlich sehen, dass die Länge der Zeichenkette `X\nY` *drei* Zeichen beträgt, da das Zeilenumbruchzeichen ein einzelnes Zeichen ist.

Wenn wir also die Zeilen in einer Datei betrachten, müssen wir uns *vorstellen*, dass es ein spezielles unsichtbares Zeichen namens Newline am Ende jeder Zeile gibt, das das Ende der Zeile markiert. Das Newline-Zeichen trennt also die Zeichen in der Datei in Zeilen.

Lesen von Dateien

Auch wenn der Dateihandler nicht die Daten für die Datei enthält, ist es recht einfach, eine 'for'-Schleife zu konstruieren, um jede der Zeilen in einer Datei durchzuarbeiten und zu zählen:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    count = count + 1
print('Line Count:', count)

# Code: https://tiny.one/py4de/code3/open.py
```

Wir können den Dateihandler als Teil unserer Schleifenkonstruktion verwenden. Unsere `for`-Schleife zählt einfach die Anzahl der Zeilen in der Datei und gibt sie aus. Die grobe Übersetzung der `for`-Schleife ins Deutsche lautet: „Für jede Zeile in der Datei, die durch den Dateihandler repräsentiert wird, füge der Variablen `count` 1 hinzu.“.

Der Grund dafür, dass die Funktion `open` nicht die gesamte Datei liest, ist, dass die Datei mit vielen Gigabytes an Daten recht groß sein kann. Die Anweisung `open` benötigt unabhängig von der Größe der Datei die gleiche Zeit. Die `for`-Schleife bewirkt, dass die Daten tatsächlich aus der Datei gelesen werden.

Wenn die Datei mit einer `for`-Schleife auf diese Weise gelesen wird, kümmert sich Python um die Aufteilung der Daten in der Datei in einzelne Zeilen mit Hilfe des Newline-Zeichens. Python liest jede Zeile durch den Zeilenumbruch und nimmt den Zeilenumbruch als letztes Zeichen in die Variable `line` für jede Iteration der Schleife auf.

Da die Schleife die Daten zeilenweise liest, kann sie effizient die Zeilen in sehr großen Dateien lesen und zählen, ohne dass der Hauptspeicher zum Speichern der Daten aufgebraucht wird. Das obige Programm kann die Zeilen in Dateien beliebiger Größe mit sehr wenig Speicherplatz zählen, da jede Zeile gelesen, gezählt und dann verworfen wird.

Wenn man weiß, dass die Datei im Vergleich zur Größe des Hauptspeichers relativ klein ist, kann man die gesamte Datei mit der Methode `read` des Dateihandlers in einen String einlesen.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print(len(inp))
94626
>>> print(inp[:20])
From stephen.marquar
```

In diesem Beispiel wird der gesamte Inhalt (alle 94,626 Zeichen) der Datei `mbox-short.txt` direkt in die Variable `inp` gelesen. Wir verwenden String-Slicing, um die ersten 20 Zeichen der in `inp` gespeicherten Zeichen auszugeben.

Wenn die Datei auf diese Weise gelesen wird, sind alle Zeichen einschließlich aller Zeilen und Zeilenumbruchzeichen eine große Zeichenkette in der Variablen `inp`. Es ist eine gute Idee, die Ausgabe von `read` als Variable zu speichern, da jeder Aufruf von `read` mit einem gewissen Aufwand verbunden ist:

```
>>> fhand = open('mbox-short.txt')
>>> print(len(fhand.read()))
94626
>>> print(len(fhand.read()))
0
```

Wir müssen daran denken, dass diese Form der `open`-Funktion nur verwendet werden sollte, wenn die Dateidaten bequem in den Hauptspeicher unseres Computers passen. Wenn die Datei zu groß ist, um in den Hauptspeicher zu passen, sollten wir unser Programm so schreiben, dass es die Datei mit Hilfe einer `for`- oder `while`-Schleife in Stücken liest.

Suchen in Dateien

Beim Durchsuchen von Daten in einer Datei ist es ein sehr gängiges Vorgehen, eine Datei durchzulesen, dabei die meisten Zeilen zu ignorieren und nur Zeilen zu verarbeiten, die eine bestimmte Bedingung erfüllen. Wir können das Muster zum Lesen einer Datei mit Zeichenkettenmethoden kombinieren, um einfache Suchmechanismen aufzubauen.

Wenn wir z. B. eine Datei lesen und nur Zeilen ausgeben wollen, die mit dem Präfix `From:` beginnen, könnten wir mit der String-Methode `startswith` nur die Zeilen mit dem gewünschten Präfix auswählen:

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    if line.startswith('From:'):
```

```
        print(line)
```

```
# Code: https://tiny.one/py4de/code3/search1.py
```

Wenn dieses Programm läuft, erhalten wir die folgende Ausgabe:

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

Die Ausgabe sieht gut aus, da die einzigen Zeilen, die wir sehen, die sind, die mit `From:` beginnen, aber warum sehen wir die zusätzlichen Leerzeilen? Das liegt an dem unsichtbaren Zeichen *Newline*. Jede der Zeilen endet mit einem Zeilenumbruch, sodass die Anweisung `print` die Zeichenkette in der Variablen `line` ausgibt, die einen Zeilenumbruch enthält, und dann fügt `print` einen weiteren Zeilenumbruch hinzu, was zu der überschüssigen Leerzeile führt, den wir sehen.

Wir könnten Zeilen-Slicing verwenden, um alle Zeichen bis auf das letzte auszugeben, aber ein besserer Ansatz ist die Verwendung der `rstrip`-Methode, die Leerzeichen auf der rechten Seite einer Zeichenkette wie folgt entfernt:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:'):
        print(line)
```

```
# Code: https://tiny.one/py4de/code3/search2.py
```

Wenn dieses Programm läuft, erhalten wir die folgende Ausgabe:

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

Wenn die Dateiverarbeitungsprogramme komplizierter werden, möchte man vielleicht die Suchschleifen mit `continue` strukturieren. Die Grundidee der Suchschleife ist, dass man nach „interessanten“ Zeilen sucht und „uninteressante“ Zeilen effektiv überspringt. Und wenn wir dann eine interessante Zeile finden, machen wir etwas mit dieser Zeile.

Wir können die Schleife wie folgt strukturieren, um uninteressante Zeilen zu überspringen:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:'):
        continue
    # Process our 'interesting' line
    print(line)

# Code: https://tiny.one/py4de/code3/search3.py
```

Die Ausgabe des Programms ist die gleiche. Die uninteressanten Zeilen sind diejenigen, die nicht mit `From:` beginnen, die wir also mit `continue` überspringen. Für die „interessanten“ Zeilen (d. h. diejenigen, die mit `From:` beginnen) führen wir die Verarbeitung auf diesen Zeilen durch.

Wir können die String-Methode `find` verwenden, um eine Texteditor-Suche zu simulieren, die Zeilen findet, in denen die gesuchte Zeichenkette irgendwo in der Zeile steht. Da `find` nach einem Vorkommen einer Zeichenkette innerhalb einer anderen Zeichenkette sucht und entweder die Position der Zeichenkette oder -1 zurückgibt, wenn die Zeichenkette nicht gefunden wurde, können wir die folgende Schleife schreiben, um Zeilen anzuzeigen, die die Zeichenkette `@uct.ac.za` (University of Cape Town in South Africa) enthalten:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1: continue
    print(line)

# Code: https://tiny.one/py4de/code3/search4.py
```

Das erzeugt die folgende Ausgabe:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

Auch hier verwenden wir verkürzte Form der `if`-Anweisung, bei der wir das `continue` in dieselbe Zeile wie das `if` setzen. Diese verkürzte Form der `if`-Anweisung funktioniert genauso, als ob das `continue` in der nächsten Zeile und eingerückt wäre.

Wahl des Dateinamens durch den Benutzer

Wir wollen nicht jedes Mal unseren Python-Code bearbeiten müssen, wenn wir eine andere Datei verarbeiten wollen. Es wäre sinnvoller, den Benutzer aufzufordern, die Zeichenkette für den Dateinamen jedes Mal einzugeben, wenn das Programm ausgeführt wird, damit er unser Programm für verschiedene Dateien verwenden kann, ohne den Python-Code zu ändern.

Dies ist recht einfach zu bewerkstelligen, indem der Dateiname vom Benutzer mittels `input` wie folgt gelesen wird:

```
fname = input('Gib eine Datei an: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print(f'Es gibt {count} Betreffzeilen in {fname}')

# Code: https://tiny.one/py4de/code3/search6.py
```

Wir lesen den Dateinamen vom Benutzer und legen ihn in eine Variable namens `fname` und öffnen diese Datei. Jetzt können wir das Programm wiederholt zur Verarbeitung verschiedener Dateien ausführen.

```
python search6.py
Gib eine Datei an: mbox.txt
Es gibt 1797 Betreffzeilen in mbox.txt

python search6.py
Gib eine Datei an: mbox-short.txt
Es gibt 27 Betreffzeilen in mbox-short.txt
```

Bevor wir uns den nächsten Abschnitt ansehen, werfen wir einen Blick auf das obige Programm und fragen uns: „Was könnte hier möglicherweise schiefgehen?“ oder „Was könnte unser freundlicher Benutzer tun, das dazu führen würde, dass unser nettes kleines Programm unschön mit einem Traceback beendet wird und uns in den Augen unserer Benutzer nicht so cool aussehen lässt?“.

Verwendung von `try`, `except` und `open`

Aber was passiert, wenn unser Benutzer etwas eingibt, das kein Dateiname ist?

```
python search6.py
Gib eine Datei an: nichtda.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
```

```
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'nichtda.txt'

python search6.py
Gib eine Datei an: ha ha ha
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
FileNotFoundError: [Errno 2] No such file or directory: 'ha ha ha'
```

Benutzer werden letztendlich alles Mögliche tun, um unsere Programme zum Absturz zu bringen, entweder aus Spaß oder sogar mit dem Ziel, Sicherheitslücken auszunutzen. Tatsächlich ist ein wichtiger Teil jedes Software-Entwicklungsteams eine Person oder Gruppe, die *Qualitätssicherung* (kurz QS) genannt wird, deren einzige Aufgabe es ist, die „verrücktesten“ Dinge zu tun, um Fehler in einer Software aufzudecken

Das QS-Team ist dafür verantwortlich, die Fehler in Programmen zu finden, bevor wir das Programm an die Endbenutzer ausliefern, die die Software vielleicht kaufen oder unser Gehalt für das Schreiben der Software bezahlen. Das QS-Team ist also der beste Freund des Programmierers.

Da wir nun den Fehler im Programm sehen, können wir ihn elegant mit der `try`/`except`-Struktur beheben. Wir müssen annehmen, dass der `open`-Aufruf fehlschlagen könnte und fügen Wiederherstellungscode hinzu, wenn das `open` wie folgt fehlschlägt:

```
fname = input('Gib eine Datei an: ')
try:
    fhand = open(fname)
except:
    print(f'Datei {fname} konnte nicht geoeffnet werden')
    exit()
count = 0
for line in fhand:
    if line.startswith('Subject:'):
        count = count + 1
print(f'Es gibt {count} Betreffzeilen in {fname}')

# Code: https://tiny.one/py4de/code3/search7.py
```

Die Funktion `exit` terminiert das Programm. Es ist eine Funktion, die wir aufrufen und die niemals zurückkehrt. Wenn unser Benutzer (oder das QS-Team) nun Dummheiten oder falsche Dateinamen eintippt, „fangen“ wir sie ab und reagieren entsprechend:

```
python search7.py
Gib eine Datei an: nichtda.txt
Datei nichtda.txt konnte nicht geoeffnet werden

python search7.py
```

```
Gib eine Datei an: ha ha ha
Datei ha ha ha konnte nicht geoeffnet werden
```

Der Schutz des `open`-Aufrufs ist ein gutes Beispiel für die richtige Verwendung von `try` und `except` in einem Python-Programm.

Sobald man mehr Erfahrung mit Python hat, kann man sich mit anderen Python-Programmierern darüber streiten, welche von zwei gleichwertigen Lösungen für ein Problem eleganter ist. Das Ziel, elegante Lösungen zu implementieren, spiegelt den Gedanken wider, dass Programmieren zum Teil Technik und zum Teil Kunst ist. Wir sind nicht immer nur daran interessiert, etwas zum Laufen zu bringen, wir wollen auch, dass unsere Lösung elegant ist und von unseren Kollegen als elegant geschätzt wird.

Schreiben von Dateien

Um eine Datei zu schreiben, müssen wir sie im Modus `w` (für *write*, schreiben) als zweitem Parameter öffnen:

```
>>> fout = open('output.txt', 'w')
>>> print(fout)
<_io.TextIOWrapper name='output.txt' mode='w' encoding='cp1252'>
```

Wenn die Datei bereits existiert, werden beim Öffnen im Schreibmodus die alten Daten gelöscht und es wird neu begonnen, also Vorsicht! Wenn die Datei nicht existiert, wird eine neue Datei erstellt.

Die Methode `write` des Dateihandler-Objekts schreibt Daten in die Datei und gibt die Anzahl der geschriebenen Zeichen zurück. Der Standard-Schreibmodus für das Schreiben (und Lesen) von Zeichenketten ist der Text-Modus.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

Auch hier merkt sich das Dateiobjekt, wo es sich befindet. Wenn Sie also `write` erneut aufrufen, fügt es die neuen Daten am Ende hinzu.

Wir müssen sicherstellen, dass wir die Zeilenenden beim Schreiben in die Datei berücksichtigen, indem wir explizit das Newline-Zeichen einfügen, wenn wir eine Zeile beenden wollen. Die Anweisung `print` fügt automatisch einen Zeilenumbruch ein, aber die Methode `write` dagegen fügt den Zeilenumbruch nicht automatisch hinzu.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
24
```

Wenn wir mit dem Schreiben fertig sind, müssen wir die Datei schließen, um sicherzustellen, dass das letzte Bit der Daten physisch auf die Festplatte geschrieben wird, damit es nicht verloren geht, wenn der Strom ausfällt.

```
>>> fout.close()
```

Wir könnten die Dateien, die wir zum Lesen öffnen, auch schließen, aber wir dürfen ein wenig nachlässig sein, wenn wir nur ein paar Dateien öffnen, da Python dafür sorgt, dass alle offenen Dateien geschlossen werden, wenn das Programm endet. Wenn wir Dateien schreiben, wollen wir die Dateien explizit schließen, um nichts dem Zufall zu überlassen.

Debugging

Beim Lesen und Schreiben von Dateien kann man auf Probleme mit Leerzeichen stoßen. Diese Fehler können schwer zu beheben sein, da Leerzeichen, Tabulatoren und Zeilenumbrüche normalerweise unsichtbar sind:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

Die eingebaute Funktion `repr` kann helfen. Sie nimmt ein beliebiges Objekt als Argument und gibt eine String-Repräsentation des Objekts zurück. Bei Zeichenketten stellt sie Leerzeichen mit Backslash-Sequenzen dar:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Dies kann bei der Fehlersuche hilfreich sein.

Ein weiteres Problem, auf das wir stoßen könnten, ist, dass verschiedene Systeme unterschiedliche Zeichen verwenden, um das Ende einer Zeile anzuzeigen. Einige Systeme verwenden einen Zeilenumbruch, dargestellt als `\n`. Andere verwenden ein Return-Zeichen, dargestellt als `\r`. Einige verwenden beides. Wenn wir Dateien zwischen verschiedenen Systemen verschieben, können diese Inkonsistenzen Probleme verursachen.

Für die meisten Systeme gibt es Anwendungen, um von einem Format in ein anderes zu konvertieren. Man findet sie (und lesen mehr über dieses Thema) unter www.wikipedia.org/wiki/Newline. Oder man kann natürlich auch selbst eines schreiben.

Glossar

Auffangen einer Ausnahme Um zu verhindern, dass eine Ausnahme ein Programm terminiert, verwenden Sie die Anweisungen `try` und `except`.

Newline Ein Sonderzeichen, das in Dateien und Zeichenketten verwendet wird, um das Ende einer Zeile anzuzeigen.

Qualitätssicherung Eine Person oder ein Team, das sich auf die Sicherstellung der Gesamtqualität eines Softwareprodukts konzentriert. QS ist oft an der Prüfung eines Produkts und der Identifizierung von Problemen beteiligt, bevor das Produkt freigegeben wird.

Textdatei Eine Folge von Zeichen, die in einem permanenten Speicher wie einer Festplatte gespeichert wird.

Übungen

Übung 1: Schreiben Sie ein Programm, das eine Datei einliest und den Inhalt der Datei (Zeile für Zeile) in Großbuchstaben ausgibt. Das Ausführen des Programms sieht wie folgt aus:

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
     BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
     SAT, 05 JAN 2008 09:14:16 -0500
```

Sie können die Datei herunterladen von tiny.one/py4de/code3/mbox-short.txt

Übung 2: Schreiben Sie ein Programm, das nach einem Dateinamen fragt, und iterieren Sie dann durch die Datei und sucht nach Zeilen der Form:

```
X-DSPAM-Confidence: 0.8475
```

Wenn Sie auf eine Zeile stoßen, die mit `X-DSPAM-Confidence:` beginnt, trennen Sie die Zeile auf, um die Fließkommazahl in der Zeile zu extrahieren. Zählen Sie diese Zeilen und berechnen Sie dann die Summe der Werte aus diesen Zeilen. Wenn Sie das Ende der Datei erreichen, geben Sie den Durchschnittswert aus.

```
Gib eine Datei an: mbox.txt
Wahrscheinlichkeit fuer Kein-Spam: 0.894128046745

Gib eine Datei an: mbox-short.txt
Wahrscheinlichkeit fuer Kein-Spam: 0.750718518519
```

Testen Sie Ihre Datei an den Dateien `mbox.txt` und `mbox-short.txt`.

Übung 3: Manchmal, wenn Programmierern langweilig ist oder sie ein bisschen Spaß haben wollen, fügen sie ein harmloses *Easter-Egg* in ihr Programm ein. Ändern Sie das Programm, das den Benutzer zur Eingabe des Dateinamens auffordert, so, dass es eine lustige Meldung ausgibt, wenn der Benutzer den genauen Dateinamen „blafabel“ eingibt. Für alle anderen existierenden und nicht existierenden Dateien sollte sich das Programm normal verhalten. Hier ist ein Beispiel für die Ausführung des Programms:

```
python egg.py
Gib eine Datei an: mbox.txt
Es gibt 1797 Betreffzeilen mbox.txt

python egg.py
Gib eine Datei an: missing.tyxt
Datei missing.tyxt konnte nicht geoeffnet werden

python egg.py
Gib eine Datei an: blafabel
Du laberst mich an?
```

Wir möchten Sie natürlich nicht dazu verführen, ständig Easter-Eggs in Ihre Programme einzubauen; dies ist nur eine Übung!

Kapitel 8

Listen

Python ist eine ideale Sprache um Daten zu verarbeiten und Daten stehen meist nicht allein, als einzelner Wert, sondern sie liegen als „Ansammlung“ vor. Entweder ungeordnet, geordnet (d. h. mit einer Reihenfolge) oder, wie in einer Kartei, unter einem Suchbegriff abgelegt. All diese Arten von „Datensammlungen“ gibt es in Python und dazu noch eine große Menge von Funktionen, um die Daten zu verarbeiten.

In den nächsten drei Kapiteln werden wir die wichtigsten *zusammengesetzten Datentypen* (englisch *Composite Data Types*) in Python behandel. Den Anfang macht der Typ *List*, den man sehr vielfältig verwenden kann.

Listen sind Folgen von Werten

Wie eine Zeichenkette ist auch eine *Liste* eine Folge von Werten. In einer Zeichenkette sind die Werte Zeichen; in einer Liste können sie von beliebigem Typ sein. Die Werte in einer Liste werden *Elemente* oder manchmal *Token* genannt.

Es gibt mehrere Möglichkeiten, eine neue Liste zu erstellen; die einfachste ist, die Elemente in eckige Klammern zu setzen (`[` und `]`):

```
[10, 20, 30, 40]
['Banane', 'Apfel', 'Kiwi']
```

Das erste Beispiel ist eine Liste mit vier Ganzzahlen. Das zweite ist eine Liste mit drei Zeichenketten. Die Elemente einer Liste müssen nicht vom gleichen Typ sein. Die folgende Liste enthält eine Zeichenkette, eine Gleitkommazahl, eine Ganzzahl und selbst eine weitere (!) Liste:

```
['spam', 2.0, 5, [10, 20]]
```

Eine Liste innerhalb einer anderen Liste ist *verschachtelt*.

Eine Liste, die keine Elemente enthält, wird als leere Liste bezeichnet; wir können eine solche mit leeren Klammern `[]` erstellen.

Man kann auch Variablen Listenwerte zuweisen:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> listoflists = [cheeses, numbers, empty]
>>> print(listoflists)
[['Cheddar', 'Edam', 'Gouda'], [17, 123], []]
```

Listen sind veränderbar

Die Syntax für den Zugriff auf die Elemente einer Liste ist die gleiche wie für den Zugriff auf die Zeichen einer Zeichenkette: der indexbasierte Zugriffsoperator oder auch Klammeroperator. Der Ausdruck innerhalb der Klammern gibt den Index an. Wir müssen daran denken, dass die Indizes bei 0 beginnen:

```
>>> print(cheeses[0])
Cheddar
```

Im Gegensatz zu Strings sind Listen veränderbar, da wir die Reihenfolge der Elemente in einer Liste ändern oder einem Element in einer Liste einen neuen Wert zuweisen können. Wenn der Klammeroperator auf der linken Seite einer Zuweisung erscheint, identifiziert er das Element der Liste, dem ein neuer Wert zugewiesen werden soll.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print(numbers)
[17, 5]
```

Das Element an der Indexposition 1 von `numbers`, welches zuvor `123` war, ist nun `5`.

Man kann sich eine Liste als eine Beziehung zwischen Indizes und Elementen vorstellen. Diese Beziehung wird als *Mapping* oder *Abbildung* bezeichnet; jeder Index wird auf eines der Elemente „abgebildet“.

Listenindizes funktionieren auf die gleiche Weise wie Zeichenketten-Indizes:

- Jeder ganzzahlige Ausdruck kann als Index verwendet werden.
- Wenn wir versuchen, ein Element zu lesen oder zu schreiben, das nicht vorhanden ist, erhalten wir einen `IndexError`.

- Wenn ein Index einen negativen Wert hat, zählt er vom Ende der Liste rückwärts.

Der `in`-Operator funktioniert auch bei Listen.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

Traversieren einer Liste

Die gebräuchlichste Art, die Elemente einer Liste zu durchlaufen, ist mit einer `for`-Schleife. Die Syntax ist die gleiche wie bei Zeichenketten:

```
for cheese in cheeses:
    print(cheese)
```

Das funktioniert gut, wenn wir die Elemente der Liste nur lesen müssen. Aber wenn wir die Elemente schreiben oder aktualisieren wollen, brauchen wir die Indizes. Ein üblicher Weg, das zu tun, ist die Kombination der Funktionen `range` und `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Diese Schleife durchläuft die Liste und aktualisiert jedes Element. `len` gibt die Anzahl der Elemente in der Liste zurück. `range` gibt eine Liste von Indizes von 0 bis $n-1$ zurück, wobei n die Länge der Liste ist. Jedes Mal, wenn die Schleife durchlaufen wird, erhält `i` den Index des nächsten Elements. Die Zuweisungsanweisung im Rumpf verwendet `i`, um den alten Wert des Elements zu lesen und den neuen Wert zuzuweisen.

Die Built-in Funktion `range` ist sehr nützlich und kann immer dann verwendet werden, wenn wir Sequenzen von ganzen Zahlen benötigen. Wenn man `range` nicht nur eines, sondern mehrere Argumente mitgibt, kann man die Sequenz von Werten weiter beeinflussen. Im folgenden Beispiel erzeugen wir eine Sequenz von -10 bis 20 (ausschließlich), bei der wir in Dreierschritten vorgehen:

```
>>> x=list(range(-10,20,3))
>>> print(x)
[-10, -7, -4, -1, 2, 5, 8, 11, 14, 17]
```

Um eine Liste zu durchlaufen und gleichzeitig die Indizes der Listenelemente zu erhalten, gibt es noch eine (im Vergleich zu `range`) etwas schönere Lösung. Die

Funktion `enumerate` liefert uns zwei Werte[1] zurück: Als ersten den Index des Elements und als zweiten den Wert des Elements. Das folgende Beispiel tut also genau dasselbe, wie die `for`-Schleife mit dem `range(len(...))` oben.

```
for i, wert in enumerate(numbers):
    numbers[i] = wert * 2
```

Eine `for`-Schleife über eine leere Liste ist übrigens kein Fehler, sie führt nur einfach den Rumpf nie aus:

```
for x in empty:
    print('This never happens.')
```

Obwohl eine Liste eine weitere Liste enthalten kann, zählt die verschachtelte Liste immer noch als ein einzelnes Element. Die Länge dieser Liste ist vier:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Listen-Operationen

Der Operator `+` verkettet Listen. Das bedeutet, die beiden listen links und rechts des `+`-Operators werden, in dieser Reihenfolge, hintereinander gehängt:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
```

In ähnlicher Weise wiederholt der Operator `*` eine Liste eine bestimmte Anzahl von Malen:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Das erste Beispiel wiederholt sich viermal. Im zweiten Beispiel wird die Liste dreimal wiederholt.

[1] Eigentlich liefert die Funktion ein 2-Tupel zurück, also *ein* Element, das aus zwei Werten besteht. Da wir bei der Zuweisung zwei Variablennamen hinschreiben, wird das Tupel „ausgepackt“, d. h. die Werte werden aus dem Tupel entnommen und den einzelnen Variablen. Dies nennt man auch *unpacking*. Um Tupel und ihre verwendung geht es in Kapitel 10.

Listen-Slicing

Der Slice-Operator, den wir bereits von den Zeichenketten kennen, funktioniert auch bei Listen:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Wenn wir den ersten Index weglassen, beginnt das Slice am Anfang. Wenn Sie den zweiten weglassen, geht das Slice bis zum Ende. Wenn wir also beide weglassen, ist das Slice eine Kopie der gesamten Liste.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Da Listen veränderbar sind, ist es oft sinnvoll, eine Kopie zu erstellen, bevor wir Operationen durchführen mit ihnen durchführen.

Ein Slice-Operator auf der linken Seite einer Zuweisung kann mehrere Elemente aktualisieren:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

Listenmethoden

Python bietet Methoden, die auf Listen operieren. Zum Beispiel fügt `append` ein neues Element an das Ende einer Liste an:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

`extend` nimmt eine Liste als Argument und fügt alle Elemente an:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

In diesem Beispiel wird `t2` nicht verändert, wohl aber t1.

`sort` ordnet die Elemente der Liste aufsteigend an. Beachten Sie bei der `sort`-Funktion, dass diese *in-place* arbeitet, d. h. sie verändert die Reihenfolge der Elemente **in** der Liste.

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

Um eine sortierte *Kopie* der Liste zu erhalten, können Sie die eingebaute Funktion `sorted` verwenden. Diese lässt sich auf alle Python Datenstrukturen anwenden, die *iterierbar* sind.

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> l = sorted(t)
>>> print(t)
['d', 'c', 'e', 'b', 'a']
>>> print(l)
['a', 'b', 'c', 'd', 'e']
>>> sorted("Hallo Welt!")
[' ', '!', 'H', 'W', 'a', 'e', 'l', 'l', 'l', 'o', 't']
```

Die meisten Listenmethoden (also solche die Sie mit der Punktnotation auf einer Liste aufrufen können) arbeiten *in-place*, haben also *keinen* Rückgabewert; sie verändern die Liste und geben `None` zurück. Wenn wir versehentlich `t = t.sort()` schreiben, werden wir von dem Ergebnis enttäuscht sein.

Löschen von Elementen

Es gibt mehrere Möglichkeiten, Elemente aus einer Liste zu löschen. Wenn wir den Index des gewünschten Elements kennen, können wir `pop` verwenden:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

`pop` ändert die Liste und gibt das Element zurück, das entfernt wurde. Wenn wir keinen Index angeben, löscht es das letzte Element und gibt es zurück.

Wenn wir den entfernten Wert nicht benötigen, können wir den Operator `del` verwenden:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

Wenn man das Element kennt, das man entfernen möchten (aber nicht den Index), kann man `remove` verwenden. `remove` entfern dabei nur das erste Vorkommen des Wertes in der Liste. Ist der gesuchte Wert mehrfach enthalten, bleiben die weiteren Vorkommen bestehen.

```
>>> t = ['a', 'b', 'c', 'a']
>>> t.remove('a')
>>> print(t)
['b', 'c', 'a']
```

Der Rückgabewert von `remove` ist `None`.

Um einen Bereich aus mehreren benachbarten Elementen zu entfernen, können wir `del` mit einem Slice-Index verwenden:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

Wie üblich wählt die Slice-Operation alle Elemente bis zu (aber nicht einschließlich) dem zweiten Index.

Listen und Funktionen

Es gibt eine Reihe von eingebauten Funktionen, die auf Listen angewendet werden können, mit denen wir schnell eine Liste durchsehen können, ohne eigene Schleifen zu schreiben:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

Die Funktion `sum()` funktioniert nur, wenn die Listenelemente Zahlen sind. Die anderen Funktionen (`max()`, `len()`, usw.) arbeiten mit Listen von Zeichenketten, aber auch anderen Datentypen, deren Werte miteinander verglichen werden können.

Wir könnten ein früheres Programm umschreiben, das den Durchschnitt einer Liste von Zahlen berechnet, die vom Benutzer eingegeben wurden.

Zunächst das Programm zur Berechnung eines Durchschnitts ohne Liste:

```
total = 0
count = 0
while (True):
    inp = input('Gib eine Zahl ein: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print('Mittelwert:', average)

# Code: https://tiny.one/py4de/code3/avenum.py
```

In diesem Programm haben wir die Variablen `count` und `total`, um die Anzahl und die laufende Summe der Zahlen des Benutzers festzuhalten, während wir den Benutzer wiederholt nach einer Zahl fragen.

Wir könnten uns einfach jede Zahl merken, so wie der Benutzer sie eingegeben hat, und eingebaute Funktionen verwenden, um die Summe und die Anzahl am Ende zu berechnen.

```
numlist = list()
while (True):
    inp = input('Gib eine Zal ein: ')
    if inp == 'done': break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print('Mittelwert:', average)

# Code: https://tiny.one/py4de/code3/avelist.py
```

Wir erstellen eine leere Liste, bevor die Schleife beginnt, und fügen dann jedes Mal, wenn wir eine neue Zahl bekommen, diese an die Liste an. Am Ende des Programms berechnen wir einfach die Summe der Zahlen in der Liste und teilen sie durch die Anzahl der Zahlen in der Liste, um den Durchschnitt zu ermitteln.

Listen und Zeichenketten

Eine Zeichenkette ist eine Folge von Zeichen und eine Liste ist eine Folge von Werten, aber eine Liste von Zeichen ist nicht dasselbe wie eine Zeichenkette. Um von einer Zeichenkette in eine Liste von Zeichen zu konvertieren, können wir `list` verwenden:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

Da `list` der Name einer eingebauten Funktion ist, sollte man es vermeiden, ihn als Variablennamen zu verwenden. Auch der Buchstabe `l` sollte vermieden werden, weil er zu sehr wie die Zahl `1` aussieht. Deshalb kann man beispielsweise `t` verwenden.

Was passiert, wenn wir zufällig einen Namen einer eingebauten Funktion für eigene Zwecke wiederverwenden, zeigt folgendes Beispiel:

```
>>> def list(*args):
...     return ['Ha', 'ha', 'ha']
...
>>> buchstaben = list("Hallo")
>>> print(buchstaben)
['Ha', 'ha', 'ha']
>>> del list
>>> buchstaben = list("Hallo")
>>> print(buchstaben)
['H', 'a', 'l', 'l', 'o']
```

Hier definieren wir eine eigene Funktion `list`; Python lässt das ohne weiteres zu. Wenn wir die Funktion `list` später verwenden, ohne uns daran zu erinnern, dass wir sie „umdefiniert“ haben, können wir ein schwer zu erklärendes Fehlverhalten haben. Um die Sache rückgängig zu machen, löchen wir mit `del` die aktuelle Bedeutung des Names `list`. So kommt die ursprüngliche Bedeutung wieder zum Vorschein und wir können die Funktion wie gewünscht verwenden.

Die Funktion `list` zerlegt eine Zeichenkette in einzelne Buchstaben. Wenn wir eine Zeichenkette in Wörter zerlegen wollen, können wir die Methode `split` verwenden:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

Nachdem wir die Zeichenkette mit `split` in eine Liste von Wörtern zerlegt haben, können wir den Indexoperator (eckige Klammer) verwenden, um ein bestimmtes Wort in der Liste zu betrachten.

Man kann `split` mit einem optionalen Argument (*Delimiter* genannt) aufrufen, das angibt, welche Zeichen als Wortgrenzen verwendet werden sollen. Das folgende Beispiel verwendet einen Bindestrich als Begrenzungszeichen:

```
>>> s = 'rama-lama-ding-dong'
>>> delimiter = '-'
>>> s.split(delimiter)
['rama', 'lama', 'ding', 'dong']
```

`join` ist die Umkehrung von `split`. Es nimmt eine Liste von Strings und verkettet die Elemente. Da `join` eine String-Methode ist, müssen wir sie auf dem Delimiter aufrufen und die Liste als Parameter übergeben:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In diesem Fall ist das Begrenzungszeichen ein Leerzeichen, also setzt `join` ein Leerzeichen zwischen die Wörter. Um Zeichenketten ohne Leerzeichen zu verketten, können wir die leere Zeichenkette als Begrenzungszeichen verwenden.

Parsen von Zeilen

Wenn wir eine Datei lesen, wollen wir normalerweise etwas anderes mit den Zeilen machen, als nur die ganze Zeile auszugeben. Oft wollen wir die „interessanten Zeilen“ finden und dann die Zeile *parsen*, um einen interessanten *Teil* der Zeile zu finden. Was wäre, wenn wir den Wochentag aus den Zeilen ausgeben wollten, die mit `From` beginnen?

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

Die Methode `split` ist bei dieser Art von Problem sehr hilfreich. Wir können ein kleines Programm schreiben, das nach Zeilen sucht, in denen die Zeile mit `From` beginnt, diese Zeilen mit `split` zerlegen, und dann das dritte Wort in der Zeile ausgibt:

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From '): continue
    words = line.split()
    print(words[2])

# Code: https://tiny.one/py4de/code3/search5.py
```

Das Programm erzeugt die folgende Ausgabe:

```
Sat
Fri
Fri
Fri
...
```

Später werden wir immer ausgefeiltere Techniken erlernen, wie wir die zu bearbeitenden Zeilen auswählen und wie wir diese Zeilen zerlegen, um genau die gesuchte Information zu finden.

Objekte und Werte

Wenn wir diese Zuweisungsanweisungen ausführen:

```
a = 'Banane'
b = 'Banane'
```

wissen wir, dass `a` und `b` sich beide auf eine Zeichenkette beziehen, aber die Frage ist, ob sie sich auf *dieselbe* Zeichenkette beziehen. Es gibt zwei denkbare Szenarien:

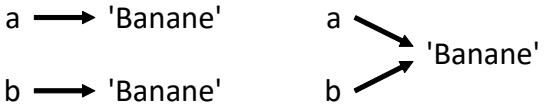

Abbildung 8.1: Variablen und Objekte

In einem Fall beziehen sich `a` und `b` auf zwei verschiedene Objekte, die den gleichen Wert haben. Im zweiten Fall beziehen sie sich auf das gleiche Objekt.

Um zu prüfen, ob zwei Variablen auf das selbe Objekt verweisen, können wir den Operator `is` verwenden.

```
>>> a = 'Banane'
>>> b = 'Banane'
>>> a is b
True
```

In diesem Beispiel war Python „schlau genug“, nur ein String-Objekt zu erzeugen; sowohl `a` als auch `b` beziehen sich darauf. Es kann aber durchaus sein, dass ein gleichlautender String mehrfach als Objekt angelegt wird. Dann kann es dazu kommen, das die Werte von `a` und `b` gleich sind, es sich aber um unterschiedliche Objekte handelt:

```
>>> a = 'Banane'
>>> b = 'B' + a[1:]
```

```
>>> print(a,b)
Banane Banane
>>> a is b
False
```

Auch, wenn wir zwei Listen mit gleichem Inhalt erstellen, erhalten wir zwei Objekte:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

In diesem Fall würden wir sagen, dass die beiden Listen *äquivalent* sind, weil sie die gleichen Elemente haben, aber nicht *identisch*, weil sie nicht das selbe Objekt sind. Wenn zwei Objekte identisch sind, sind sie auch äquivalent, aber wenn sie äquivalent sind, sind sie nicht unbedingt identisch.

Bis jetzt haben wir „Objekt“ und „Wert“ austauschbar verwendet, aber es ist präziser zu sagen, dass ein Objekt einen Wert hat. Wenn wir `a = [1,2,3]` ausführen, bezieht sich `a` auf ein Listenobjekt, dessen Wert eine bestimmte Folge von Elementen ist. Wenn eine andere Liste die gleichen Elemente hat, würden wir sagen, sie hat den gleichen Wert.

Aliase

Wenn sich `a` auf ein Objekt bezieht und wir `b = a` zuweisen, dann beziehen sich beide Variablen auf das selbe Objekt:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Die Assoziation einer Variablen mit einem Objekt wird als *Referenz* bezeichnet. In diesem Beispiel gibt es zwei Referenzen auf das selbe Objekt.

Ein Objekt mit mehr als einem Verweis hat mehr als einen Namen. Diese Namen bezeichnen wir als *Aliase* (Plural für *Alias*).

Wenn das referenzierte Objekt veränderbar ist, wirken sich Änderungen, die mit einem Alias vorgenommen werden, auch auf die anderen Aliase aus:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Dieses Verhalten kann zwar nützlich sein, ist aber fehleranfällig. Im Allgemeinen ist es sicherer, Aliasing zu vermeiden, wenn wir mit veränderlichen Objekten arbeiten.

Bei unveränderlichen Objekten wie Zeichenketten ist das Aliasing nicht so problematisch. In dem folgenden Beispiel macht es fast nie einen Unterschied, ob sich `a` und `b` auf dieselbe Zeichenkette beziehen oder nicht.

```
a = 'Banane'
b = 'Banane'
```

Listen als Funktionsargumente

Wenn wir eine Liste an eine Funktion übergeben, erhält die Funktion eine Referenz auf die Liste. Wenn die Funktion einen Listenparameter modifiziert, ist diese Änderung auch in der aufrufenden Umgebung sichtbar. Zum Beispiel entfernt `delete_head` das erste Element aus einer Liste:

```
def delete_head(t):
    del t[0]
```

Die Funktion wird folgendermaßen verwendet:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
```

Der Parameter `t` und die Variable `letters` sind Aliasnamen für das selbe Objekt.

Übung 1: Schreiben Sie eine Funktion namens `remove_all`, die eine Liste und einen Wert entgegennimmt und alle Vorkommen des Wertes aus der Liste entfernt. Die Funktion soll die Liste *in-place* verändern und als Ergebnis `None` zurückgeben.

Es ist wichtig, zwischen Operationen zu unterscheiden, die Listen verändern, und Operationen, die neue Listen erzeugen. Zum Beispiel verändert die Methode `append` eine Liste, aber der Operator `+` erzeugt eine neue Liste:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None

>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t2 is t3
False
```

Dieser Unterschied ist wichtig, wenn wir Funktionen schreiben, die Listen verändern sollen. Zum Beispiel löscht diese Funktion *nicht* den Kopf einer Liste:

```
def bad_delete_head(t):
    t = t[1:]               # FALSCH!
```

Der Slice-Operator erzeugt eine neue Liste und die Zuweisung macht `t` zu einem Verweis auf diese Liste, aber nichts davon hat irgendeinen Effekt auf die Liste, die als Argument übergeben wurde.

Eine Alternative ist, eine Funktion zu schreiben, die eine neue Liste erzeugt und zurückgibt. Zum Beispiel gibt `tail` alle Elemente einer Liste bis auf das erste zurück:

```
def tail(t):
    return t[1:]
```

Diese Funktion lässt die ursprüngliche Liste unangetastet. So wird sie verwendet:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print(rest)
['b', 'c']
```

Übung 2: Schreiben Sie eine Funktion namens `chop`, die eine Liste entgegennimmt und sie modifiziert, indem sie das erste und letzte Element entfernt und `None` zurückgibt. Schreiben Sie dann eine Funktion namens `middle`, die eine Liste annimmt und eine neue Liste zurückgibt, die alle Elemente außer dem ersten und letzten enthält.

Debugging

Der unvorsichtige Umgang mit Listen (und anderen veränderbaren Objekten) kann zu stundenlanger Fehlersuche führen. Hier sind einige häufige Fallstricke und Möglichkeiten, sie zu vermeiden:

1. Vergessen wir nicht, dass die meisten Listenmethoden das Argument modifizieren und `None` zurückgeben. Dies ist das Gegenteil der String-Methoden, die eine neue Zeichenkette zurückgeben und das Original unberührt lassen.

   Wenn man es gewohnt ist, String-Code wie diesen zu schreiben:

   ```
   word = word.strip()
   ```

   dann ist es vermutlich verlockend, auch Listencode wie diesen zu schreiben:

   ```
   t = t.sort()            # FALSCH!
   ```

Da `sort None` zurückgibt, wird die nächste Operation, die wir mit `t` durchführen, wahrscheinlich fehlschlagen.

Bevor man die Methoden und Operatoren von Listen verwendet, sollten man die Dokumentation sorgfältig lesen und sie dann im interaktiven Modus testen. Die Methoden und Operatoren, die Listen mit anderen Sequenzen (z. B. Zeichenketten) gemeinsam haben, sind dokumentiert unter:

docs.python.org/library/stdtypes.html#common-sequence-operations

Die Methoden und Operatoren, die nur für veränderbare Sequenzen gelten, sind hier dokumentiert:

docs.python.org/library/stdtypes.html#mutable-sequence-types

2. Gewöhnen Sie sich einen Stil an.

   Ein Teil des Problems mit Listen ist, dass es zu viele Möglichkeiten gibt, Dinge zu tun. Um zum Beispiel ein Element aus einer Liste zu entfernen, können Sie `pop`, `remove`, `del` oder sogar eine Slice-Zuweisung verwenden.

   Um ein Element hinzuzufügen, kann man die Methode `append` oder den Operator `+` verwenden. Aber nicht vergessen, dass das hier korrekt ist:

   ```
   t.append(x)
   t = t + [x]
   t += [x]
   ```

   Alle drei Anweisungen hängen das Element an die Liste `t` an. Die erste Anweisung verändert das bestehende Listenobjekt `t`. Die zweite Anweiung erzeugt mit `t + [x]` eine neue Liste, die sich durch die Verkettung von `t` und einer Liste mit nur einem Elemnt `x` entsteht. Dieser neuen Liste wird der Namen `t` zugewiesen. und damit das „alte“ `t` verworfen.

   Die dritte Anweisung ist eine Kurzschreibweise des `+`-Operators. Im Allgemeinen (also vor allem bei Zahlen) entspricht `a += b` der Schreibweise `a = a + b`. Auf Listen angewendet, hat dies den Effekt, dass die Liste selbst, also wie bei `append` verändert wird. Wir behalten damit also das alte Listenobjekt.

   Oft ist es unerheblich, ob man mit einem neuen Objekt oder dem bestehenden weiterarbeitet. Wenn es aber zu einem unerwarteten Programmverhalten kommt, können darin die Ursachen bestehen. Wenn Sie genau wissen wollen, ob sich ein Objekt geändert hat, können Sie die eingebaute Funktion `id()` verwenden. `id` zeigt Ihnen zu jedem Objekt die zugehörige, eindeutige ID an. Wenn Sie diese vor und nach einer Zuweisung aufrufen, wissen Sie, ob ein neues Objekt angelegt wurde.

   Neben den richtigen Anweisungen, um eine Liste zu erweitern, gibt es noch viele, die recht ähnlich aussehen, aber leider nicht das gewünschte Ergebnis liefer. Hier einige Beispiele:

   ```
   t.append([x])          # FALSCH!
   t = t.append(x)        # FALSCH!
   t + [x]                # FALSCH!
   t = t + x              # FALSCH!
   ```

Probieren Sie einmal jedes dieser Beispiele im interaktiven Modus aus, um nachzuvollziehen, was die Anweisungen tun. Man sollte beachten, dass nur das letzte Beispiel einen Laufzeitfehler verursacht; die anderen drei sind legal, aber sie tun das Falsche.

3. Kopien erstellen, um die Nutzung von Aliasen zu vermeiden.

   Wenn wir eine Methode wie `sort` verwenden wollen, die das Argument verändert, wir aber die ursprüngliche Liste ebenfalls behalten möchten, müssen wir eine Kopie erstellen.

   ```
   orig = t[:]
   t.sort()
   ```

   In diesem Beispiel könnten wir auch die eingebaute Funktion `sorted` verwenden, die eine neue, sortierte Liste zurückgibt und das Original in Ruhe lässt. Aber in diesem Fall sollte man es vermeiden, `sorted` als Variablennamen zu verwenden!

4. Listen, `split` und Dateien

   Beim Lesen und Parsen von Dateien gibt es viele Gelegenheiten, auf Eingaben zu stoßen, die unser Programm zum Absturz bringen können. Deshalb ist es eine gute Idee, wie bereits in vorherigen Kapiteln mit „Wächtern“ zu arbeiten, wenn es darum geht, Programme zu schreiben, die eine Datei durchlaufen und nach der „Nadel im Heuhaufen“ suchen.

   Schauen wir uns noch einmal unser Programm an, das in den `From`-Zeilen unserer Datei nach dem Wochentag sucht:

   ```
   From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
   ```

   Da wir diese Zeile in Wörter aufteilen, könnten wir auf die Verwendung von `startswith` verzichten und einfach auf das erste Wort der Zeile schauen, um festzustellen, ob wir überhaupt an der Zeile interessiert sind. Wir können `continue` verwenden, um Zeilen, die nicht `From` als erstes Wort haben, wie folgt zu überspringen:

   ```
   fhand = open('mbox-short.txt')
   for line in fhand:
       words = line.split()
       if words[0] != 'From': continue
       print(words[2])
   ```

   Das sieht viel einfacher aus und wir müssen nicht einmal `rstrip` verwenden, um den Zeilenumbruch am Ende der Datei zu entfernen. Aber ist das wirklich besser?

   ```
   python search8.py
   Sat
   Traceback (most recent call last):
     File "search8.py", line 5, in <module>
       if words[0] != 'From': continue
   IndexError: list index out of range
   ```

Es funktioniert zwar irgendwie und wir sehen den Tag aus der ersten Zeile (Sat), aber dann schlägt das Programm mit einem Traceback-Fehler fehl. Was ist schief gelaufen? Welche verpfuschten Daten haben dazu geführt, dass unser elegantes und cleveres Programm fehlschlägt?

Man könnte lange darauf starren und rätseln oder jemanden um Hilfe bitten, aber der schnellere und klügere Ansatz ist, eine `print`-Anweisung hinzuzufügen. Die beste Stelle, um die `print`-Anweisung hinzuzufügen, ist direkt vor der Zeile, in der das Programm fehlgeschlagen ist, und die Daten auszugeben, die den Fehler zu verursachen scheinen.

Nun mag dieser Ansatz eine Menge Zeilen an Ausgabe erzeugen, aber zumindest haben wir sofort einen Anhaltspunkt für das vorliegende Problem. Wir fügen also einen Ausgabe der Variablen `words` direkt vor Zeile 5 ein. Wir fügen der Zeile sogar ein Präfix `Debug:` hinzu, damit wir unsere reguläre Ausgabe von der Debug-Ausgabe getrennt halten können.

```
for line in fhand:
    words = line.split()
    print('Debug:', words)
    if words[0] != 'From': continue
    print(words[2])
```

Wenn wir das Programm ausführen, läuft eine Menge Ausgabe über den Bildschirm, aber am Ende sehen wir unsere Debug-Ausgabe und den Traceback, sodass wir wissen, was kurz vor dem Traceback passiert ist.

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From': continue
IndexError: list index out of range
```

Jede Debug-Zeile gibt die Liste der Wörter aus, die wir erhalten, wenn wir die Zeile mit `split` in Wörter zerlegen. Wenn das Programm fehlschlägt, ist die Liste der Wörter leer `[]`. Wenn wir die Datei in einem Texteditor öffnen und uns die Datei ansehen, sieht sie zu diesem Zeitpunkt wie folgt aus:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Der Fehler tritt auf, wenn unser Programm auf eine Leerzeile stößt! Natürlich stehen in einer Leerzeile 0 Wörter. Warum haben wir nicht daran gedacht, als wir den Code geschrieben haben? Wenn der Code nach dem ersten Wort (`word[0]`) sucht, um zu prüfen, ob es mit `From` übereinstimmt, erhalten wir einen `index out of range`-Fehler.

Dies ist natürlich der perfekte Ort, um einen „Wächter“ einzufügen, der verhindert, dass das erste Wort geprüft wird, wenn das erste Wort nicht vorhanden ist. Es gibt viele Möglichkeiten, diesen Code zu schützen; wir werden uns dafür entscheiden, die Anzahl der vorhandenen Wörter zu prüfen, bevor wir uns das erste Wort ansehen:

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    # print('Debug:', words)
    if len(words) == 0: continue
    if words[0] != 'From': continue
    print(words[2])
```

Zuerst haben wir die Debug-Print-Anweisung auskommentiert, anstatt sie zu entfernen, für den Fall, dass unsere Änderung fehlschlägt und wir erneut debuggen müssen. Dann haben wir eine Wächter-Anweisung hinzugefügt, die prüft, ob wir 0 Wörter haben, und wenn ja, verwenden wir `continue`, um zur nächsten Zeile in der Datei zu springen.

Wir können uns die beiden `continue`-Anweisungen so vorstellen, dass sie uns helfen, die Menge der Zeilen zu reduzieren, die für uns potenziell „interessant“ sind und die wir weiter verarbeiten wollen. Eine Zeile, die keine Wörter enthält, ist für uns „uninteressant“, also springen wir zur nächsten Zeile. Eine Zeile, deren erstes Wort nicht `From` ist, ist für uns uninteressant, also überspringen wir sie.

Das Programm in der geänderten Form läuft erfolgreich, also ist es vielleicht richtig. Unsere Wächter-Anweisung stellt zwar sicher, dass die `Worte[0]` niemals fehlschlagen werden, aber vielleicht ist das nicht genug. Wenn wir programmieren, müssen wir immer daran denken: „Was könnte schiefgehen?“.

Übung 3: Finden Sie heraus, welche Zeile des obigen Programms noch nicht richtig geschützt ist. Versuchen Sie, eine Textdatei zu konstruieren, die das Programm zum Scheitern bringt. Ändern Sie dann das Programm so, dass die Zeile richtig geschützt ist, und testen Sie, ob es Ihre neue Textdatei verarbeitet.

Übung 4: Schreiben Sie den Wächter-Code im obigen Beispiel ohne zwei `if`-Anweisungen um. Verwenden Sie stattdessen einen zusammengesetzten logischen Ausdruck mit dem logischen Operator `or` und einer einzigen `if`-Anweisung.

Glossar

Alias Ein Umstand, bei dem zwei oder mehr Variablen auf das gleiche Objekt verweisen.

Delimiter Ein Zeichen oder eine Zeichenkette, das bzw. die angibt, wo eine Zeichenkette aufgeteilt werden soll.

Element Einer der Werte in einer Liste (oder einer anderen Sequenz); auch Item genannt.

äquivalent Den gleichen Wert haben.

Index Ein ganzzahliger Wert, der ein Element in einer Liste indiziert.
identisch Dasselbe Objekt sein (was Äquivalenz impliziert).
Liste Eine Folge von Werten.
Traversieren einer Liste Der sequentielle Zugriff auf jedes Element in einer Liste.

verschachtelte Liste Eine Liste, die ein Element einer anderen Liste ist.
Objekt Etwas, auf das sich eine Variable beziehen kann. Ein Objekt hat einen Typ und einen Wert.
Referenz Der Assoziation zwischen einer Variablen und ihrem Wert.

Übungen

Übung 5: Alle einzigartigen Wörter in einer Datei finden

Shakespeare hat über 20,000 verschiedene Wörter in seinen Werken verwendet. Aber wie könnte man das feststellen? Wie könnte man eine Liste aller Wörter erstellen, die Shakespeare verwendet hat? Würde man sein gesamtes Werk herunterladen, es lesen und alle einzigartigen Wörter von Hand nachverfolgen?

Lassen Sie uns stattdessen Python verwenden, um das zu erreichen. Listen Sie alle eindeutigen Wörter auf, sortiert in alphabetischer Reihenfolge, die in der Datei `romeo.txt` gespeichert sind, welche eine Teilmenge von Shakespeares Werk enthält.

Um loszulegen, laden Sie eine Kopie der Datei herunter:

tiny.one/py4de/code3/romeo.txt

Erstellen Sie eine Liste mit eindeutigen Wörtern. Schreiben Sie dazu ein Programm, um die Datei `romeo.txt` zu öffnen und Zeile für Zeile zu lesen. Teilen Sie jede Zeile mit der Funktion `split` in eine Liste von Wörtern auf. Prüfen Sie für jedes Wort, ob das Wort bereits in der Liste der eindeutigen Wörter enthalten ist. Wenn das Wort nicht in der Liste der eindeutigen Wörter enthalten ist, fügen Sie es der Liste hinzu. Wenn das Programm mit dem Lesen der Datei fertig, sortieren Sie die Liste der eindeutigen Wörter in alphabetischer Reihenfolge und geben Sie sie aus.

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

Übung 6: Ein minimalistischer E-Mail-Client.

MBOX (Mailbox) ist ein beliebtes Dateiformat zum Speichern und Freigeben einer Sammlung von E-Mails. Dies wurde von frühen E-Mail-Servern und Desktop-Apps verwendet. Ohne zu sehr ins Detail zu gehen, ist MBOX eine Textdatei, die E-Mails fortlaufend speichert. E-Mails werden durch eine spezielle Zeile voneinander getrennt, die mit `From` beginnt (beachten Sie das angehängte Leerzeichen). Wichtig ist, dass Zeilen, die mit `From:` beginnen (beachten Sie den Doppelpunkt), die E-Mail selbst beschreiben und nicht als Trennzeichen fungieren. Stellen Sie sich vor,

Sie schreiben eine minimalistische E-Mail-App, die die E-Mails der Absender im Posteingang des Benutzers auflistet und die Anzahl der E-Mails zählt.

Schreiben Sie ein Programm, das die Daten der Mailbox durchläuft, und wenn Sie eine Zeile finden, die mit `From` beginnt, teilen Sie die Zeile mit der Funktion `split` in Wörter auf. Wir sind daran interessiert, wer die Nachricht gesendet hat, also was das zweite Wort in der `From`-Zeile ist.

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

Sie parsen die `From`-Zeile und geben das zweite Wort für jede `From`-Zeile aus, dann zählen Sie auch die Anzahl der `From`-Zeilen (nicht `From:`) und geben am Ende die Anzahl aus. Dies ist eine Beispielausgabe, bei der ein paar Zeilen entfernt wurden:

```
python fromcount.py
Gib eine Datei an: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
Es gibt 27 Zeilen mit 'From' als erstem Wort.
```

Übung 7: Schreiben Sie das Programm um, das den Benutzer zur Eingabe einer Liste von Zahlen auffordert und am Ende das Maximum und Minimum der Zahlen ausgibt, wenn der Benutzer „done“ eingibt. Schreiben Sie das Programm so, dass es die Zahlen, die der Benutzer eingibt, in einer Liste speichert und die Funktionen `max()` und `min()` verwendet, um die größte und kleinste Zahl nach Abschluss der Schleife zu ermitteln.

```
Bitte eine Zahl eingeben: 6
Bitte eine Zahl eingeben: 2
Bitte eine Zahl eingeben: 9
Bitte eine Zahl eingeben: 3
Bitte eine Zahl eingeben: 5
Bitte eine Zahl eingeben: done
Maximum: 9.0
Minimum: 2.0
```

Kapitel 9

Dictionarys

Listen sind sehr gut geeignet, um alles Mögliche zu speichern, was eine Reihenfolge hat. Sie können bei einer Liste über den *Index* das entsprechende Element finden, also z. B. das erste, das fünfte oder auch das letzte Element. Oftmals ist beim Abspeichern der Daten die Reihenfolge allerdings unerheblich, es kommt eher darauf an, Datensätze schnell „wiederzufinden“.

Stellen Sie sich die Mitarbeiterkartei einer Personalabteilung vor. Als Liste abgelegt würde es bedeuten, dass wir alle Akten übereinander stapeln. Wir könnten zwar schnell eine beliebige Akte herausziehen, um eine bestimmte Akte zu finden, müssten wir aber im Stapel suchen und im Zweifelsfall jede öffnen, um herauszufinden, zu welcher Person sie gehört. Daher legt man Akten unter einem geeigneten Stichwort ab, in diesem Fall z. B. dem Namen der Person. In diesem Kapitel geht es um *Dictionarys*, einem zusammengesetzten Datentyp in Python, mit dem genau solche Ablage- und Zugriffsmöglichkeiten für beliebige Daten möglich sind.

Was ist ein Dictionary

Ein *Dictionary* (zu Deutsch *Wörterbuch*) ist wie eine Liste, aber allgemeiner. In einer Liste müssen die Indexpositionen ganze Zahlen sein; in einem Dictionary können die Indizes (fast) jeden Typs sein.

Man kann sich ein Dictionary als eine Abbildung zwischen einem Satz von Indizes, die *Schlüssel* (englisch *Key*) genannt werden, und einem Satz von Werten (englisch *Value*) vorstellen. Jeder Schlüssel wird auf einen Wert abgebildet. Die Verbindung zwischen einem Schlüssel und einem Wert wird als *Schlüssel-Wert-Paar* (englisch *Key-Value-Pair*) bezeichnet.

Als Beispiel erstellen wir ein Dictionary, das englische und deutsche Wörter abbildet, sodass die Schlüssel und die Werte allesamt Zeichenketten sind.

Die Funktion `dict` erzeugt ein neues Dictionary ohne Elemente. Da `dict` der Name einer eingebauten Funktion ist, sollte man es vermeiden, ihn als Variablennamen zu verwenden.

```
>>> eng2de = dict()
>>> print(eng2de)
{}
```

Die geschweiften Klammern `{}` stehen für ein leeres Dictionary. Um Elemente zum Dictionary hinzuzufügen, können wir eckige Klammern verwenden:

```
>>> eng2de['one'] = 'eins'
```

Diese Zeile erzeugt ein Element, das vom Schlüssel `'one'` auf den Wert `'eins'` abbildet. Wenn wir das Dictionary erneut ausgeben, sehen wir ein Schlüssel-Wert-Paar mit einem Doppelpunkt zwischen dem Schlüssel und dem Wert:

```
>>> print(eng2de)
{'one': 'eins'}
```

Dieses Ausgabeformat ist auch ein Eingabeformat. Wir können zum Beispiel ein neues Dictionary mit drei Einträgen erstellen. Aber wenn man `eng2de` ausgeben möchte, wird man vielleicht überrascht sein:

```
>>> eng2de = {'one': 'eins', 'two': 'zwei', 'three': 'drei'}
>>> print(eng2de)
{'one': 'eins', 'three': 'drei', 'two': 'zwei'}
```

Die Reihenfolge der Schlüssel-Wert-Paare ist nicht die gleiche. Wenn wir das gleiche Beispiel auf unserem Computer eingeben, erhalten wir möglicherweise ein anderes Ergebnis. Im Allgemeinen ist die Reihenfolge der Elemente in einem Dictionary unvorhersehbar.

Das ist aber kein Problem, weil die Elemente eines Dictionarys nie mit ganzzahligen Indizes indiziert sind. Stattdessen verwenden wir die Schlüssel, um die entsprechenden Werte nachzuschlagen:

```
>>> print(eng2de['two'])
'zwei'
```

Der Schlüssel `'two'` bildet immer auf den Wert `'zwei'` ab, sodass die Reihenfolge der Elemente keine Rolle spielt.

Wenn der Schlüssel nicht im Dictionary enthalten ist, erhalten wir eine Ausnahme:

```
>>> print(eng2de['four'])
KeyError: 'four'
```

Die Funktion `len` arbeitet mit Dictionarys; sie gibt die Anzahl der Schlüssel-Wert-Paare zurück:

```
>>> len(eng2de)
3
```

Der `in`-Operator arbeitet mit Dictionarys; er sagt Ihnen, ob etwas als *Schlüssel* im Dictionary vorkommt (als Wert zu erscheinen ist nicht gut genug).

```
>>> 'one' in eng2de
True
>>> 'eins' in eng2de
False
```

Um zu sehen, ob etwas als Wert in einem Dictionary vorkommt, können wir die Methode `values` verwenden, die die Werte als einen Typ zurückgibt, der in eine Liste konvertiert werden kann. Dann verwenden wir den Operator `in`:

```
>>> vals = list(eng2de.values())
>>> 'eins' in vals
True
```

Der Operator `in` verwendet unterschiedliche Algorithmen für Listen und Dictionarys. Für Listen verwendet er einen linearen Suchalgorithmus. Je länger die Liste wird, desto länger wird die Suchzeit im direkten Verhältnis zur Länge der Liste. Für Dictionarys verwendet Python einen Algorithmus, der eine *Hashtabelle* verwendet und eine bemerkenswerte Eigenschaft hat: Der `in`-Operator benötigt jedes Mal ungefähr die gleiche konstante Zeit, egal wie viele Elemente in einem Dictionary vorhanden sind. Ich werde nicht erklären, warum Hash-Funktionen so schnell sind, aber man kann mehr darüber unter de.wikipedia.org/wiki/Hashtabelle lesen.

Übung 1: Laden Sie diese Datei herunter: tiny.one/py4de/code3/words.txt

Schreiben Sie ein Programm, das die Wörter in `words.txt` liest und sie als Schlüssel in einem Dictionary speichert. Es spielt keine Rolle, wie die Werte lauten. Dann können Sie den `in`-Operator als schnelle Möglichkeit verwenden, um zu prüfen, ob eine Zeichenfolge im Dictionary enthalten ist.

Ein Dictionary zum Zählen verwenden

Angenommen, wir verfügen über eine Zeichenkette und möchten zählen, wie oft jeder Buchstabe vorkommt. Es gibt mehrere Möglichkeiten, wie wir das tun können:

1. Wir könnten 26 Variablen erstellen, eine für jeden Buchstaben des Alphabets. Dann könnten wir die Zeichenfolge durchlaufen und für jedes Zeichen den entsprechenden Zähler inkrementieren, wahrscheinlich unter Verwendung einer verketteten Bedingung.
2. Wir könnten eine Liste mit 26 Elementen erstellen. Dann könnten wir jedes Zeichen in eine Zahl umwandeln (mit der eingebauten Funktion `ord`), die Zahl als Index in der Liste verwenden und den entsprechenden Zähler inkrementieren.

3. Wir könnten ein Dictionary mit Zeichen als Schlüssel und Zählern als den entsprechenden Werten erstellen. Das erste Mal, wenn wir ein Zeichen sehen, würden wir dem Dictionary ein Element hinzufügen. Danach würden wir den Wert eines vorhandenen Elements inkrementieren.

Jede dieser Optionen führt dieselbe Berechnung durch, aber jede von ihnen implementiert diese Berechnung auf eine andere Weise.

Eine *Implementierung* ist eine Art und Weise, eine Berechnung durchzuführen; einige Implementierungen sind besser als andere. Ein Vorteil der Dictionaryimplementierung ist zum Beispiel, dass wir nicht im Voraus wissen müssen, welche Buchstaben in der Zeichenkette vorkommen, und wir müssen nur Platz für die Buchstaben schaffen, die tatsächlich vorkommen.

So könnte der Code aussehen:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

Wir berechnen effektiv ein *Histogramm*, was ein statistischer Begriff für einen Satz von Zählern (oder Häufigkeiten) ist.

Die `for`-Schleife durchläuft die Zeichenkette. Jedes Mal, wenn die Schleife durchlaufen wird und das Zeichen, das an `c` zugewiesen wird, nicht im Dictionary enthalten ist, wird ein neues Element mit dem Schlüssel `c` und dem Anfangswert 1 angelegt (da wir diesen Buchstaben schon einmal gesehen haben). Wenn `c` bereits im Dictionary ist, inkrementieren wir `d[c]`.

Hier ist die Ausgabe des Programms:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

Das Histogramm zeigt an, dass die Buchstaben „a“ und „b“ einmal vorkommen; „o“ erscheint zweimal, und so weiter.

Dictionarys haben eine Methode namens `get`, die einen Schlüssel und einen Standardwert annimmt. Wenn der Schlüssel im Dictionary vorkommt, gibt `get` den entsprechenden Wert zurück; andernfalls wird der Standardwert zurückgegeben. Zum Beispiel:

```
>>> counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

Wir können `get` verwenden, um unsere Histogramm-Schleife übersichtlicher zu schreiben. Da die Methode `get` automatisch den Fall behandelt, dass ein Schlüssel nicht in einem Dictionary enthalten ist, können wir vier Zeilen auf eine reduzieren und die `if`-Anweisung eliminieren.

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print(d)
```

Die Verwendung der `get`-Methode zur Vereinfachung dieser Zählschleife ist schließlich ein sehr häufig verwendetes Idiom in Python und wir werden es im weiteren Verlauf des Buches noch oft verwenden. Wir sollten uns also einen Moment Zeit nehmen und die Schleife mit der `if`-Anweisung und dem `in`-Operator mit der Schleife mit der `get`-Methode vergleichen. Wir tun genau das Gleiche, aber die eine ist wesentlich übersichtlicher.

Dictionarys und Dateien

Eine der häufigsten Verwendungen eines Dictionarys ist das Zählen des Vorkommens von Wörtern in einer Datei mit einem geschriebenen Text. Beginnen wir mit einer sehr einfachen Datei mit Wörtern aus dem Text von *Romeo and Juliet*.

Für die erste Reihe von Beispielen werden wir eine verkürzte und vereinfachte Version des Textes ohne Interpunktion verwenden. Später werden wir mit dem Text der Szene mit enthaltener Zeichensetzung arbeiten.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Wir werden ein Python-Programm schreiben, das die Zeilen der Datei liest, jede Zeile in eine Liste von Wörtern zerlegt und dann in einer Schleife jedes der Wörter in der Zeile durchläuft und jedes Wort mithilfe eines Dictionarys zählt.

Wir werden erkennen, dass wir zwei `for`-Schleifen haben. Die äußere Schleife liest die Zeilen der Datei und die innere Schleife iteriert durch jedes der Wörter in dieser bestimmten Zeile. Dies ist ein Beispiel für ein Muster, das *geschachtelte Schleifen* genannt wird, weil eine der Schleifen die *äußere* Schleife und die andere Schleife die *innere* Schleife ist.

Da die innere Schleife alle ihre Iterationen jedes Mal ausführt, wenn die äußere Schleife eine einzelne Iteration macht, können wir uns die innere Schleife als „schneller“ und die äußere Schleife als „langsamer“ iterierend vorstellen.

Die Kombination der beiden verschachtelten Schleifen stellt sicher, dass wir jedes Wort in jeder Zeile der Eingabedatei zählen.

```
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: https://tiny.one/py4de/code3/count1.py
```

In unserer `else`-Anweisung verwenden wir die kompaktere Alternative zur Inkrementierung einer Variablen. `counts[word] += 1` ist äquivalent zu `counts[word] = counts[word] + 1`. Beide Methoden können verwendet werden, um den Wert einer Variablen um einen beliebigen Betrag zu ändern. Ähnliche Kurzschreibweisen gibt es auch für andere Operatoren:

- `a -= b` für `a = a - b`
- `a *= b` für `a = a * b`
- `a /= b` für `a = a / b`

Wenn wir das Programm ausführen, sehen wir einen *Dump* mit allen Zählungen in unsortierter Hash-Reihenfolge. (die Datei *romeo.txt* ist verfügbar unter tiny.one/py4de/code3/romeo.txt)

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

Es ist etwas umständlich, das Dictionary zu durchforsten, um die häufigsten Wörter und ihre Anzahl zu finden, also müssen wir noch etwas Python-Code hinzufügen, um die Ausgabe zu erhalten, die nützlicher für uns sein wird.

Schleifen und Dictionarys

Wenn man ein Dictionary als Sequenz in einer `for`-Anweisung verwendet, durchläuft diese die Schlüssel des Dictionarys. Diese Schleife gibt jeden Schlüssel und den entsprechenden Wert aus:

```
counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
for key in counts:
    print(key, counts[key])
```

So sieht die Ausgabe aus:

```
jan 100
chuck 1
annie 42
```

Auch hier sind die Schlüssel in keiner bestimmten Reihenfolge.

Wir können dieses Muster verwenden, um die verschiedenen Schleifen-Idiome zu implementieren, die wir zuvor beschrieben haben. Wenn wir zum Beispiel alle Einträge in einem Dictionary mit einem Wert über zehn finden wollen, könnten wir folgenden Code schreiben:

```
counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
for key in counts:
    if counts[key] > 10:
        print(key, counts[key])
```

Die `for`-Schleife iteriert über die *Schlüssel* des Dictionarys, also müssen wir den Indexoperator verwenden, um den entsprechenden *Wert* für jeden Schlüssel abzurufen. So sieht die Ausgabe aus:

```
jan 100
annie 42
```

Wir sehen nur die Einträge mit einem Wert über 10.

Wenn wir die Schlüssel in alphabetischer Reihenfolge ausgeben möchten, erstellen wir zunächst eine Liste der Schlüssel des Dictionarys mit der in Dictionaryobjekten verfügbaren Methode `keys`. Dann sortieren diese Liste und iterieren anschließend durch sie, wobei wir jeden Schlüssel nachschlagen und die Schlüssel-Wert-Paare wie folgt ausgeben:

```
counts = { 'chuck': 1 , 'annie': 42, 'jan': 100}
lst = list(counts.keys())
print(lst)
lst.sort()
for key in lst:
    print(key, counts[key])
```

So sieht die Ausgabe aus:

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

Zuerst sehen wir die Liste der Schlüssel in unsortierter Reihenfolge, die wir von der Methode `keys` erhalten. Dann sehen wir die Schlüssel-Wert-Paare in der Reihenfolge aus der `for`-Schleife.

Fortgeschrittene Textanalyse

Im obigen Beispiel mit der Datei `romeo.txt` haben wir die Datei so einfach wie möglich gemacht, indem wir alle Satzzeichen von Hand entfernt haben. Der ursprüngliche Text enthält viele Satzzeichen, wie unten gezeigt.

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Da die Python-Funktion `split` nach Leerzeichen sucht und Wörter als durch Leerzeichen getrennte Token behandelt, würden wir die Wörter „soft!“ und „soft“ als *unterschiedliche* Wörter behandeln und für jedes Wort einen eigenen Dictionary-Eintrag erstellen.

Da die Datei außerdem Groß- und Kleinschreibung aufweist, würden wir „who“ und „Who“ als unterschiedliche Wörter mit unterschiedlicher Zählung behandeln.

Wir können diese beiden Probleme mit den String-Methoden `lower`, `punctuation`, `translate` und `maketrans` lösen.

Die Funktion `maketrans()` wird verwendet, um eine Ersetzungstabelle zu erstellen. Darin werden die Zeichen angegeben, die in der gesamten Zeichenfolge ersetzt bzw. gelöscht werden sollen. Der folgende Aufruf erzeugt eine Ersetzungstabelle `ttable`, bei der die Zeichen in `fromstr` durch die jeweils an der gleichen Position stehenden Zeichen in `tostr` ersetzt werden. Die in der Liste `deletestr` stehenden Zeichen werden gelöscht. `fromstr` und `tostr` können leere Strings sein und der Parameter `deletestr` kann weggelassen werden.

```
ttable = str.maketrans(fromstr, tostr, deletestr)
```

Mit der Methode `translate` kann die Ersetzungstabelle dann auf eine Zeichenkette angewndet werden:

```
line.translate(ttable)
```

Für unseren Zweck werden wir den Parameter `tostr` nicht angeben, aber wir werden den Parameter `deletestr` verwenden, um alle Satzzeichen zu löschen. Wir lassen uns sogar von Python die Liste der Zeichen mitteilen, die es als „Interpunktion“ betrachtet:

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Wir nehmen die folgenden Änderungen an unserem Programm vor:

```
import string

fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    exit()

counts = dict()
for line in fhand:
    line = line.rstrip()
    line = line.translate(
                line.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print(counts)

# Code: https://tiny.one/py4de/code3/count2.py
```

Ein Teil des Erlernens der „Kunst von Python“ ist die Erkenntnis, dass Python oft eingebaute Fähigkeiten für viele gängige Datenanalyseprobleme hat. Mit der Zeit werden wir genug Beispielcode gesehen haben und genug von der Dokumentation gelesen haben, um zu wissen, wo man nachschauen muss. Vielleicht hat jemand bereits etwas geschrieben, das unsere Arbeit viel einfacher macht.

Im Folgenden finden wir eine gekürzte Version der Ausgabe:

```
Gib einen Dateinamen ein: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

Das Durchsehen dieser Ausgabe ist immer noch unhandlich, aber wir können Python verwenden, um uns genau das ausgeben zu lassen, wonach wir suchen. Aber dazu müssen wir zunächst *Tupel* kennenlernen. Wir werden dieses Beispiel aufgreifen, sobald wir etwas über Tupel gelernt haben.

Debugging

Wenn wir mit größeren Datensätzen arbeiten, kann das Debuggen durch Ausgeben und Prüfen der Daten von Hand umständlich werden. Hier sind einige Vorschläge für die Fehlersuche in großen Datensätzen:

Reduzieren der Datenmenge Reduzieren wir, wenn möglich, die Größe des Datensatzes. Wenn das Programm z. B. eine Textdatei liest, beginnen wir nur mit den ersten 10 Zeilen, oder mit dem kleinsten Beispiel, das wir finden können. Wir können entweder die Dateien selbst bearbeiten, oder (besser) das Programm so modifizieren, dass es nur die ersten `n` Zeilen liest.

Wenn ein Fehler vorliegt, können wir `n` auf den kleinsten Wert reduzieren, der den Fehler manifestiert, und ihn dann schrittweise erhöhen, während wir Fehler finden und korrigieren.

Informationen extrahieren und Datentypen prüfen Anstatt den gesamten Datensatz auszugeben und zu prüfen, sollten wir in Erwägung ziehen, nur bestimmte Informationen auszugeben: z. B. die Anzahl der Elemente in einem Dictionary oder die Gesamtsumme einer Liste von Zahlen.

Eine häufige Ursache für Laufzeitfehler ist ein Wert, der nicht den richtigen Typ hat. Zur Fehlersuche bei dieser Art von Fehlern reicht es oft aus, den Typ eines Wertes auszugeben.

Plausibilitätsprüfungen Manchmal können wir Code schreiben, um automatisch auf Fehler zu prüfen. Wenn wir beispielsweise den Durchschnitt einer Liste von Zahlen berechnen, könnten wir prüfen, dass das Ergebnis nicht größer als das größte Element in der Liste oder kleiner als das kleinste ist. Dies wird als „Plausibilitätsprüfungen“ bezeichnet, weil es Ergebnisse erkennt, die „völlig unlogisch“ sind.

Eine andere Art der Prüfung vergleicht die Ergebnisse von zwei verschiedenen Berechnungen, um zu sehen, ob sie konsistent sind. Dies wird als „Konsistenzprüfung“ bezeichnet.

Formatierung der Debugging-Ausgabe Eine Formatierung der Debugging-Ausgabe kann es erleichtern, einen Fehler zu erkennen.

Auch hier kann die Zeit, die wir mit dem Aufbau eines stabilen „Gerüsts“ verbringen, die Zeit reduzieren, die wir für die Fehlersuche aufwenden müssen.

Glossar

Dictionary Eine Abbildung von einer Menge von Schlüsseln auf ihre entsprechenden Werte.

Hash-Tabelle Der Algorithmus, der zur Implementierung von Python-Dictionarys verwendet wird.

Hashfunktion Eine Funktion, die von einer Hashtabelle verwendet wird, um den Speicherort für einen Schlüssel zu berechnen.

Histogramm Ein Menge von Zählern.

Implementierung Eine Art, eine Berechnung durchzuführen.

Schlüssel Ein Objekt, das in einem Dictionary als erster Teil eines Schlüssel-Wert-Paares erscheint.

Schlüssel-Wert-Paar Die Darstellung des Mappings von einem Schlüssel auf einen Wert.

Lookup Eine Dictionaryoperation, die einen Schlüssel nimmt und den entsprechenden Wert findet (Lookup = Nachschlagen im Dictionary).

verschachtelte Schleifen Wenn es eine oder mehrere Schleifen innerhalb einer anderen Schleife gibt. Die innere Schleife läuft jedes Mal zu Ende, wenn die äußere Schleife einmal läuft.

Wert Ein Objekt, das in einem Dictionary als zweiter Teil eines Schlüssel-Wert-Paares erscheint. Dies ist spezifischer als unsere bisherige Verwendung des Begriffs *Wert*.

Übungen

Übung 2: Schreiben Sie ein Programm, das jede E-Mail-Nachricht danach kategorisiert, an welchem Wochentag sie versendet wurde. Suchen Sie dazu nach Zeilen, die mit „From“ beginnen, suchen Sie dann nach dem dritten Wort und führen Sie eine laufende Zählung der einzelnen Wochentage durch. Am Ende des Programms geben Sie den Inhalt Ihres Dictionarys aus (die Reihenfolge spielt keine Rolle).

Beispiel:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

Beispiel:

```
python dow.py
Gib eine Datei an: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

Übung 3: Schreiben Sie ein Programm, um ein E-Mail-Protokoll zu lesen. Erzeugen Sie ein Histogramm mithilfe eines Dictionarys, um zu zählen, wie viele Nachrichten von den einzelnen E-Mail-Adressen gekommen sind. Geben Sie das Dictionary dann aus.

```
Gib eine Datei an: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

Übung 4: Fügen Sie folgendes dem obigen Programmcode hinzu, um herauszufinden, wer die meisten Nachrichten in der Datei erhalten hat. Nachdem alle Daten

gelesen und das Dictionary erstellt wurde, durchsuchen Sie das Dictionary mithilfe einer Schleife nach dem Maximum (siehe Kapitel 5: Maximum und Minimum ermitteln), um herauszufinden, wer die meisten Nachrichten hat, und geben Sie aus, wie viele Nachrichten die Person bekommen hat.

```
Gib eine Datei an: mbox-short.txt
cwen@iupui.edu 5

Gib eine Datei an: mbox.txt
zqian@umich.edu 195
```

Übung 5: Dieses Programm zeichnet nur den Domänennamen (anstelle der Adresse) auf, von dem die Nachricht gesendet wurde, also nicht, von welchem konkreten Absender die E-Mail kam (das wäre die gesamte E-Mail-Adresse). Geben Sie am Ende des Programms den Inhalt Ihres Dictionarys aus.

```
python schoolcount.py
Gib eine Datei an: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

Kapitel 10

Tupel

Den Begriff *Tupel* sollten Sie aus der Mathematik kennen. Dort bezeichnet er eine (endlich lange) Liste von nicht notwendigerweise voneinander verschiedenen Objekten. Da wir bereits Listen in Python haben, benötigen wir keine Tupel, richtig? Falsch! Denn im Gegensatz zu Listen können bei einem Tupel die einzelnen Werte zur Laufzeit des Programms nicht verändert werden. Und diese Eigenschaft macht tatsächlich einen großen Unterschied.

Tupel sind unveränderbar

Ein Tupel ist eine Folge von Werten. Die in einem Tupel gespeicherten Werte können von beliebigem Typ sein und sie werden durch Ganzzahlen indiziert. Alles genau wie bei Listen, nur eben mit dem wichtigen Unterschied, dass Tupel *unveränderlich* sind. Tupel sind außerdem *vergleichbar* und können *gehasht* werden, sodass wir Listen von ihnen sortieren und Tupel als Schlüsselwerte in Python-Dictionarys verwenden können.

Syntaktisch ist ein Tupel eine durch Kommata getrennte Liste von Werten:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Obwohl es nicht notwendig ist, ist es üblich, Tupel in Klammern einzuschließen, damit wir Tupel schnell identifizieren können, wenn wir uns Python-Code ansehen:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Um ein Tupel mit einem einzelnen Element zu erstellen, müssen wir das abschließende Komma einfügen:

```
>>> t1 = (1,)
>>> type(t1)
<type 'tuple'>
```

Ohne das Komma würde die `(1)` für den Python-Interpreter wie eine geklammerte Ganzzahl aussehen und damit wie ein ganz normaler Ausdruck. Erst das Komma vor der Klammer macht deutlich, dass hier etwas anderes, nämlich ein *Eintupel* gemeint ist.

Eine weitere Möglichkeit, ein Tupel zu konstruieren, ist die eingebaute Funktion `tuple`. Ohne Argument erzeugt sie ein leeres Tupel:

```
>>> t = tuple()
>>> print(t)
()
```

Wenn das Argument eine Sequenz (String, Liste oder Tupel) ist, ist das Ergebnis des Aufrufs von `tuple` ein Tupel mit den Elementen der Sequenz:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Da `tuple` der Name eines *Konstruktors* ist, sollten wir es vermeiden, ihn als Variablennamen zu verwenden.

Die meisten Listenoperatoren funktionieren auch auf Tupeln. Der Indexoperator indiziert ein Element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

Und der Slice-Operator selektiert einen Bereich von Elementen.

```
>>> print(t[1:3])
('b', 'c')
```

Wenn wir jedoch versuchen, eines der Elemente des Tupels zu ändern, erhalten wir einen Fehler:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Wir können die Elemente eines Tupels nicht ändern, aber wir können ein Tupel durch ein anderes ersetzen:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

Vergleichen von Tupeln

Die Vergleichsoperatoren funktionieren mit Tupeln und anderen Sequenzen. Python beginnt mit dem Vergleich des jeweils ersten Elements aus jeder Sequenz. Wenn beide gleich sind, geht es weiter zum nächsten Element und so weiter, bis zwei Elemente gefunden wurden, die sich unterscheiden. Diese werden entsprechend dem verwendeten Operator verglichen. Damit endet der Vergleich der Tupel. Alle danach kommenden Elemente werden also nicht mehr berücksichtigt.

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Die Funktion `sort` arbeitet auf die gleiche Weise. Sie sortiert primär nach dem ersten Element, aber im Falle eines Gleichstandes nach dem zweiten Element und so weiter.

Diese Funktion eignet sich für ein Muster namens *DSU*:

Decorate „Dekorieren“ einer Sequenz, indem eine Liste von Tupeln mit einem oder mehreren Sortierschlüsseln den Elementen der Sequenz vorangestellt werden.
Sort Sortieren der Liste der Tupel mit dem in Python eingebauten `sort`.
Undecorate Extrahieren der sortierten Elemente der Sequenz.

Angenommen, wir haben eine Liste von Wörtern und möchten diese nach den Wortlängen absteigend sortieren:

```
txt = 'Es war die Nachtigall und nicht die Lerche'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print(res)

# Code: https://tiny.one/py4de/code3/soft.py
```

Die erste Schleife baut eine Liste von Tupeln auf, wobei jedes Tupel ein Wort ist, dem seine Länge vorangestellt ist.

`sort` vergleicht das erste Element (die Länge) zuerst und berücksichtigt nur das zweite Element, um Gleichstände aufzulösen. Das Schlüsselwortargument `reverse=True` sagt `sort`, dass es in absteigender Reihenfolge vorgehen soll.

Die zweite Schleife durchläuft die Liste aller Tupel und baut eine Liste der Wörter nach absteigender Wortlänge auf. Gleichlange Wörter werden in *umgekehrter* alphabetischer Reihenfolge sortiert, sodass „war“ in der folgenden Liste vor „und“ erscheint.

Die Ausgabe des Programms ist wie folgt:

```
['Nachtigall', 'Lerche', 'nicht', 'war', 'und', 'die', 'die', 'Es']
```

Natürlich verliert die Zeile viel von ihrer poetischen Wirkung, wenn man sie in eine Python-Liste verwandelt und in absteigender Reihenfolge der Wortlänge sortiert.

Tupel-Zuweisung

Eine der einzigartigen syntaktischen Eigenschaften der Sprache Python ist die Fähigkeit, ein Tupel auf der linken Seite einer Zuweisung haben zu können. Dadurch können wir mehr als eine Variable auf einmal zuweisen, wenn die linke Seite eine Sequenz ist.

In diesem Beispiel haben wir eine zweielementige Liste (die eine Sequenz ist) und weisen das erste und zweite Element der Sequenz den Variablen `x` und `y` in einer einzigen Anweisung zu.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

Es ist keine Magie, Python übersetzt die Tupel-Zuweisungssyntax in etwa so, dass sie wie folgt aussieht:[1]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

[1] Python übersetzt die Syntax nicht wörtlich. Wenn wir dies zum Beispiel mit einem Dictionary versuchen, wird es nicht so funktionieren, wie wir es vielleicht erwarten.

Stilistisch gesehen lassen wir die Klammern weg, wenn wir ein Tupel auf der linken Seite der Zuweisungsanweisung verwenden, aber das Folgende ist eine ebenso gültige Schreibweise:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

Eine besonders clevere Anwendung der Tupel-Zuweisung erlaubt es uns, die Werte zweier Variablen in einer einzigen Anweisung zu *tauschen*:

```
>>> a, b = b, a
```

Beide Seiten dieser Anweisung sind Tupel, aber die linke Seite ist ein Tupel von Variablen; die rechte Seite ist ein Tupel von Ausdrücken. Jeder Wert auf der rechten Seite wird der entsprechenden Variablen auf der linken Seite zugewiesen. Alle Ausdrücke auf der rechten Seite werden vor ihrer Zuweisungen ausgewertet.

Die Anzahl der Variablen auf der linken Seite und die Anzahl der Werte auf der rechten Seite müssen zueinander passen:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Wir sollten noch ein wenig näher darauf eingehen, was „zueinander passen“ hier bedeutet. `a` und `b` sind in der obigen Zuweisung einfach nur Namen. Python, als dynamisch typisierte Programmiersprache, wird zur Laufzeit entscheiden, welche Werte `a` und `b` bekommen sollen. Nun könnte einer der beiden Namen für eine Liste stehen, in der mehrere Werte untergebracht sind. Dann würde die Zuweisung oben wieder „passen“. Nur kann Python hier nicht wissen, ob `a` oder `b` diese Liste sein soll.

Um das Beispiel oben doch funktionstüchtig zu machen, müssen wir die Information hinzufügen, welcher Name für eine Sequenz von Werten stehen soll. Das tun wir mit einem vorangestellten Sternchen `*` (englisch *Asterisk*):

```
>>> a, *b = (1,2,3)
>>> b
[2, 3]
```

Es gibt noch weitere Stellen in Python, an denen man mit einem vorangestellten Asterisk ausdrückt, dass mehrere einzelne Werte *zusammengefasst* oder aber eine Sequenz zu einzelnen Werten *entpackt* wird. Dieses Konzept nennt man dementsprechend *Packing* bzw. *Unpacking*. Funktionen verwenden es z. B., um eine beliebig lange Liste von Parametern zuzulassen.

Beachten Sie übrigens, dass im Beispiel oben das Resultat in `b` eine Liste ist, obwohl auf der linken Seite der Zuweisung ein Tupel steht. Beim Packing werden die einzelenen Elemente immer in eine Liste gepackt, ganz gleich aus welcher Datenstruktur sie entnommen werden.

(Un)packing lässt sich nicht nur auf numerische Werte anwenden, sondern auch auf andere Typen wie etwa Zeichenketten. Um z. B. eine E-Mail-Adresse in einen Benutzernamen und eine Domäne aufzuteilen, könnten wir schreiben:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Der Rückgabewert von `split` ist eine Liste mit zwei Elementen; das erste Element wird `uname` zugewiesen, das zweite `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

Dictionarys und Tupel

Dictionarys haben eine Methode namens `items`, die eine Liste von Tupeln zurückgibt, wobei jedes Tupel ein Schlüssel-Wert-Paar ist:

```
>>> d = {'b':1, 'a':13, 'c':7}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 13), ('c', 7)]
```

Wie wir es vermutlich von einem Dictionary erwarten, sind die Begriffe in keiner bestimmten Reihenfolge angeordnet.

Da die Liste der Tupel jedoch eine Liste ist und Tupel vergleichbar sind, können wir nun die Liste der Tupel sortieren. Das Konvertieren eines Dictionarys in eine Liste von Tupeln ist eine Möglichkeit, den Inhalt eines Dictionarys nach Schlüssel sortiert auszugeben:

```
>>> d = {'b':1, 'a':13, 'c':7}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 13), ('c', 7)]
>>> t.sort()
>>> t
[('a', 13), ('b', 1), ('c', 7)]
```

Die neue Liste wird in aufsteigender alphabetischer Reihenfolge nach dem Schlüsselwert sortiert.

Mehrfachzuweisung mit Dictionarys

Durch die Kombination von `items`, Tupel-Zuweisung und `for` können wir ein schönes Codeschema für das Durchlaufen der Schlüssel und Werte eines Dictionarys in einer einzigen Schleife sehen:

```
d = {'b':1, 'a':13, 'c':7}
for key, val in list(d.items()):
    print(val, key)
```

Diese Schleife hat zwei *Iterationsvariablen*, weil `items` eine Liste von Tupeln zurückgibt und `key, val` eine Tupel- bzw. *Unpacking*-Zuweisung ist, die nacheinander durch jedes der Schlüssel-Wert-Paare im Dictionary iteriert.

Bei jeder Iteration durch die Schleife werden sowohl `key` als auch `val` zum nächsten Schlüssel-Wert-Paar im Dictionary vorgerückt (immer noch in Hash-Reihenfolge).

Der Ausgang dieser Schleife ist:

```
1 b
13 a
7 c
```

Auch hier gilt wieder die Reihenfolge der Hash-Schlüssel (d. h. letztendlich keine bestimmte Reihenfolge).

Wenn wir diese beiden Techniken kombinieren, können wir den Inhalt eines Dictionarys sortiert nach dem *Wert*, der in jedem Schlüssel-Wert-Paar gespeichert ist, ausgeben.

Um dies zu tun, erstellen wir zunächst eine Liste von Tupeln, wobei jedes Tupel `(value, key)` ist. Die Methode `items` würde uns eine Liste von `(key, value)`-Tupeln liefern, aber dieses Mal wollen wir nach Wert und nicht nach Schlüssel sortieren. Sobald wir die Liste mit den Wert-Schlüssel-Tupeln aufgebaut haben, ist es eine einfache Sache, die Liste in umgekehrter Reihenfolge zu sortieren und die neue, sortierte Liste auszugeben.

```
>>> d = {'b':1, 'a':13, 'c':7}
>>> l = list()
>>> for key, val in d.items():
...     l.append( (val, key) )
...
>>> l
[(1, 'b'), (13, 'a'), (7, 'c')]
>>> l.sort(reverse=True)
>>> l
[(13, 'a'), (7, 'c'), (1, 'b')]
>>>
```

Indem wir die Liste der Tupel sorgfältig so konstruieren, dass der Wert das erste Element jedes Tupels ist, können wir die Liste der Tupel sortieren und erhalten den Inhalt unseres Dictionarys nach dem Wert sortiert.

Worthäufigkeit zählen

Wenn wir zu unserem laufenden Beispiel des Textes aus *Romeo und Juliet* Akt 2, Szene 2 zurückkehren, können wir unser Programm erweitern, um diese Technik zu verwenden und die zehn häufigsten Wörter im Text wie folgt auszugeben:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(str.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
for key, val in list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]:
    print(key, val)

# Code: https://tiny.one/py4de/code3/count3.py
```

Der erste Teil des Programms, der die Datei liest und das Dictionary erstellt, welches jedes Wort auf die Anzahl der Wörter im Dokument abbildet, bleibt unverändert. Aber anstatt einfach `counts` auszugeben und das Programm zu beenden, konstruieren wir eine Liste von `(val, key)`-Tupeln und sortieren die Liste dann in umgekehrter Reihenfolge.

Da der Wert an erster Stelle steht, wird er für die Vergleiche verwendet. Wenn es mehr als ein Tupel mit demselben Wert gibt, wird das zweite Element (der Schlüssel) betrachtet, sodass Tupel, bei denen der Wert gleich ist, weiter nach der alphabetischen Reihenfolge des Schlüssels sortiert werden.

Am Ende schreiben wir eine `for`-Schleife, die in jeder Iteration eine Mehrfachzuweisung durchführt und die zehn häufigsten Wörter ausgibt, indem sie durch einen Ausschnitt der Liste (`lst[:10]`) iteriert.

Jetzt sieht die Ausgabe endlich so aus, wie wir es uns für unsere Worthäufigkeitsanalyse wünschen.

```
61 i
```

```
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

Die Tatsache, dass dieses komplexe Parsen und Analysieren von Daten mit einem leicht verständlichen 19-zeiligen Python-Programm durchgeführt werden kann, ist ein Grund, warum Python eine gute Wahl als Sprache für die Erforschung von Informationen ist.

Tupel als Schlüssel in Dictionarys

Da Tupel *gehasht* werden können und Listen nicht, müssen wir ein Tupel als Schlüssel verwenden, wenn wir einen *zusammengesetzten* Schlüssel zur Verwendung in einem Dictionary erstellen wollen.

Wir würden auf einen zusammengesetzten Schlüssel stoßen, wenn wir ein Telefonverzeichnis erstellen wollten, das von Nachnamen-Vornamen-Paaren auf Telefonnummern abbildet. Unter der Annahme, dass wir die Variablen `last`, `first` und `number` definiert haben, könnten wir eine Dictionaryzuweisung wie folgt schreiben:

```
directory[last,first] = number
```

Der Ausdruck in Klammern ist ein Tupel. Wir könnten die Tupel-Zuweisung in einer `for`-Schleife verwenden, um dieses Dictionary zu durchlaufen.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

Diese Schleife durchläuft die Schlüssel in `directory`, bei denen es sich um Tupel handelt. Sie ordnet die Elemente jedes Tupels `last` und `first` zu und gibt dann den Namen und die entsprechende Telefonnummer aus.

Zeichenketten, Listen und Tupel

Ich habe mich hierbei auf Listen von Tupeln konzentriert, aber fast alle Beispiele in diesem Kapitel arbeiten auch mit Listen von Listen, Tupeln von Tupeln und Tupeln von Listen. Um die Aufzählung der möglichen Kombinationen zu vermeiden, ist es manchmal einfacher, von „Sequenzen von Sequenzen“ zu sprechen.

In vielen Kontexten können die verschiedenen Arten von Sequenzen (Strings, Listen und Tupel) austauschbar verwendet werden. Wie und warum wählen wir also eine der beiden Arten aus?

Um mit dem Offensichtlichen zu beginnen: Zeichenketten sind begrenzter als andere Sequenzen, da die Elemente Zeichen sein müssen. Außerdem sind sie unveränderlich. Wenn wir die Möglichkeit benötigen, die Zeichen in einer Zeichenfolge zu ändern (im Gegensatz zur Erstellung einer neuen Zeichenfolge), sollten wir stattdessen eine Liste von Zeichen verwenden.

Listen sind üblicher als Tupel, hauptsächlich weil sie veränderbar sind. Aber es gibt ein paar Fälle, in denen wir Tupel bevorzugen könnten:

1. In manchen Kontexten, wie z. B. einer `return`-Anweisung, ist es syntaktisch einfacher, ein Tupel zu erstellen als eine Liste. In anderen Kontexten bevorzugen wir vielleicht eine Liste.

2. Wenn wir eine Sequenz als Dictionaryschlüssel verwenden möchten, müssen wir einen unveränderlichen Typ wie ein Tupel oder einen String verwenden.

3. Wenn wir eine Sequenz als Argument an eine Funktion übergeben, verringert die Verwendung von Tupeln das Potenzial für unerwartetes Verhalten aufgrund von Aliasbildung.

Da Tupel unveränderlich sind, bieten sie keine Methoden wie `sort` und `reverse`, die bestehende Listen verändern. Python bietet jedoch die eingebauten Funktionen `sorted` und `reversed`, die eine beliebige Sequenz als Parameter nehmen und eine neue Sequenz mit denselben Elementen in einer anderen Reihenfolge zurückgeben.

Debugging

Listen, Dictionarys und Tupel sind allgemein als *Datenstrukturen* bekannt; in diesem Kapitel beginnen wir, zusammengesetzte Datenstrukturen zu nutzen, wie Listen von Tupeln und Dictionarys, die Tupel als Schlüssel und Listen als Werte enthalten. Zusammengesetzte Datenstrukturen sind nützlich, aber sie sind anfällig für das, was man als *Formatfehler* bezeichnen könnte; das heißt, Fehler, die entstehen, wenn eine Datenstruktur den falschen Typ, die falsche Größe oder die falsche Zusammensetzung hat, oder wenn wir vielleicht etwas Code schreiben und das Format der Daten vergessen und einen Fehler einführen. Wenn wir zum Beispiel eine Liste mit einer Ganzzahl erwarten und ich dem Programm eine einfache Ganzzahl (nicht in einer Liste) gebe, wird es nicht funktionieren.

Glossar

vergleichbar Ein Datentyp, bei dem ein Wert daraufhin überprüft werden kann, ob er größer, kleiner oder gleich einem anderen Wert desselben Typs ist. Typen, die vergleichbar sind, können in eine Liste eingefügt und sortiert werden.

Datenstruktur Eine Sammlung zusammengehöriger Werte, oft organisiert in Listen, Dictionarys, Tupeln usw.

DSU Abkürzung für „Decorate-Sort-Undecorate“, ein Muster, bei dem eine Liste von Tupeln erstellt, sortiert und ein Teil des Ergebnisses extrahiert wird.

hashbar Ein Datentyp, der eine Hash-Funktion hat. Unveränderliche Typen wie Ganzzahlen, Fließkommazahlen und Zeichenketten sind hashfähig; veränderliche Typen wie Listen und Dictionarys können nicht gehasht werden.

Singleton Eine Liste (oder andere Sequenz) mit einem einzelnen Element.

Tupel Eine unveränderliche Folge von Elementen.

Tupelzuweisung Eine Zuweisung mit einer Sequenz auf der rechten Seite und einem Tupel von Variablen auf der linken Seite. Die rechte Seite wird ausgewertet und dann werden ihre Elemente den Variablen auf der linken Seite zugewiesen.

Übungen

Übung 1: Das vorherige Programm soll auf folgende Weise überarbeitet werden: Die `From`-Zeilen sollen gelesen und geparst werden, um daraus die Adressen zu extrahieren. Dabei soll mit Hilfe eines Dictionarys die Anzahl der Nachrichten von jeder Person gezählt werden.

Nachdem alle Daten gelesen wurden, soll die Person mit den meisten Nachrichten ausgegeben werden, indem eine Liste von `(count, email)`-Tupel aus dem Dictionary erstellt wird. Dann muss die Liste in umgekehrter Reihenfolge sortiert und schließlich die Person mit den meisten Nachrichten ausgegeben werden.

```
Gib eine Datei an: mbox-short.txt
cwen@iupui.edu 5

Gib eine Datei an: mbox.txt
zqian@umich.edu 195
```

Übung 2: Schreiben Sie ein Programm, welches die Häufigkeit von Nachrichten pro Tageszeit (ganze Stunden) ermittelt. Die Stunden sollen aus der `From`-Zeile extrahiert werden, indem die Uhrzeit gefunden und diese Zeichenfolge dann anhand des Doppelpunkts in Teile zergliedert wird. Sobald die Nachrichtenhäufigkeit für jede Stunde gesammelt wurde, sollen diese wie weiter unten gezeigt ausgegeben werden (eine pro Zeile, sortiert nach Stunde).

```
python timeofday.py
Gib eine Datei an: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
```

```
17 2
18 1
19 1
```

Übung 3: Schreiben Sie ein Programm, das eine Datei liest und die *Buchstaben* in absteigender Reihenfolge der Häufigkeit ausgibt. Das Programm sollte alle Eingaben in Kleinbuchstaben umwandeln und nur die Buchstaben a-z zählen. Es soll keine Leerzeichen, Ziffern, Interpunktionszeichen oder irgendetwas anderes als die Buchstaben a-z zählen. Dann sollen Textbeispiele in verschiedenen Sprachen als Eingabe dienen, um zu analysieren, wie die Buchstabenhäufigkeit zwischen verschiedenen Sprachen variiert. Vergleichen Sie ihre Ergebnisse mit den Tabellen auf https://wikipedia.org/wiki/Letter_frequencies

Kapitel 11

Reguläre Ausdrücke

Bisher haben wir uns durch Dateien gelesen, nach Mustern gesucht und verschiedene Teile von Zeilen extrahiert, die wir interessant finden. Wir haben String-Methoden wie `split` und `find` benutzt sowie Listen- und String-Slicing verwendet, um Teile von Texten zu extrahieren.

Diese Aufgabe des Suchens und Extrahierens begegnet einem so häufig, dass Python eine sehr mächtige Bibliothek für *reguläre Ausdrücke* hat, die viele dieser Aufgaben recht elegant erledigt. Der Grund, warum wir reguläre Ausdrücke nicht früher im Buch eingeführt haben, ist, dass sie zwar sehr mächtig, aber auch etwas kompliziert sind und ihre Syntax etwas gewöhnungsbedürftig ist.

Reguläre Ausdrücke (englisch *regular Expressions* oder kurz *regex*) sind fast eine eigene kleine Programmiersprache zum Suchen und Parsen von Zeichenketten. Tatsächlich sind ganze Bücher über das Thema reguläre Ausdrücke geschrieben worden und auch der Python-Interpreter verwendet, immer wenn er unsere Programme „verstehen“ möchte, reguläre Ausdrücke an.

In diesem Kapitel werden wir nur die Grundlagen der regulären Ausdrücke behandeln. Weitere Details zu regulären Ausdrücken finden sich unter:

https://de.wikipedia.org/wiki/Regulärer_Ausdruck

https://docs.python.org/library/re.html

Die Bibliothek für reguläre Ausdrücke `re` muss in das Programm importiert werden, bevor sie verwendet werden kann. Die einfachste Verwendung der Bibliothek für reguläre Ausdrücke ist die Funktion `search()`. Das folgende Programm demonstriert eine triviale Verwendung der Suchfunktion.

```
# Finde Zeilen, die ein 'From' beinhalten
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print(line)
```

```
# Code: https://tiny.one/py4de/code3/re01.py
```

Wir öffnen die Datei, durchlaufen jede Zeile in einer Schleife und verwenden `search()`, um nur Zeilen auszugeben, die die Zeichenkette `From:` enthalten. Dieses Programm nutzt nicht die wirkliche Macht der regulären Ausdrücke, da wir genauso gut `line.find()` hätten verwenden können, um das gleiche Ergebnis zu erzielen.

Die Stärke der regulären Ausdrücke kommt zum Tragen, wenn wir dem Suchstring Sonderzeichen hinzufügen, mit denen wir genauer steuern können, welche Zeilen auf den String passen. Durch das Hinzufügen dieser Sonderzeichen zu unserem regulären Ausdruck können wir anspruchsvolle Abgleiche und Extraktionen durchführen und dabei sehr wenig Code schreiben.

Der Zirkumflex (^) wird zum Beispiel in regulären Ausdrücken verwendet, um den Anfang einer Zeile zu finden. Wir könnten unser Programm so ändern, dass es nur mit Zeilen übereinstimmt, in denen `From:` am Anfang der Zeile steht:

```
# Finde Zeilen, die mit 'From' beginnen
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print(line)

# Code: https://tiny.one/py4de/code3/re02.py
```

Jetzt werden wir nur Zeilen abgleichen, die *mit* der Zeichenkette `From:` beginnen. Dies ist immer noch ein sehr einfaches Beispiel, das wir äquivalent mit der Methode `startswith()` aus der String-Bibliothek hätten erledigen können. Aber es dient dazu, die Vorstellung einzuführen, dass reguläre Ausdrücke spezielle Zeichen enthalten, die uns mehr Kontrolle darüber geben, was mit dem regulären Ausdruck übereinstimmen wird.

Wildcards

Es gibt eine Reihe weiterer Sonderzeichen, mit denen sich noch mächtigere reguläre Ausdrücke erstellen lassen. Das am häufigsten verwendete Sonderzeichen ist der Punkt, der auf jedes beliebige Zeichen passt.

Im folgenden Beispiel würde der reguläre Ausdruck `F..m:` auf jede der Zeichenketten „From:“, „Fxxm:“, „F12m:“ oder „F!@m:“ passen, da die Punkte im regulären Ausdruck auf jedes Zeichen reagieren. Daher spricht man von *Wildcards*, was mit *Platzhalter* übersetzt werden kann.

```
# Finde Zeilen die mit einem 'F' beginnen, gefolgt
# von 2 beliebigen Zeichen, gefolgt von einem 'm:'
```

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)

# Code: https://tiny.one/py4de/code3/re03.py
```

Dies ist besonders effektiv in Kombination mit der Möglichkeit, anzugeben, dass ein Zeichen beliebig oft wiederholt werden kann, indem Sie die Zeichen `*` oder `+` in Ihrem regulären Ausdruck verwenden. Diese Sonderzeichen bedeuten, dass sie nicht auf ein einzelnes Zeichen in der Suchzeichenfolge passen, sondern auf null oder mehr Zeichen (im Fall des Sterns) oder auf ein oder mehr der Zeichen (im Fall des Pluszeichens).

Wir können die übereinstimmenden Zeilen weiter eingrenzen, indem wir im folgenden Beispiel ein wiederholtes *Wildcard*-Zeichen verwenden:

```
# Finde Zeilen, die mit 'From' beginnen
# und ein at-Zeichen beinhalten
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)

# Code: https://tiny.one/py4de/code3/re04.py
```

Die Suchzeichenfolge `^From:.+@` passt auf Zeilen, die mit `From:` beginnen, gefolgt von einem oder mehreren beliebigen Zeichen (`.+`), gefolgt von einem at-Zeichen. Dies wird also auf die folgende Zeile passen:

```
From: stephen.marquard@uct.ac.za
```

Wir können uns den Platzhalter `.+` so vorstellen, dass er auf alle Zeichen zwischen dem Doppelpunkt und dem at-Zeichen passt.

Extrahieren von Daten

Wenn wir in Python Daten aus einer Zeichenkette extrahieren wollen, können wir die Methode `findall()` verwenden, um alle Teilstrings zu extrahieren, die einem regulären Ausdruck entsprechen. Nehmen wir als Beispiel, dass wir alles, was wie eine E-Mail-Adresse aussieht, aus jeder Zeile extrahieren wollen, unabhängig vom Format. Zum Beispiel wollen wir die E-Mail-Adressen aus jeder der folgenden Zeilen extrahieren:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

Wir wollen nicht für jeden der Zeilentypen separaten Code schreiben. Das folgende Programm verwendet `findall()`, um die Zeilen mit E-Mail-Adressen darin zu finden und eine oder mehrere Adressen aus jeder dieser Zeilen zu extrahieren.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)

# Code: https://tiny.one/py4de/code3/re05.py
```

Die Methode `findall()` durchsucht die Zeichenkette im zweiten Argument und gibt eine Liste aller Zeichenketten zurück, die wie E-Mail-Adressen aussehen. Wir verwenden eine zweistellige Zeichenfolge, die auf ein Zeichen ohne Leerzeichen (`\S`) passt.

Die Ausgabe des Programms würde lauten:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Der reguläre Ausdruck sucht nach Teilzeichenfolgen, die mindestens ein Nicht-Leerzeichen enthalten, gefolgt von einem at-Zeichen, gefolgt von mindestens einem weiteren Nicht-Leerzeichen. Das `\S+` passt auf so viele Nicht-Leerzeichen wie möglich.

Der reguläre Ausdruck würde zweimal passen (auf `csev@umich.edu` und auf `cwen@iupui.edu`), aber er würde nicht auf die Zeichenkette „@2PM“ passen, weil es keine Nicht-Leerzeichen *vor* dem at-Zeichen gibt. Wir können diesen regulären Ausdruck in einem Programm verwenden, um alle Zeilen in einer Datei zu lesen und alles auszugeben, was wie eine E-Mail-Adresse aussieht:

```
# Finde Zeilen, die ein at-Zeichen zwischen zwei Zeichen haben:
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Code: https://tiny.one/py4de/code3/re06.py
```

Wir lesen jede Zeile und extrahieren dann alle Teilzeichenfolgen, die unserem regulären Ausdruck entsprechen. Da `findall()` eine Liste zurückgibt, prüfen wir

einfach, ob die Anzahl der Elemente in unserer zurückgegebenen Liste größer als 0 ist, um nur Zeilen auszugeben, in denen wir mindestens eine Teilzeichenkette gefunden haben, die wie eine E-Mail-Adresse aussieht.

Wenn wir das Programm auf `mbox.txt` anwenden, erhalten wir die folgende Ausgabe:

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
```

Einige unserer E-Mail-Adressen haben falsche Zeichen wie `<` oder `;` am Anfang oder Ende. Lassen wir uns definieren, dass wir nur an dem Teil der Zeichenkette interessiert sind, der mit einem Buchstaben oder einer Zahl beginnt und endet.

Dazu verwenden wir eine weitere Funktion regulärer Ausdrücke. Eckige Klammern werden verwendet, um eine Menge von mehreren akzeptablen Zeichen anzugeben, die wir als übereinstimmend betrachten. In gewissem Sinne bittet das `\S` darum, auf die Menge der „Nicht-Leerzeichen“ zu passen. Jetzt werden wir ein wenig expliziter in Bezug auf die Zeichen, die wir abgleichen werden.

Hier ist unser neuer regulärer Ausdruck:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

Das wird jetzt etwas kompliziert und wir sehen, warum reguläre Ausdrücke eine eigene kleine Sprache für sich sind. Dieser regulärer Ausdruck sucht nach Teilzeichenketten, die mit einem *einzigen* Kleinbuchstaben, Großbuchstaben oder einer Zahl `[a-zA-Z0-9]` beginnen, gefolgt von keinem oder mehr Nicht-Leerzeichen (`\S*`), von einem at-Zeichen, von keinem oder mehr Nicht-Leerzeichen (`\S*`) und schließlich von einem Groß- oder Kleinbuchstaben. Es ist zu beachten, dass wir von `+` zu `*` gewechselt haben, um kein oder mehr Nicht-Leerzeichen anzuzeigen, da `[a-zA-Z0-9]` bereits ein Nicht-Leerzeichen ist. Außerdem ist zu beachten, dass das `*` oder `+` auf das einzelne Zeichen unmittelbar links vom Plus oder Sternchen zutrifft.

Wenn wir diesen Ausdruck in unserem Programm verwenden, sind unsere Daten viel sauberer:

```
# Finde Zeilen, mit einem at-Zeichen zwischen zwei Zeichen. Die
# Zeichenfolge vor dem at muss mit einem Buchstaben oder einer
# Ziffer beginnen; die Zeichenfolge nach dem at muss mit einem
# Buchstaben enden.
import re
hand = open('mbox-short.txt')
for line in hand:
```

```
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0:
        print(x)

# Code: https://tiny.one/py4de/code3/re07.py
```

```
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Es ist wichtig zu beachten, dass unser regulärer Ausdruck in den Zeilen mit `source@collab.sakaiproject.org` die zwei Buchstaben `>;` am Ende der Zeichenfolge entfernt hat. Das liegt daran, dass wir, wenn wir `[a-zA-Z]` an das Ende unseres regulären Ausdrucks anhängen, verlangen, dass jede Zeichenfolge, die der Parser des regulären Ausdrucks findet, mit einem Buchstaben enden muss. Wenn er also das `>` am Ende von `sakaiproject.org>;` sieht, bleibt er einfach bei dem letzten „passenden“ Buchstaben stehen, den er gefunden hat (d. h. das „g“ war die letzte Übereinstimmung).

Zuletzt muss beachtet werden, dass die Ausgabe des Programms eine Python-Liste ist, die eine Zeichenkette als einziges Element in der Liste hat.

Kombination von Suchen und Extrahieren

Wenn wir Zahlen in Zeilen finden wollen, die mit der Zeichenkette `X-` beginnen, wie z. B.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

dann suchen wir nicht einfach beliebige Fließkommazahlen aus beliebigen Zeilen, sondern wir möchten nur Zahlen aus Zeilen extrahieren, die die obige Syntax aufweisen.

Wir können den folgenden regulären Ausdruck verwenden, um diese Zeilen auszuwählen:

```
^X-.*: [0-9.]+
```

Übersetzt bedeutet dieser Ausdruck: Wir wollen Zeilen, die mit `X-` beginnen, gefolgt von keinem oder beliebig vielen Zeichen (`.*`), gefolgt von einem Doppelpunkt (`:`)

und dann einem Leerzeichen. Nach dem Leerzeichen suchen wir nach einem oder mehreren Zeichen, die entweder eine Ziffer (0-9) oder ein Punkt `[0-9.]+` sind. Der Punkt innerhalb der eckigen Klammern stimmt mit einem tatsächlichen Punkt überein (d. h. es handelt sich nicht um einen Platzhalter)!

Dies ist ein sehr genau definierter Ausdruck, der wirklich nur die Zeilen trifft, an denen wir interessiert sind:

```
# Finde Zeilen, die mit 'X' beginnen, gefolgt von beliebig vielen
# (nicht-Leer-)Zeichen, gefolgt von einem ':' und einem
# Leerzeichen. Danach kann eine Zahl mit einer oder mehreren
# Ziffern und Dezimalpunkten folgen.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)

# Code: https://tiny.one/py4de/code3/re10.py
```

Wenn wir das Programm ausführen, sehen wir nur die Zeilen, nach denen wir tatsächlich suchen.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

Aber jetzt müssen wir das Problem des Extrahierens der Zahlen lösen. Während es einfach genug wäre, `split` zu verwenden, können wir eine andere Funktion von regulären Ausdrücken nutzen, um die Zeile gleichzeitig zu suchen und zu analysieren.

Klammern sind ein weiteres Sonderzeichen in regulären Ausdrücken. Wenn wir einem regulären Ausdruck Klammern hinzufügen, werden diese beim Abgleich der Zeichenkette ignoriert. Wenn wir jedoch `findall()` verwenden, zeigen die Klammern an, dass wir zwar wollen, dass der gesamte Ausdruck übereinstimmt, dass wir aber nur daran interessiert sind, einen Teil der Zeichenkette zu extrahieren, der mit dem regulären Ausdruck übereinstimmt.

Wir nehmen also folgende Änderung an unserem Programm vor:

```
# Finde Zeilen, die mit 'X' beginnen, gefolgt von beliebig vielen
# (nicht-Leer-)Zeichen, gefolgt von einem ':' und einem
# Leerzeichen. Danach kann eine Zahl mit einer oder mehreren
# Ziffern und Dezimalpunkten folgen. Diese Zahl ist dann der Match.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
```

```
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Code: https://tiny.one/py4de/code3/re11.py
```

Anstatt `search()` aufzurufen, fügen wir Klammern um den Teil des regulären Ausdrucks hinzu, der die Fließkommazahl repräsentiert, um anzugeben, dass `findall()` uns nur den Fließkommazahlenanteil der passenden Zeichenkette zurückgeben soll.

Die Ausgabe dieses Programms ist wie folgt:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
..
```

Die Zahlen sind immer noch in einer Liste und müssen von Zeichenketten in Fließkommazahlen umgewandelt werden. Aber wir haben gelernt, wie man reguläre Ausdrücke nutzt, um sowohl zu suchen als auch Informationen zu extrahieren, die wir interessant finden.

Als weiteres Beispiel für diese Technik, gibt es eine Reihe von Zeilen der Form:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

Wenn wir alle Revisionsnummern (die ganzzahlige Zahl am Ende dieser Zeilen) mit der gleichen Technik wie oben extrahieren wollten, könnten wir das folgende Programm schreiben:

```
# Finde Zeilen, die mit 'Details:' beginnen und auf 'rev=' enden
# gefolgt von einer Zahl. Diese Zahl ist dann der Match.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9]+)', line)
    if len(x) > 0:
        print(x)

# Code: https://tiny.one/py4de/code3/re12.py
```

Bei der Übersetzung unseres regulären Ausdrucks suchen wir nach Zeilen, die mit `Details:` beginnen, gefolgt von einer beliebigen Anzahl von Zeichen (`.*`), gefolgt von `rev=`, und dann von einer oder mehreren Ziffern. Wir wollen Zeilen finden, die mit dem gesamten Ausdruck übereinstimmen, aber wir wollen nur die ganzzahlige Zahl am Ende der Zeile extrahieren, also umgeben wir `[0-9]+` mit Klammern.

Wenn wir das Programm ausführen, erhalten wir die folgende Ausgabe:

```
['39772']
['39771']
['39770']
['39769']
...
```

Wir müssen bedenken, dass das `[0-9]+` „gierig“ ist und versucht, eine möglichst große Ziffernfolge zu bilden, bevor es diese Ziffern extrahiert. Dieses „gierige“ Verhalten ist der Grund, warum wir alle fünf Ziffern für jede Zahl erhalten. Der reguläre Ausdruck expandiert in beide Richtungen, bis er auf eine Nicht-Ziffer oder den Anfang oder das Ende einer Zeile stößt.

Jetzt können wir reguläre Ausdrücke verwenden, um eine Übung von früher im Buch zu wiederholen, bei der wir uns für die Tageszeit jeder Mail-Nachricht interessierten. Wir haben nach Zeilen dieser Form gesucht:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

Ziel war es, aus jeder Zeile die volle Stunde der Uhrzeit zu extrahieren. Das haben wir mit zwei Aufrufen von `split` bewerkstelligt. Zunächst haben wir die Zeile in Worte zerlegt und dann das sechste Wort entnommen (hier `09:14:16`). Dieses haben wir an den Doppelpunkten erneut zerlegt, um dann so die ersten beiden Ziffern zu erlangen.

Das hat zwar funktioniert, führt aber zu einem ziemlich unhandlichen Code, der davon ausgeht, dass die Zeilen schön formatiert sind. Wenn wir genug Fehlerprüfung (oder einen großen try/except-Block) hinzufügen würden, um sicherzustellen, dass Ihr Programm niemals fehlschlägt, wenn es mit falsch formatierten Zeilen konfrontiert wird, würde der Code auf 10 bis 15 Zeilen Code anwachsen.

Wir können dies auf eine viel einfachere Weise mit dem folgenden regulären Ausdruck lösen:

```
^From .* [0-9][0-9]:
```

Die Übersetzung dieses regulären Ausdrucks würde lauten: Wir suchen nach Zeilen, die mit `From` beginnen (das angehängte Leerzeichen beachten), gefolgt von einer beliebigen Anzahl von Zeichen (`.*`), von einem Leerzeichen, von zwei Ziffern `[0-9][0-9]` und schließlich von einem Doppelpunkt. Dies ist die Definition der Arten von Zeilen, nach denen wir suchen.

Um mit `findall()` nur die Stunde herauszuholen, fügen wir Klammern um die beiden Ziffern wie folgt hinzu:

```
^From .* ([0-9][0-9]):
```

Daraus ergibt sich das folgende Programm:

```
# Finde Zeilen, die mit 'From ' beginnen und dann irgendwann eine
# Zahl mit genau zwei Ziffern folgt. Vor der zweistellingen Zahl
```

```
# muss ein Leerzeichen stehen, danach ein ':'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0: print(x)

# Code: https://tiny.one/py4de/code3/re13.py
```

Wenn das Programm läuft, erzeugt es die folgende Ausgabe:

```
['09']
['18']
['16']
['15']
...
```

Escapezeichen

Wir verwenden spezielle Zeichen in regulären Ausdrücken, um den Anfang (^) oder das Ende ($) einer Zeile abzugleichen oder um Platzhalter (.) anzugeben. Aber was machen wir, wenn wir tatsächlich nach einem dieser Zeichen suchen möchten? Wir benötigen also eine Möglichkeit, solche Symbole in regulären Ausdrücken auszeichnen zu können.

Genau das erreichen wir, indem wir diesem Zeichen einen Rückstrich voranstellen. Zum Beispiel können wir Dollarbeträge mit dem folgenden regulären Ausdruck finden.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Da wir dem Dollarzeichen einen Rückstrich voranstellen, passt er tatsächlich auf das Dollarzeichen in der Eingabezeichenkette, statt auf das Zeilenende, und der Rest des regulären Ausdrucks passt auf eine oder mehrere Ziffern oder das Punktzeichen. *Hinweis:* Innerhalb eckiger Klammern sind die Zeichen nicht „speziell“. Wenn wir also `[0-9.]` sagen, bedeutet das in Wirklichkeit Ziffern oder einen Punkt. Außerhalb von eckigen Klammern ist der Punkt ein Platzhalter und passt zu jedem Zeichen. Innerhalb eckiger Klammern ist der Punkt ein Punkt.

Zusammenfassung

Obwohl dies nur an der Oberfläche der regulären Ausdrücke gekratzt hat, haben wir ein wenig über die Sprache der regulären Ausdrücke gelernt. Es handelt sich um

Suchzeichenfolgen mit Sonderzeichen, die dem System mitteilen, was als „übereinstimmend“ definiert und was aus den übereinstimmenden Zeichenfolgen extrahiert wird. Hier sind einige dieser Sonderzeichen und Zeichenfolgen:

`^` Zeilenanfang

`$` Zeilenende

`.` genau ein beliebiges Zeichen (Platzhalter/Wildcard)

`\s` genau ein Leerzeichen

`\S` genau ein Nicht-Leerzeichen (Gegenteil von `\s`).

`*` kein bis beliebig viele Zeichen

`+` ein bis beliebig viele Zeichen

`?` kein oder genau ein Zeichen

`*?` kein bis beliebig viele Zeichen (im non-greedy-Modus)

`+?` ein bis beliebig viele Zeichen (im non-greedy-Modus)

`??` kein oder genau ein Zeichen (im non-greedy-Modus)

Die Fragezeichen in `*?`, `+?` und `??` fordern, dass der Abgleich im sogenannten non-greedy-Modus durchgeführt werden soll. Bei einem non-greedy-Match wird versucht, die *kleinstmögliche* übereinstimmende Zeichenfolge zu finden. Im normalen Modus (greedy) dagegen wird versucht, die *größtmögliche* übereinstimmende Zeichenfolge zu finden.

`[aeiou]` Stimmt mit einem einzelnen Zeichen überein, sofern dieses Zeichen in der angegebenen Menge enthalten ist. In diesem Beispiel würde es auf „a“, „e“, „i“, „o“ oder „u“ passen, aber nicht auf andere Zeichen.

`[a-z0-9]` Wir können Zeichenbereiche mit dem Minuszeichen angeben. Dieses Beispiel ist ein einzelnes Zeichen, das ein Kleinbuchstabe oder eine Ziffer sein muss.

`[^A-Za-z]` Wenn das erste Zeichen in der Mengenschreibweise ein Zirkumflex ist, wird die Logik invertiert. Dieses Beispiel passt auf ein einzelnes Zeichen, das *etwas anderes als* ein Groß- oder Kleinbuchstabe ist.

`( )` Wenn Klammern zu einem regulären Ausdruck hinzugefügt werden, werden sie während des Abgleichs ignoriert, ermöglichen es aber, eine bestimmte Teilzeichenfolge zu extrahieren, wenn wir `findall()` verwenden.

`\b` leere Zeichenkette (aber nur am Anfang oder Ende eines Wortes)

`\B` leere Zeichenkette (aber *nicht* am Anfang oder Ende eines Wortes)

`\d` Dezimalziffer; entspricht der Menge `[0-9]`

`\D` jedes Zeichen, das keine Ziffer ist; entspricht der Menge `[^0-9]`

Bonuskapitel für Unix/Linux-Benutzer

Die Unterstützung für das Durchsuchen von Dateien mit regulären Ausdrücken wurde seit den 1960er Jahren in das Unix-Betriebssystem eingebaut und ist heute in fast allen Programmiersprachen in der einen oder anderen Form vorhanden.

Tatsächlich gibt es ein in Unix eingebautes Kommandozeilenprogramm namens *grep* (Generalized Regular Expression Parser), das so ziemlich dasselbe tut wie die `search()`-Beispiele in diesem Kapitel. Wer also ein Macintosh- oder Linux-System hat, kann die folgenden Befehle im Kommandozeilenfenster ausprobieren.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

Damit wird `grep` angewiesen, Zeilen anzuzeigen, die mit der Zeichenkette `From:` in der Datei `mbox-short.txt` beginnen. Wenn wir ein wenig mit dem Befehl `grep` experimentieren und die Dokumentation zu `grep` lesen, werden wir einige subtile Unterschiede zwischen der Unterstützung regulärer Ausdrücke in Python und der Unterstützung regulärer Ausdrücke in `grep` feststellen. Zum Beispiel unterstützt `grep` nicht das Nicht-Leerzeichen `\S`, sodass die etwas komplexere Mengenschreibweise `[^ ]` verwendet werden muss.

Debugging

Python hat eine einfache und rudimentäre eingebaute Dokumentation, die sehr hilfreich sein kann, wenn wir eine schnelle Auffrischung benötigen, um sich an den genauen Namen einer bestimmten Methode zu erinnern. Diese Dokumentation kann im Python-Interpreter im interaktiven Modus eingesehen werden.

Mit `help()` können wir ein interaktives Hilfesystem aufrufen.

```
>>> help()

help> modules
```

Wenn wir wissen, welches Modul wir verwenden möchten, können wir den Befehl `dir()` verwenden, um die Methoden im Modul wie folgt zu finden:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

Wir können auch mit dem `help`-Befehl einen kleinen Teil der Dokumentation zu einer bestimmten Methode abrufen.

```
>>> help(re.search)
Help on function search in module re:
```

```
search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern,
    returning a match object, or None if no match was found.
>>>
```

Die integrierte Dokumentation ist nicht sehr umfangreich, kann aber hilfreich sein, wenn wir in Eile sind oder keinen Zugriff auf einen Webbrowser oder eine Suchmaschine haben.

Glossar

greedy Matching Bei einem non-greedy-Match wird versucht, die *kleinstmögliche* übereinstimmende Zeichenfolge zu finden. Im normalen Modus (greedy) wird dagegen versucht, die *größtmögliche* übereinstimmende Zeichenfolge zu finden.

grep Ein in den meisten Unix-Systemen verfügbarer Befehl, der Texte nach Zeilen durchsucht, die regulären Ausdrücken entsprechen. Der Name des Befehls steht für „Generalized Regular Expression Parser“.

regulärer Ausdruck (Regex) Eine Sprache zum Definieren komplexerer Suchzeichenfolgen.

Wildcard Ein Platzhalter, der auf ein beliebiges Zeichen passt. In regulären Ausdrücken ist das Platzhalterzeichen der Punkt.

Übungen

Übung 1: Es soll ein einfaches Programm geschrieben werden, das die Funktionsweise des Befehls `grep` unter Unix simuliert. Der Benutzer soll aufgefordert werden, einen regulären Ausdruck einzugeben, mit dem dann die Anzahl der Zeilen gezählt werden, die mit dem regulären Ausdruck übereinstimmen:

```
$ python grep.py
Gib eine RegEx an: ^Author
mbox.txt hat 1798 Zeilen die auf den Ausdruck "^Author" passen

$ python grep.py
Gib eine RegEx an: ^X-
mbox.txt hat 14368 Zeilen die auf den Ausdruck "^X-" passen

$ python grep.py
Gib eine RegEx an: java$
mbox.txt hat 4175 Zeilen die auf den Ausdruck "java" passen
```

Übung 2: Es soll ein Programm geschrieben werden, das nach Zeilen der folgenden Form sucht:

```
New Revision: 39772
```

Die Zahl soll aus jeder der Zeilen mit einem regulären Ausdruck und der Methode `findall()` extrahiert werden. Es soll dann der Durchschnitt der Zahlen berechnet und als Ganzzahl ausgegeben werden.

```
Gib eine Datei an: mbox.txt
38549

Gib eine Datei an: mbox-short.txt
39756
```

Kapitel 12

Vernetzen von Programmen

In den vorangegangenen Beispielen dieses Buches haben wir Daten vor allem aus Dateien gelesen, die bei uns im Sekundärspeicher (also in der Regel auf der Festplatte) abgespeichert waren. Betrachtet man allerdings die Tatsache, dass Computer heutzutage nahezu *immer online* sind, wird klar, dass das Internet eine wichtige Datenquelle für Programme ist.

In diesem Kapitel werden wir Programme schreiben, die wie Webbrowser auf Internetseiten zugreifen und Daten über das Hypertext Transfer Protocol (HTTP) abrufen. Wir werden die Daten der Webseite einlesen und parsen, um bestimmte Informationen von diesen Seiten zu extrahieren.

Hypertext Transfer Protocol – HTTP

Das Netzwerkprotokoll, das das Web antreibt, ist eigentlich recht einfach und es gibt eine eingebaute Unterstützung in Python namens `socket`, die es sehr einfach macht, Netzwerkverbindungen herzustellen und Daten über diese Sockets in einem Python-Programm abzurufen.

Ein *Socket* ist ähnlich wie eine Datei, mit dem Unterschied, dass ein einzelner Socket eine bidirektionale Verbindung zwischen zwei Programmen ermöglicht. Wir können sowohl von einem Socket lesen als auch auf einen Socket schreiben. Wenn wir etwas in einen Socket schreiben, wird es an die Anwendung am anderen Ende des Sockets gesendet. Wenn wir aus dem Socket lesen, erhalten wir die Daten, die die andere Anwendung gesendet hat.

Falls wir aber versuchen, einen Socket zu lesen, wenn das Programm am anderen Ende des Sockets noch keine Daten gesendet hat, sitzen wir nur da und warten. Wenn die Programme an beiden Enden des Sockets nur auf Daten warten, ohne etwas zu senden, werden sie sehr lange warten. Daher ist es für Programme, die über das Internet kommunizieren, wichtig, über eine Art Protokoll zu verfügen. Ein Protokoll ist ein Satz präziser Regeln, die festlegen, wer zuerst sendet, was er zu tun hat, was die Antworten auf diese Nachricht sind, wer als nächstes sendet und so weiter. Es gibt viele Dokumente, die diese Netzwerkprotokolle beschreiben. Das Hypertext Transfer Protocol wird in dem folgenden Dokument beschrieben:

https://www.w3.org/Protocols/rfc2616/rfc2616.txt

Dies ist ein langes und komplexes 176-seitiges Dokument mit vielen Details. Wer es interessant findet, kann es gerne ganz lesen, notwendig ist das aber nicht, um HTTP zu verwenden. Auf Seite 36 von RFC2616 finden wir die Syntax für den GET-Request. Um ein Dokument von einem Webserver anzufordern, stellen wir eine Verbindung zum `www.pr4e.org`-Server auf Port 80 her und senden dann eine Zeile der Form

```
GET http://data.pr4e.org/romeo.txt HTTP/1.0
```

wobei der zweite Parameter die von uns angeforderte Webseite ist. Außerdem senden wir auch eine Leerzeile. Der Webserver antwortet mit einigen Header-Informationen über das Dokument und einer Leerzeile, gefolgt von dem Dokumentinhalt.

Der einfachste Webbrowser der Welt

Der vielleicht einfachste Weg zu zeigen, wie das HTTP-Protokoll funktioniert, ist, ein sehr einfaches Python-Programm zu schreiben, das eine Verbindung zu einem Webserver herstellt und den Regeln des HTTP-Protokolls folgt, um ein Dokument anzufordern und anzuzeigen, welches der Server zurückschickt.

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt HTTP/1.0\r\n\r\n'.encode()
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if len(data) < 1:
        break
    print(data.decode(),end='')

mysock.close()

# Code: https://tiny.one/py4de/code3/socket1.py
```

Zunächst stellt das Programm eine Verbindung zu Port 80 auf dem Server www.py4e.com her. Da unser Programm die Rolle des Webbrowsers spielt, sagt das HTTP-Protokoll, dass wir den GET-Befehl gefolgt von einer Leerzeile senden müssen. `\r\n` bedeutet ein EOL (End-of-Line), also steht `\r\n\r\n` für „nichts“ zwischen zwei EOL-Sequenzen. Das ist das Äquivalent zu einer Leerzeile.

Sobald wir diese Leerzeile gesendet haben, schreiben wir eine Schleife, die Daten in 512-Zeichen-Blöcken vom Socket empfängt und die Daten ausgibt, bis es keine weiteren Daten mehr zu lesen gibt (d. h., bis `recv()` eine leere Zeichenkette liefert).

Das Programm erzeugt die folgende Ausgabe:

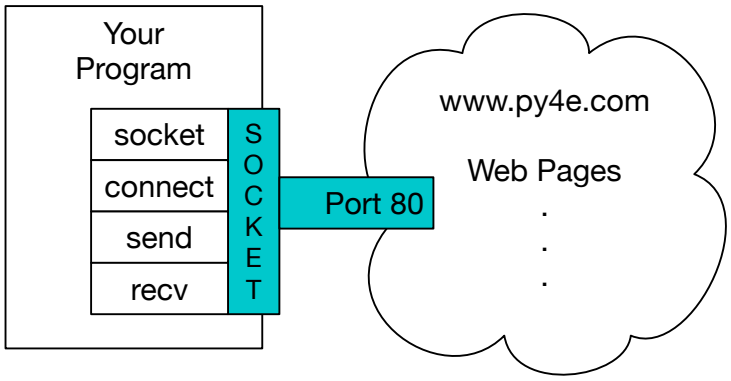

Abbildung 12.1: Eine Socketverbindung

```
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:52:55 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Die Ausgabe beginnt mit Headern, die der Webserver sendet, um das Dokument zu beschreiben. Der Header `Content-Type` zeigt zum Beispiel an, dass es sich um ein reines Textdokument (`text/plain`) handelt.

Nachdem der Server uns die Kopfzeilen gesendet hat, fügt er eine Leerzeile ein, um das Ende der Kopfzeilen anzuzeigen, und sendet dann die eigentlichen Daten der Datei `romeo.txt`.

Dieses Beispiel zeigt, wie wir eine Low-Level-Netzwerkverbindung mit Sockets herstellen. Sockets können für die Kommunikation mit einem Webserver oder mit einem Mailserver oder vielen anderen Arten von Servern verwendet werden. Man muss nur das Dokument finden, das das Protokoll beschreibt, und dann den entsprechenden Code verwenden, um die Daten gemäß dem Protokoll zu senden und zu empfangen.

Da das von uns am häufigsten verwendete Protokoll jedoch das HTTP-Webprotokoll ist, verfügt Python über eine spezielle Bibliothek zur Unterstützung des HTTP-Protokolls für den Abruf von Dokumenten und Daten über das Web.

Eine der Voraussetzungen für die Verwendung des HTTP-Protokolls ist die Notwendigkeit, Daten als Byte-Objekte anstelle von Strings zu senden und zu empfangen. Im vorangegangenen Beispiel wandeln die Methoden `encode()` und `decode()` Strings in Byte-Objekte und wieder zurück.

Das nächste Beispiel verwendet die Notation `b''`, um anzugeben, dass eine Variable als Byte-Objekt gespeichert werden soll. `encode()` und `b''` sind gleichwertig.

```
>>> b'Hello world'
b'Hello world'
>>> 'Hello world'.encode()
b'Hello world'
```

Abrufen eines Bildes über HTTP

Im obigen Beispiel haben wir eine reine Textdatei abgerufen, die Zeilenumbrüche in der Datei hatte, und wir haben die Daten einfach auf den Bildschirm kopiert, während das Programm lief. Wir können ein ähnliches Programm verwenden, um ein Bild über HTTP abzurufen. Anstatt die Daten bei der Ausführung des Programms auf den Bildschirm zu kopieren, sammeln wir die Daten in einer Zeichenkette, schneiden die Kopfzeilen ab und speichern die Bilddaten dann wie folgt in einer Datei:

```
import socket
import time

HOST = 'data.pr4e.org'
PORT = 80
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect((HOST, PORT))
mysock.sendall(
    b'GET http://data.pr4e.org/cover3.jpg HTTP/1.0\r\n\r\n')
count = 0
picture = b""

while True:
    data = mysock.recv(5120)
    if len(data) < 1: break
    #time.sleep(0.25)
    count = count + len(data)
    print(len(data), count)
    picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)
pos = picture.find(b"\r\n\r\n")
print('Header length', pos)
```

```
print(picture[:pos].decode())

# Skip past the header and save the picture data
picture = picture[pos+4:]
fhand = open("stuff.jpg", "wb")
fhand.write(picture)
fhand.close()

# Code: https://tiny.one/py4de/code3/urljpeg.py
```

Wenn das Programm läuft, erzeugt es die folgende Ausgabe:

```
$ python urljpeg.py
5120 5120
5120 10240
4240 14480
5120 19600
...
5120 214000
3200 217200
5120 222320
5120 227440
3167 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 18:54:09 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Wir können sehen, dass bei dieser URL der `Content-Type`-Header anzeigt, dass der Rumpf des Dokuments ein Bild ist (`image/jpeg`). Sobald das Programm beendet ist, können wir die Bilddaten anzeigen, indem wir die Datei `stuff.jpg` in einem Bildbetrachter öffnen.

Während das Programm läuft, können wir sehen, dass wir nicht jedes Mal 5120 Zeichen erhalten, wenn wir die Methode `recv()` aufrufen. Wir erhalten nur so viele Zeichen wie vom Webserver bis zu dem jeweiligen Aufruf von `recv()` tatsächlich über das Netzwerk zu uns übertragen wurden.

Unsere Ergebnisse können je nach der Netzwerkgeschwindigkeit unterschiedlich sein. Beachten wir auch, dass wir beim letzten Aufruf von `recv()` 3167 Bytes erhalten, was das Ende des Streams ist, und dass wir beim nächsten Aufruf von `recv()` eine Zeichenkette der Länge 0 erhalten, die uns mitteilt, dass der Server an seinem Ende des Sockets `close()` aufgerufen hat und keine weiteren Daten mehr anstehen.

Wir können unsere aufeinanderfolgenden `recv()`-Aufrufe verlangsamen, indem wir den Aufruf von `time.sleep()` wieder einbinden. Auf diese Weise warten wir nach jedem Aufruf eine Viertelsekunde, damit der Server uns „zuvorkommen“ und weitere Daten an uns senden kann, bevor wir `recv()` erneut aufrufen. Mit der Verzögerung an Ort und Stelle wird das Programm wie folgt ausgeführt:

```
$ python urljpeg.py
5120 5120
5120 10240
5120 15360
...
5120 225280
5120 230400
207 230607
Header length 393
HTTP/1.1 200 OK
Date: Wed, 11 Apr 2018 21:42:08 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Mon, 15 May 2017 12:27:40 GMT
ETag: "38342-54f8f2e5b6277"
Accept-Ranges: bytes
Content-Length: 230210
Vary: Accept-Encoding
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: image/jpeg
```

Abgesehen vom letzten Aufruf von `recv()` erhalten wir jetzt jedes Mal 5120 Zeichen, wenn wir neue Daten anfordern.

Es gibt einen Puffer zwischen dem Server, der `send()`-Requests stellt, und unserer Anwendung, die `recv()`-Requests stellt. Wenn wir das Programm mit der Verzögerung laufen lassen, kann es sein, dass der Server irgendwann den Puffer im Socket füllt und gezwungen ist, eine Pause einzulegen, bis unser Programm beginnt, den Puffer zu leeren. Das Anhalten entweder der sendenden oder der empfangenden Anwendung wird *Flusskontrolle* (englisch *flow control*) genannt.

Abrufen von Webseiten mit `urllib`

Während wir mit der Socket-Bibliothek manuell Daten über HTTP senden und empfangen können, gibt es einen viel einfacheren Weg, diese Aufgabe in Python zu erledigen, indem wir die `urllib`-Bibliothek verwenden.

Mit `urllib` können wir eine Web-Seite ähnlich wie eine Datei behandeln. Wir geben einfach an, welche Webseite wir abrufen möchten, und `urllib` kümmert sich um alle Details des HTTP-Protokolls und der Header.

Der äquivalente Code zum Lesen der Datei `romeo.txt` aus dem Web mit `urllib` sieht wie folgt aus:

```
import urllib.request

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')
for line in fhand:
    print(line.decode().strip())

# Code: https://tiny.one/py4de/code3/urllib1.py
```

Sobald die Webseite mit `urllib.urlopen` geöffnet wurde, können wir sie wie eine Datei behandeln und mit einer `for`-Schleife durchlaufen.

Wenn das Programm läuft, sehen wir nur die Ausgabe des Inhalts der Datei. Die Kopfzeilen werden immer noch gesendet, aber der `urllib`-Code entfernt die Kopfzeilen und gibt nur die Daten an uns zurück.

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

Als Beispiel können wir ein Programm schreiben, um die Daten für `romeo.txt` abzurufen und die Häufigkeit jedes Wortes in der Datei folgendermaßen zu berechnen:

```
import urllib.request, urllib.parse, urllib.error

fhand = urllib.request.urlopen('http://data.pr4e.org/romeo.txt')

counts = dict()
for line in fhand:
    words = line.decode().split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1
print(counts)

# Code: https://tiny.one/py4de/code3/urlwords.py
```

Auch hier können wir, nachdem wir die Webseite geöffnet haben, sie wie eine lokale Datei lesen.

Lesen von Binärdateien mit `urllib`

Manchmal möchten wir eine Binärdatei abrufen, wie z. B. eine Bild- oder Videodatei. Die Daten in diesen Dateien sind in der Regel nicht zum Ausgeben geeignet, aber wir können mit `urllib` leicht eine Datei von einer URL auf unsere Festplatte kopieren.

Das Vorgehen besteht darin, die URL zu öffnen und mit `read` den gesamten Inhalt des Dokuments in eine String-Variable (`img`) herunterzuladen und diese Informationen dann wie folgt in eine lokale Datei zu schreiben:

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen(
          'http://data.pr4e.org/cover3.jpg').read()
fhand = open('cover3.jpg', 'wb')
fhand.write(img)
fhand.close()

# Code: https://tiny.one/py4de/code3/curl1.py
```

Dieses Programm liest alle Daten auf einmal über das Netzwerk ein und speichert sie in der Variablen `img` im Hauptspeicher des Computers, erstellt und öffnet dann die neue Datei `cover.jpg` im Schreibmodus und schreibt die Daten auf die Festplatte. Das Argument `wb` für `open()` öffnet eine Binärdatei nur zum Schreiben. Dieses Programm funktioniert, solange die Größe der Datei kleiner ist als die Größe des Speichers unseres Computers.

Wenn es sich jedoch um eine große Audio- oder Videodatei handelt, kann dieses Programm abstürzen oder zumindest extrem langsam laufen, wenn der Speicher unseres Computers erschöpft ist. Um ein Überschreiten der Speicherkapazität zu vermeiden, werden die Daten in Blöcken (oder Puffern) abgerufen und dann jeder Block auf die Festplatte geschrieben, bevor der nächste Block abgerufen wird. Auf diese Weise kann das Programm Dateien beliebiger Größe lesen, ohne den gesamten Speicher des Computers zu verbrauchen.

```
import urllib.request, urllib.parse, urllib.error

img = urllib.request.urlopen('http://data.pr4e.org/cover3.jpg')
fhand = open('cover3.jpg', 'wb')
size = 0
while True:
    info = img.read(100000)
    if len(info) < 1: break
    size = size + len(info)
    fhand.write(info)

print(size, 'characters copied.')
fhand.close()

# Code: https://tiny.one/py4de/code3/curl2.py
```

In diesem Beispiel werden jeweils nur 100.000 Zeichen gelesen und dann in die Datei `cover.jpg` geschrieben, bevor die nächsten 100.000 Zeichen an Daten aus dem Web abgerufen werden.

Dieses Programm läuft so ab:

```
python curl2.py
230210 characters copied.
```

Parsen von HTML und Erkunden des Webs

Eine der häufigsten Verwendungen der `urllib` in Python ist das *Scraping* des Webs. Beim Web-Scraping schreiben wir ein Programm, das vorgibt, ein Web-Browser zu sein, Seiten abruft und dann die Daten auf diesen Seiten auf Muster untersucht.

Ein Beispiel: Eine Suchmaschine wie Google schaut sich die Quelle einer Webseite an, extrahiert die Links zu anderen Seiten und ruft diese Seiten auf, indem sie die Links extrahiert. Dies wird dann einfach wiederholt. Mit dieser Technik erkundet Google seinen Weg durch fast alle Seiten im Web.

Google verwendet auch die Häufigkeit der Links von Seiten, die es zu einer bestimmten Seite findet, als ein Maß dafür, wie „wichtig“ eine Seite ist und wie weit oben die Seite in den Suchergebnissen erscheinen soll.

Parsen von HTML mit regulären Ausdrücken

Eine einfache Möglichkeit, HTML zu parsen, ist die Verwendung regulärer Ausdrücke, um wiederholt nach Teilzeichenketten zu suchen und diejenigen zu extrahieren, die einem bestimmten Muster entsprechen.

Hier ist eine einfache Web-Seite:

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

Wir können einen wohlgeformten regulären Ausdruck konstruieren, um die Links aus dem obigen Text folgendermaßen zu extrahieren:

```
href="http[s]?://.+?"
```

Unser regulärer Ausdruck sucht nach Zeichenfolgen, die mit `href="http://` oder `href="https://` beginnen, gefolgt von einem oder mehreren Zeichen (`.+?`) und einem weiteren doppelten Anführungszeichen. Das Fragezeichen hinter dem `[s]` zeigt an, dass nach der Zeichenkette `http` nach keinem oder einem `s` gesucht werden soll. Das Fragezeichen hinter dem `.+` gibt an, dass der Abgleich im non-greedy-Modus (also nicht „greedy“) durchgeführt werden soll. Bei einem non-greedy-Match wird versucht, die *kleinstmögliche* übereinstimmende Zeichenfolge zu finden. Bei einem greedy-Match hingegen wird versucht, die *größtmögliche* übereinstimmende Zeichenfolge zu finden.

Wir fügen unserem regulären Ausdruck Klammern hinzu, um anzugeben, welchen Teil der übereinstimmenden Zeichenfolge wir extrahieren möchten:

```
# Search for link values within URL input
import urllib.request, urllib.parse, urllib.error
import re
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
links = re.findall(b'href="(http[s]?://.*?)"', html)
for link in links:
    print(link.decode())

# Code: https://tiny.one/py4de/code3/urlregex.py
```

Die `ssl`-Bibliothek ermöglicht diesem Programm den Zugriff auf Websites, die HTTPS strikt erzwingen. Die Methode `read` gibt den HTML-Quellcode als Byte-Objekt zurück, anstatt ein HTTP-Response-Objekt zu liefern. Die Methode `findall` für reguläre Ausdrücke liefert uns eine Liste aller Zeichenketten, die mit unserem regulären Ausdruck übereinstimmen, und gibt nur den Linktext zwischen den Anführungszeichen zurück.

Wenn wir das Programm ausführen und eine URL eingeben, erhalten wir diese Ausgabe:

```
Enter - https://docs.python.org
https://docs.python.org/3/index.html
https://www.python.org/
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
https://www.python.org/
https://www.python.org/psf/donations/
http://sphinx.pocoo.org/
```

Reguläre Ausdrücke funktionieren sehr gut, wenn das HTML gut formatiert und vorhersehbar ist. Da es aber eine ganze Menge „defekter“ HTML-Seiten gibt, könnte eine Lösung, die ausschließlich reguläre Ausdrücke verwendet, entweder einige gültige Links übersehen oder unbrauchbare Daten liefern.

Dies kann durch die Verwendung einer robusten HTML-Parsing-Bibliothek gelöst werden.

Parsen von HTML mit BeautifulSoup

Auch wenn HTML wie XML aussieht[1] und einige Seiten sorgfältig so konstruiert sind, dass sie XML-konform sind, ist das meiste HTML in der Regel so fehlerhaft, dass ein XML-Parser die gesamte HTML-Seite als nicht korrekt formatiert zurückweist.

Es gibt eine Reihe von Python-Bibliotheken, die helfen können, HTML zu parsen und Daten aus den Seiten zu extrahieren. Jede der Bibliotheken hat ihre Stärken und Schwächen und wir müssen eine entsprechend unseren Bedürfnissen auswählen.

Als Beispiel werden wir einfach einige HTML-Eingaben parsen und Links mit Hilfe der *BeautifulSoup*-Bibliothek extrahieren. BeautifulSoup toleriert hochgradig fehlerhaftes HTML und lässt uns trotzdem einfach die benötigten Daten extrahieren. Auf der folgenden Seite können wir den BeautifulSoup-Code herunterladen und installieren:

https://pypi.python.org/pypi/beautifulsoup4

Informationen zur Installation von BeautifulSoup mit dem Python Package Index Tool `pip` finden sich unter:

https://packaging.python.org/tutorials/installing-packages/

Wir werden `urllib` benutzen, um die Seite zu lesen und dann `BeautifulSoup` benutzen, um die `href`-Attribute aus den Anker-Tags (`a`) zu extrahieren.

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: https://tiny.one/py4de/code3/urllinks.py
```

[1] Das XML-Format wird im nächsten Kapitel beschrieben.

Das Programm fragt nach einer Webadresse, öffnet dann die Webseite, liest die Daten ein und übergibt die Daten an den BeautifulSoup-Parser, der dann alle Anker-Tags abruft und das `href`-Attribut für jedes Tag ausgibt.

Wenn das Programm läuft, erzeugt es die folgende Ausgabe:

```
Enter - https://docs.python.org
genindex.html
py-modindex.html
https://www.python.org/
#
whatsnew/3.6.html
whatsnew/index.html
tutorial/index.html
library/index.html
reference/index.html
using/index.html
howto/index.html
installing/index.html
distributing/index.html
extending/index.html
c-api/index.html
faq/index.html
py-modindex.html
genindex.html
glossary.html
search.html
contents.html
bugs.html
about.html
license.html
copyright.html
download.html
https://docs.python.org/3.8/
https://docs.python.org/3.7/
https://docs.python.org/3.5/
https://docs.python.org/2.7/
https://www.python.org/doc/versions/
https://www.python.org/dev/peps/
https://wiki.python.org/moin/BeginnersGuide
https://wiki.python.org/moin/PythonBooks
https://www.python.org/doc/av/
genindex.html
py-modindex.html
https://www.python.org/
#
copyright.html
https://www.python.org/psf/donations/
bugs.html
http://sphinx.pocoo.org/
```

Diese Liste ist viel länger, weil einige HTML-Anker-Tags relative Pfade (z. B. `tutorial/index.html`) oder seiteninterne Verweise (z. B. `#`) sind, die nicht `http://`

oder `https://` enthalten, was in unserem regulären Ausdruck noch eine Voraussetzung war.

Wir können auch BeautifulSoup verwenden, um verschiedene Teile der einzelnen Tags zu extrahieren:

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

from urllib.request import urlopen
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, "html.parser")

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    # Look at the parts of a tag
    print('TAG:', tag)
    print('URL:', tag.get('href', None))
    print('Contents:', tag.contents[0])
    print('Attrs:', tag.attrs)

# Code: https://tiny.one/py4de/code3/urllink2.py
```

```
python urllink2.py
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: ['\nSecond Page']
Attrs: [('href', 'http://www.dr-chuck.com/page2.htm')]
```

`html.parser` ist der HTML-Parser, der in der Standardbibliothek von Python 3 enthalten ist. Informationen zu anderen HTML-Parsern finden sich unter:

http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

Diese Beispiele zeigen nur ansatzweise die Möglichkeiten von BeautifulSoup, wenn es um das Parsen von HTML geht.

Bonuskapitel für Unix-/Linux-User

Wenn wir einen Linux-, Unix- oder Macintosh-Computer haben, haben wir wahrscheinlich Befehle in das Betriebssystem eingebaut, die sowohl Klartext- als auch Binärdateien über die Protokolle HTTP oder File Transfer (FTP) abrufen. Einer dieser Befehle ist `curl`:

```
$ curl -O http://www.py4e.com/cover.jpg
```

Der Befehl `curl` ist die Abkürzung für „Client for URLs“ und so heißen die beiden zuvor aufgeführten Beispiele zum Abrufen von Binärdateien mit `urllib` auf www.py4e.com/code3 schlauerweise `curl1.py` und `curl2.py`, da sie eine ähnliche Funktionalität wie der Befehl `curl` implementieren. Es gibt auch ein Beispielprogramm `curl3.py`, das diese Aufgabe etwas effektiver erledigt (für den Fall, dass jemand dies tatsächlich in einem Programm verwenden möchte).

Ein zweiter Befehl, der sehr ähnlich funktioniert, ist `wget`:

```
$ wget http://www.py4e.com/cover.jpg
```

Diese beiden Befehle machen das Abrufen von Webseiten und Dateien zu einer einfachen Aufgabe.

Glossar

BeautifulSoup Eine Python-Bibliothek zum Parsen von HTML-Dokumenten und zum Extrahieren von Daten aus HTML-Dokumenten, die die meisten erwartbaren Unzulänglichkeiten im HTML-Code kompensiert, die auch von Browsern im Allgemeinen ignoriert werden. Der BeautifulSoup-Code kann auf der folgenden Seite heruntergeladen werden: www.crummy.com.

Port Eine Nummer, die im Allgemeinen angibt, mit welcher Anwendung wir Kontakt aufnehmen, wenn wir eine Socket-Verbindung zu einem Server herstellen. Ein Beispiel: Der Webverkehr verwendet normalerweise Port 80, während der E-Mail-Verkehr Port 25 verwendet.

Web-Scraping Wenn ein Programm vorgibt, ein Webbrowser zu sein, und eine Webseite abruft, dann schaut es sich den Inhalt der Webseite an. Oft folgen Programme den Links auf einer Seite, um die nächste Seite zu finden, sodass sie ein Netzwerk von Seiten oder ein soziales Netzwerk durchqueren können.

Socket Eine Netzwerkverbindung zwischen zwei Anwendungen, bei der die Anwendungen Daten in beide Richtungen senden und empfangen können.

Spider Der Vorgang, bei dem eine Web-Suchmaschine eine Seite abruft und dann alle verlinkten Seiten auf dieser ebenfalls aufruft, bis sie fast alle Seiten im Internet besucht hat. Diese können dann zum Aufbau eines Suchindexes verwendet werden.

Übungen

Übung 1: Das Socket-Programm `socket1.py` soll so geändert werden, dass es den Benutzer nach der URL fragt und jede Webseite lesen kann. Hierbei kann `split('/')` verwendet werden, um die URL in ihre Bestandteile zu zerlegen, damit wir den Hostnamen für den Socket-Aufruf `connect` extrahieren können. Dabei soll eine Fehlerprüfung mit `try` und `except` hinzugefügt werden für den Fall, dass der Benutzer eine falsch formatierte oder nicht existierende URL eingibt.

Übung 2: Das Socket-Programm soll so geändert werden, dass es die Anzahl der empfangenen Zeichen zählt und keinen Text mehr anzeigt, nachdem es 3000 Zeichen ausgegeben hat. Das Programm soll das gesamte Dokument abrufen und die Gesamtzahl der Zeichen zählen und diese am Ende des Dokuments anzeigen.

Übung 3: Es soll `urllib` verwendet werden, um die vorherige Übung zu wiederholen: (1) Abrufen des Dokuments von einer URL, (2) Anzeigen von bis zu 3000 Zeichen und (3) Zählen der Gesamtanzahl der Zeichen im Dokument. Dabei sollen diesmal die Header ignoriert und lediglich die ersten 3000 Zeichen des Dokumentinhalts angezeigt werden.

Übung 4: Das Programm `urllinks.py` soll so geändert werden, dass es Paragraph-Tags (`p`) aus dem abgerufenen HTML-Dokument extrahiert und zählt sowie die Anzahl der Absätze als Ausgabe des Programms anzeigt. Der Text soll nicht angezeigt werden. Das Programm sollte auf mehreren kleinen sowie einigen größeren Webseiten getestet werden.

Übung 5: (Erweiterung) Das Socket-Programm soll so geändert werden, dass es erst Daten anzeigt, nachdem die Header und eine Leerzeile empfangen wurden. Dabei muss bedacht werden, dass `recv` Zeichen empfängt, nicht aber ganze Zeilen.

Kapitel 13

Web-Services

Wir haben gesehen, wie einfach es ist, Dokumente über HTTP abzurufen und zu parsen. Anschließend haben wir gelernt, wie wir selbst Dokumente erzeugen, welche dann von anderen Programmen verarbeitet werden können.

Es gibt zwei gängige Formate, die wir beim Austausch von Daten über das Web verwenden. eXtensible Markup Language (XML) wird schon sehr lange verwendet und eignet sich am besten für den Austausch von dokumentenähnlichen Daten. Wenn Programme nur Dictionarys, Listen oder andere interne Informationen miteinander austauschen wollen, verwenden wir dagegen die JavaScript Object Notation (JSON) (siehe www.json.org). Wir werden uns beide Formate ansehen.

eXtensible Markup Language – XML

XML sieht HTML zwar sehr ähnlich, ist aber strenger strukturiert. Hier ist ein Beispiel für ein XML-Dokument:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

Jedes Paar von öffnenden (z. B. `<person>`) und schließenden Tags (`</person>`) repräsentiert ein *Element* oder einen *Knoten* mit dem gleichen Namen wie das Tag (hier `person`). Jedes Element kann Text, Attribute (wie bspw. `hide`) und andere verschachtelte Elemente enthalten. Wenn ein XML-Element leer ist (also keinen Inhalt hat), kann es durch ein selbstschließendes Tag dargestellt werden (`<email />`).

Oft ist es hilfreich, sich ein XML-Dokument als Baumstruktur vorzustellen, in der es ein oberstes Element (hier `person`) gibt und andere Tags wie `phone` als *Kinder* ihrer *Eltern*-Elemente dargestellt werden.

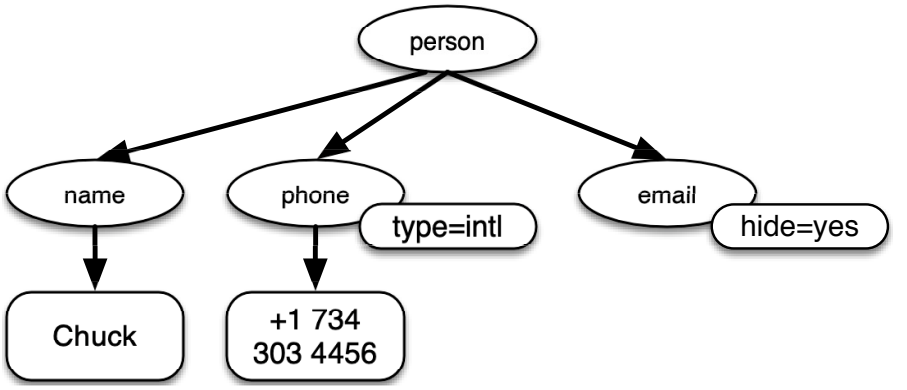

Abbildung 13.1: Eine Baumdarstellung von XML

Parsen von XML

Hier ist eine einfache Anwendung, die ein XML-Dokument parst und einige Datenelemente daraus extrahiert:

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Code: https://tiny.one/py4de/code3/xml1.py
```

Das dreifache einfache Anführungszeichen (`'''`) sowie das dreifache doppelte Anführungszeichen (`"""`) ermöglichen die Erstellung von Zeichenketten, die sich über mehrere Zeilen erstrecken.

Der Aufruf von `fromstring` konvertiert die String-Darstellung des XML in einen Baum von XML-Elementen. Wenn das XML in einem Baum vorliegt, haben wir eine Reihe von Methoden, die wir aufrufen können, um Teile der Daten aus dem XML-String zu extrahieren. Die Funktion `find` durchsucht den XML-Baum und ruft das Element ab, das mit dem angegebenen Tag übereinstimmt.

```
Name: Chuck
Attr: yes
```

Die Verwendung eines XML-Parsers wie `ElementTree` hat den Vorteil, dass es – obwohl das XML in diesem Beispiel recht einfach ist – viele verbindliche Regeln für gültiges XML gibt. Die Verwendung von `ElementTree` ermöglicht es uns, Daten zu extrahieren, ohne uns um die Regeln der XML-Syntax kümmern zu müssen.

Iterieren durch Knoten

Oft hat das XML mehrere Knoten und wir benötigen eine Schleife, um alle Knoten zu verarbeiten. Im folgenden Programm durchlaufen wir in einer Schleife alle `user`-Knoten:

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get('x'))

# Code: https://tiny.one/py4de/code3/xml2.py
```

Die Methode `findall` ruft eine Python-Liste von Teilbäumen ab, die die `user`-Strukturen im XML-Baum darstellen. Dann können wir eine `for`-Schleife schreiben, die jeden der `user`-Knoten betrachtet und die Textelemente `name` und `id` sowie das Attribut `x` des `user`-Knotens ausgibt.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
```

```
Id 009
Attribute 7
```

Es ist wichtig, alle Tags der *direkt* übergeordneten Ebene (also `users/user`) in die `findall`-Anweisung aufzunehmen. Andernfalls wird Python die gewünschten Knoten nicht finden. Das Top-Level-Tag (`stuff`) muss in unserem Fall nicht angegeben werden.

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')
print('User count:', len(lst))

lst2 = stuff.findall('user')
print('User count:', len(lst2))
```

Zum besseren Verständnis: `lst` speichert alle `user`-Elemente, die in ihrem `users`-Elternteil verschachtelt sind. `lst2` sucht nach `user`-Elementen, die nicht innerhalb des übergeordneten `stuff`-Elements verschachtelt sind. Von diesen gibt es allerdings keine.

```
User count: 2
User count: 0
```

JavaScript Object Notation – JSON

Das JSON-Format wurde durch das in der Sprache JavaScript verwendete Objekt- und Array-Format inspiriert. Da Python jedoch vor JavaScript erfunden wurde, hat die Syntax von Python für Dictionarys und Listen die Syntax von JSON beeinflusst. Das Format von JSON ist also fast identisch mit einer Kombination aus Python-Listen und Python-Dictionarys.

Hier ist eine JSON-Kodierung, die in etwa dem einfachen XML-Dokument von oben entspricht:

```
{
  "name": "Chuck",
  "phone": {
    "type": "intl",
    "number": "+1 734 303 4456"
   },
   "email": {
     "hide": "yes"
   }
}
```

Man stellt sofort einige Unterschiede fest. Erstens können wir in XML Attribute wie `intl` dem `phone`-Tag hinzufügen. In JSON verwenden wir dafür Schlüssel-Wert-Paare. Auch das XML-Tag `person` ist verschwunden und wurde durch eine Reihe von geschweiften Klammern ersetzt.

Im Allgemeinen sind JSON-Strukturen einfacher als XML, weil JSON weniger Möglichkeiten bietet als XML. Aber JSON hat den Vorteil, dass es *direkt* auf eine Kombination von Dictionarys und Listen abgebildet werden kann. Da fast alle Programmiersprachen Elemente haben, die den Dictionarys und Listen von Python entsprechen, ist JSON ein sehr naheliegendes Format, um zwischen zwei zusammenwirkenden Programmen Daten auszutauschen.

JSON hat sich aufgrund seiner relativen Einfachheit im Vergleich zu XML schnell zum Format der Wahl für fast jeden Datenaustausch zwischen Programmen entwickelt.

Parsen von JSON

Wir konstruieren unser JSON, indem wir Dictionarys und Listen nach Bedarf verschachteln. In diesem Beispiel stellen wir eine Liste von Benutzern dar, wobei jeder Benutzer ein Satz von Schlüssel-Wert-Paaren ist (also ein Dictionary). Somit erhalten wir also eine Liste von Dictionarys.

Im folgenden Programm verwenden wir die eingebaute `json`-Bibliothek, um JSON zu parsen und die Daten einzulesen. Das wollen wir nun genau mit den entsprechenden XML-Daten und dem Code von oben vergleichen. Das JSON hat weniger Details, daher müssen wir im Voraus wissen, dass wir eine Liste erhalten und dass die Liste aus Benutzern besteht und jeder Benutzer ein Satz von Schlüssel-Wert-Paaren ist. Das JSON ist prägnanter (ein Vorteil), aber auch weniger selbstbeschreibend (ein Nachteil).

```
import json

data = '''
[
```

```
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: https://tiny.one/py4de/code3/json2.py
```

Wenn wir den Code zum Extrahieren von Daten aus dem geparsten JSON und XML vergleichen, sehen wir, dass das, was wir durch `json.loads()` erhalten, eine Python-Liste ist, die wir mit einer `for`-Schleife durchlaufen. Jedes Element innerhalb dieser Liste ist ein Python-Dictionary. Sobald das JSON geparst wurde, können wir den Python-Indexoperator verwenden, um die verschiedenen Daten für jeden Benutzer zu extrahieren. Wir müssen nicht die JSON-Bibliothek verwenden, um uns durch das geparste JSON zu wühlen, da die zurückgegebenen Daten bereits native Python-Strukturen sind!

Die Ausgabe dieses Programms ist genau die gleiche wie die der obigen XML-Version.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

Im Allgemeinen gibt es bei Webservices in der IT-Branche einen Trend weg von XML und hin zu JSON. Da JSON einfacher ist und direkter auf native Datenstrukturen abgebildet wird, die in verschiedenen Programmiersprachen bereits vorhanden sind, ist der Code für das Parsen und die Datenextraktion bei der Verwendung von JSON normalerweise einfacher. XML ist jedoch selbstbeschreibender als JSON und daher gibt es einige Anwendungen, in denen XML weiterhin einen Vorteil bietet. So speichern die meisten Textverarbeitungsprogramme Dokumente intern in XML und nicht in JSON.

Application Programming Interfaces – API

Wir haben inzwischen die Fähigkeit erlernt, Daten zwischen Anwendungen über das HyperText Transport Protocol (HTTP) auszutauschen und wir haben eine Möglichkeit kennengelernt, komplexe Daten, die wir zwischen diesen Anwendungen hin- und herschicken möchten, mittels eXtensible Markup Language (XML) oder JavaScript Object Notation (JSON) darzustellen.

Der nächste Schritt besteht darin, *Schnittstellen* zwischen Anwendungen zu definieren und zu dokumentieren. Der allgemeine Name für eine Schnittstelle lautet *Application Program Interface* (kurz *API*). Bei der Bereitstellung einer API stellt ein Programm eine Reihe von *Diensten* für die Nutzung durch andere Anwendungen zur Verfügung. Um auf diese Dienste zugreifen zu können, müssen die anderen Programme die Regeln der entsprechenden API befolgen. Daher werden die Regeln und die Dokumentation der API normalerweise veröffentlicht.

Wenn wir beginnen, unsere Programme so aufzubauen, dass die Funktionalität unseres Programms den Zugriff auf Dienste beinhaltet, die von anderen Programmen bereitgestellt werden, bezeichnen wir diesen Ansatz als *Service orientierte Architektur* (SOA). Ein SOA-Ansatz ist ein solcher, bei dem unsere Gesamtanwendung auf die Dienste anderer Anwendungen zurückgreift. Ein Nicht-SOA-Ansatz ist einer, bei dem die Anwendung eine einzelne, eigenständige Anwendung ist, die den gesamten für die Implementierung der Anwendung erforderlichen Code enthält.

Wir sehen viele Beispiele für SOA, wenn wir das Web nutzen. Wir können auf eine einzige Website gehen und Flugreisen, Hotels und Autos buchen – alles von einer einzigen Seite aus. Die Daten für Hotels werden nicht auf den Computern der Fluggesellschaft gespeichert. Stattdessen kontaktieren die Computer der Fluggesellschaften die Dienste auf den Hotelservern und rufen die Hoteldaten ab und präsentieren sie dem Kunden. Wenn der Kunde zustimmt, eine Hotelreservierung über die Website der Fluggesellschaft vorzunehmen, verwendet die Website der Fluggesellschaft wieder einen anderen Webservice des Hotelsystemens, um die Reservierung tatsächlich vorzunehmen. Und wenn es an der Zeit ist, die Kreditkarte für die gesamte Transaktion zu belasten, werden noch weitere Computer in den Prozess einbezogen.

Eine serviceorientierte Architektur hat viele Vorteile, darunter: (1) Wir halten immer nur eine Kopie der Daten vor (dies ist besonders wichtig für Dinge wie Hotelreservierungen, bei denen wir nicht zu viele Verpflichtungen eingehen wollen) und (2) die Eigentümer der Daten können die Regeln für die Verwendung ihrer Daten festlegen. Mit diesen Vorteilen muss ein SOA-System sorgfältig entworfen werden, um leistungsstark zu sein und die Bedürfnisse der Benutzer zu erfüllen.

Wenn eine Anwendung einen Satz von Diensten in ihrer API über das Web verfügbar macht, nennen wir diese Dienste *Webdienste*.

Sicherheit und API-Nutzung

Es ist durchaus üblich, dass wir einen API-Schlüssel benötigen, um die API eines Anbieters nutzen zu können. Der Grund dafür ist, dass der Anbieter wissen möchte,

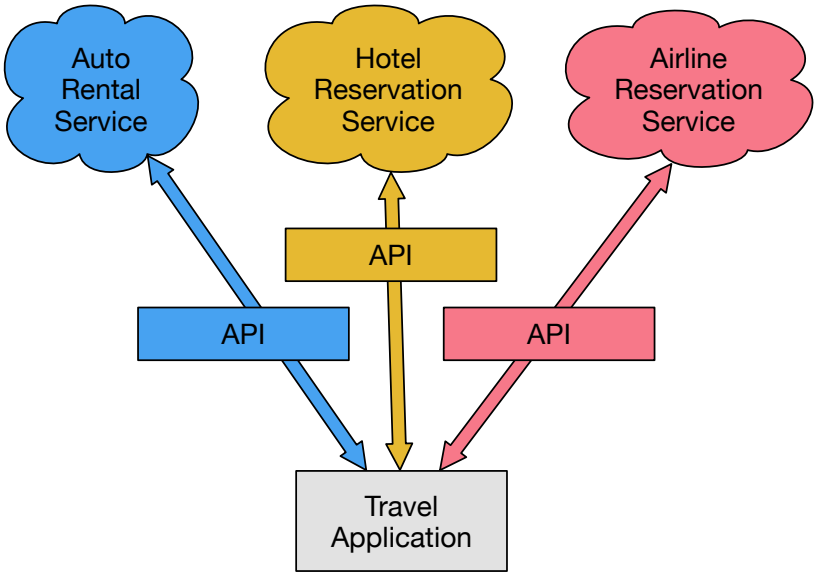

Abbildung 13.2: Serviceorientierte Architektur

wer seine Dienste nutzt und wie viel Endbenutzer auf sie zugreifen. Häufig gibt es eine kostenlose und eine kostenpflichtige Version der Dienste oder es gibt eine Richtlinie, die die Anzahl der Anfragen begrenzt, die eine einzelne Person während eines bestimmten Zeitraums stellen kann. Den API-Schlüssel fügen wir als Teil der POST-Daten oder auch als Parameter in der URL beim Aufruf der API ein. Das kann je nach Dienst variieren.

In anderen Fällen möchte der Anbieter eine erhöhte Sicherheit bezüglich der Quelle der Anfragen erzielen und erwartet daher, dass wir kryptografisch signierte Nachrichten mit gemeinsamen *Keys* und *Secrets* senden. Eine sehr verbreitete Technologie, die zum Signieren von Anfragen über das Internet verwendet wird, heißt *OAuth*. Mehr über das OAuth-Protokoll kann man unter www.oauth.net erfahren.

Glücklicherweise gibt es eine Reihe von praktischen und kostenlosen OAuth-Bibliotheken, sodass es nicht nötig ist, eine OAuth-Anbindung von Grund auf zu implementieren. Dafür müssen wir uns in die Spezifikation der Bibliothek einarbeiten. Diese Bibliotheken sind von unterschiedlicher Komplexität und haben einen unterschiedlichen Funktionsumfang. Auf der OAuth-Website finden sich Informationen über verschiedene OAuth-Bibliotheken.

Glossar

API Application Program Interface – Eine Schnittstelle zwischen Anwendungen, welche die Interaktion zwischen zwei Anwendungskomponenten definiert.

ElementTree Eine integrierte Python-Bibliothek, die zum Parsen von XML-Daten verwendet wird.

JSON JavaScript Object Notation – Ein Format, das die Auszeichnung (Markup)

von strukturierten Daten basierend auf der Syntax von JavaScript-Objekten ermöglicht.

SOA Service-Oriented Architecture – Komponenten von Anwendungen, die über ein Netzwerk verbunden sind.

XML eXtensible Markup Language – Ein Format, das die Auszeichnung von strukturierten Daten ermöglicht.

Anwendungsbeispiel 1: Google Geocoding Web Service

Google hat einen ausgezeichneten Webdienst, der es uns ermöglicht, seine große Datenbank mit geografischen Informationen zu nutzen. Wir können einen geografischen Suchstring wie „Iserlohn, Sauerland“ an die Geocoding-API übermitteln und Google gibt eine Vermutung darüber zurück, wo auf der Karte wir diesen Ort finden könnten, und informiert uns über die Sehenswürdigkeiten in der Nähe.

Der Geocoding-Dienst ist zwar kostenlos, die Nutzungsfrequenz der API ist jedoch begrenzt, sodass wir die API nicht uneingeschränkt in einer kommerziellen Anwendung nutzen können. Aber wenn wir eine Suchanfrage nach einem Ort haben, die ein Endbenutzer in ein Freitextfeld eingegeben hat, können wir diese API verwenden, um die Daten recht gut zu bereinigen und nach diesem Ort zu suchen.

Hinweis zur Nutzung freier Webdienste: Wenn wir eine kostenlose API wie die Geocoding-API von Google verwenden, müssen wir bei der Nutzung dieser Ressourcen gewissenhaft vorgehen. Wenn zu viele Leute den Dienst missbrauchen, könnte Google seinen kostenlosen Dienst einstellen oder erheblich einschränken.

Die Dokumentation für diesen Dienst ist online verfügbar. Wir können den Dienst sogar im Browser testen, indem wir die folgende URL eingeben:

http://maps.googleapis.com/maps/api/geocode/json?address=Iserlohn+Sauerland

Der Dienst antwortet dann mit der Fehlermeldung, dass ein API-Key benötigt wird.

Im Folgenden sehen wir eine einfache Anwendung, die den Benutzer zur Eingabe eines Suchstrings auffordert, die Google Geocoding-API aufruft und Informationen aus dem zurückgegebenen JSON extrahiert.

```
import urllib.request, urllib.parse
import json
import ssl

api_key = False
# If you have a Google Places API key, enter it here
# api_key = 'AIzaSy___IDByT70'
# https://developers.google.com/maps/documentation/geocoding/intro

if api_key is False:
    api_key = 42
```

```
    serviceurl = 'http://py4e-data.dr-chuck.net/json?'
else:
    serviceurl = 'https://maps.googleapis.com/maps/api/geocode/json?'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    address = input('Enter location: ')
    if len(address) < 1: break

    parms = dict()
    parms['address'] = address
    if api_key is not False: parms['key'] = api_key
    url = serviceurl + urllib.parse.urlencode(parms)

    print('Retrieving', url)
    uh = urllib.request.urlopen(url, context=ctx)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')

    try:
        js = json.loads(data)
    except:
        js = None

    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue

    print(json.dumps(js, indent=4))

    lat = js['results'][0]['geometry']['location']['lat']
    lng = js['results'][0]['geometry']['location']['lng']
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)

# Code: https://tiny.one/py4de/code3/geojson.py
```

Das Programm nimmt den Suchstring und konstruiert eine URL mit dem Suchstring als korrekt kodierten Parameter und verwendet dann `urllib`, um den Text von der Google Geocoding-API abzurufen. Im Gegensatz zu einer festen Webseite hängen die Daten, die wir erhalten, von den Parametern ab, die wir senden, und von den geografischen Daten, die auf den Servern von Google gespeichert sind.

Sobald wir die JSON-Daten abgerufen haben, parsen wir sie mit der `json`-Bibliothek

und führen ein paar Prüfungen durch, um sicherzustellen, dass wir korrekte Daten erhalten haben. Dann extrahieren wir die Informationen, nach denen wir suchen.

Die Ausgabe des Programms sieht wie folgt aus (ein Teil des zurückgegebenen JSON wurde für eine bessere Übersichtlichkeit entfernt):

```
$ python3 geojson.py
Enter location: Ann Arbor, MI
Retrieving http://py4e-data.dr-chuck.net/json?address=Ann+Arbor%2C+MI&key=42
Retrieved 1736 characters

{
    "results": [
        {
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "short_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ]
                },
                {
                    "long_name": "Washtenaw County",
                    "short_name": "Washtenaw County",
                    "types": [
                        "administrative_area_level_2",
                        "political"
                    ]
                },
                {
                    "long_name": "Michigan",
                    "short_name": "MI",
                    "types": [
                        "administrative_area_level_1",
                        "political"
                    ]
                },
                {
                    "long_name": "United States",
                    "short_name": "US",
                    "types": [
                        "country",
                        "political"
                    ]
                }
            ],
            "formatted_address": "Ann Arbor, MI, USA",
            "geometry": {
```

```
                "bounds": {
                    "northeast": {
                        "lat": 42.3239728,
                        "lng": -83.6758069
                    },
                    "southwest": {
                        "lat": 42.222668,
                        "lng": -83.799572
                    }
                },
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                },
                "location_type": "APPROXIMATE",
                "viewport": {
                    "northeast": {
                        "lat": 42.3239728,
                        "lng": -83.6758069
                    },
                    "southwest": {
                        "lat": 42.222668,
                        "lng": -83.799572
                    }
                }
            },
            "place_id": "ChIJMx9D1A2wPIgR4rXIhkb5Cds",
            "types": [
                "locality",
                "political"
            ]
        }
    ],
    "status": "OK"
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA


Enter location:
```


Wir können www.py4e.com/code3/geoxml.py herunterladen, um die XML-Variante der Google Geocoding-API zu erkunden.

Übung 1: Es soll py4e.com/code3/geojson.py oder py4e.com/code3/geoxml.py abgeändert werden, um den zweistelligen Ländercode aus den abgerufenen Daten auszugeben. Es muss eine Fehlerprüfung hinzugefügt werden, damit das Programm keinen Traceback auslöst, wenn der Ländercode nicht vorhanden ist. Sobald es zum Laufen gebracht wurde, soll nach dem „Atlantic Ocean“ gesucht und sichergestellt werden, dass es auch mit Orten umgehen kann, die in keinem Land liegen.

Anwendungsbeispiel 2: Twitter

Als die Twitter-API immer mehr an Bedeutung gewann, ging Twitter von einer offenen und öffentlichen API zu einer API über, die die Verwendung von OAuth-Signaturen bei jeder API-Anfrage erfordert.

Für das nächste Beispielprogramm laden wir die Dateien `twurl.py`, `hidden.py`, `oauth.py` und `twitter1.py` von www.py4e.com/code herunter und legen alle in einem Ordner auf dem Computer ab.

Um diese Programme nutzen zu können, müssen wir ein Twitter-Konto haben und unseren Python-Code als Anwendung autorisieren. Es muss dabei ein Key, Secret, Token und Token Secret eingerichtet werden. Dann muss die Datei `hidden.py` bearbeitet und diese vier Zeichenfolgen in die entsprechenden Variablen in der Datei eingefügt werden:

```
# Keep this file separate

# https://apps.twitter.com/
# Create new App and get the four strings

def oauth():
    return {"consumer_key": "h7Lu...Ng",
            "consumer_secret": "dNKenAC3New...mmn7Q",
            "token_key": "10185562-eibxCp9n2...P4GEQQOSGI",
            "token_secret": "H0ycCFemmC4wyf1...qoIpBo"}

# Code: https://tiny.one/py4de/code3/hidden.py
```

Der Zugriff auf den Twitter-Webdienst erfolgt über eine URL wie diese:

https://api.twitter.com/1.1/statuses/user_timeline.json

Sobald jedoch alle Sicherheitsinformationen hinzugefügt wurden, sieht die URL eher so aus:

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

Wir können die OAuth-Spezifikation lesen, wenn wir mehr über die Bedeutung der verschiedenen Parameter erfahren möchten, die hinzugefügt werden, um die Sicherheitsanforderungen von OAuth zu erfüllen.

Für die Programme, die wir mit Twitter ausführen, verstecken wir die ganze Komplexität in den Dateien `oauth.py` und `twurl.py`. Wir setzen einfach die Secrets in `hidden.py` und senden dann die gewünschte URL an die Funktion `twurl.augment()` und der Bibliothekscode fügt alle notwendigen Parameter für uns an die URL an.

Dieses Programm ruft die Timeline für einen bestimmten Twitter-Benutzer ab und gibt sie im JSON-Format in einer Zeichenkette an uns zurück. Wir geben dann einfach die ersten 250 Zeichen des Strings aus:

```
import urllib.request, urllib.parse, urllib.error
import twurl
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    print(data[:250])
    headers = dict(connection.getheaders())
    # print headers
    print('Remaining', headers['x-rate-limit-remaining'])

# Code: https://tiny.one/py4de/code3/twitter1.py
```

Wenn das Programm läuft, erzeugt es die folgende Ausgabe:

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178

Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
```

```
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

Enter Twitter Account:
```

Zusammen mit den zurückgegebenen Timeline-Daten gibt Twitter auch Metadaten über die Anfrage in den HTTP-Antwort-Headern zurück. Ein Header im Besonderen, `x-rate-limit-remaining`, informiert uns darüber, wie viele Anfragen wir noch stellen können, bevor wir für eine kurze Zeitspanne gesperrt werden. Wir können sehen, dass unsere verbleibenden Abrufe bei jeder Anfrage an die API um eins sinken.

Im folgenden Beispiel rufen wir die Twitter-Freunde eines Benutzers ab, parsen das zurückgegebene JSON und extrahieren einige der Informationen über diese Freunde. Außerdem geben wir das JSON nach dem Parsen aus und drucken es mit einem Einzug von vier Zeichen aus, damit wir die Daten durchforsten können, wenn wir weitere Felder extrahieren möchten.

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()

    js = json.loads(data)
    print(json.dumps(js, indent=2))

    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])
```

```
    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
            print('   * No status found')
            continue
        s = u['status']['text']
        print('  ', s[:50])
```

 # Code: https://tiny.one/py4de/code3/twitter2.py

Da das JSON zu einer Reihe von verschachtelten Python-Listen und -Dictionarys wird, können wir eine Kombination aus Indexzugriff und `for`-Schleifen verwenden, um die zurückgegebenen Datenstrukturen mit sehr wenig Pythoncode zu durchwandern.

Die Ausgabe des Programms sieht so aus (einige der Daten sind gekürzt, damit sie auf die Seite passen):

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14
```


```
{
  "next_cursor": 1444171224491980205,
  "users": [
    {
      "id": 662433,
      "followers_count": 28725,
      "status": {
        "text": "@jazzychad I just bought one .__.",
        "created_at": "Fri Sep 20 08:36:34 +0000 2013",
        "retweeted": false,
      },
      "location": "San Francisco, California",
      "screen_name": "leahculver",
      "name": "Leah Culver",
    },
    {
      "id": 40426722,
      "followers_count": 2635,
      "status": {
        "text": "RT @WSJ: Big employers like Google ...",
        "created_at": "Sat Sep 28 19:36:37 +0000 2013",
      },
      "location": "Victoria Canada",
      "screen_name": "_valeriei",
      "name": "Valerie Irvine",
    }
  ],
 "next_cursor_str": "1444171224491980205"
}
```

```
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

Im letzten Teil der Ausgabe sehen wir, wie die `for`-Schleife die fünf neuesten Freunde des Twitter-Kontos *@drchuck* ausliest und den neuesten Status für jeden Freund ausgibt. In dem zurückgegebenen JSON sind noch viel mehr Daten vorhanden. Wenn wir in die Ausgabe des Programms schauen, können wir auch sehen, dass das Finden der Freunde eines bestimmten Kontos eine andere Zugriffsratenbeschränkung hat als die Anzahl der Timeline-Abfragen, die wir pro Zeitspanne ausführen dürfen.

Durch die individuellen API-Schlüssel hat Twitter die Kontrolle darüber, wer ihre API und ihre Daten nutzt und in welchem Umfang. Die Beschränkung der Zugriffsraten erlaubt uns einfache persönliche Datenabrufe, aber er gestattet es uns nicht, eine Software zu entwickeln, die Millionen von Daten pro Tag aus der API abruft.

Kapitel 14

Objektorientierte Programmierung

Verwaltung größerer Programme

Zu Beginn dieses Buches haben wir vier grundlegende Programmiermuster betrachtet, die wir zur Konstruktion von Programmen verwenden:

- Sequentieller Code (Folge von Anweisungen)
- Bedingter Code (if-Anweisungen)
- Wiederholter Code (Schleifen)
- Wiederverwendung von Code (Funktionen)

In den ersten Kapiteln haben wir vorwiegend *elementare Datentypen* wie ganze Zahlen oder Fließkommazahlen verwendet und in den späteren Kapiteln dann *zusammengesetzte Datentypen* wie Listen, Tupeln und Dictionarys eingeführt.

Wenn wir Programme erstellen, verwenden wir Datentypen, um eigene Datenstrukturen zu entwerfen, und schreiben Code, um diese Datenstrukturen zu manipulieren. Es gibt viele Möglichkeiten, Programme zu schreiben, und inzwischen haben wir wahrscheinlich einige Programme geschrieben, die „nicht so elegant“ sind, wie sie sein könnten, und andere, die uns vielleicht etwas „eleganter“ geraten sind. Auch wenn unsere bisherigen Programme eher überschaubar sind, fängt man an zu begreifen, dass ein gewisses Gespür für das Schreiben von gut lesbarem und übersichtlichem Code hilfreich ist.

Wenn Programme Millionen von Zeilen lang werden, wird es immer wichtiger, Code zu schreiben, der leicht zu verstehen ist. Wenn wir an einem millionen Zeilen langen Programm arbeiten, können wir nie einen Überblick über das gesamte Programm gleichzeitig im Kopf behalten. Wir brauchen also Möglichkeiten, große Programme in mehrere kleinere Teile zu zerlegen, damit wir weniger zu beachten haben, wenn wir ein Problem lösen, einen Fehler beheben oder eine neue Funktionalität hinzufügen.

In gewisser Weise ist die objektorientierte Programmierung eine Möglichkeit, Code so anzuordnen, dass wir in 50 Zeilen des Codes hineinzoomen und ihn verstehen können,

während wir die anderen 999.950 Zeilen des Codes für den Moment ignorieren können.

Schon gehts los

Wie bei vielen Aspekten der Programmierung ist es notwendig, die Konzepte der objektorientierten Programmierung zu erlernen, bevor wir sie effektiv einsetzen können. Dieses Kapitel gehen wir so an, dass wir zunächst einige Begriffe und Konzepte kennenlernen und dann ein paar einfache Beispiele durcharbeiten, um eine Grundlage für das weitere Lernen zu schaffen.

Das wichtigste Ergebnis dieses Kapitels ist, ein grundlegendes Verständnis dafür zu erlangen, wie Objekte aufgebaut sind und wie sie funktionieren und vor allem, wie wir die Funktionen von Objekten nutzen, die uns von Python und den Python-Bibliotheken zur Verfügung gestellt werden.

Handhabung von Objekten

Wie sich herausstellt, haben wir in diesem Buch die ganze Zeit über Objekte verwendet. Python stellt uns viele eingebaute Objekte zur Verfügung. Hier ist ein einfaches Beispiel, bei dem man die ersten paar Zeilen recht intuitiv nachvollziehen kann.

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: https://tiny.one/py4de/code3/party1.py
```

Wir wollen uns in den nächsten Abschnitten ansehen, was hier aus der Sicht der objektorientierten Programmierung tatsächlich passiert. Dabei ist es nicht schlimm, wenn wir nicht alle Details sofort verstehen. Am Ende des Kapitels oder spätestens nach einem zweiten Lesedurchgang werden wir über ein gutes Verständnis der Objektorientierung verfügen.

Die erste Zeile *konstruiert* ein Objekt vom Typ `list`, die zweite und dritte Zeile *ruft* die `append()`-Methode auf, die vierte Zeile ruft die `sort()`-Methode auf, und die fünfte Zeile *liefert* das Element an Position 0.

Die sechste Zeile ruft die Methode `__getitem__()` in der Liste `stuff` mit einem Parameter 0 auf.

```
print (stuff.__getitem__(0))
```

Die siebte Zeile ist ein noch ausführlicherer Weg, um das nullte Element in der Liste abzurufen.

```
print (list.__getitem__(stuff,0))
```

In diesem Code rufen wir die Methode `__getitem__` der Klasse `list` auf und *übergeben* die Liste und das Element, das wir aus der Liste abrufen wollen, als Parameter.

Die letzten drei Zeilen des Programms sind gleichwertig, aber es ist bequemer, einfach die Syntax der eckigen Klammern zu verwenden, um ein Element an einer bestimmten Position in einer Liste nachzuschlagen.

Wir können einen Blick auf die Fähigkeiten eines Objekts werfen, indem wir uns die Ausgabe der Funktion `dir()` ansehen:

```
>>> stuff = list()
>>> dir(stuff)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

Im weiteren Verlauf dieses Kapitels werden alle oben genannten Begriffe definiert. Nach dem Durcharbeiten dieses Kapitels sollten wir die obigen Abschnitte erneut lesen, um unser Verständnis zu überprüfen.

Betrachtung von außen

Ein Programm in seiner einfachsten Form nimmt Eingaben entgegen, führt Verarbeitungen durch und erzeugt Ausgaben. Das folgende Programm zum Konvertieren einer Etagennummer demonstriert ein sehr kurzes, aber vollständiges Programm, das alle drei dieser Schritte zeigt.

```
usf = input('Enter the US Floor Number: ')
wf = int(usf) - 1
print('Non-US Floor Number is',wf)

# Code: https://tiny.one/py4de/code3/elev.py
```

Wenn wir ein bisschen mehr über dieses Programm nachdenken, gibt es die „Außenwelt“ und das Programm. Die Eingabe- und Ausgabeaspekte sind die Stellen,

an denen das Programm mit der Außenwelt interagiert. Innerhalb des Programms haben wir Code und Daten, um die Aufgabe zu erfüllen, die das Programm lösen soll.

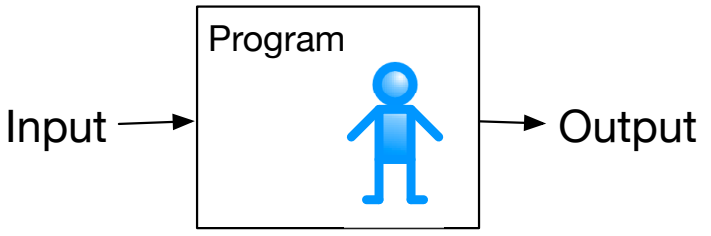

Abbildung 14.1: Ein Programm

Eine Möglichkeit, objektorientierte Programmierung zu verstehen, besteht in der Vorstellung, dass sie unser Programm in mehrere „Zonen“ aufteilt. Jede Zone enthält etwas Code und Daten (wie ein Programm) und hat gut definierte Interaktionen mit der Außenwelt und den anderen Zonen innerhalb des Programms.

Wenn wir auf das Programm zur Link-Extraktion zurückblicken, bei der wir die BeautifulSoup-Bibliothek verwendet haben, können wir ein Programm sehen, das durch das Verbinden verschiedener Objekte konstruiert wird, um eine Aufgabe zu erfüllen:

```
# To run this, download the BeautifulSoup zip file
# http://www.py4e.com/code3/bs4.zip
# and unzip it in the same directory as this file

import urllib.request, urllib.parse, urllib.error
from bs4 import BeautifulSoup
import ssl

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

url = input('Enter - ')
html = urllib.request.urlopen(url, context=ctx).read()
soup = BeautifulSoup(html, 'html.parser')

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
    print(tag.get('href', None))

# Code: https://tiny.one/py4de/code3/urllinks.py
```

Wir lesen die URL in einen String und übergeben diesen dann an `urllib`, um die Daten aus dem Web abzurufen. Die `urllib`-Bibliothek verwendet die `socket`-

Bibliothek, um die eigentliche Netzwerkverbindung zum Abrufen der Daten herzustellen. Wir nehmen die Zeichenkette, die `urllib` zurückgibt und übergeben sie an BeautifulSoup zum Parsen. BeautifulSoup verwendet das Objekt `html.parser`[1] und gibt wiederum ein Objekt zurück. Wir rufen die Funktion `tags()` für das zurückgegebene Objekt auf, die ein Dictionary mit Tag-Objekten zurückgibt. Wir laufen in einer Schleife durch die Tags und rufen die Funktion `get()` für jedes Tag auf, um das Attribut `href` auszugeben.

Wir können uns so ein Bild von diesem Programm machen und davon, wie die Objekte zusammenarbeiten.

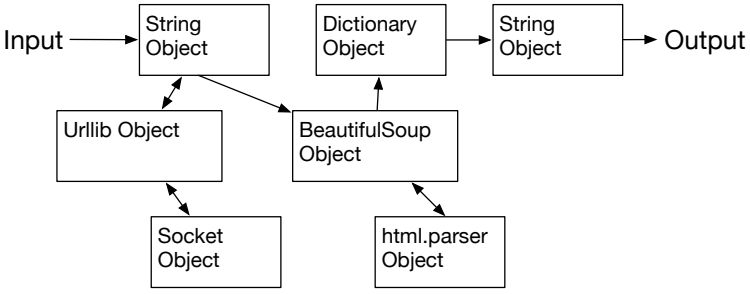

Abbildung 14.2: Ein Programm als Netzwerk von Objekten

Das Ziel ist hier nicht, perfekt zu verstehen, wie dieses Programm funktioniert, sondern zu sehen, wie wir ein Netzwerk aus interagierenden Objekten aufbauen und die Bewegung von Informationen zwischen den Objekten steuern, um ein Programm zu erstellen. Es ist auch wichtig zu bemerken, dass das Programm in den vorherigen Kapiteln vollständig verstanden werden konnte, ohne überhaupt zu erkennen, dass das Programm die Bewegung von Informationen zwischen den Objekten steuert. Es waren einfach nur Codezeilen, die die Aufgabe erledigt haben.

Unterteilen eines Problems

Einer der Vorteile des objektorientierten Ansatzes ist, dass er Komplexität verbergen kann. Zum Beispiel müssen wir zwar wissen, wie man den `urllib`- und BeautifulSoup-Code verwendet, aber wir müssen nicht wissen, wie diese Bibliotheken intern funktionieren. Das erlaubt es uns, uns auf den Teil des Problems zu konzentrieren, den wir lösen müssen, und die anderen Teile des Programms zu ignorieren.

Diese Fähigkeit, sich ausschließlich auf den Teil eines Programms zu fokussieren, der uns interessiert, und den Rest zu ignorieren, ist auch für die Programmierer der Objekte hilfreich. Zum Beispiel müssen die Programmierer, die BeautifulSoup entwickeln, nicht wissen oder sich darum kümmern, wie wir unsere HTML-Seite abrufen, welche Teile wir lesen wollen oder was wir mit den Daten vorhaben, die wir aus der Webseite extrahieren.

[1] https://docs.python.org/3/library/html.parser.html

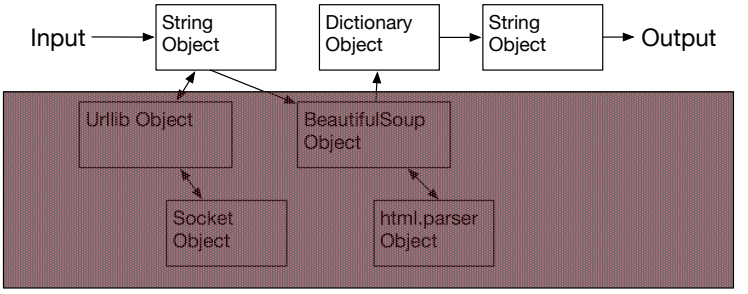

Abbildung 14.3: Ignorieren von Details bei der Verwendung eines Objekts

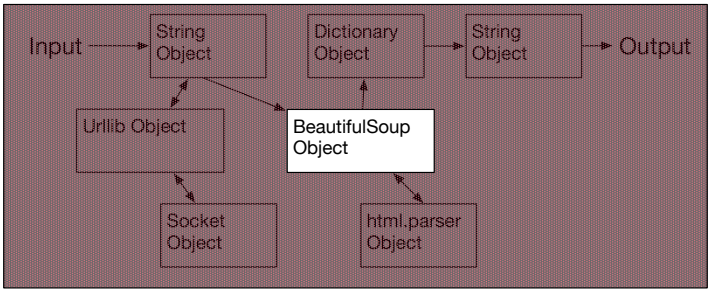

Abbildung 14.4: Ignorieren von Details beim Entwickeln eines Objekts

Unser erstes Python-Objekt

Auf einer grundlegenden Ebene ist ein Objekt einfach etwas Code plus Datenstrukturen, die kleiner sind als ein ganzes Programm. Das Definieren einer Funktion ermöglicht es uns, ein Stück Code zu speichern und ihm einen Namen zu geben und diesen Code dann später mit dem Namen der Funktion aufzurufen.

Ein Objekt kann eine Reihe von Funktionen (die wir *Methoden* nennen) sowie Daten enthalten, die von diesen Methoden verwendet werden. Wir nennen Datenelemente, die Teil des Objekts sind, *Attribute*.

Wir verwenden das Schlüsselwort `class`, um die Daten und den Code zu definieren, aus denen jedes der Objekte bestehen wird. Das Schlüsselwort `class` wird gefolgt vom Namen der Klasse und beginnt einen eingerückten Codeblock, in den die Attribute (Daten) und Methoden (Code) aufgenommen werden.

```
class PartyAnimal:

    def party(self):
        try:
            self.x = self.x + 1
        except AttributeError:
            self.x = 1
        print("Partys bisher:", self.x)
```

```
an = PartyAnimal()
an.party()
an.party()
an.party()
PartyAnimal.party(an)

# Code: https://tiny.one/py4de/code3/party2.py
```

Jede Methode sieht aus wie eine Funktion, die mit dem Schlüsselwort `def` beginnt und aus einem eingerückten Codeblock besteht. Dieses Objekt hat nur eine Methode (`party`). Die Methoden haben einen speziellen ersten Parameter, den wir per Konvention `self` nennen.

Genauso wie das Schlüsselwort `def` nicht die Ausführung von Funktionscode bewirkt, wird mit dem Schlüsselwort `class` kein Objekt erzeugt. Stattdessen definiert das Schlüsselwort `class` eine Art Vorlage, die angibt, welche Daten und welcher Code in jedem Objekt des Typs `PartyAnimal` enthalten sein werden. Die Klasse ist wie eine Ausstechform und die mit der Klasse erzeugten Objekte sind die Kekse. Wir kleben keinen Zuckerguss auf die Ausstechform, wir kleben Zuckerguss stattdessen auf die Kekse, und wir können auf jeden Keks einen anderen Zuckerguss kleben.

Abbildung 14.5: Eine Klasse und zwei Objekte[2]

Wenn wir uns das Beispielprogramm nun weiter ansehen, sehen wir die erste ausführbare Codezeile:

```
an = PartyAnimal()
```

Hier weisen wir Python an, ein *Objekt* oder eine *Instanz* der Klasse `PartyAnimal` zu konstruieren (d. h. zu erzeugen). Python konstruiert das Objekt mit den richtigen Daten und Methoden und gibt das Objekt zurück, das dann der Variablen `an` zugewiesen wird. In gewisser Weise ähnelt dies der folgenden Zeile, die wir schon die ganze Zeit verwendet haben:

[2]Bildquelle: Didriks. snowman cookie cutter. URL: https://www.flickr.com/photos/dinnerseries/23570475099. Lizenz: Creative Commons Attribution 2.0 Generic (CC BY 2.0) https://creativecommons.org/licenses/by/2.0/legalcode

```
counts = dict()
```

Hier weisen wir Python an, ein Objekt unter Verwendung der Vorlage `dict` (in Python bereits vorhanden) zu konstruieren, die Instanz des Dictionarys zurückzugeben und sie der Variablen `counts` zuzuweisen.

Wenn die Klasse `PartyAnimal` verwendet wird, um ein Objekt zu erstellen, wird die Variable `an` verwendet, um auf dieses Objekt zu zeigen. Wir verwenden `an`, um auf den Code und die Daten für diese bestimmte Instanz der Klasse `PartyAnimal` zuzugreifen.

Jedes `Partyanimal`-Objekt/jede Partyanimal-Instanz enthält in sich eine Methode namens `party`. Wir rufen die Methode `party` in dieser Zeile auf:

```
an.party()
```

Möglicherweise erscheint uns der Aufruf von `party()` nicht ganz vollständig. In der Implementierung haben wir ja gesagt, dass die Methode einen Parameter namens `self` besitzt, dieser Parameter wird anscheinend beim obigen Aufruf nicht angegeben. Dieser Anschein trügt allerdings, denn wir rufen die Methode schließlich *auf* einem `PartyAnimal`-Objekt, nämlich auf dem Objekt `an` auf. Dieses Objekt wird an den ersten Parameter einer Methode (den wir per Konvention `self` nennen) übergeben. Natürlich kann eine Methode neben der Selbstreferenz weitere Parameter besitzen, die bei einem Aufruf dann innerhalb der Klammern angegeben werden.

Eine Methode kann nicht nur über ein Objekt aufgerufen werden, sondern auch über seine Klasse. Statt `an.party()` könnte man also auch folgenden gleichbedeutenden Aufruf verwenden:

```
PartyAnimal.party(an)
```

Hier sagen wir, welche `party()` Methode wir meinen (die aus der Klasse `PartyAnimal`) und welches Objekt wir der Methode übergeben wollen (nämlich das Objekt `an`). Dies ist quasi die Langform eines Methodenaufrufs, die in der Praxis eher seltener verwendet wird.

Wenn wir nun in die Implementierung der Methode `party` gehen, sehen wir die folgenden Zeilen:

```
try:
    self.x = self.x + 1
except AttributeError:
    self.x = 1
```

Wir versuchen hier, ein Datenattribut `x`, das zu unserem `PartyAnimal`-Objekt gehört, um den Wert `1` zu erhöhen. Beim ersten Aufruf der Methode hat unser Objekt noch kein solches Attribut. Daher wird der Ausdruck `self.x + 1` einen `AttributeError` erzeugen, den wir behandeln, indem wir ein solches Attribut mit `self.x = 1` erzeugen und ihm den Wert `1` zuweisen.

Wenn das Programm ausgeführt wird, erzeugt es die folgende Ausgabe:

```
Partys bisher: 1
Partys bisher: 2
Partys bisher: 3
Partys bisher: 4
```

Das Objekt wird konstruiert und die Methode `party` wird viermal aufgerufen, wobei der Wert für `x` innerhalb des Objekts `an` sowohl inkrementiert als auch ausgegeben wird.

Klassen als Datentypen

Wie wir gesehen haben, haben in Python alle Variablen einen Typ. Wir können die eingebaute Funktion `dir` verwenden, um die Fähigkeiten einer Variablen zu untersuchen. Wir können `type` und `dir` auch mit Klassen verwenden, die wir erstellen.

```
class PartyAnimal:

    def party(self):
        try:
            self.x = self.x + 1
        except AttributeError:
            self.x = 1
        print("Partys bisher:", self.x)

an = PartyAnimal()
print ("Type", type(an))
print ("Dir ", dir(an))
print ("Type", type(an.party))

# Code: https://tiny.one/py4de/code3/party3.py
```

Bei der Ausführung erzeugt das Programm folgende Ausgabe:

```
Type <class '__main__.PartyAnimal'>
Dir  ['__class__', '__delattr__', ...
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'party']
Type <class 'int'>
Type <class 'method'>
```

Wir können sehen, dass wir mit dem Schlüsselwort `class` einen neuen Typ erstellt haben. Anhand der `dir`-Ausgabe können wir sehen, dass die Methode `party` im Objekt verfügbar ist.

Lebenszyklus von Objekten

In den vorherigen Beispielen definieren wir eine Klasse, verwenden diese Klasse, um eine Instanz dieser Klasse (ein Objekt) zu erstellen, und verwenden dann diese Instanz. Wenn das Programm beendet ist, werden alle Variablen verworfen. Normalerweise denken wir nicht viel über das Erzeugen und Zerstören von Objekten nach. Dies wird allerdings von Bedeutung, wenn wir eigene Klassen definieren. Dann müssen wir einige Aktionen innerhalb des Objekts durchführen, um Dinge einzurichten, während das Objekt konstruiert wird, und möglicherweise Dinge aufzuräumen, wenn das Objekt verworfen wird.

Selbst bei unserem einfachen `PartyAnimal`-Beispiel sollten wir etwas tun, wenn wir ein neues Objekt erzeugen. Statt bei einem ersten Aufruf von `party()` das Attribut `x` zu erzeugen, sollten wir es beim Erstellen des Objekts anlegen. Gleichermaßen können wir auch „Aufräumarbeiten“ leisten, die immer dann ausgeführt werden, wenn ein Objekt verworfen wird.

Um bestimmte Aktionen beim Erstellen oder Löschen eines Objekts durchzuführen, fügen wir zu unserer Klasse speziell benannte Methoden hinzu:

```
class PartyAnimal:

   def __init__(self):
     self.x = 0
     print("PartyAnimal wird erstellt")

   def party(self) :
     self.x = self.x + 1
     print("Partys bisher:", self.x)

   def __del__(self):
     print("Zerstört nach", self.x, "Partys")

an = PartyAnimal()
an.party()
an.party()
an = 42
print("an speichert nun", an)

# Code: https://tiny.one/py4de/code3/party4.py
```

Das Programm erzeugt nachfolgende Ausgabe:

```
PartyAnimal wird erstellt
Partys bisher: 1
Partys bisher: 2
Zerstört nach 2 Partys
an speichert nun 42
```

Während Python unser Objekt konstruiert, ruft es unsere Methode `__init__` auf, um uns die Möglichkeit zu geben, einige Standard- oder Anfangswerte für das Objekt einzurichten. Wenn Python auf die Zeile stößt

```
an = 42
```

wird unser Objekt tatsächlich „zerstört“, damit es die Variable `an` wiederverwenden kann, um den Wert `42` zu speichern. Genau in dem Moment, in dem unser `an`-Objekt zerstört wird, wird unser Destruktor-Code (`__del__`) aufgerufen. Wir können nicht verhindern, dass unsere Variable zerstört wird, aber wir können alle notwendigen Aufräumarbeiten durchführen, kurz bevor unser Objekt nicht mehr existiert.

Bei der Entwicklung von Objekten ist es üblich, einen Konstruktor zu einer Klasse anzugeben, um Anfangswerte für Objekte einzurichten. Es ist hingegen relativ selten, dass man einen Destruktor für ein Objekt benötigt.

Mehrere Instanzen

Bisher haben wir eine Klasse definiert, daraus ein einzelnes Objekt konstruiert, dieses Objekt verwendet und dann das Objekt verworfen. Der eigentliche Vorteil in der objektorientierten Programmierung liegt jedoch darin, mehrere Instanzen unserer Klasse konstruieren zu können.

Wenn wir mehrere Objekte aus unserer Klasse konstruieren, möchten wir vielleicht unterschiedliche Anfangswerte für jedes der Objekte einrichten. Wir können Daten an die Konstruktoren übergeben, um jedem Objekt einen anderen Anfangswert zu geben:

```
class PartyAnimal:

   def __init__(self, nam):
     self.x = 0
     self.name = nam
     print(self.name," wird erstellt")

   def party(self) :
     self.x = self.x + 1
     print(self.name, "hat", self.x, "Party(s) besucht")

s = PartyAnimal('Sally')
j = PartyAnimal('Jim')

s.party()
j.party()
s.party()

# Code: https://tiny.one/py4de/code3/party5.py
```

Der Konstruktor hat sowohl einen `self`-Parameter, der auf die Objektinstanz zeigt, als auch zusätzliche Parameter, die beim Aufbau des Objekts an den Konstruktor übergeben werden:

```
s = PartyAnimal('Sally')
```

Innerhalb des Konstruktors weist die zweite Anweisung den Parameter `nam` dem Attribut `name` innerhalb der Objektinstanz zu.

```
self.name = nam
```

Die Ausgabe des Programms zeigt, dass jedes der Objekte (`s` und `j`) seine eigenen unabhängigen Kopien von `x` und `nam` enthält:

```
Sally  wird erstellt
Jim  wird erstellt
Sally hat 1 Party(s) besucht
Jim hat 1 Party(s) besucht
Sally hat 2 Party(s) besucht
```

Vererbung

Eine weitere mächtige Eigenschaft der objektorientierten Programmierung ist die Möglichkeit, eine neue Klasse durch die Erweiterung einer bestehenden Klasse zu erstellen. Bei der Erweiterung einer Klasse nennen wir die ursprüngliche Klasse die *Basisklasse* und die neue Klasse die *abgeleitete Klasse*.

Für dieses Beispiel verschieben wir unsere Klasse `PartyAnimal` in eine eigene Datei. Dann können wir die Klasse `PartyAnimal` in eine neue Datei importieren und sie wie folgt erweitern:

```
from party import PartyAnimal

class CricketFan(PartyAnimal):

   def __init__(self, nam):
        super().__init__(nam)
        self.points = 0

   def six(self):
      self.points = self.points + 6
      self.party()
      print(self.name, "hat", self.points, "Punkte")

s = PartyAnimal("Sally")
s.party()
j = CricketFan("Jim")
```

```
j.party()
j.six()

# Code: https://tiny.one/py4de/code3/party6.py
```

Wenn wir die Klasse `CricketFan` definieren, geben wir an, dass wir die Klasse `PartyAnimal` erweitern. Das bedeutet, dass alle Variablen (`x`) und Methoden (`party`) der Klasse `PartyAnimal` von der Klasse `CricketFan` *geerbt* werden. Zum Beispiel rufen wir innerhalb der Methode `six` in der Klasse `CricketFan` die Methode `party` aus der Basisklasse `PartyAnimal` auf.

In unserem Konstruktor `__init__` rufen wir den Konstruktor der *Basisklasse* über die Funktion `super()` auf. Damit tun wir für unser `CricketFan`-Objekt alles, was wir auch beim Erzeugen eines `PartyAnimal`-Objektes tun würden. Zusätzlich fügen wir dem Objekt noch ein Attribut `points` hinzu.

Während das Programm ausgeführt wird, erzeugen wir `s` und `j` als unabhängige Instanzen von `PartyAnimal` und `CricketFan`. Das `j`-Objekt hat zusätzliche Fähigkeiten, welche über die des `s`-Objektes hinausgehen.

```
Sally  wird erstellt
Sally hat 1 Party(s) besucht
Jim  wird erstellt
Jim hat 1 Party(s) besucht
Jim hat 2 Party(s) besucht
Jim hat 6 Punkte
```

Zusammenfassung

Dies war eine sehr schnelle Einführung in die objektorientierte Programmierung, die sich hauptsächlich auf die Terminologie und die Syntax der Definition und Verwendung von Objekten konzentriert. Der Code am Anfang dieses Kapitels sollte nun jedoch besser verständlich sein:

```
stuff = list()
stuff.append('python')
stuff.append('chuck')
stuff.sort()
print (stuff[0])
print (stuff.__getitem__(0))
print (list.__getitem__(stuff,0))

# Code: https://tiny.one/py4de/code3/party1.py
```

Die erste Zeile konstruiert ein `list`-Objekt. Wenn Python das Objekt `list` erzeugt, ruft es den Konstruktor (namens `__init__`) auf, um die internen Datenattribute zu initialisieren, die zum Speichern der Listendaten verwendet werden. Wir haben keine Parameter an den Konstruktor übergeben. Wenn der Konstruktor seine Arbeit

beendet, verwenden wir die Variable `stuff`, um auf die zurückgegebene Instanz der Klasse `list` zu zeigen.

Die zweite und dritte Zeile rufen die Methode `append` mit einem Parameter auf, um ein neues Element am Ende der Liste hinzuzufügen, indem die Attribute innerhalb von `stuff` aktualisiert werden. Dann, in der vierten Zeile, rufen wir die Methode `sort` ohne Parameter auf, um die Daten innerhalb des `stuff`-Objekts zu sortieren.

Wir geben dann den ersten Eintrag in der Liste aus, indem wir die eckigen Klammern verwenden, die eine Abkürzung für den Aufruf der Methode `__getitem__` innerhalb von `stuff` sind. Dies ist äquivalent zum Aufruf der Methode `__getitem__` in der Klasse `list` und der Übergabe des Objekts `stuff` als ersten Parameter und der gesuchten Position als zweiten Parameter.

Am Ende des Programms wird das `stuff`-Objekt verworfen, aber nicht bevor der Destruktor (namens `__del__`) automatisch aufgerufen wurde, damit das Objekt bei Bedarf Aufräumarbeiten durchführen kann.

Das sind die Grundlagen der objektorientierten Programmierung. Es gibt viele zusätzliche Details, wie man objektorientierte Ansätze bei der Entwicklung großer Anwendungen und Bibliotheken am besten einsetzt, die den Rahmen dieses Kapitels aber sprengen würden.[3]

Glossar

Klasse Eine Art Vorlage, die zum Konstruieren eines Objekts verwendet werden kann. Sie definiert die Attribute und Methoden, aus denen das Objekt bestehen wird.

Attribut Eine Variable, die Teil einer Klasse ist.

Methode Eine Funktion, die in einer Klasse und den Objekten, die aus der Klasse konstruiert werden, enthalten ist.

Konstruktor Eine optionale, speziell benannte Methode (`__init__`), die in dem Moment aufgerufen wird, in dem eine Klasse verwendet wird, um ein Objekt zu konstruieren. Normalerweise wird dies verwendet, um Anfangswerte für das Objekt einzurichten.

Destruktor Eine optionale, speziell benannte Methode (`__del__`), die in dem Moment aufgerufen wird, kurz bevor ein Objekt zerstört wird.

Objekt Eine Instanz einer Klasse. Ein Objekt enthält alle Attribute und Methoden, die von der Klasse definiert wurden. Häufig werden die Begriffe „Instanz“ und „Objekt“ austauschbar verwendet.

Vererbung Wenn wir eine neue abgeleitete Klasse erstellen, indem wir eine vorhandene Klasse (Basisklasse) erweitern. Die untergeordnete Klasse hat alle Attribute und Methoden der übergeordneten Klasse plus zusätzliche Attribute und Methoden, die von der untergeordneten Klasse neu definiert werden.

abgeleitete Klasse Eine neue Klasse, die man erstellt, indem man eine übergeordnete Klasse (Basisklasse) erweitert. Die untergeordnete Klasse erbt alle Attribute und Methoden der übergeordneten Klasse.

[3] Wenn Sie neugierig sind, wo die Klasse `list` definiert ist, werfen Sie einen Blick auf https://github.com/python/cpython/blob/master/Objects/listobject.c. Die Klasse `list` ist in einer Sprache namens „C“ geschrieben.

Basisklasse Die Klasse, die erweitert wird, um eine neue abgeleitete Klasse zu erstellen. Die übergeordnete Klasse bringt alle ihre Methoden und Attribute in die neue untergeordnete Klasse ein.

Kapitel 15

Datenbanken und SQL

Was ist eine Datenbank?

Eine *Datenbank* ist eine Datei, die zum Speichern von Daten verwendet wird. Die meisten Datenbanken sind wie ein Dictionary in dem Sinne organisiert, dass sie von Schlüsseln auf Werte abbilden. Der größte Unterschied besteht darin, dass sich die Datenbank auf der Festplatte (oder einem anderen permanenten Speicher) befindet, sodass sie auch nach Beendigung des Programms bestehen bleibt. Da eine Datenbank auf einem permanenten Speicher gespeichert ist, kann sie viel mehr Daten speichern als ein Dictionary, das auf die Größe des Hauptspeichers beschränkt ist.

Ähnlich wie ein Dictionary ist Datenbanksoftware darauf ausgelegt, das Einfügen und den Zugriff auf Daten sehr schnell zu halten, auch bei großen Datenmengen. Datenbanksoftware hält ihre Leistung aufrecht, indem sie *Indizes* aufbaut, wenn Daten in die Datenbank eingefügt werden, damit der Computer schnell zu einem bestimmten Eintrag springen kann.

Es gibt viele verschiedene Datenbanksysteme, die für die unterschiedlichsten Zwecke eingesetzt werden, darunter: Oracle, MySQL, Microsoft SQL Server, PostgreSQL und SQLite. Wir konzentrieren uns in diesem Buch auf SQLite, weil es eine sehr verbreitete Datenbank ist, die bereits in Python integriert ist. SQLite ist dafür ausgelegt, in andere Anwendungen *eingebettet* zu werden, um Datenbankunterstützung innerhalb der Anwendung zu bieten. Zum Beispiel verwendet der Firefox-Browser wie viele andere Produkte auch intern die SQLite-Datenbank.

http://sqlite.org/

SQLite eignet sich gut für einige der Datenverarbeitungsprobleme, die uns in der Informatik begegnen, wie z. B. die Twitter-Spider-Anwendung, die wir in diesem Kapitel beschreiben.

Datenbankkonzepte

Auf den ersten Blick sieht eine Datenbank aus wie eine Tabellenkalkulation mit mehreren Sheets. Die primären Datenstrukturen in einer Datenbank sind: *Tabellen*, *Zeilen* und *Spalten*.

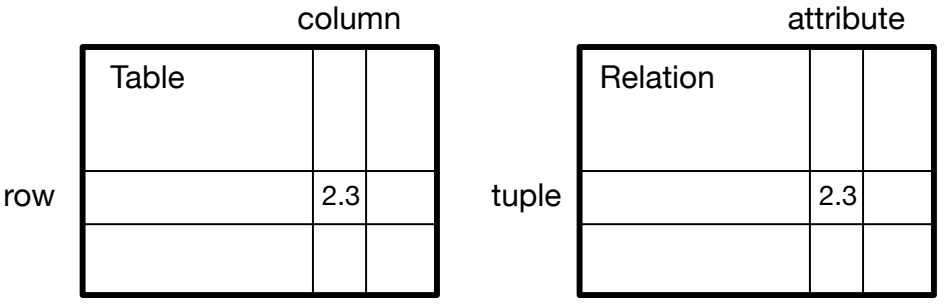

Abbildung 15.1: Relationale Datenbanken

In technischen Beschreibungen von relationalen Datenbanken werden die Konzepte Tabelle, Zeile und Spalte formaler als *Relation*, *Tupel* und *Attribut* bezeichnet. Wir werden in diesem Kapitel die weniger formalen Begriffe verwenden.

Datenbankbrowser für SQLite

Während sich dieses Kapitel auf die Verwendung von Python für die Arbeit mit Daten in SQLite-Datenbankdateien konzentriert, können viele Operationen bequemer mit einer Software namens *Database Browser for SQLite* durchgeführt werden, die frei erhältlich ist:

http://sqlitebrowser.org/

Mit dem Browser können wir ganz einfach Tabellen erstellen, Daten einfügen, Daten bearbeiten oder einfache SQL-Abfragen zu den Daten in der Datenbank ausführen.

In gewisser Weise ähnelt der Datenbankbrowser einem Texteditor beim Arbeiten mit Textdateien. Wenn wir eine oder sehr wenige Operationen an einer Textdatei durchführen möchten, können wir sie einfach in einem Texteditor öffnen und die gewünschten Änderungen vornehmen. Wenn wir viele Änderungen an einer Textdatei vornehmen müssen, werden wir oft ein einfaches Python-Programm schreiben. Das gleiche Vorgehen werden wir bei der Arbeit mit Datenbanken finden. Wir werden einfache Operationen im Datenbankmanager durchführen und komplexere Operationen werden am bequemsten in Python erledigt.

Erstellen einer Datenbanktabelle

Datenbanken erfordern eine strenger definierte Struktur als Python-Listen oder Python-Dictionarys[1].

Wenn wir eine *Datenbanktabelle* erstellen, müssen wir der Datenbank im Voraus die Namen der einzelnen *Spalten* in der Tabelle und die Art der Daten mitteilen, die wir in jeder *Spalte* zu speichern beabsichtigen. Wenn die Datenbanksoftware den Typ der Daten in jeder Spalte kennt, kann sie den effizientesten Weg zum Speichern und Nachschlagen der Daten auf der Grundlage des Datentyps wählen.

Die verschiedenen Datentypen, die von SQLite unterstützt werden, können wir unter der folgenden URL einsehen:

http://www.sqlite.org/datatypes.html

Das Definieren einer Struktur für unsere Daten im Voraus mag anfangs umständlich erscheinen, aber der Vorteil ist ein schneller Zugriff auf unsere Daten, selbst wenn die Datenbank eine große Menge an Daten enthält.

Der Code zum Erstellen einer Datenbankdatei und einer Tabelle namens `Tracks` mit zwei Spalten in der Datenbank lautet wie folgt:

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()

# Code: https://tiny.one/py4de/code3/db1.py
```

Die Operation `connect` stellt eine *Verbindung* zu der Datenbank her, die in der Datei `music.sqlite` im aktuellen Verzeichnis gespeichert ist. Wenn die Datei nicht vorhanden ist, wird sie erstellt. Der Grund, warum dies als *Verbindung* bezeichnet wird, ist, dass die Datenbank manchmal auf einem anderen *Datenbankserver* als dem Server gespeichert ist, auf dem wir unsere Anwendung ausführen. In unseren einfachen Beispielen ist die Datenbank nur eine lokale Datei im gleichen Verzeichnis wie der Python-Code, den wir ausführen.

Ein *Cursor* ist wie ein Dateihandler, mit dem wir Operationen mit den in der Datenbank gespeicherten Daten durchführen können. Der Aufruf von `cursor()` ist in etwa vergleichbar mit dem Aufruf von `open()` beim Umgang mit Textdateien.

Sobald wir den Cursor erhalten haben, können wir damit beginnen, mithilfe der Methode `execute()` Datenbankoperationen auszuführen.

[1] SQLite erlaubt zwar eine gewisse Flexibilität bei der Speicherung von Daten in einer Tabellenspalte, aber wir werden uns strikt an bestimmte Datentypen halten, damit die hier vorgestellten Konzepte auch auf andere Datenbanksysteme wie MySQL übertragbar sind.

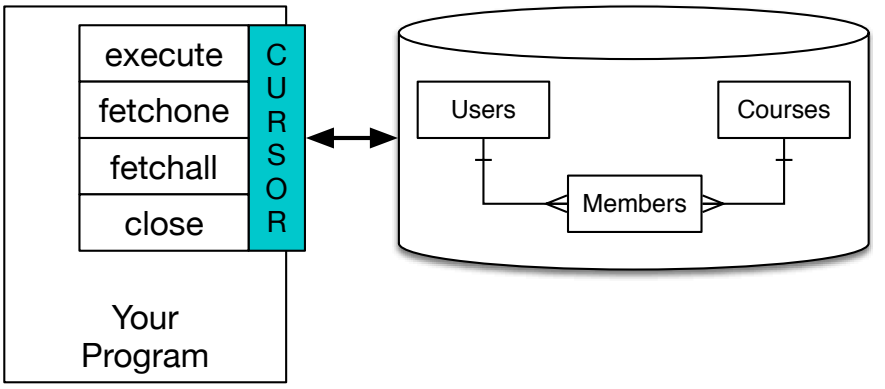

Abbildung 15.2: Ein Datenbankcursor

Datenbankbefehle werden in einer speziellen Sprache ausgedrückt, die über viele verschiedene Datenbankanbieter hinweg standardisiert wurde, sodass wir nur eine einzige Datenbanksprache erlernen müssen. Die Datenbanksprache wird *Structured Query Language* oder kurz *SQL* genannt. Eine Datenbankabfrage wird entsprechend als *Query* bezeichnet.

https://de.wikipedia.org/wiki/SQL

In unserem Beispiel führen wir zwei SQL-Befehle in unserer Datenbank aus. Per Konvention schreiben wir die SQL-Schlüsselwörter in Großbuchstaben und die Teile des Befehls, die wir hinzufügen (z. B. unsere Tabellen- und Spaltennamen), in Kleinbuchstaben. Das Datenbanksystem selbst unterscheidet nicht zwischen Groß- und Kleinschreibung.

Der erste SQL-Befehl entfernt die Tabelle `Tracks` aus der Datenbank, falls sie schon existieren sollte. Dies dient lediglich dazu, dass wir das gleiche Programm zum Erstellen der Tabelle `Tracks` immer wieder ausführen können, ohne einen Fehler zu verursachen. Es ist zu beachten, dass der Befehl `DROP TABLE` die Tabelle und ihren gesamten Inhalt sofort aus der Datenbank löscht. Diese Operation kann ohne Weiteres nicht rückgängig gemacht werden.

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

Der zweite Befehl erstellt eine Tabelle namens `Tracks` mit einer Textspalte namens `title` und einer Integer-Spalte namens `plays`.

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

Nachdem wir nun eine Tabelle namens `Tracks` erstellt haben, können wir mit der SQL-Operation `INSERT` Daten in diese Tabelle einfügen. Wieder beginnen wir damit, eine Verbindung zur Datenbank herzustellen und einen „Cursor“ zu erhalten. Dann können wir SQL-Befehle mit dem Cursor ausführen.

Der `INSERT`-Befehl gibt zunächst an, in welche Tabelle wir die Daten einfügen möchten. In runden Klammern folgen die Spalten, in die die Werte geschrieben

werden sollen. Zuletzt werden die Werte (`VALUES`) aufgelistet. Statt konkreter Werte verwenden wir hier zwei Fragezeichen als Platzhalter. Als zweites Argument von `execute()` übergeben wir dann ein Python-Tupel, das die eigentlichen Werte enthält. Intern ersetzt die `execute()`-Methode dann die Platzhalter durch die tatsächlichen Werte.

```
import sqlite3

conn = sqlite3.connect('music.sqlite')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
    ('Thunderstruck', 20))
cur.execute('INSERT INTO Tracks (title, plays) VALUES (?, ?)',
    ('My Way', 15))
conn.commit()

print('Tracks:')
cur.execute('SELECT title, plays FROM Tracks')
for row in cur:
     print(row)

cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()

cur.close()

# Code: https://tiny.one/py4de/code3/db2.py
```

Zuerst fügen wir mit `INSERT` zwei Zeilen (Datensätze) in unsere Tabelle ein und benutzen `commit()`, um das Schreiben der Daten in die Datenbankdatei zu veranlassen.

Tracks

title	plays
Thunderstruck	20
My Way	15

Abbildung 15.3: Eine Tabelle mit zwei Datensätzen

Dann verwenden wir den Befehl `SELECT`, um die Zeilen, die wir gerade eingefügt haben, aus der Tabelle abzurufen. Beim `SELECT`-Befehl geben wir an, welche Spalten wir abfragen möchten `(title, plays)` und aus welcher Tabelle wir die Daten abrufen möchten. Nachdem wir die `SELECT`-Anweisung ausgeführt haben, können wir den Cursor in einer `for`-Schleife durchlaufen. Um die Effizienz zu erhöhen, liest

der Cursor nicht alle Daten aus der Datenbank, wenn wir die Anweisung `SELECT` ausführen. Stattdessen werden die Daten bei Bedarf gelesen, während wir in der Schleife durch die Zeilen iterieren.

Die Ausgabe des Programms sieht dann so aus:

```
Tracks:
('Thunderstruck', 20)
('My Way', 15)
```

Unsere `for`-Schleife findet zwei Zeilen, und jede Zeile ist ein Python-Tupel mit dem ersten Wert als `title` und dem zweiten Wert als die Anzahl der `plays`.

Hinweis: In anderen Büchern oder im Internet sehen wir möglicherweise Zeichenketten, die mit `u'` beginnen. Dies war in Python 2 ein Hinweis darauf, dass es sich bei den Strings um *Unicode*-Strings handelt, die in der Lage sind, nicht-lateinische Zeichensätze zu speichern. In Python 3 sind alle Zeichenketten standardmäßig Unicode-Zeichenketten.

Ganz am Ende des Programms führen wir einen SQL-Befehl aus, um die gerade erstellten Zeilen mit `DELETE` zu löschen, damit wir das Programm immer wieder ausführen können. Der `DELETE`-Befehl zeigt die Verwendung einer `WHERE`-Klausel, mit der wir ein Auswahlkriterium angeben. Damit wird der Befehl nur auf die Datensätze angewendet, die das Kriterium erfüllen. In diesem Beispiel trifft das Kriterium auf alle Zeilen zu, also leeren wir die Tabelle, damit wir das Programm wiederholt ausführen können. Nachdem `DELETE` ausgeführt wurde, rufen wir wieder `commit()` auf, um die Entfernung der Daten aus der Datenbank zu veranlassen.

Zusammenfassung von SQL

Bisher haben wir die Structured Query Language in unseren Python-Beispielen verwendet und einige wesentliche Grundlagen ausgewählter SQL-Befehle behandelt. In diesem Abschnitt gehen wir auf die SQL-Sprache im Speziellen ein und geben einen Überblick über die SQL-Syntax.

Da es so viele verschiedene Datenbankanbieter gibt, wurde die Structured Query Language (SQL) standardisiert, damit wir in einer portablen Weise mit Datenbanksystemen verschiedener Anbieter kommunizieren können.

Eine relationale Datenbank setzt sich aus Tabellen, Zeilen und Spalten zusammen. Die Spalten haben im Allgemeinen einen Typ, z. B. Text, numerische Daten oder Datumsdaten. Wenn wir eine Tabelle erstellen, geben wir die Namen und Typen der Spalten an:

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

Um eine Zeile in eine Tabelle einzufügen, verwenden wir den SQL-Befehl `INSERT`:

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

Die Anweisung `INSERT` gibt den Tabellennamen an, dann eine Liste der Spalten, in die wir Werte schreiben möchten, und dann das Schlüsselwort `VALUES` und eine Liste der entsprechenden Werte für jedes der Felder.

Der SQL-Befehl `SELECT` wird zum Abrufen von Zeilen und Spalten aus einer Datenbank verwendet. Mit der `SELECT`-Anweisung können wir angeben, welche Spalten wir abrufen möchten, sowie eine `WHERE`-Klausel definieren, um die Zeilen auszuwählen, die wir sehen möchten. Sie erlaubt auch eine optionale `ORDER BY`-Klausel, um die Sortierung der zurückgegebenen Zeilen zu steuern.

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

Die Verwendung von `*` zeigt an, dass die Datenbank alle Spalten für jede Zeile zurückgeben soll, die mit der `WHERE`-Klausel übereinstimmt.

Es ist zu beachten, dass – anders als in Python – in einer SQL-`WHERE`-Klausel ein *einzelnes* Gleichheitszeichen verwendet wird, um einen Test auf Gleichheit durchzuführen, und nicht ein doppeltes Gleichheitszeichen. Andere logische Operationen, die in einer `WHERE`-Klausel erlaubt sind, sind `<`, `>`, `<=`, `>=`, `!=`, sowie `AND` und `OR` und Klammern, um unsere logischen Ausdrücke aufzubauen.

Wir können anfordern, dass die zurückgegebenen Zeilen nach einem der Felder wie folgt sortiert werden:

```
SELECT title,plays FROM Tracks ORDER BY title
```

Um eine Zeile zu entfernen, benötigen wir eine `WHERE`-Klausel in einer SQL `DELETE`-Anweisung. Die `WHERE`-Klausel bestimmt, welche Zeilen gelöscht werden sollen:

```
DELETE FROM Tracks WHERE title = 'My Way'
```

Es ist möglich, eine oder mehrere Spalten in einer oder mehreren Zeilen einer Tabelle mit der SQL-Anweisung `UPDATE` wie folgt zu aktualisieren:

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

Die `UPDATE`-Anweisung gibt eine Tabelle an, gefolgt von einer Liste von Spalten und Werten (nach `SET`), die geändert werden sollen, und schließlich eine optionale `WHERE`-Klausel, um die Zeilen auszuwählen, die aktualisiert werden sollen. Eine einzelne `UPDATE`-Anweisung ändert alle Zeilen, die mit der `WHERE`-Klausel übereinstimmen. Wenn keine `WHERE`-Klausel angegeben ist, wird die `UPDATE`-Anweisung für alle Zeilen in der Tabelle ausgeführt.

Diese vier grundlegenden SQL-Befehle (`INSERT`, `SELECT`, `UPDATE` und `DELETE`) ermöglichen die vier grundlegenden Operationen, die zum Erstellen und Verwalten von Daten erforderlich sind.

Auslesen von Twitter-Daten mithilfe einer Datenbank

In diesem Abschnitt werden wir ein einfaches Spider-Programm erstellen, das Twitter-Konten durchgeht und eine Datenbank von ihnen erstellt. **Hinweis:** Vorsicht, wenn das Programm ausgeführt wird. Wir möchten nicht zu viele Daten abrufen oder das Programm zu lange laufen lassen, damit der Zugang nicht gesperrt wird.

Eines der Probleme von Spider-Programmen ist, dass sie in der Lage sein müssen, viele Male angehalten und neu gestartet zu werden, ohne das wir dabei die Daten verlieren, die wir bisher abgerufen haben. Wir wollen unsere Datenabfrage nicht immer wieder von vorne beginnen, also wollen wir die Daten speichern, während wir sie abrufen, damit unser Programm wieder starten und dort weitermachen kann, wo es aufgehört hat.

Wir beginnen damit, dass wir die Twitter-Freunde einer Person und deren Status („schon verarbeitet“/„noch nicht verarbeitet“) abrufen, die Liste der Freunde in einer Schleife durchlaufen und jeden der Freunde zu einer Datenbank hinzufügen, um sie in Zukunft abrufen zu können. Nachdem wir die Twitter-Freunde einer Person verarbeitet haben, schauen wir in unserer Datenbank nach und rufen einen der Freunde des Freundes ab. Wir machen das immer wieder, wählen eine „nicht besuchte“ Person aus, rufen ihre Freundesliste ab und fügen die Freunde, die wir bisher nicht besucht haben, unserer Liste der noch nicht besuchten Freunde hinzu.

Wir verfolgen auch, wie oft wir einen bestimmten Freund in der Datenbank gesehen haben, um ein Gefühl für seine Beliebtheit zu bekommen.

In einer Datenbank speichern wir die Liste der bekannten Konten, deren Beliebtheit und die Information, ob wir das Konto bereits besucht (d. h. schon abgearbeitet) haben. Dadurch können wir unser Programm beliebig oft anhalten und neu starten.

Dieses Programm ist recht komplex. Es basiert auf dem Code aus der Übung weiter oben im Buch, die die Twitter-API verwendet.

Hier ist der Quellcode für unsere Twitter Spider-Anwendung:

```
from urllib.request import urlopen
import urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()

cur.execute('''
            CREATE TABLE IF NOT EXISTS Twitter
```

```
            (name TEXT, retrieved INTEGER, friends INTEGER)''')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
    if (len(acct) < 1):
        cur.execute(
            'SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print('No unretrieved Twitter accounts found')
            continue

    url = twurl.augment(
        TWITTER_URL, {'screen_name': acct, 'count': '20'})
    print('Retrieving', url)
    connection = urlopen(url, context=ctx)
    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Remaining', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    # Debugging
    # print json.dumps(js, indent=4)

    cur.execute(
        'UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute(
            'SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
            (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute(
                'UPDATE Twitter SET friends = ? WHERE name = ?',
                (count+1, friend))
            countold = countold + 1
        except:
            cur.execute(
```

```
                '''INSERT INTO Twitter (name, retrieved, friends)
                VALUES (?, 0, 1)''', (friend, ))
            countnew = countnew + 1
    print('New accounts=', countnew, ' revisited=', countold)
    conn.commit()

cur.close()

# Code: https://tiny.one/py4de/code3/twspider.py
```

Unsere Datenbank ist in der Datei `spider.sqlite` gespeichert und sie hat eine Tabelle namens `Twitter`. Jede Zeile in der `Twitter`-Tabelle hat eine Spalte für den Kontonamen, ob wir die Freunde dieses Kontos abgerufen haben und wie oft dieses Konto „befreundet“ wurde.

In der Hauptschleife des Programms wird der Benutzer aufgefordert, den Namen eines Twitter-Kontos einzugeben oder das Programm mit `quit` zu beenden. Wenn der Benutzer einen Twitter-Account eingibt, rufen wir die Liste der Freunde und den Status dieses Users ab und fügen jeden Freund zur Datenbank hinzu, wenn er nicht bereits in der Datenbank vorhanden ist. Wenn der Freund bereits in der Liste ist, fügen wir 1 zum Feld `friends` in der Zeile in der Datenbank hinzu.

Wenn der Benutzer die Eingabetaste drückt, suchen wir in der Datenbank nach dem nächsten Twitter-Konto, das wir noch nicht abgerufen haben, rufen die Freunde und den Status dieses Kontos ab, fügen sie der Datenbank hinzu oder aktualisieren sie und erhöhen die Anzahl ihrer `friends`.

Sobald wir die Liste der Freunde und deren Status abgerufen haben, gehen wir in einer Schleife durch alle `user`-Elemente im zurückgegebenen JSON und rufen den `screen_name` für jeden Benutzer ab. Dann verwenden wir die Anweisung `SELECT`, um zu sehen, ob wir diesen bestimmten `screen_name` bereits in der Datenbank gespeichert haben und rufen die Anzahl der Freunde (`friends`) ab, wenn der Datensatz existiert.

```
countnew = 0
countold = 0
for u in js['users']:
    friend = u['screen_name']
    print(friend)
    cur.execute(
        'SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
        (friend, ))
    try:
        count = cur.fetchone()[0]
        cur.execute(
            'UPDATE Twitter SET friends = ? WHERE name = ?',
            (count+1, friend))
        countold = countold + 1
    except:
        cur.execute(
            '''INSERT INTO Twitter (name, retrieved, friends)
```

```
            VALUES (?, 0, 1)''', (friend, ))
        countnew = countnew + 1
print('New accounts=', countnew, ' revisited=', countold)
conn.commit()
```

Sobald der Cursor (`cur`) die `SELECT`-Anweisung ausführt, müssen wir die Zeilen abrufen. Wir könnten dies mit einer `for`-Schleife tun, aber da wir nur eine Zeile abrufen (`LIMIT 1`), können wir die Methode `fetchone()` verwenden, um die erste (und einzige) Zeile zu laden, die durch `SELECT` ermittelt wurde. Da `fetchone()` die Zeile als *Tupel* zurückgibt (auch wenn es nur ein Feld gibt), nehmen wir den ersten Wert aus dem Tupel, um die aktuelle Anzahl der Freunde in die Variable `count` zu speichern.

Wenn dieser Abruf erfolgreich ist, verwenden wir die SQL-Anweisung `UPDATE` mit einer `WHERE`-Klausel, um der Spalte `friends` für die Zeile, die dem Konto des Freundes entspricht, um eins zu erhöhen. Es ist zu beachten, dass es zwei Platzhalter (Fragezeichen) in der SQL-Anweisung gibt und der zweite Parameter von `execute()` ein Tupel mit zwei Elementen ist, das die Werte enthält, die in der SQL-Anweisung anstelle der Fragezeichen eingesetzt werden sollen.

Wenn der Code im `try`-Block fehlschlägt, liegt es wahrscheinlich daran, dass kein Datensatz mit der `WHERE name = ?`-Klausel der `SELECT`-Anweisung übereinstimmt. Im `except`-Block verwenden wir also die SQL-Anweisung `INSERT`, um den `screen_name` des Freundes zur Tabelle hinzuzufügen, mit dem Hinweis, dass wir den `screen_name` noch nicht abgerufen haben, und setzen die Anzahl der Freunde auf eins.

Wenn das Programm also zum ersten Mal ausgeführt wird und wir ein Twitter-Konto eingeben, läuft das Programm wie folgt:

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

Da wir das Programm zum ersten Mal ausführen, ist die Datenbank leer. Wir erstellen die Datenbank in der Datei `spider.sqlite` und fügen eine Tabelle namens `Twitter` zur Datenbank hinzu. Dann rufen wir einige Freunde ab und fügen sie alle zur Datenbank hinzu, da die Datenbank leer ist.

An dieser Stelle möchten wir vielleicht einen einfachen Datenbank-„Dumper“ implementieren, um zu sehen, was in unserer Datei `spider.sqlite` steht:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
    count = count + 1
```

```
print(count, 'rows.')
cur.close()

# Code: https://tiny.one/py4de/code3/twdump.py
```

Dieses Programm öffnet einfach die Datenbank und wählt alle Spalten aller Zeilen in der Tabelle `Twitter` aus, geht dann in einer Schleife durch die Zeilen und gibt jede Zeile aus.

Wenn wir dieses Programm nach der ersten Ausführung unseres obigen Twitter-Spiders ausführen, sieht seine Ausgabe so aus:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 rows.
```

Wir sehen eine Zeile für jeden `screen_name`. Weiterhin sehen wir, dass wir die Daten für diesen `screen_name` noch nicht abgerufen haben und dass jeder User in der Datenbank einen Freund hat.

Jetzt spiegelt unsere Datenbank den Abruf der Freunde unseres ersten Twitter-Accounts wider (*drchuck*). Wir können das Programm erneut starten und ihm sagen, dass es die Freunde des nächsten „unverarbeiteten“ Kontos abrufen soll, indem wir die Eingabetaste anstelle eines Twitter-Kontos drücken:

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

Da wir die Eingabetaste gedrückt haben (d. h. wir haben kein Twitter-Konto angegeben), wird der folgende Code ausgeführt:

```
if(len(acct) < 1):
    cur.execute(
        'SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print('No unretrieved twitter accounts found')
        continue
```

Wir verwenden die SQL-Anweisung `SELECT`, um den Namen des ersten (`LIMIT 1`) Benutzers abzurufen, bei dem der Wert für „haben wir diesen Benutzer abgerufen?“ noch auf Null gesetzt ist. Wir verwenden auch `fetchone()[0]` innerhalb eines `try/except`-Blocks, um entweder einen `screen_name` aus den abgerufenen Daten zu extrahieren oder eine Fehlermeldung auszugeben und die Schleife erneut zu starten.

Wenn wir erfolgreich einen unverarbeiteten `screen_name` abgerufen haben, rufen wir dessen Daten folgendermaßen ab:

```
url=twurl.augment(TWITTER_URL,{'screen_name': acct,'count': '20'})
print('Retrieving', url)
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)

cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?',
            (acct, ))
```

Sobald wir die Daten erfolgreich abgerufen haben, verwenden wir die Anweisung `UPDATE`, um die Spalte `retrieved` auf 1 zu setzen, um damit anzuzeigen, dass wir den Abruf der Freunde dieses Kontos abgeschlossen haben. Dies verhindert, dass wir die gleichen Daten immer wieder abrufen, und sorgt dafür, dass wir im Netzwerk der Twitter-Freunde vorankommen.

Wenn wir das *friend*-Programm ausführen und zweimal die Eingabetaste drücken, um die Freunde des nächsten nicht besuchten Freundes abzurufen, und dann das Dumper-Programm ausführen, erhalten wir die folgende Ausgabe:

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 rows.
```

Wir können sehen, dass wir `lhawthorn` und `opencontent` besucht haben. Auch die Konten `cnxorg` und `kthanos` haben bereits zwei Follower. Da wir nun die Freunde von drei Personen (`drchuck`, `opencontent` und `lhawthorn`) abgerufen haben, hat unsere Tabelle 55 Zeilen mit Freunden zum Abrufen.

Jedes Mal, wenn wir das Programm starten und die Eingabetaste drücken, wählt es das nächste nicht besuchte Konto aus (z. B. wird das nächste Konto `steve_coppin` sein), ruft dessen Freunde ab, markiert sie als abgerufen und fügt sie für jeden der Freunde von `steve_coppin` entweder am Ende der Datenbank hinzu oder aktualisiert ihre Freundesanzahl, wenn sie bereits in der Datenbank sind.

Da die Daten des Programms alle auf der Festplatte in einer Datenbank gespeichert werden, kann die Spider-Aktivität beliebig oft ohne Datenverlust unterbrochen und wieder aufgenommen werden.

Grundlagen der Datenmodellierung

Die eigentliche Leistung einer relationalen Datenbank besteht darin, dass wir mehrere Tabellen erstellen und zwischen diesen Tabellen Verknüpfungen herstellen können. Der Entwurfsprozess, in welchem wir unsere Anwendungsdaten in mehrere Tabellen aufteilen und die Beziehungen zwischen den Tabellen herstellen, wird *Datenmodellierung* genannt. Ein Diagramm, das die Tabellen und ihre Beziehungen darstellt, wird als *Datenmodell* bezeichnet.

Die Datenmodellierung ist eine relativ anspruchsvolle Aufgabe und wir werden in diesem Abschnitt nur die grundlegendsten Konzepte der relationalen Datenmodellierung vorstellen. Weitere Details zur Datenmodellierung können hier eingesehen werden:

https://de.wikipedia.org/wiki/Relationale_Datenbank

Nehmen wir an, für unsere Twitter-Spider-Anwendung wollten wir nicht nur die Freunde einer Person zählen, sondern eine Liste aller eingehenden Beziehungen führen, damit wir eine Liste aller Personen finden, die einem bestimmten Konto folgen.

Da jeder User viele Konten haben kann, die ihm folgen, können wir nicht einfach eine einzelne Spalte zu unserer `Twitter`-Tabelle hinzufügen. Also erstellen wir eine neue Tabelle, die die Freundespaare speichert. Im Folgenden sehen wir eine einfache Möglichkeit, eine solche Tabelle zu erstellen:

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

Jedes Mal, wenn wir auf eine Person stoßen, der `drchuck` folgt, würden wir eine Zeile dieser Form einfügen:

```
INSERT INTO Pals (from_friend, to_friend)
    VALUES ('drchuck', 'lhawthorn')
```

Da wir die 20 Freunde aus dem Twitter-Feed von `drchuck` verarbeiten, fügen wir 20 Datensätze mit `drchuck` als erstem Parameter ein, sodass die Zeichenkette am Ende viele Male in der Datenbank auftaucht.

Diese redundante Datenhaltung verstößt gegen eines der Prinzipien der *Normalisierung* von Datenbanken, die im Grunde besagt, dass wir dieselben Daten nie mehr als einmal in die Datenbank aufnehmen sollten. Wenn wir die Daten mehr als einmal benötigen, erstellen wir einen numerischen *Schlüssel* (einen *Primärschlüssel*) für die Daten und *referenzieren* die eigentlichen Daten über diesen Schlüssel.

In der Praxis nimmt eine Zeichenkette viel mehr Platz auf der Festplatte und im Speicher unseres Computers ein als eine Ganzzahl und benötigt mehr Prozessorzeit

beim Vergleichen und Sortieren. Wenn wir nur ein paar hundert Einträge haben, spielt die Speicher- und Prozessorzeit kaum eine Rolle. Aber wenn wir eine Million Personen in unsere Datenbank aufnehmen und möglicherweise 100 Millionen Freundschaftsverbindungen, ist es wichtig, die Daten so schnell wie möglich durchsuchen zu können.

Wir speichern unsere Twitter-Konten in einer Tabelle namens `People` anstelle der im vorherigen Beispiel verwendeten Tabelle `Twitter`. Die Tabelle `People` hat eine zusätzliche Spalte, um den numerischen Schlüssel zu speichern, der mit der Zeile für diesen Twitter-Benutzer verbunden ist. SQLite verfügt über eine Funktionalität, die automatisch den Schlüsselwert (`INTEGER PRIMARY KEY`) für jede Zeile hinzufügt, die wir in eine Tabelle einfügen.

Wir können die Tabelle `People` mit dieser zusätzlichen Spalte `id` wie folgt erstellen:

```
CREATE TABLE People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

Es ist zu beachten, dass wir nicht mehr in jeder Zeile der Tabelle `People` die Anzahl der Freunde mitführen. Wenn wir `INTEGER PRIMARY KEY` als Typ unserer `id`-Spalte wählen, geben wir an, dass SQLite diese Spalte verwaltet und jeder Zeile, die wir einfügen, automatisch einen eindeutigen numerischen Schlüssel zuweist. Wir fügen auch das Schlüsselwort `UNIQUE` hinzu, um anzugeben, dass wir SQLite nicht erlauben, zwei Zeilen mit demselben Wert für `name` einzufügen.

Statt der obigen Tabelle `Pals` erstellen wir nun eine Tabelle namens `Follows` mit zwei Integer-Spalten `from_id` und `to_id` und der Einschränkung (englisch *Constraint*) für die Tabelle, dass die *Kombination* von `from_id` und `to_id` in dieser Tabelle eindeutig sein muss (d. h. wir können keine zwei identischen Zeilen einfügen).

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))
```

Wenn wir `UNIQUE`-Klauseln zu unseren Tabellen hinzufügen, vereinbaren wir eine Reihe von Regeln, die die Datenbank befolgen wird, wenn wir versuchen, Datensätze einzufügen. Wir erstellen diese Regeln aus Bequemlichkeit in unseren Programmen. Die Regeln halten uns davon ab, Fehler zu machen, und erleichtern uns obendrein die Programmierung.

Im Wesentlichen modellieren wir bei der Erstellung dieser `Follows`-Tabelle eine *Beziehung* (Relation), bei der eine Person einer anderen „folgt“, und stellen sie mit einem Zahlenpaar dar, das anzeigt, dass (a) die Personen miteinander verbunden sind und (b) die Richtung der Beziehung verdeutlicht.

Arbeiten mit mehreren Tabellen

Wir werden nun das Twitter-Spider-Programm mit zwei Tabellen, Primärschlüsseln und Schlüsselreferenzen wie oben beschrieben neu erstellen. Hier ist der Code für die neue Version des Programms:

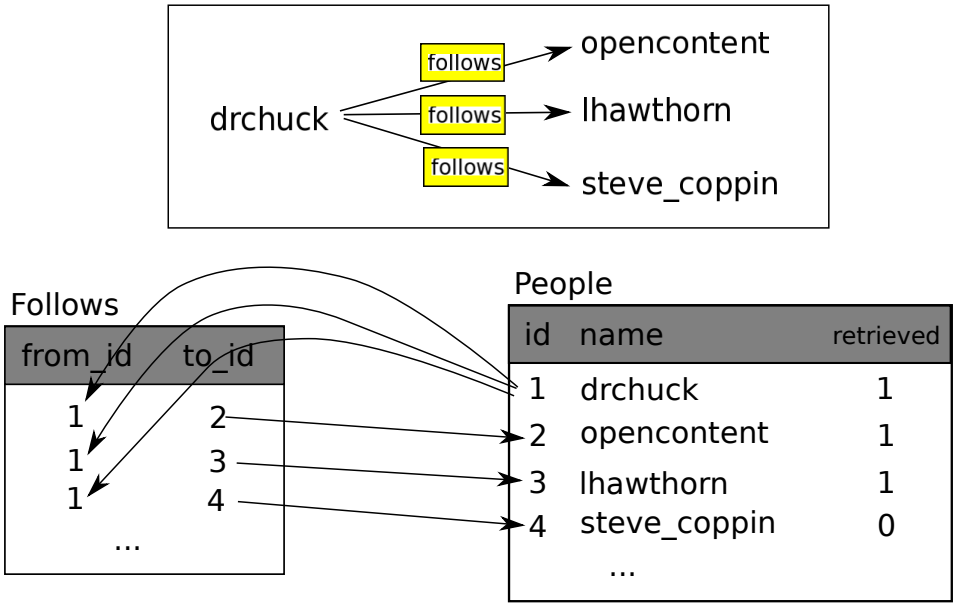

Abbildung 15.4: Beziehungen zwischen Tabellen

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute(
    '''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE,
    retrieved INTEGER)''')
cur.execute(
    '''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input('Enter a Twitter account, or quit: ')
    if (acct == 'quit'): break
```

```
if (len(acct) < 1):
    cur.execute(
        'SELECT id, name FROM People WHERE retrieved=0 LIMIT 1')
    try:
        (id, acct) = cur.fetchone()
    except:
        print('No unretrieved Twitter accounts found')
        continue
else:
    cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                (acct, ))
    try:
        id = cur.fetchone()[0]
    except:
        cur.execute('''INSERT OR IGNORE INTO People
                    (name, retrieved) VALUES (?, 0)''',
                    (acct, ))
        conn.commit()
        if cur.rowcount != 1:
            print('Error inserting account:', acct)
            continue
        id = cur.lastrowid

url = twurl.augment(
          TWITTER_URL, {'screen_name': acct, 'count': '100'})
print('Retrieving account', acct)
try:
    connection = urllib.request.urlopen(url, context=ctx)
except Exception as err:
    print('Failed to Retrieve', err)
    break

data = connection.read().decode()
headers = dict(connection.getheaders())

print('Remaining', headers['x-rate-limit-remaining'])

try:
    js = json.loads(data)
except:
    print('Unable to parse json')
    print(data)
    break

# Debugging
# print(json.dumps(js, indent=4))

if 'users' not in js:
    print('Incorrect JSON received')
    print(json.dumps(js, indent=4))
```

```
        continue

    cur.execute(
        'UPDATE People SET retrieved=1 WHERE name = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
                    (friend, ))
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute(
                '''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Error inserting account:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute(
            '''INSERT OR IGNORE INTO Follows (from_id, to_id)
            VALUES (?, ?)''', (id, friend_id))
    print('New accounts=', countnew, ' revisited=', countold)
    print('Remaining', headers['x-rate-limit-remaining'])
    conn.commit()
cur.close()
```

 # Code: https://tiny.one/py4de/code3/twfriends.py

Dieses Programm ist recht komplex, aber es veranschaulicht die Vorgehensweisen, die wir verwenden müssen, wenn wir Integer-Schlüssel zur Verknüpfung von Tabellen verwenden. Das grundlegende Vorgehen ist:

1. Erstellen der Tabellen mit Primärschlüsseln und Constraints.
2. Wenn wir einen logischen Schlüssel für eine Person haben (z. B. den Kontonamen) und den `id`-Wert für die Person benötigen, müssen wir je nachdem, ob die Person bereits in der Tabelle `People` enthalten ist oder nicht, entweder (1) die Person in der Tabelle `People` nachschlagen und den Wert `id` für die Person abrufen oder (2) die Person zur Tabelle `People` hinzufügen und den Wert `id` für die neu hinzugefügte Zeile abrufen.
3. Einfügen der Zeilen, die die `Follows`-Beziehung erfasst.

Wir werden jeden dieser Punkte nacheinander behandeln.

Constraints in Datenbanktabellen

Wenn wir unsere Tabellenstrukturen entwerfen, können wir dem Datenbanksystem mitteilen, dass wir möchten, dass es gewisse Regeln verfolgt. Diese Regeln helfen uns, Fehler zu vermeiden und falsche Daten in unsere Tabellen einzufügen.

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE,
    retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

Wenn wir unsere Tabellen erstellen, dann geben wir an, dass die Spalte `name` in der Tabelle `People` eindeutig (`UNIQUE`) sein muss. Wir geben außerdem an, dass die Kombination der beiden Zahlen in jeder Zeile der Tabelle `Follows` eindeutig sein muss. Diese Einschränkungen verhindern, dass wir Fehler machen, z. B. dieselbe Beziehung mehr als einmal hinzufügen.

Wir können diese Einschränkungen im folgenden Code ausnutzen:

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
    VALUES ( ?, 0)''', ( friend, ))
```

Wir fügen die Klausel `OR IGNORE` zu unserer `INSERT`-Anweisung hinzu, um anzugeben, dass das Datenbanksystem die `INSERT`-Anweisung ignorieren soll, wenn dieser eine Verletzung der Regel, dass `name` eindeutig sein muss, verursachen würde. Wir verwenden die Datenbankbeschränkung als Sicherheitsnetz, um zu gewährleisten, dass wir nicht versehentlich etwas Falsches tun.

In ähnlicher Weise stellt der folgende Code sicher, dass wir nicht genau die gleiche `Follows`-Beziehung zweimal hinzufügen.

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id))
```

Auch hier sagen wir der Datenbank, dass sie ein `INSERT` ignorieren soll, wenn die `UNIQUE`-Constraint verletzen werden würde, die wir für die `Follows`-Zeilen definiert haben.

Abrufen und Einfügen eines Datensatzes

Wenn wir den Benutzer nach einem Twitter-Konto fragen, müssen wir, sofern das Konto existiert, den `id`-Wert nachschlagen. Wenn das Konto noch nicht in der Tabelle `People` vorhanden ist, müssen wir den Datensatz einfügen und den Wert `id` aus der eingefügten Zeile holen.

Dies ist ein sehr häufiges Schema und wird in dem obigen Programm zweimal ausgeführt. Dieser Code zeigt, wie wir die `id` für das Konto eines Freundes nachschlagen, wenn wir einen `screen_name` aus einem `user`-Knoten im abgerufenen Twitter JSON extrahiert haben.

Da es im Laufe der Zeit immer wahrscheinlicher wird, dass das Konto bereits in der Datenbank vorhanden ist, prüfen wir zunächst mit einer SELECT-Anweisung, ob der Datensatz People existiert.

Wenn alles gut geht[2], rufen wir den Datensatz mit fetchone() ab und erhalten dann das erste (und einzige) Element des zurückgegebenen Tupels und speichern es in friend_id.

Wenn der SELECT fehlschlägt, schlägt fetchone()[0] fehl und die Kontrolle wird in den except Abschnitt übertragen.

```
    friend = u['screen_name']
    cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
        (friend, ))
    try:
        friend_id = cur.fetchone()[0]
        countold = countold + 1
    except:
        cur.execute(
            '''INSERT OR IGNORE INTO People (name, retrieved)
            VALUES ( ?, 0)''', ( friend, ))
        conn.commit()
        if cur.rowcount != 1:
            print('Error inserting account:',friend)
            continue
        friend_id = cur.lastrowid
        countnew = countnew + 1
```

Wenn wir im except-Teil landen, bedeutet das schlichtweg, dass die Zeile nicht gefunden wurde, also müssen wir die Zeile einfügen. Wir verwenden INSERT OR IGNORE nur, um Fehler zu vermeiden und rufen dann commit() auf, um die Aktualisierung der Datenbank zu erzwingen. Nachdem der Schreibvorgang abgeschlossen ist, können wir die cur.rowcount überprüfen, um zu sehen, wie viele Zeilen betroffen waren. Da wir versuchen, eine einzelne Zeile einzufügen, ist es ein Fehler, wenn die Anzahl der betroffenen Zeilen etwas anderes als 1 ist.

Wenn das INSERT erfolgreich war, können wir uns cur.lastrowid ansehen, um herauszufinden, welchen Wert die Datenbank der Spalte id in unserer neu erstellten Zeile zugewiesen hat.

Speichern der Freundschaftsbeziehung

Sobald wir den Schlüsselwert sowohl für den Twitter-Benutzer als auch für den Freund im JSON kennen, ist es einfach, die beiden Zahlen mit dem folgenden Code in die Tabelle Follows einzufügen:

```
cur.execute(
    'INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)',
    (id, friend_id))
```

[2]Wenn ein Satz schon mit „wenn alles gut geht“ beginnt, ist es häufig ratsam, den Code in try/except-Blöcke einzuschließen.

Es ist wichtig zu beachten, dass wir es der Datenbank überlassen, uns vor dem doppelten Einfügen einer Beziehung zu schützen, indem wir die Tabelle mit einer `UNIQUE`-Einschränkung erstellt und dann `OR IGNORE` zu unserer `INSERT`-Anweisung hinzugefügt haben.

Hier ist ein Beispiel für die Ausführung dieses Programms:

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

Wir haben mit dem Konto `drchuck` begonnen und dann das Programm automatisch die nächsten beiden Konten auswählen lassen, um sie abzurufen und zu unserer Datenbank hinzuzufügen.

Hier sind die ersten paar Zeilen der Tabellen `People` und `Follows`, nachdem der Programmlauf abgeschlossen ist:

```
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

Wir können die Felder `id`, `name` und `visited` in der Tabelle `People` sehen und in der Tabelle `Follows` die IDs der Konten, die in einer Freundschaftsbeziehung zueinander stehen. In der Tabelle `People` können wir sehen, dass die ersten drei Personen besucht und ihre Daten abgerufen wurden. Die Daten in der Tabelle `Follows` zeigen, dass `drchuck` (Benutzer 1) ein Freund aller Personen ist, die in den ersten fünf Zeilen angezeigt werden. Dies ergibt Sinn, denn die ersten Daten, die wir abgerufen und gespeichert haben, waren die Twitter-Freunde von `drchuck`. Wenn wir weitere Zeilen aus der Tabelle `Follows` ausgeben würden, würden wir auch die Freunde der Benutzer 2 und 3 sehen.

Drei Arten von Schlüsseln

Nachdem wir nun mit dem Aufbau eines Datenmodells begonnen haben, in dem wir unsere Daten in mehreren verknüpften Tabellen ablegen und die Zeilen in diesen Tabellen mit *Schlüsseln* verknüpfen, müssen wir uns nun mit der Terminologie rund um Schlüssel beschäftigen. Im Allgemeinen gibt es drei Arten von Schlüsseln, die in einem Datenbankmodell verwendet werden.

- Ein *logischer Schlüssel* ist ein Schlüssel, den die „reale Welt“ zum Nachschlagen eines Datensatzes verwenden könnte. In unserem Beispiel-Datenmodell ist das Feld `name` ein logischer Schlüssel. Es ist der Benutzername des Benutzers und wir schlagen die Zeile eines Benutzers in der Tat mehrmals im Programm über das Feld `name` nach. Wir werden oft feststellen, dass es sinnvoll ist, eine ‘UNIQUE’-Einschränkung zu einem logischen Schlüssel hinzuzufügen. Da der logische Schlüssel einen Datensatz identifiziert, ergibt es wenig Sinn, mehrere Zeilen mit demselben Wert in der Tabelle zuzulassen.

- Ein *Primärschlüssel* (englisch *Primary Key*, kurz *PK*) ist meist eine Nummer, die von der Datenbank automatisch vergeben wird. Er hat in der Regel keine Bedeutung außerhalb des Programms und wird nur verwendet, um Zeilen aus verschiedenen Tabellen miteinander zu verknüpfen. Wenn wir eine Zeile in einer Tabelle nachschlagen wollen, ist die Suche nach der Zeile über den Primärschlüssel normalerweise der schnellste Weg, die Zeile zu finden. Da es sich bei den Primärschlüsseln um ganze Zahlen handelt, nehmen sie sehr wenig Speicherplatz ein und können sehr schnell verglichen oder sortiert werden. In unserem Datenmodell ist das Feld `id` ein Beispiel für einen Primärschlüssel.

- Ein *Fremdschlüssel* (englisch *Foreign Key* , kurz *FK*) ist normalerweise eine Zahl, die auf den Primärschlüssel einer zugehörigen Zeile in einer *anderen* Tabelle verweist. Ein Beispiel für einen Fremdschlüssel in unserem Datenmodell ist die Spalte `from_id`.

Wir verwenden die Namenskonvention, den Primärschlüssel immer `id` zu nennen und das Suffix `_id` an jeden Spaltennamen anzuhängen, der ein Fremdschlüssel ist.

Abrufen von Daten mit `JOIN`

Da wir nun die Regeln der Datenbanknormalisierung befolgt haben und die Daten in zwei Tabellen aufgeteilt sind, die über Primär- und Fremdschlüssel miteinander verknüpft sind, müssen wir in der Lage sein, einen `SELECT` zu implementieren, der die Daten in den Tabellen wieder zusammensetzt.

SQL verwendet die `JOIN`-Klausel, um diese Tabellen wieder zu verbinden. In der `JOIN`-Klausel geben wir die Felder an, die zum Wiederverbinden der Zeilen zwischen den Tabellen verwendet werden.

Im Folgenden sehen wir ein Beispiel für einen `SELECT` mit einer `JOIN`-Klausel:

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

Die JOIN-Klausel gibt an, dass die ausgewählten Felder sowohl die Follows- als auch die People-Tabelle betreffen. Die ON-Klausel gibt an, wie die beiden Tabellen verbunden werden sollen: Es wird eine passende Zeile aus Follows entnommen und an die Zeile aus People angefügt, bei der das Feld from_id in Follows gleich dem Wert id in der Tabelle People ist.

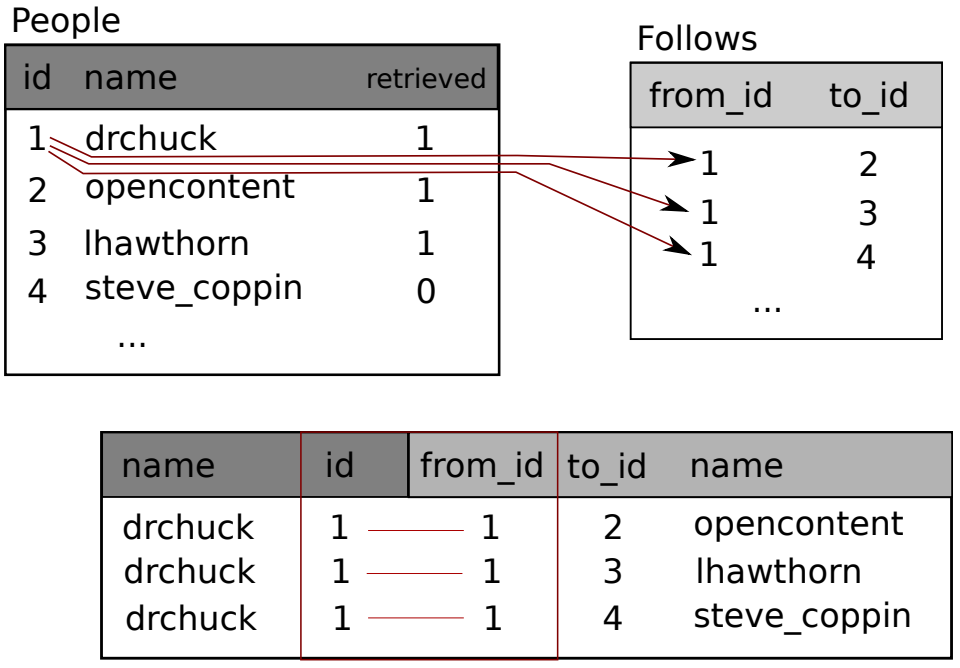

Abbildung 15.5: Datenabfrage über mehrere Tabellen mit JOIN

Das Ergebnis des JOIN sind Tabellenzeilen, die sowohl die Felder aus People als auch die passenden Felder aus Follows enthalten. Wenn es mehr als eine Übereinstimmung zwischen dem Feld id aus People und der from_id aus People gibt, dann erzeugt JOIN eine Zeile für *jedes* der übereinstimmenden Zeilenpaare und dupliziert Daten bei Bedarf.

Der folgende Code zeigt die Daten, die wir in der Datenbank erhalten, nachdem das obige Spider-Programm mehrmals ausgeführt wurde.

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print('People:')
```

```
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.execute('SELECT * FROM Follows')
count = 0
print('Follows:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.execute('''SELECT * FROM Follows JOIN People
            ON Follows.to_id = People.id
            WHERE Follows.from_id = 2''')
count = 0
print('Connections for id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'rows.')

cur.close()

# Code: https://tiny.one/py4de/code3/twjoin.py
```

In diesem Programm werden zuerst die `People`- und `Follows`-Tabellen und dann eine Teilmenge der Daten in den miteinander verbundenen Tabellen ausgegeben.

Hier ist die Ausgabe des Programms:

```
python twjoin.py
People:
(1, 'drchuck', 1)
(2, 'opencontent', 1)
(3, 'lhawthorn', 1)
(4, 'steve_coppin', 0)
(5, 'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, 'drchuck', 1)
(2, 28, 28, 'cnxorg', 0)
(2, 30, 30, 'kthanos', 0)
```

```
(2, 102, 102, 'SomethingGirl', 0)
(2, 103, 103, 'ja_Pac', 0)
20 rows.
```

Wir sehen die Spalten aus den Tabellen `People` und `Follows`. Der letzte Satz von Zeilen ist das Ergebnis des `SELECT` mit der `JOIN`-Klausel.

Im letzten `SELECT` suchen wir nach Konten, die mit `opencontent` befreundet sind (d. h. `People.id=2`).

In jeder der Zeilen des letzten `SELECT` stammen die ersten beiden Spalten aus der Tabelle `Follows`, gefolgt von den Spalten drei bis fünf aus der Tabelle `People`. Wir können auch sehen, dass die zweite Spalte (`Follows.to_id`) mit der dritten Spalte (`People.id`) in jeder der verknüpften Zeilen übereinstimmt.

Zusammenfassung

Datenbanktabellen anzulegen, zu verwalten und Datensätze einzufügen und abzufragen, ist mit einem gewissen programmiertechnischen Aufwand verbunden. Die Verwendung von Python-Datenstrukturen wie zum Beispiel Dictionarys ist dagegen recht simpel. Datenbanken verfügen jedoch über Eigenschaften und Funktionalitäten, wegen der man sie in bestimmten Szenarien präferiert.

Die Situationen, in denen eine Datenbank sehr nützlich sein kann, sind: (1) Wenn die Anwendung viele kleine, zufällige Aktualisierungen innerhalb eines großen Datensatzes vornehmen muss, (2) wenn die Daten so groß sind, dass sie nicht in ein Dictionary passen und wir wiederholt Informationen nachschlagen müssen, oder (3) wenn wir einen lange laufenden Prozess haben, den wir stoppen und neu starten möchten und die Daten von einem Lauf zum nächsten beibehalten wollen.

Wir können eine einfache Datenbank mit einer einzigen Tabelle erstellen, um viele Anwendungsanforderungen zu erfüllen, aber die meisten Probleme werden mehrere Tabellen und Beziehungen zwischen diesen erfordern. Wenn wir damit beginnen, Verknüpfungen zwischen Tabellen herzustellen, ist es wichtig, ein durchdachtes Design zu entwerfen und die Regeln der Datenbanknormalisierung zu befolgen, um die Fähigkeiten der Datenbank optimal nutzen zu können. Da die Hauptmotivation für die Verwendung einer Datenbank darin besteht, mit einer großen Menge von Daten zu arbeiten, ist es wichtig, Daten effizient zu modellieren, um damit auch die Performanz unserer Programme zu optimieren.

Debugging

Ein häufiges Vorgehen, wenn wir ein Python-Programm zur Verbindung mit einer SQLite-Datenbank entwickeln, wird sein, dass wir ein Python-Programm ausführen und die Ergebnisse mit dem Datenbankbrowser für SQLite überprüfen. Mit dem Browser können wir schnell überprüfen, ob unser Programm richtig funktioniert.

Wir müssen vorsichtig sein, weil SQLite darauf achtet, dass nicht zwei Programme gleichzeitig dieselben Daten ändern. Wenn wir z. B. eine Datenbank im Browser

öffnen und eine Änderung an der Datenbank vornehmen und noch nicht die Schaltfläche „Speichern“ im Browser betätigt haben, sperrt der Browser die Datenbankdatei und verhindert, dass ein anderes Programm auf die Datei zugreift. Insbesondere unser Python-Programm kann nicht auf die Datei zugreifen, wenn sie gesperrt ist.

Eine Lösung ist also, sicherzustellen, dass wir entweder den Datenbankbrowser schließen oder das Menü *Datei* verwenden, um die Datenbank im Browser zu schließen, bevor wir versuchen, von Python aus auf die Datenbank zuzugreifen.

Glossar

Attribut Einer der Werte innerhalb eines Tupels. Häufiger als Spalte oder Feld bezeichnet.

Constraint Eine Anweisung an die Datenbank, eine Regel für ein Feld oder eine Zeile in einer Tabelle zu erzwingen. Eine übliche Einschränkung ist, darauf zu bestehen, dass es in einem bestimmten Feld keine doppelten Werte geben darf, also alle Werte eindeutig sein müssen.

Cursor Mit einem Cursor können wir SQL-Befehle in einer Datenbank ausführen und Daten aus der Datenbank abrufen. Ein Cursor ist vergleichbar mit einem Socket oder einem Dateihandler für Netzwerkverbindungen bzw. Dateien.

Datenbankenbrowser Eine Software, die es ermöglicht, eine direkte Verbindung zu einer Datenbank herzustellen und die Datenbank direkt zu manipulieren, ohne dafür ein gesondertes Programm schreiben zu müssen.

Fremdschlüssel Ein numerischer Schlüssel, der auf den Primärschlüssel einer Zeile in einer anderen Tabelle verweist. Fremdschlüssel stellen Beziehungen zwischen Zeilen her, die in verschiedenen Tabellen gespeichert sind.

Index Zusätzliche Daten, die vom Datenbanksystem verwaltet und in eine Tabelle einfügt werden, um Abfragen sehr schnell zu machen.

Logischer Schlüssel Ein Schlüssel, den die „Außenwelt“ verwendet, um eine bestimmte Zeile nachzuschlagen. In einer Tabelle mit Benutzerkonten könnte z. B. die E-Mail-Adresse einer Person ein guter Kandidat als logischer Schlüssel für die Daten des Benutzers sein.

Normalisierung Entwurf eines Datenmodells mit dem Ziel, dass keine Daten repliziert werden müssen. Wir speichern jedes Datenelement an einer Stelle in der Datenbank und referenzieren es an anderer Stelle über einen Fremdschlüssel.

Primärschlüssel Ein numerischer Schlüssel, der jeder Zeile zugewiesen wird und dazu dient, von einer anderen Tabelle aus auf eine Zeile in einer Tabelle zu verweisen. Oft ist die Datenbank so konfiguriert, dass Primärschlüssel automatisch zugewiesen werden, wenn Zeilen eingefügt werden.

Relation Ein Datensatz (eine Zeile) in einer Datenbanktabelle.

Tupel Ein einzelner Eintrag in einer Datenbanktabelle, der eine Menge von Attributen darstellt. Er wird typischerweise als Zeile bezeichnet.

Kapitel 16

Visualisierung von Daten

Bisher haben wir Python erlernt und uns damit beschäftigt, wie man Python, das Internet und Datenbanken verwendet, um Daten zu verarbeiten.

In diesem Kapitel werfen wir einen Blick auf drei vollständige Anwendungen, die all diese Dinge zusammenbringen, um Daten zu verwalten und zu visualisieren. Wir können diese Anwendungen als Beispielcode verwenden, um mit dem Lösen eines realen Problems zu beginnen.

Jede der Anwendungen liegt als ZIP-Datei vor, die wir herunterladen, entpacken und ausführen können.

Erstellen einer OpenStreetMap aus Geodaten

In diesem Projekt verwenden wir die OpenStreetMap-API, um die Standorte einiger vom Benutzer eingegebenen Universitätsnamen auf einer aktuellen OpenStreetMap zu platzieren.

Um loszulegen, laden wir die Anwendung hier herunter:

www.py4e.com/code3/opengeo.zip

Das erste zu lösende Problem ist, dass die Geocoding-APIs auf eine bestimmte Anzahl von Anfragen pro Tag begrenzt sind. Wenn wir viele Daten haben, müssen wir den Lookup-Prozess möglicherweise mehrmals anhalten und neu starten. Also teilen wir das Problem in zwei Phasen auf.

In der ersten Phase nehmen wir die Daten (Universitätsnamen) in der Datei `where.data` und lesen sie zeilenweise ein, rufen die Geoinformationen von Google ab und speichern sie in einer Datenbank `geodata.sqlite`. Bevor wir die Geocoding-API für jeden vom Benutzer eingegebenen Ort verwenden, prüfen wir zunächst, ob wir die Daten für diese bestimmte Eingabezeile bereits haben. Die Datenbank fungiert als lokaler *Zwischenspeicher* (*Cache*) unserer Geodaten, um sicherzustellen, dass wir Google nie zweimal nach denselben Daten fragen.

Wir können den Prozess jederzeit neu starten, indem wir die Datei `geodata.sqlite` löschen.

Abbildung 16.1: OpenStreetMap

Nun führen wir das Programm `geoload.py` aus. Das Programm liest die Eingabezeilen in `where.data` und prüft für jede Zeile, ob sie bereits in der Datenbank vorhanden ist. Wenn wir die Daten für den Ort nicht haben, wird es die Geocoding-API aufrufen, um die Daten abzurufen und in der Datenbank zu speichern.

Hier ist ein Beispiellauf, nachdem bereits einige Daten in der Datenbank vorhanden sind:

```
Found in database AGH University of Science and Technology

Found in database Academy of Fine Arts Warsaw Poland

Found in database American University in Cairo

Found in database Arizona State University

Found in database Athens Information Technology

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=BITS+Pilani
Retrieved 794 characters {"type":"FeatureColl

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=Babcock+University
Retrieved 760 characters {"type":"FeatureColl

Retrieving https://py4e-data.dr-chuck.net/
   opengeo?q=Banaras+Hindu+University
Retrieved 866 characters {"type":"FeatureColl
...
```

Die ersten fünf Orte sind bereits in der Datenbank und werden daher übersprungen. Das Programm scannt bis zu dem Punkt, an dem es neue Orte findet und beginnt, diese abzurufen.

Das Programm `geoload.py` kann jederzeit gestoppt werden und es gibt einen Zähler, mit dem wir die Anzahl der Aufrufe der Geocoding-API für jeden Lauf begrenzen können. In Anbetracht der Tatsache, dass die Datei `where.data` nur ein paar hundert Datenelemente enthält, sollten wir nicht an das tägliche Google-Zugriffslimit stoßen, aber wenn wir mehr Daten hätten, könnte es mehrere Durchläufe über mehrere Tage hinweg erfordern, bis unsere Datenbank alle Geodaten für unsere Eingabe enthält.

Sobald wir einige Daten in `geodata.sqlite` geladen haben, können wir die Daten mit dem Programm `geodump.py` visualisieren. Dieses Programm liest die Datenbank und schreibt die Datei `where.js` mit dem Standort, dem Breitengrad und dem Längengrad in Form von ausführbarem JavaScript-Code.

Ein Durchlauf des Programms `geodump.py` sieht folgendermaßen aus:

```
AGH University of Science and Technology, Czarnowiejska,
Czarna Wieś, Krowodrza, Kraków, Lesser Poland
Voivodeship, 31-126, Poland 50.0657 19.91895

Academy of Fine Arts, Krakowskie Przedmieście,
Northern Śródmieście, Śródmieście, Warsaw, Masovian
Voivodeship, 00-046, Poland 52.239 21.0155
...
260 lines were written to where.js
Open the where.html file in a web browser to view the data.
```

Die Datei `where.html` besteht aus HTML und JavaScript zur Visualisierung einer Google-Karte. Sie liest die aktuellsten Daten aus `where.js`, um die zu visualisierenden Daten zu erhalten. Hier ist das Format der `where.js` Datei:

```
myData = [
[50.0657,19.91895,
'AGH University of Science and Technology, Czarnowiejska,
Czarna Wieś, Krowodrza, Kraków, Lesser Poland
Voivodeship, 31-126, Poland '],
[52.239,21.0155,
'Academy of Fine Arts, Krakowskie Przedmieściee,
Śródmieście Północne, Śródmieście, Warsaw,
Masovian Voivodeship, 00-046, Poland'],
   ...
];
```

Dies ist eine JavaScript-Variable, die eine Liste von Listen enthält. Die Syntax für JavaScript-Listenkonstanten ist der von Python sehr ähnlich, daher sollte uns die Syntax vertraut sein.

Nun muss die `where.html` in einem Browser geöffnet werden, um die Standorte zu sehen. Wir können den Mauszeiger über jeden Pin bewegen, um den Standort

zu finden, den die Geocoding-API für die vom Benutzer eingegebene Eingabe zurückgegeben hat. Wenn wir beim Öffnen der Datei `where.html` keine Daten vorfinden, sollten wir die JavaScript- oder Entwicklerkonsole für unseren Browser überprüfen.

Visualisierung von Netzwerken

In dieser Anwendung werden wir einige der Funktionen einer Suchmaschine nachbilden. Wir werden zunächst eine kleine Teilmenge des Webs durchforsten und eine eigene vereinfachte Version des Page-Ranking-Algorithmus von Google ausführen, um festzustellen, welche Seiten am stärksten miteinander verbunden sind, und dann den Page-Rank und die Verbindung in einem kleinen Teil des Webs visualisieren. Wir werden die D3-JavaScript-Visualisierungsbibliothek http://d3js.org/ verwenden, um die Visualisierung zu erzeugen.

Wir können diese Anwendung hier herunterladen:

www.py4e.com/code3/pagerank.zip

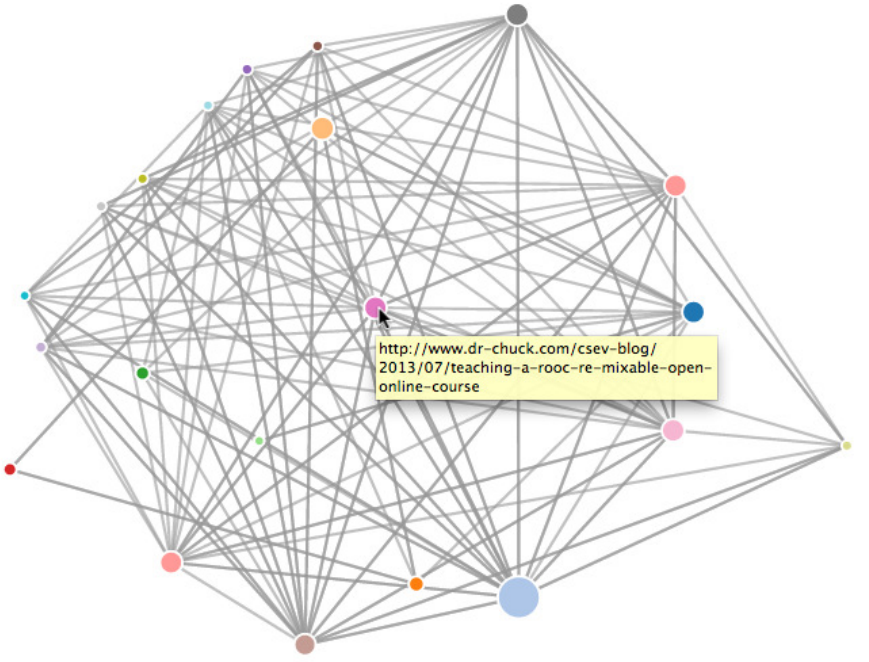

Abbildung 16.2: Page-Ranking

Das erste Programm (`spider.py`) durchsucht eine Website und holt eine Reihe von Seiten in die Datenbank (`spider.sqlite`), wobei die Links zwischen den Seiten aufgezeichnet werden. Wir können den Prozess jederzeit neu starten, indem wir die Datei `spider.sqlite` entfernen und `spider.py` erneut ausführen.

```
Enter web url or enter: http://www.dr-chuck.com/
```

```
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

In diesem Beispieldurchlauf haben wir das Programm angewiesen, eine Website zu durchforsten und zwei Seiten abzurufen. Wenn wir das Programm neu starten und ihm sagen, dass es weitere Seiten besuchen soll, wird es keine Seiten, die sich bereits in der Datenbank befinden, erneut besuchten. Beim Neustart wechselt es zu einer zufälligen, nicht besuchten Seite und beginnt dort. Somit verhält sich jeder erneute Lauf von `spider.py` additiv.

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

Wir können mehrere Startpunkte in derselben Datenbank haben – im Programm werden diese als `webs` bezeichnet. Der Spider wählt nach dem Zufallsprinzip aus allen nicht besuchten Links über alle `webs` hinweg die nächste zu durchsuchende Seite aus.

Wenn wir den Inhalt der Datei `spider.sqlite` ausgeben möchten, können wir `spdump.py` wie folgt ausführen:

```
(5, None, 1.0, 3, 'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

Dies zeigt die Anzahl der eingehenden Links, den alten Page-Rank, den neuen Page-Rank, die ID der Seite und die URL der Seite. Das Programm `spdump.py` zeigt nur Seiten an, die mindestens einen eingehenden Link auf sie haben.

Sobald wir ein paar Websites in der Datenbank haben, können wir mit dem Programm `sprank.py` das Page-Ranking in Gang setzen. Wir teilen ihm einfach mit, wie viele Iterationen es ausführen soll.

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

Wir können die Datenbank erneut ausgeben, um zu sehen, dass der Page-Rank aktualisiert wurde:

```
(5, 1.0, 0.985, 3, 'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, 'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, 'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, 'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

Wir können `sprank.py` so oft ausführen wie wir möchten und es wird das Page-Ranking jedes Mal verfeinern. Wir können `sprank.py` sogar ein paar Mal laufen lassen und dann ein paar weitere Seiten mit `spider.py` durchforsten und dann `sprank.py` laufen lassen, um die Page-Ranking-Werte neu zu berechnen. Eine Suchmaschine lässt normalerweise sowohl das Crawling- als auch das Ranking-Programm ständig laufen.

Wenn wir die Page-Ranking-Berechnungen neu starten wollen, ohne die Webseiten neu zu laden, können wir `spreset.py` verwenden und dann `sprank.py` neu starten.

```
How many iterations:50
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
42 0.000109076928206
43 9.91987599002e-05
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05
[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

Für jede Iteration des Page-Ranking-Algorithmus wird die durchschnittliche Änderung des Page-Ranks pro Seite ausgegeben. Das Netzwerk ist anfangs recht unausgewogen, sodass sich die einzelnen Page-Rank-Werte zwischen den Iterationen stark ändern. Aber in ein paar kurzen Iterationen konvergiert der Page-Rank. Wir müssen `sprank.py` lange genug laufen lassen, damit die Page-Ranking-Werte konvergieren.

Wenn wir die aktuellen Top-Seiten in Bezug auf den Page-Rank visualisieren möchten, führen wir `spjson.py` aus, um die Datenbank zu lesen und die Daten für die am stärksten verlinkten Seiten in ein JSON-Format zu schreiben, das in einem Webbrowser angezeigt werden kann.

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

Wir können diese Daten betrachten, indem wir die Datei `force.html` in unserem Webbrowser öffnen. Dies zeigt ein automatisches Layout der Knoten und Links. Wir können jeden Knoten anklicken und ziehen und wir können auch auf einen Knoten doppelklicken, um die URL zu sehen, die durch den Knoten dargestellt wird.

Wenn wir die anderen Dienstprogramme erneut ausführen, führen wir `spjson.py` ebenfalls erneut aus. Dann drücken wir im Browser auf Aktualisieren, um die neuen Daten aus `spider.json` zu erhalten.

Visualisierung von Maildaten

Bis zu diesem Punkt im Buch sind wir mit unseren Datendateien `mbox-short.txt` und `mbox.txt` ziemlich vertraut geworden. Jetzt ist es an der Zeit, unsere Analyse der E-Mail-Daten auf die nächste Stufe zu bringen.

In der realen Welt müssen wir manchmal Mail-Daten von Servern abrufen. Das kann einige Zeit in Anspruch nehmen und die Daten können inkonsistent und fehlerhaft sein und eine Menge Bereinigungen oder Anpassungen erfordern. In diesem Abschnitt arbeiten wir mit einer Anwendung, die bisher die komplexeste ist, und rufen fast ein Gigabyte an Daten ab und visualisieren diese.

Abbildung 16.3: Eine Wortwolke aus der Sakai-Entwicklerliste

Wir können diese Anwendung hier herunterladen:

https://www.py4e.com/code3/gmane.zip

Wir werden Daten von einem freien E-Mail-Listen-Archivierungsdienst namens http://www.gmane.org verwenden. Dieser Dienst ist sehr beliebt bei Open-Source-Projekten, weil er ein schönes durchsuchbares Archiv der E-Mail-Aktivitäten bietet. Sie haben auch eine sehr liberale Richtlinie bezüglich des Zugriffs auf Daten durch ihre API. Die API hat keine Zugriffsbeschränkungen, wir sollten jedoch darauf achten, den Dienst nicht zu überlasten und nur die Daten zu verwenden, die wir benötigen. Die allgemeinen Geschäftsbedingungen von *gmane* sind auf der folgenden Seite zu finden:

http://www.gmane.org/export.php

Hinweis: Es ist sehr wichtig, dass wir die Daten von **gmane.org** verantwortungsvoll nutzen, indem wir unseren Zugriff auf deren Dienste verzögern und lang laufende Aufträge über einen längeren Zeitraum verteilen. So missbrauchen wir diesen kostenlosen Dienst nicht und ruinieren ihn nicht für andere Entwickler.

Als die Sakai-E-Mail-Daten mit dieser Software gesichtet wurden, produzierte dies fast ein Gigabyte an Daten und erforderte eine Reihe von Durchläufen an mehreren Tagen. Die Datei `README.txt` im obigen ZIP-Archiv enthält möglicherweise Anweisungen, wie wir eine vorab erstellte Kopie der Datei `content.sqlite` für einen Großteil des Sakai-E-Mail-Korpus herunterladen können, damit wir nicht fünf Tage lang „spidern“ müssen, bevor wir unsere Programme endlich auszuführen können. Wenn wir den vorbereiteten Inhalt herunterladen, sollten wir trotzdem den Spider-Prozess ausführen, um neuere Nachrichten einzuholen.

Der erste Schritt besteht darin, das gmane-Repository zu durchforsten. Die Basis-URL ist in `gmane.py` fest codiert und mit der Sakai-Entwicklerliste verknüpft. Wir können ein anderes Repository spidern, indem wir diese Basis-URL ändern. Es ist sicherzustellen, dass die Datei `content.sqlite` gelöscht wird, wenn wir die Basis-URL ändern.

Das Programm Datei `gmane.py` arbeitet als verantwortlicher Caching-Spider, indem es langsam läuft und nur eine Mail-Nachricht pro Sekunde abruft, um nicht von gmane gedrosselt zu werden. Das Programm speichert alle Daten in einer Datenbank und kann so oft wie nötig unterbrochen und neu gestartet werden. Es kann viele Stunden dauern, bis alle Daten heruntergeladen sind. Wir müssen also möglicherweise mehrmals neu starten.

Hier ist ein Lauf von `gmane.py`, der die letzten fünf Nachrichten der Sakai-Entwicklerliste abruft:

```
How many messages:10
http://download.gmane.org/gmane.comp.cms.sakai.devel/51410/51411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51411/51412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51412/51413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51413/51414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51414/51415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51415/51416 0
```

```
Does not start with From
```

Das Programm durchsucht die Datei `content.sqlite` von der allerersten bis zur ersten noch nicht verarbeiteten Nachricht und beginnt das Spidering bei dieser Nachricht. Es fährt mit dem Spidering fort, bis es die gewünschte Anzahl von Nachrichten durchforstet hat oder es eine Seite erreicht, die nicht als ordnungsgemäß formatierte Nachricht erscheint.

Manchmal fehlt auf **gmane.org** eine Nachricht. Vielleicht können Administratoren Nachrichten löschen oder vielleicht gehen sie verloren. Wenn unser Spider anhält und es scheint, dass er auf eine fehlende Nachricht gestoßen ist, gehen wir in den SQLite Manager und fügen eine Zeile mit der fehlenden ID hinzu, wobei wir alle anderen Felder leer lassen und `gmane.py` neu starten. Dies wird den Spider-Prozess wieder freigeben und ihm erlauben, fortzufahren. Diese leeren Nachrichten werden in der nächsten Phase des Prozesses ignoriert.

Eine schöne Sache ist, dass wir, sobald wir alle Nachrichten gespidert haben und sie in `content.sqlite` vorliegen, `gmane.py` erneut ausführen können, um neue Nachrichten zu erhalten, sobald sie in die Liste übertragen werden.

Die `content.sqlite`-Daten sind unbearbeitet, mit einem ineffizienten Datenmodell gespeichert, und nicht komprimiert. Dies ist beabsichtigt, da es Ihnen ermöglicht, `content.sqlite` im SQLite-Manager zu betrachten, um Probleme mit dem Spider-Prozess zu beheben. Es wäre eine schlechte Idee, irgendwelche Abfragen über diese Datenbank laufen zu lassen, da sie ziemlich langsam wären.

Der zweite Vorgang ist die Ausführung des Programms `gmodel.py`. Dieses Programm liest die Rohdaten aus `content.sqlite` und erzeugt eine bereinigte und gut modellierte Version der Daten in der Datei `index.sqlite`. Diese Datei wird viel kleiner sein (oft 10 Mal kleiner) als `content.sqlite`, weil auch der Header- und Body-Text komprimiert wird.

Jedes Mal, wenn `gmodel.py` läuft, löscht es die Datei `index.sqlite` und baut sie neu auf, sodass wir die Parameter anpassen und die Mapping-Tabellen in `content.sqlite` bearbeiten können, um den Datenbereinigungsprozess zu optimieren. Dies ist ein Beispiellauf von `gmodel.py`. Jedes Mal, wenn 250 Mail-Nachrichten verarbeitet werden, wird eine Zeile ausgegeben, damit wir den Fortschritt sehen können, denn dieses Programm kann eine Weile laufen und fast ein Gigabyte an Mail-Daten verarbeiten.

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

Das Programm `gmodel.py` übernimmt eine Reihe von Datenbereinigungsaufgaben.

Domainnamen werden auf zwei Domainlevel für `.com`, `.org`, `.edu` und `.net` gekürzt. Andere Domain-Namen werden auf drei Level gekürzt. So wird `si.umich.edu` zu `umich.edu` und `caret.cam.ac.uk` zu `cam.ac.uk`. E-Mail-Adressen werden ebenfalls in Kleinschreibung überführt, und Adressen wie

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org
```

werden in die echte Adresse umgewandelt, wenn es an anderer Stelle im Nachrichtenkorpus eine passende echte E-Mail-Adresse gibt.

In der Datenbank `mapping.sqlite` gibt es zwei Tabellen, die es Ihnen ermöglichen, sowohl Domänennamen als auch einzelne E-Mail-Adressen zuzuordnen, die sich während der Lebensdauer der E-Mail-Liste ändern. Zum Beispiel hat Steve Githens die folgenden E-Mail-Adressen verwendet, als er während der Lebensdauer der Sakai-Entwicklerliste den Arbeitsplatz wechselte:

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

Wir können zwei Einträge in die Mapping-Tabelle in `mapping.sqlite` hinzufügen, sodass `gmodel.py` alle drei auf eine Adresse abbildet:

```
s-githens@northwestern.edu ->  swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

Wir können auch ähnliche Einträge in der DNSMapping-Tabelle vornehmen, wenn es mehrere DNS-Namen gibt, die wir einem einzigen DNS zuordnen möchten. Die folgende Zuordnung wurde zu den Sakai-Daten hinzugefügt:

```
iupui.edu -> indiana.edu
```

So werden alle Konten der verschiedenen Standorte der Indiana University zusammen verfolgt.

Wir können `gmodel.py` immer wieder ausführen, während wir uns die Daten ansehen, und Mappings hinzufügen, um die Daten immer mehr zu bereinigen. Wenn wir fertig sind, haben wir eine ordentlich indizierte Version der E-Mail in `index.sqlite`. Dies ist die Datenbankdatei, die für die Datenanalyse verwendet wird. Mit ihr wird die Datenanalyse wirklich schnell sein.

Die erste und einfachste Datenanalyse besteht darin, festzustellen, wer die meisten Mails verschickt hat und welche Organisation die meisten Mails verschickt hat. Dies wird mit `gbasic.py` durchgeführt:

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584

Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184
```

```
Top 5 Email list organizations
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

Es ist zu beachten, wie viel schneller `gbasic.py` im Vergleich zu `gmane.py` oder sogar `gmodel.py` läuft. Sie arbeiten alle mit den gleichen Daten, aber `gbasic.py` verwendet die komprimierten und normalisierten Daten in `index.sqlite`. Wenn wir viele Daten zu verwalten haben, kann ein mehrstufiger Prozess wie in dieser Anwendung etwas länger dauern, aber es kann viel Zeit gespart werden, wenn Daten untersucht und visualisiert werden.

Eine einfache Visualisierung der Worthäufigkeit in den Betreffzeilen können wir mit `gword.py` erzeugen:

```
Range of counts: 33229 129
Output written to gword.js
```

Dies erzeugt die Datei `gword.js`, die wir mit `gword.htm` visualisieren können, um eine Wortwolke ähnlich der am Anfang dieses Abschnitts zu erzeugen.

Eine zweite Visualisierung wird von `gline.py` erzeugt. Sie berechnet die E-Mail-Aktivität von Organisationen im Laufe der Zeit.

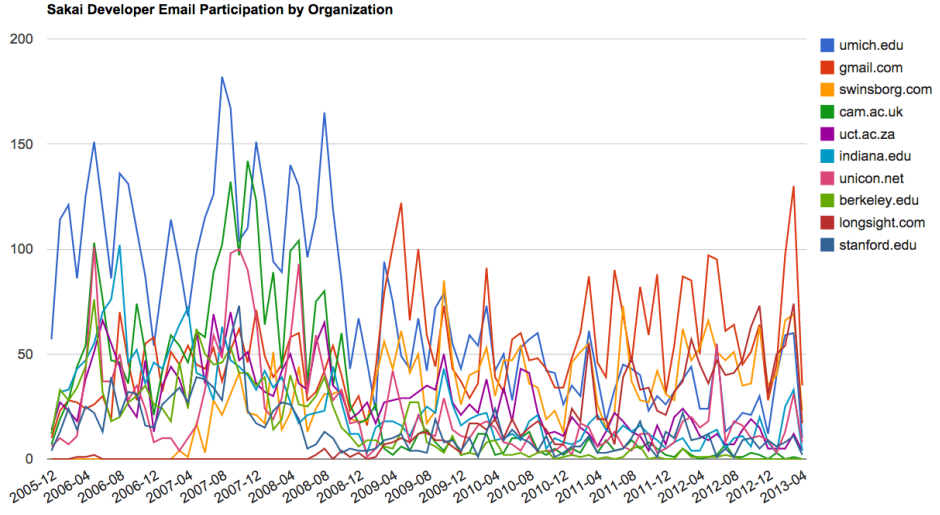

Abbildung 16.4: Sakai-Mail-Aktivität pro Organisation

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

Die Ausgabe wird in `gline.js` geschrieben, die mit `gline.htm` visualisiert wird.

Dies ist eine relativ komplexe und ausgefeilte Anwendung und sie verfügt über Funktionen zum Abrufen, Bereinigen und Visualisieren von Daten.

Anhang A

Mitwirkende

Mitwirkende an „Python for Everybody“

Andrzej Wójtowicz, Elliott Hauser, Stephen Catto, Sue Blumenberg, Tamara Brunnock, Mihaela Mack, Chris Kolosiwsky, Dustin Farley, Jens Leerssen, Naveen KT, Mirza Ibrahimovic, Naveen (@togarnk), Zhou Fangyi, Alistair Walsh, Erica Brody, Jih-Sheng Huang, Louis Luangkesorn, and Michael Fudge

Einzelheiten zu den Beiträgen finden Sie unter:

https://github.com/csev/py4e/graphs/contributors

Mitwirkende an „Python for Informatics“

Bruce Shields for copy editing early drafts, Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, Hedemarrie Dussan,

Vorwort von „Think Python“

Die seltsame Geschichte von „Think Python“

(Allen B. Downey)

Im Januar 1999 bereitete ich mich darauf vor, einen Einführungskurs in die Programmierung mit Java zu geben. Ich hatte ihn bereits dreimal unterrichtet und

war frustriert. Die Durchfallquote in der Klasse war zu hoch, und selbst bei den Schülern, die erfolgreich waren, war das Leistungsniveau insgesamt zu niedrig.

Eines der Probleme, die ich sah, waren die Bücher. Sie waren zu umfangreich, enthielten zu viele unnötige Details über Java und nicht genug Anleitungen von guter Qualität, wie man programmiert. Außerdem hatten die Bücher folgenden Fallstrick: Sie fingen gemächlich an, gingen allmählich voran, und dann plötzlich, etwa ab Kapitel 5, zog das Tempo enorm an. Die Studenten lernten zu schnell zu viel neues Material, und ich musste den Rest des Semesters damit verbringen, die Scherben aufzusammeln.

Zwei Wochen vor dem ersten Unterrichtstag beschloss ich, mein eigenes Buch zu schreiben. Meine Ziele waren:

- Es kurz halten. Es ist besser für die Schüler, 10 Seiten zu lesen, als 50 Seiten nicht zu lesen.
- Vorsichtig mit dem Vokabular sein. Ich habe versucht, den Jargon einfach zu halten und jeden Begriff bei der ersten Verwendung zu definieren.
- Schrittweises Vorgehen. Um zu vermeiden, dass die Leser abgehängt werden, habe ich die schwierigsten Themen in eine Reihe kleiner Schritte aufgeteilt.
- Konzentrieren auf die Programmierung, nicht auf die Programmiersprache. Ich habe die kleinste nützliche Teilmenge von Java einbezogen und den Rest weggelassen.

Ich brauchte einen Titel und entschied mich aus einer Laune heraus für *How to Think Like a Computer Scientist*.

Meine erste Version war holprig, aber sie funktionierte. Die Schüler haben die Lektüre gelesen und genug verstanden, sodass ich die Unterrichtszeit auf die schwierigen und interessanten Themen verwenden und (was am wichtigsten ist) die Schüler üben lassen konnte.

Ich habe das Buch unter der GNU Free Documentation License veröffentlicht, die es den Benutzern erlaubt, das Buch zu kopieren, zu verändern und zu verbreiten.

Was dann passierte, ist der coole Teil. Jeff Elkner, ein Highschool-Lehrer in Virginia, nahm sich meines Buches an und übersetzte es in Python. Er schickte mir eine Kopie seiner Übersetzung, und ich machte die ungewöhnliche Erfahrung, Python zu lernen, indem ich mein eigenes Buch las.

Jeff und ich überarbeiteten das Buch, fügten eine Fallstudie von Chris Meyers ein und veröffentlichten im Jahr 2001 *How to Think Like a Computer Scientist: Learning with Python*, ebenfalls unter der GNU Free Documentation License. Als Green Tea Press veröffentlichte ich das Buch und begann mit dem Verkauf von gedruckten Exemplaren über Amazon.com und College-Buchläden. Andere Bücher von Green Tea Press sind unter greenteapress.com erhältlich.

Im Jahr 2003 begann ich am Olin College zu unterrichten, und ich durfte zum ersten Mal Python unterrichten. Der Unterschied zu Java war frappierend. Die Studenten hatten weniger Mühe, lernten mehr, arbeiteten an interessanteren Projekten und hatten allgemein viel mehr Spaß.

In den letzten fünf Jahren habe ich das Buch weiterentwickelt, Fehler korrigiert, einige der Beispiele verbessert und Material hinzugefügt, insbesondere Übungen. Im Jahr 2008 begann ich mit der Arbeit an einer umfassenden Überarbeitung – zur gleichen Zeit wurde ich von einem Redakteur der Cambridge University Press kontaktiert, der an der Veröffentlichung der nächsten Ausgabe interessiert war. Gutes Timing!

Ich wünsche Ihnen viel Spaß bei der Arbeit mit diesem Buch und hoffe, dass es Ihnen hilft, zu programmieren und zumindest ein bisschen wie ein Informatiker zu denken.

Danksagungen für „Think Python“

(Allen B. Downey)

Zunächst und vor allem danke ich Jeff Elkner, der mein Java-Buch in Python übersetzt hat, was dieses Projekt ins Rollen brachte und mich in die Sprache einführte, die sich als meine Lieblingssprache herausgestellt hat.

Ich danke auch Chris Meyers, der mehrere Abschnitte zu *How to Think Like a Computer Scientist* beigetragen hat.

Und ich danke der Free Software Foundation für die Entwicklung der GNU Free Documentation License, die meine Zusammenarbeit mit Jeff und Chris erst möglich gemacht hat.

Ich danke auch den Redakteuren bei Lulu, die an *How to Think Like a Computer Scientist* gearbeitet haben.

Ich danke allen Studierenden, die an früheren Fassungen dieses Buches mitgearbeitet haben, und allen Autoren (die in einem Anhang aufgeführt sind), die mir Korrekturen und Vorschläge zugesandt haben.

Und ich danke meiner Frau Lisa für ihre Arbeit an diesem Buch, an Green Tea Press und an allem anderen auch.

Allen B. Downey
Needham MA
Allen Downey ist außerordentlicher Professor für Computerwissenschaften am Franklin W. Olin College of Engineering.

Mitwirkende an „Think Python“

(Allen B. Downey)

Mehr als 100 aufmerksame Leser haben in den letzten Jahren Vorschläge und Korrekturen eingesandt. Ihre Beiträge und ihre Begeisterung für dieses Projekt waren eine große Hilfe.

Einzelheiten über die Art der Beiträge dieser Personen finden Sie im Text „Think Python“.

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, Fernando Tardio, and Paul Stoop.

Anhang B

Hinweise zum Urheberrecht

Ich hätte es vorgezogen, das Buch unter der weniger restriktiven CC-BY-SA-Lizenz zu lizenzieren. Aber leider gibt es ein paar skrupellose Organisationen, die nach frei lizenzierten Büchern suchen und diese finden und dann praktisch unveränderte Kopien der Bücher über einen Print-on-Demand-Dienst wie LuLu oder KDP veröffentlichen und verkaufen. KDP hat (dankenswerterweise) eine Richtlinie eingeführt, die den Wünschen des eigentlichen Urheberrechtsinhabers Vorrang gegenüber einem Nicht-Urheberrechtsinhaber einräumt, der versucht, ein frei lizenziertes Werk zu veröffentlichen. Leider gibt es viele Print-on-Demand-Dienste, und nur wenige haben eine so gut durchdachte Politik wie KDP.

Bedauerlicherweise sah ich mich dazu gezwungen, der Lizenz für dieses Buch das NC-Element hinzugefügt, um mir einen Rechtsanspruch zu sichern, falls jemand versucht, dieses Buch zu vervielfältigen und es kommerziell zu verkaufen. Leider schränkt das Hinzufügen des NC-Elements die Nutzung dieses Materials ein. Daher habe ich diesen Abschnitt des Dokuments hinzugefügt, um bestimmte Fälle zu beschreiben, in denen ich im Voraus meine Erlaubnis gebe, das Material in diesem Buch in Situationen zu verwenden, die manche als kommerziell ansehen könnten.

- Wenn Sie eine begrenzte Anzahl von Kopien des gesamten Buches oder eines Teils davon zur Verwendung in einem Kurs drucken (z. B. als Kurspaket), erhalten Sie für diesen Zweck eine CC-BY-Lizenz für diese Materialien.

- Wenn Sie als Hochschullehrer dieses Buch in eine andere Sprache als Englisch übersetzen und mit dem übersetzten Buch unterrichten, können Sie sich mit mir in Verbindung setzen, und ich gewähre Ihnen eine CC-BY-SA-Lizenz für diese Materialien im Hinblick auf die Veröffentlichung Ihrer Übersetzung. Insbesondere wird Ihnen gestattet, das übersetzte Buch kommerziell zu verkaufen.

Wenn Sie beabsichtigen, das Buch zu übersetzen, sollten Sie sich mit mir in Verbindung setzen, damit wir sicherstellen können, dass Sie alle zugehörigen Kursmaterialien haben, damit Sie diese ebenfalls übersetzen können.

Natürlich können Sie mich gerne kontaktieren und um Erlaubnis bitten, wenn Ihnen diese Klauseln nicht ausreichen. In jedem Fall wird die Erlaubnis zur Wiederverwendung und zum umarbeiten dieses Materials erteilt, solange ein klarer Mehrwert oder Nutzen für Schüler oder Lehrer durch die Bearbeitung entsteht.

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
September 9, 2013

Index