

Johannes Ernesti
Peter Kaiser

Migration
von Python
2.6 auf 3

Python 3

Das umfassende Handbuch

- Einstieg, Praxis, Referenz
- Sprachgrundlagen, Objektorientierung, Modularisierung
- Migration, Debugging, Interoperabilität mit C, GUIs, Netzwerkkommunikation u.v.m.

Galileo Computing



Johannes Ernesti, Peter Kaiser

Python 3

Das umfassende Handbuch

Liebe Leserin, lieber Leser,

mit der Version 3 hat Python einen großen Sprung nach vorn gemacht. Die Sprache wurde von Inkonsistenzen befreit, sodass in Python 3 geschriebener Code noch einfacher und klarer strukturiert ist. Die Zukunft gehört Python 3. Der Umstieg lohnt sich!

In diesem umfassenden Handbuch werden Sie alles finden, was Sie für Ihre Arbeit mit Python brauchen. Die Sprachgrundlagen werden genauso ausführlich behandelt wie professionelle Techniken. Egal, ob Sie gerade anfangen, mit Python zu programmieren oder schon länger mit Python arbeiten: Dieses Buch ist genau das richtige für Sie. Es bietet einen leichten Einstieg in die Python-Programmierung und lässt sich als Referenz für die tägliche Arbeit mit Python nutzen. Wenn Sie schon mit älteren Python-Versionen gearbeitet haben, können Sie sich im Migrationskapitel einen Überblick über die wichtigsten Änderungen zwischen den Versionen 2.x und 3 verschaffen.

Das Buch wurde komplett zu Python 3 geschrieben. Für den Fall, dass Sie Informationen zu älteren Python-Versionen benötigen, schauen Sie doch einfach auf die Buch-CD. Dort finden Sie das Handbuch der Autoren zur Version 2.5 als HTML-Version. Außerdem bietet die CD-ROM Python für verschiedene Plattformen sowie viele nützliche Tools. Sie können also sofort loslegen.

Wenn Sie Fragen oder Anregungen zu diesem Buch haben, können Sie sich gern an mich wenden. Ich freue mich auf Ihre Rückmeldung.

Viel Freude beim Lesen wünscht Ihnen

Ihre Judith Stevens-Lemoine

Lektorat Galileo Computing

judith.stevens@galileo-press.de

www.galileocomputing.de

Galileo Press • Rheinwerkallee 4 • 53227 Bonn

Auf einen Blick

Teil I Einstieg in Python	15
1 Einleitung	17
2 Überblick über Python	23
3 Die Arbeit mit Python	27
4 Der interaktive Modus	35
5 Grundlegendes zu Python-Programmen	43
6 Kontrollstrukturen	51
7 Das Laufzeitmodell	67
8 Basisdatentypen	79
9 Dateien	171
10 Funktionen	181
Teil II Fortgeschrittene Programmiertechniken	221
11 Modularisierung	223
12 Objektorientierung	235
13 Weitere Spracheigenschaften	279
Teil III Die Standardbibliothek	325
14 Mathematik	327
15 Strings	351
16 Datum und Zeit	385
17 Schnittstelle zum Betriebssystem	405
18 Parallele Programmierung	437
19 Datenspeicherung	459
20 Netzwerkkommunikation	515
21 Debugging	591
Teil IV Weiterführende Themen	633
22 Distribution von Python-Projekten	635
23 Optimierung	645
24 Grafische Benutzeroberflächen	651
25 Anbindung an andere Programmiersprachen	719
26 Insiderwissen	747
27 Von Python 2.6 nach Python 3.0	755
A Anhang	765

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch *Eppur se muove* (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

Gerne stehen wir Ihnen mit Rat und Tat zur Seite:

judith.stevens@galileo-press.de bei Fragen und Anmerkungen zum Inhalt des Buches
service@galileo-press.de für versandkostenfreie Bestellungen und Reklamationen
julia.bruch@galileo-press.de für Rezensionen- und Schulungsexemplare

Lektorat Judith Stevens-Lemoine, Anne Scheibe

Korrektur Petra Biedermann, Reken

Cover Barbara Thoben, Köln

Typografie und Layout Vera Brauner

Herstellung Steffi Ehrentraut

Satz Typographie & Computer, Krefeld

Druck und Bindung Bercker Graphischer Betrieb, Kevelaer

Dieses Buch wurde gesetzt aus der Linotype Syntax Serif (9,25/13,25 pt) in FrameMaker.
Gedruckt wurde es auf chlorfrei gebleichtem Offsetpapier.

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

ISBN 978-3-8362-1412-4

© Galileo Press, Bonn 2009

2., aktualisierte Auflage 2009

Das vorliegende Werk ist in all seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten, insbesondere das Recht der Übersetzung, des Vortrags, der Reproduktion, der Vervielfältigung auf fotomechanischem oder anderen Wegen und der Speicherung in elektronischen Medien. Ungeachtet der Sorgfalt, die auf die Erstellung von Text, Abbildungen und Programmen verwendet wurde, können weder Verlag noch Autor, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Die in diesem Werk wiedergegebenen Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. können auch ohne besondere Kennzeichnung Marken sein und als solche den gesetzlichen Bestimmungen unterliegen.

Inhalt

TEIL I Einstieg in Python

1	Einleitung	17
1.1	Über dieses Buch	17
1.1.1	Warum haben wir dieses Buch geschrieben?	17
1.1.2	Was leistet dieses Buch, was nicht?	18
1.1.3	Wie ist dieses Buch aufgebaut?	19
1.1.4	Wer sollte dieses Buch wie lesen?	20
1.1.5	Neuerungen in der zweiten Auflage	21
1.1.6	Danksagung	21
2	Überblick über Python	23
2.1	Geschichte und Entstehung	23
2.2	Grundlegende Konzepte	24
2.3	Einsatzmöglichkeiten und Stärken	25
2.4	Aktuelle Einsatzgebiete	26
3	Die Arbeit mit Python	27
3.1	Die Verwendung von Python	27
3.1.1	Windows	29
3.1.2	Linux	29
3.1.3	Mac OS X	30
3.2	Tippen, kompilieren, testen	30
3.2.1	Shebang	32
3.2.2	Interne Abläufe	32
4	Der interaktive Modus	35
4.1	Ganze Zahlen	36
4.2	Gleitkommazahlen	37
4.3	Zeichenketten	37
4.4	Variablen	38
4.5	Logische Ausdrücke	39
4.6	Bildschirm Ausgaben	41

5	Grundlegendes zu Python-Programmen	43
5.1	Grundstruktur eines Python-Programms	43
5.2	Das erste Programm	45
5.3	Kommentare	47
5.4	Der Fehlerfall	48
6	Kontrollstrukturen	51
6.1	Fallunterscheidungen	51
6.1.1	if, elif, else	51
6.1.2	Conditional Expressions	55
6.2	Schleifen	56
6.2.1	While-Schleife	56
6.2.2	Vorzeitiger Abbruch einer Schleife	58
6.2.3	Vorzeitiger Abbruch eines Schleifendurchlaufs	59
6.2.4	For-Schleife	61
6.3	Die pass-Anweisung	65
7	Das Laufzeitmodell	67
7.1	Die Struktur von Instanzen	69
7.2	Referenzen und Instanzen freigeben	74
7.3	Mutable vs. immutable Datentypen	75
8	Basisdatentypen	79
8.1	Operatoren	79
8.2	Das Nichts – NoneType	81
8.3	Numerische Datentypen	82
8.3.1	Ganzzahlen – int	85
8.3.2	Gleitkommazahlen – float	91
8.3.3	Boolesche Werte – bool	93
8.3.4	Komplexe Zahlen – complex	98
8.4	Methoden und Parameter	100
8.5	Sequentielle Datentypen	103
8.5.1	Listen – list	111
8.5.2	Unveränderliche Listen – tuple	122
8.5.3	Strings – str, bytes	124
8.6	Mappings	151
8.6.1	Dictionary – dict	151

8.7	Mengen	160
8.7.1	Mengen – set	167
8.7.2	Unveränderliche Mengen – frozenset	168

9 Dateien 171

9.1	Datenströme	171
9.2	Daten aus einer Datei auslesen	172
9.3	Daten in eine Datei schreiben	176
9.4	Verwendung des Dateiobjekts	177

10 Funktionen 181

10.1	Schreiben einer Funktion	183
10.2	Funktionsparameter	187
10.2.1	Optionale Parameter	187
10.2.2	Schlüsselwortparameter	188
10.2.3	Beliebige Anzahl von Parametern	189
10.2.4	Seiteneffekte	192
10.3	Lokale Funktionen	195
10.4	Anonyme Funktionen	196
10.5	Namensräume	196
10.5.1	Zugriff auf globale Variablen – global	197
10.5.2	Zugriff auf übergeordnete Namensräume – nonlocal	198
10.6	Rekursion	200
10.7	Vordefinierte Funktionen	201

TEIL II Fortgeschrittene Programmiertechniken

11 Modularisierung 223

11.1	Einbinden externer Programmbibliotheken	224
11.2	Eigene Module	226
11.2.1	Modulinterne Referenzen	227
11.3	Pakete	228
11.3.1	Absolute und relative Import-Anweisungen	230
11.3.2	Importieren aller Module eines Pakets	231
11.3.3	Die Built-in Function <code>__import__</code>	232

12 Objektorientierung 235

12.1	Klassen	240
12.1.1	Definieren von Methoden	242
12.1.2	Konstruktor, Destruktor und die Erzeugung von Attributen	243
12.1.3	Private Member	246
12.1.4	Versteckte Setter und Getter	250
12.1.5	Statische Member	252
12.2	Vererbung	255
12.2.1	Mehrfachvererbung	258
12.3	Magic Members	262
12.3.1	Allgemeine Magic Members	263
12.3.2	Datentypen emulieren	269
12.4	Objektphilosophie	277

13 Weitere Spracheigenschaften 279

13.1	Exception Handling	279
13.1.1	Eingebaute Exceptions	280
13.1.2	Werfen einer Exception	282
13.1.3	Abfangen einer Exception	283
13.1.4	Eigene Exceptions	287
13.1.5	Erneutes Werfen einer Exception	289
13.1.6	Exception Chaining	291
13.2	Comprehensions	292
13.2.1	List Comprehensions	293
13.2.2	Dict Comprehensions	295
13.2.3	Set Comprehensions	296
13.3	Docstrings	296
13.4	Generatoren	298
13.5	Iteratoren	301
13.6	Interpreter im Interpreter	310
13.7	Geplante Sprachelemente	312
13.8	Die with-Anweisung	313
13.9	Function Annotations	316
13.10	Function Decorator	318
13.11	assert	321
13.12	Weitere Aspekte der Syntax	322
13.12.1	Umbrechen langer Zeilen	322
13.12.2	Zusammenfügen mehrerer Zeilen	323

TEIL III Die Standardbibliothek

14 Mathematik 327

14.1	Mathematische Funktionen – math, cmath	327
14.1.1	Mathematische Konstanten	328
14.1.2	Zahlentheoretische Funktionen	328
14.1.3	Exponential- und Logarithmusfunktionen	330
14.1.4	Trigonometrische Funktionen	331
14.1.5	Winkelfunktionen	333
14.1.6	Hyperbolische Funktionen	333
14.1.7	Funktionen aus cmath	334
14.2	Zufallszahlengenerator – random	334
14.3	Präzise Dezimalzahlen – decimal	339
14.3.1	Verwendung des Datentyps	340
14.3.2	Nichtnumerische Werte	343
14.3.3	Das Context-Objekt	343
14.4	Spezielle Generatoren – itertools	345

15 Strings 351

15.1	Reguläre Ausdrücke – re	351
15.1.1	Syntax regulärer Ausdrücke	352
15.1.2	Verwendung des Moduls	362
15.1.3	Ein einfaches Beispielprogramm – Searching	372
15.1.4	Ein komplexeres Beispielprogramm – Matching	373
15.2	Lokalisierung von Programmen – gettext	376
15.2.1	Beispiel für die Verwendung von gettext	377
15.3	Hash-Funktionen – hashlib	380
15.3.1	Verwendung des Moduls	382
15.3.2	Beispiel	383

16 Datum und Zeit 385

16.1	Elementare Zeitfunktionen – time	385
16.2	Komfortable Datumsfunktionen – datetime	392
16.2.1	datetime.date	393
16.2.2	datetime.time	397
16.2.3	datetime.datetime	399

17 Schnittstelle zum Betriebssystem 405

17.1	Funktionen des Betriebssystems – os	405
17.1.1	Zugriff auf den eigenen Prozess und andere Prozesse	406
17.1.2	Zugriff auf das Dateisystem	407
17.2	Umgang mit Pfaden – os.path	413
17.3	Zugriff auf die Laufzeitumgebung – sys	418
17.3.1	Konstanten	418
17.3.2	Exceptions	421
17.3.3	Hooks	422
17.3.4	Sonstige Funktionen	423
17.4	Informationen über das System – platform	425
17.4.1	Funktionen	425
17.5	Kommandozeilenparameter – optparse	425
17.5.1	Taschenrechner – ein einfaches Beispiel	426
17.5.2	Weitere Verwendungsmöglichkeiten	428
17.6	Kopieren von Instanzen – copy	430
17.7	Zugriff auf das Dateisystem – shutil	433
17.8	Das Programmende – atexit	435

18 Parallele Programmierung 437

18.1	Prozesse, Multitasking und Threads	437
18.2	Die Thread-Unterstützung in Python	440
18.3	Das Modul thread	441
18.3.1	Datenaustausch zwischen Threads – locking	443
18.4	Das Modul threading	448
18.4.1	Locking im threading-Modul	450
18.4.2	Worker-Threads und Queues	454
18.4.3	Ereignisse definieren – threading.Event	457
18.4.4	Eine Funktion zeitlich versetzt ausführen – threading.Timer	457

19 Datenspeicherung 459

19.1	Komprimierte Dateien lesen und schreiben – gzip	459
19.2	XML	461
19.2.1	DOM – Document Object Model	463
19.2.2	SAX – Simple API for XML	474
19.2.3	ElementTree	479

19.3	Datenbanken	483
19.3.1	Pythons eingebaute Datenbank – sqlite3	487
19.4	Serialisierung von Instanzen – pickle	502
19.5	Das Tabellenformat CSV – csv	506
19.6	Temporäre Dateien – tempfile	511

20 Netzwerkkommunikation 515

20.1	Socket API	517
20.1.1	Client-Server-Systeme	518
20.1.2	UDP	520
20.1.3	TCP	522
20.1.4	Blockierende und nicht-blockierende Sockets	525
20.1.5	Verwendung des Moduls	526
20.1.6	Netzwerk-Byte-Order	531
20.1.7	Multiplexende Server – select	532
20.1.8	socketserver	536
20.2	URLs	540
20.2.1	Zugriff auf Ressourcen im Internet – urllib.request	540
20.2.2	Verarbeiten einer URL – urllib.parse	544
20.3	FTP – ftplib	549
20.4	E-Mail	557
20.4.1	SMTP – smtplib	557
20.4.2	POP3 – poplib	561
20.4.3	IMAP4 – imaplib	566
20.4.4	Erstellen komplexer E-Mails – email	572
20.5	Telnet – telnetlib	577
20.6	XML-RPC	580
20.6.1	Der Server	581
20.6.2	Der Client	585
20.6.3	Multicall	587
20.6.4	Einschränkungen	588

21 Debugging 591

21.1	Der Debugger	591
21.2	Inspizieren von Instanzen – inspect	594
21.2.1	Datentypen, Attribute und Methoden	595
21.2.2	Quellcode	597
21.2.3	Klassen und Funktionen	598
21.3	Formatierte Ausgabe von Instanzen – pprint	602

21.4	Logdateien – logging	605
21.4.1	Das Meldungsformat anpassen	607
21.4.2	Logging Handler	609
21.5	Automatisiertes Testen	611
21.5.1	Testfälle in Docstrings – doctest	611
21.5.2	Unit Tests – unittest	615
21.6	Traceback-Objekte – traceback	619
21.7	Analyse des Laufzeitverhaltens	622
21.7.1	Laufzeitmessung – timeit	623
21.7.2	Profiling – cProfile	626
21.7.3	Tracing – trace	629

TEIL IV Weiterführende Themen

22 Distribution von Python-Projekten 635

22.1	Erstellen von Distributionen – distutils	635
22.1.1	Schreiben des Moduls	636
22.1.2	Das Installationsscript	638
22.1.3	Erstellen einer Quellcodedistribution	642
22.1.4	Erstellen einer Binärdistribution	643
22.2	Distributionen installieren	644

23 Optimierung 645

23.1	Die Optimize-Option	646
23.2	Mutable vs. immutable	646
23.3	Funktionsaufrufe	647
23.4	Schleifen	648
23.5	C	648
23.6	Lookup	649
23.7	Exceptions	649
23.8	Keyword Arguments	650

24 Grafische Benutzeroberflächen 651

24.1	Toolkits	651
24.2	Einführung in tkinter	654
24.2.1	Ein einfaches Beispiel	654
24.2.2	Steuerelementvariablen	656
24.2.3	Der Packer	658
24.2.4	Ausrichtung	660

24.2.5	Padding	660
24.2.6	Übersicht	661
24.2.7	Events	663
24.2.8	Die Steuerelemente	669
24.2.9	Die Klasse Tk	705
24.2.10	Weitere Module	707

25 Anbindung an andere Programmiersprachen 719

25.1	Dynamisch ladbare Bibliotheken – ctypes	720
25.1.1	Ein einfaches Beispiel	720
25.1.2	Die eigene Bibliothek	721
25.1.3	Schnittstellenbeschreibung	725
25.1.4	Verwendung des Moduls	726
25.2	Schreiben von Extensions	728
25.2.1	Ein einfaches Beispiel	728
25.2.2	Exceptions	732
25.2.3	Erzeugen der Extension	733
25.2.4	Reference Counting	734
25.3	Python als eingebettete Scriptsprache	736
25.3.1	Ein einfaches Beispiel	736
25.3.2	Ein komplexeres Beispiel	738
25.3.3	Python-API-Referenz	741

26 Insiderwissen 747

26.1	URLs im Standardbrowser öffnen – webbrowser	747
26.2	Funktionsschnittstellen vereinfachen – functools	748
26.3	Versteckte Passworteingaben – getpass	750
26.4	Kommandozeilen-Interpreter – cmd	750

27 Von Python 2.6 nach Python 3.0 755

27.1	Die wichtigsten Unterschiede	755
27.1.1	Ein-/Ausgabe	756
27.1.2	Iteratoren	757
27.1.3	Strings	757
27.1.4	Ganze Zahlen	758
27.1.5	Exception Handling	759
27.1.6	Standardbibliothek	759
27.1.7	Neue Sprachelemente in Python 3.0	760
27.2	Automatische Konvertierung	761

A	Anhang	765
A.1	Entwicklungsumgebungen	765
A.1.1	Eclipse	766
A.1.2	Eric4	767
A.1.3	Komodo IDE	768
A.1.4	Wing IDE	769
A.2	Reservierte Wörter	770
A.3	Operatorrangfolge	770
A.4	Built-in Exceptions	771
A.5	Built-in Functions	775
	Index	779



Teil I
Einstieg in Python

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

1 Einleitung

1.1 Über dieses Buch

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

1.1.1 Warum haben wir dieses Buch geschrieben?

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

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

Programmierung auf, weil man nun mit den einführenden Tutorien nicht mehr vorankommt, trotzdem aber noch nicht in der Lage ist, die zumeist sehr technische Dokumentation von Python zur Weiterbildung zu nutzen.

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

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

1.1.2 Was leistet dieses Buch, was nicht?

Das Ziel dieses Buchs ist es, dem Leser fundierte Python-Kenntnisse zu vermitteln, damit er auch professionellen Aufgaben gewachsen ist. Dazu wird die Sprache Python umfassend eingeführt. Die Einführung erfolgt systematisch vom ersten einfachen Programm bis hin zu komplexen objektorientierten Programmen. Das Buch stellt den praxisbezogenen Umgang mit Python in den Vordergrund. Es ist nicht das Ziel dieses Buchs, Ihnen fundierte theoretische Kenntnisse über Disziplinen der Informatik zu vermitteln.

Abgesehen von der Einführung in die Sprache selbst, werden große Teile von Pythons Standardbibliothek besprochen. Bei der Standardbibliothek handelt es sich um eine Sammlung von Hilfsmitteln, die das Arbeiten mit Python erleichtern und eine der größten Stärken von Python darstellen. Abhängig von der Bedeutung und Komplexität des jeweiligen Themas werden konkrete Beispielprogramme zur Demonstration erstellt, was zum einen im Umgang mit der Sprache Python schult und zum anderen als Grundlage für eigene Projekte dienen kann. Der Quelltext der Beispielprogramme ist sofort ausführbar und befindet sich auf der CD, die diesem Buch beiliegt. Bei wichtigen Themen wird zusätzlich eine Referenz geboten, die das Buch auch als Nachschlagewerk nutzbar macht.

Dieses Buch ist keinesfalls als Einführung in die Programmierung allgemein oder gar in die Informatik anzusehen. Wir behandeln weder Datenstrukturen noch Algorithmen noch die dahinterstehende Theorie. Der Hauptfokus liegt auf der praktischen Arbeit mit Python, weshalb wir uns auf die Lösung von Problemen mithilfe der Sprache konzentrieren.

1.1.3 Wie ist dieses Buch aufgebaut?

Dieses Buch ist in vier Teile gegliedert, deren Inhalt im Folgenden kurz zusammengefasst wird. Sollten Sie mit den Begriffen im Moment noch nichts anfangen können, seien Sie unbesorgt – an dieser Stelle dienen alle genannten Begriffe zur Orientierung und werden im jeweiligen Kapitel des Buchs ausführlich erklärt.

1. Der erste Teil bietet einen Einstieg in die Arbeit mit Python. Dabei legen wir sehr viel Wert darauf, dass der Leser schon früh seine ersten eigenen Programme entwickeln und testen kann, denn wie bei der Programmierung allgemein gilt auch in Python, dass *Learning by Doing* die erfolgversprechendste Lernmethode ist. Die Einführung in die Grundelemente von Python haben wir so aufgebaut, dass größtenteils auf das Begriffsgebäude der Objektorientierung verzichtet wurde, um Umsteigern von nicht objektorientierten Sprachen den Einstieg zu erleichtern. Neben der Sprache selbst werden die eingebauten Datentypen und ihre Verwendung behandelt.
2. Im zweiten Teil stehen dann die Konzepte im Vordergrund, die die Arbeit mit Python erst so richtig angenehm machen, allerdings für den unerfahrenen Leser auch völliges Neuland darstellen können. Als große Oberthemen sind dabei Modularisierung und Objektorientierung zu nennen, die in Python eine zentrale Rolle spielen. Außerdem werden moderne Programmierstechniken wie Exception Handling, Iteratoren und Generatoren behandelt.
3. Der dritte Teil konzentriert sich auf Pythons *Batteries-included*-Philosophie, wonach Python nach Möglichkeit alles in der Standardbibliothek mitbringen sollte, was für die Entwicklung eigener Anwendungen erforderlich ist. Wir werden in diesem Teil auf viele der mitgelieferten Module eingehen und auch das ein oder andere Drittanbietermodul erklären. Insbesondere ist auch die Suche nach Fehlern in Python-Programmen und deren Behebung Thema dieses Teils.

Der dritte Teil ist eher als Nachschlagewerk zu konkreten Problemen gedacht und sollte nicht einfach in einem Rutsch von vorn bis hinten gelesen werden.

4. Im letzten Teil werden wir weiterführende Themen wie die Weitergabe von fertigen Python-Programmen und -Modulen an Endanwender bzw. andere Entwickler behandeln. Neben der Programmoptimierung und der Auslagerung lauffeithritischer Programmteile in effizientere Sprachen wie C wird auch die Entwicklung von grafischen Benutzeroberflächen mit Tkinter besprochen. Außerdem werden kleine Kniffe gezeigt, die das Arbeiten mit Python noch effektiver machen können. Am Ende des Buchs besprechen wir die Unterschiede zwischen den Python-Versionen 2.6 und 3.0 und zeigen, was getan werden muss, damit alte Programme wieder unter der neuen Python-Version laufen.

1.1.4 Wer sollte dieses Buch wie lesen?

Dieses Buch richtet sich im Wesentlichen an zwei Typen von Lesern: diejenigen, die in die Programmierung mit Python einsteigen möchten und idealerweise bereits grundlegende Kenntnisse der Programmierung besitzen, und diejenigen, die mit der Sprache Python bereits mehr oder weniger vertraut sind und ihr Wissen vertiefen möchten. Für beide Typen ist dieses Buch bestens geeignet, da sowohl eine vollständige Einführung in die Programmiersprache als auch eine umfassende Referenz zur Anwendung von Python in vielen Bereichen geboten werden.

Im Folgenden möchten wir eine Empfehlung an Sie richten, wie Sie dieses Buch, abhängig von Ihrem Kenntnisstand, lesen sollten.

Sollten Sie bereits einige grundlegende Erfahrungen in einer Programmiersprache, beispielsweise PHP oder Java, gesammelt haben, so bringen Sie im Prinzip bereits alle Voraussetzungen zum Lesen dieses Buchs mit, da der erste Teil des Buchs einen umfassenden Einstieg in die Sprache Python beinhaltet und wichtige Begriffe an Ort und Stelle erklärt werden. Dennoch sollten Sie sich darauf gefasst machen, dass der Anspruch in den folgenden drei Teilen des Buchs rasch zunimmt, denn unser Buch soll Sie in die Lage versetzen, Python professionell einzusetzen. Ein Leser, der das Buch zum Einstieg in die Programmiersprache Python verwenden möchte, sollte sich auf die beiden ersten Teile konzentrieren und diese vollständig durcharbeiten.

Wenn Sie selbst als »alter Hase« von C oder einer anderen Programmiersprache wechseln und eine moderne Sprache kennenlernen möchten, haben Sie mit diesem Buch die richtige Wahl getroffen. In den ersten beiden Teilen können Sie Ihre Programmierkenntnisse auf Python übertragen, wobei Sie über Erklärungen bekannter Begriffe hinweglesen können. Teil 4 bietet Ihnen dann Informationen zu weiterführenden Themen. Die Besprechung zentraler Module ist in Teil 3 angesiedelt, der als Nachschlagewerk dient.

Als letzte Zielgruppe kommen erfahrene Python-Programmierer in Betracht. Sollte der Umgang mit Python für Sie zum alltäglichen Geschäft gehören, können Sie im ersten und zweiten Teil Ihr Wissen vertiefen und festigen oder beide Teile einfach querlesen. Für Sie werden die letzten beiden Teile interessanter sein, die Ihnen als hilfreiches Nachschlagewerk dienen und Ihnen weiterführende Informationen zu speziellen Themen wie zur Entwicklung grafischer Benutzeroberflächen anbieten. Außerdem bietet dieses Buch einige interessante Praxistipps, mit denen Sie Ihre Ziele schneller als bisher erreichen können.

1.1.5 Neuerungen in der zweiten Auflage

Seitdem die erste Auflage dieses Buchs erschienen ist, hat sich in der Python-Welt einiges bewegt. So ist mit Python 3.0 die Sprache grundlegend überarbeitet worden, wodurch sich für den Python-Programmierer einiges ändert. Insbesondere ist Python 3.0 nicht mehr kompatibel zu früheren Python-Versionen.

Aus diesem Grund wurde das Buch von uns vollständig überarbeitet und auf den neusten Stand gebracht. Wir haben besonderen Wert darauf gelegt, zu verdeutlichen, wie sich Python 3.0 von früheren Versionen unterscheidet. Am Ende dieses Buchs stellen wir in einem Migrationskapitel die Unterschiede zwischen den Python-Versionen im Detail gegenüber und beschreiben, wie bestehende Python-Programme komfortabel an die neueste Version angepasst werden können.

1.1.6 Danksagung

Nachdem wir Ihnen das Buch vorgestellt und hoffentlich schmackhaft gemacht haben, möchten wir uns noch bei denjenigen bedanken, die uns bei der Ausarbeitung des Manuskripts begleitet, unterstützt und uns immer wieder zum Schreiben angetrieben haben.

Besonderer Dank gilt Peters Vater, Prof. Dr. Ulrich Kaiser, der mit seiner konstruktiven Kritik und unzähligen Stunden des Korrekturlesens die Qualität des Buchs deutlich verbessert hat. Außerdem ist es seiner Initiative zu verdanken, dass wir überhaupt dazu gekommen sind, ein Buch zu schreiben. Wir sind sehr glücklich, dass wir von seiner Sachkenntnis und Erfahrung profitieren konnten.

Neben der fachlichen Korrektheit trägt auch die verwendete Sprache maßgeblich zur Qualität des Buchs bei. Dass sich dieses Buch so gut liest, wie es sich liest, haben wir Peters Mutter Angelika Kaiser zu verdanken, die auch noch so kompliziert verschachtelte Satzgefüge in klare, gut verständliche Formulierungen verwandeln konnte.

Wir danken auch Johannes' Vater Herbert Ernesti dafür, dass er das fertige Werk noch einmal als Ganzes unter die Lupe genommen hat und viele nützliche Verbesserungsvorschläge machen konnte.

Die Anfängerfreundlichkeit der Erklärungen wurde von Peters Schwester Anne Kaiser experimentell erprobt und für gut befunden – vielen Dank dafür.

Außerdem danken wir allen Mitarbeitern von Galileo Press, die an der Erstellung dieses Buchs beteiligt waren. Namentlich hervorheben möchten wir dabei unseren Lektorinnen Judith Stevens-Lemoine und Anne Scheibe, die uns geholfen ha-

ben, sich durch den Autorendschungel zu schlagen, und uns dabei alle Freiheiten für eigene Ideen gelassen haben.

Zum Schluss möchten wir uns noch bei allen Lesern der ersten Auflage bedanken, die mit ausführlichem Feedback und konstruktiver Kritik dazu beigetragen haben, die zweite Auflage noch besser zu gestalten.

Johannes Ernesti – je@lpe-media.de

Peter Kaiser – p@penguin-p.de

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

2 Überblick über Python

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

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

2.1 Geschichte und Entstehung

Python wurde Anfang der 90er Jahre von dem Holländer *Guido van Rossum* am *Centrum voor Wiskunde en Informatica* (CWI) in Amsterdam entwickelt. Ursprünglich war Python als Scriptsprache für das verteilte Betriebssystem Amoeba gedacht. Der Name Python lehnt sich nicht etwa an die Schlangenart an, sondern ist eine Hommage an die britische Komikertruppe Monty Python.

Vor der Entwicklung von Python hatte van Rossum an der Entwicklung der Programmiersprache ABC mitgewirkt, die mit dem Ziel entworfen wurde, möglichst einfach zu sein, so dass sie problemlos einem interessierten Laien ohne Programmiererfahrung beigebracht werden kann. Die Erfahrung aus positiver und negativer Kritik an ABC nutzte van Rossum für die Entwicklung von Python. Er schuf somit eine Programmiersprache, die mächtig und zugleich einfach und leicht zu erlernen sein sollte. Inzwischen liegt Python in den Versionen 2.6 und 3.0 vor.

Mit Python 3.0, das im Dezember 2008 erschien, wurde die Sprache von Grund auf überarbeitet. Dabei sind viele kleine Inkonsistenzen und Design-Fehler be-

reinhaltet, die man in bisherigen Versionen aufgrund der Abwärtskompatibilität stets in der Sprache behalten musste.

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

2.2 Grundlegende Konzepte

Grundsätzlich handelt es sich bei Python um eine imperative Programmiersprache, die jedoch noch weitere *Programmierparadigmen* in sich vereint. So ist es beispielsweise möglich, mit Python objektorientiert und funktional zu programmieren. Sollten Sie mit diesen Begriffen im Moment noch nichts anfangen können, seien Sie unbesorgt, schließlich soll Ihnen die Programmierung mit Python und damit die Anwendung der verschiedenen Paradigmen in diesem Buch beigebracht werden.

Obwohl Python viele Sprachelemente gängiger Scriptsprachen implementiert, handelt es sich um eine interpretierte Programmiersprache. Der Unterschied zwischen einer Programmier- und einer Scriptsprache liegt im sogenannten *Compiler*. Ähnlich wie Java oder C# verfügt Python über einen Compiler, der aus dem Quelltext ein Kompilat erzeugt, den sogenannten *Byte-Code*. Dieser Byte-Code wird dann in einer virtuellen Maschine, dem *Python-Interpreter*, ausgeführt.

Ein weiteres Konzept, das Python zum Beispiel mit Java gemeinsam hat, ist die *Plattformunabhängigkeit*. Der Python-Interpreter läuft unter verschiedensten Betriebssystemen und ermöglicht, dass ein und dasselbe Python-Programm unmodifiziert unter verschiedenen Systemen lauffähig ist. Insbesondere werden die drei großen Desktop-Betriebssysteme Windows, Linux und Mac OS X unterstützt.

Im Lieferumfang von Python ist neben dem Interpreter und dem Compiler eine umfangreiche *Standardbibliothek* enthalten. Diese Standardbibliothek ermöglicht es dem Programmierer, in kurzer Zeit übersichtliche Programme zu schreiben, die allerdings sehr komplexe Aufgaben verrichten können. So bietet die Standardbibliothek beispielsweise umfassende Möglichkeiten zur Netzwerkkommunikation oder zur Datenspeicherung. Da die Standardbibliothek die Programmiermöglichkeiten in Python wesentlich bereichert, widmen wir ihr im dritten und teilweise auch vierten Teil dieses Buchs besondere Aufmerksamkeit.

Ein Nachteil der Programmiersprache ABC, den Guido van Rossum bei der Entwicklung von Python beheben wollte, war ihre fehlende Flexibilität. Ein grundlegendes Konzept von Python ist es daher, es dem Programmierer so einfach wie möglich zu machen, die Standardbibliothek beliebig zu erweitern. Da Python selbst, als abstrakte Programmiersprache, nur eingeschränkte Möglichkeiten zur maschinennahen Programmierung bietet, können maschinennahe oder zeitkritische Erweiterungen problemlos in C geschrieben werden. Das ermöglicht die sogenannte *Python API*.

Als letztes grundlegendes Konzept von Python soll erwähnt werden, dass Python unter der *PSF-Lizenz* steht. Das ist eine von der Python Software Foundation entworfene Lizenz für Open-Source-Software, die wesentlich weniger restriktiv ist als beispielsweise die GNU General Public License. So erlaubt es die PSF-Lizenz, den Python-Interpreter lizenzkostenfrei in größere, kommerzielle Anwendungen einzubetten und mit diesen auszuliefern. Diese Politik macht Python auch für kommerzielle Anwendungen attraktiv.

2.3 Einsatzmöglichkeiten und Stärken

Die größte Stärke von Python ist *Flexibilität*. So kann Python beispielsweise als Programmiersprache für kleine und große Applikationen, als serverseitige Programmiersprache im Internet oder als Scriptsprache für eine größere C- oder C++-Anwendung verwendet werden. Auch abseits des klassischen Markts breitet sich Python beispielsweise im Embedded-Bereich aus. So existieren bereits Python-Interpreter für diverse Mobiltelefone oder PDAs.

Python ist aufgrund seiner *einfachen Syntax* sehr leicht zu erlernen und gut zu lesen. Außerdem erlauben es die automatische Speicherverwaltung und die umfangreiche Standardbibliothek, mit relativ kleinen Programmen bereits sehr komplexe Probleme anzugehen. Aus diesem Grund eignet sich Python zum sogenannten *Rapid Prototyping*. Bei dieser Art der Entwicklung geht es darum, in möglichst kurzer Zeit einen lauffähigen Prototyp als eine Art Machbarkeitsstudie einer größeren Software zu erstellen, die dann später in einer anderen Programmiersprache implementiert werden soll. Mithilfe eines solchen Prototyps lassen sich Probleme und Designfehler bereits entdecken, bevor die tatsächliche Entwicklung der Software begonnen wird.

Eine weitere Stärke Pythons ist die bereits im vorherigen Abschnitt angesprochene *Erweiterbarkeit*. Aufgrund dieser Erweiterbarkeit können Python-Entwickler aus einem reichen Fundus von Drittanbieterbibliotheken und Anbindungen

an viele bekannte Bibliotheken schöpfen. So existieren beispielsweise Anbindungen an die gängigsten GUI-Toolkits, die somit das Erstellen eines Python-Programms mit grafischer Benutzeroberfläche ermöglichen.

2.4 Aktuelle Einsatzgebiete

Python erfreut sich immer größerer Bekanntheit und Verbreitung. Viele große Firmen setzen bereits erfolgreich die freie Programmiersprache ein. Die wohl bekannteste dieser Firmen ist *Google*, bei der der Python-Erfinder Guido van Rossum arbeitet. Neben Google nutzen beispielsweise auch die amerikanische Spezialeffekte-Schmiede *Industrial Light & Magic* und sogar die *NASA* Python. Auch die bekannte Website *YouTube* ist fast vollständig in Python geschrieben.

Ein weiteres interessantes Einsatzgebiet ist der von der gemeinnützigen Organisation *One Laptop per Child* entwickelte 100-Dollar-Laptop. Dabei wurde die Benutzeroberfläche des Laptops in Python geschrieben.

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

3 Die Arbeit mit Python

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

3.1 Die Verwendung von Python

Die jeweils aktuelle Version von Python können Sie von der offiziellen Python-Website unter <http://www.python.org> als Installationsdatei für Ihr Betriebssystem herunterladen und installieren. Alternativ finden Sie Python 3.0 auf der CD, die diesem Buch beiliegt.

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

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

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

Bei *IDLE* (**I**ntegrated **D**eve**L**opment **E**nvironment) handelt es sich um eine rudimentäre Python-Entwicklungsumgebung mit grafischer Benutzeroberfläche. Beim Starten von IDLE wird zunächst nur ein Fenster geöffnet, das eine Python-Shell beinhaltet. Zudem kann in IDLE über den Menüpunkt **FILE • NEW WINDOW** eine neue Python-Programmdatei erstellt und editiert werden.

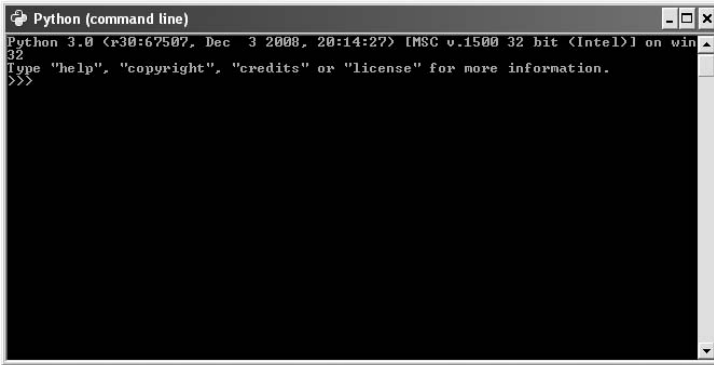


Abbildung 3.1 Python im interaktiven Modus (Python-Shell)

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



Abbildung 3.2 Die Entwicklungsumgebung IDLE

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

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

3.1.1 Windows

Sie finden die Windows-Installationsdatei von Python 3.0 auf der dem Buch beigelegten CD-ROM.

Nach der Installation von Python unter Windows sehen Sie im Wesentlichen zwei neue Einträge im Startmenü: PYTHON (COMMAND LINE) und IDLE (PYTHON GUI). Ersterer startet den interaktiven Modus von Python in der Kommandozeile (»schwarzes Fenster«) und Letzterer die grafische Entwicklungsumgebung IDLE.

3.1.2 Linux

Beachten Sie, dass Python bei vielen Linux-Distributionen bereits im Lieferumfang enthalten ist. Die meisten Distributionen werden dabei standardmäßig Python 2.x mitbringen. Python 3.0 muss eventuell über den Paketmanager Ihrer Distribution nachinstalliert werden. Die beiden Versionen können aber problemlos gleichzeitig installiert sein.

Sollten Sie eine Distribution ohne Paketmanager einsetzen oder sollte Python 3.0 nicht verfügbar sein, müssen Sie den Quellcode von Python selbst kompilieren und installieren. Dazu können Sie den Anweisungen der im Quelltext enthaltenen *Readme*-Datei folgen.

Sie finden den Quellcode von Python 3.0 auf der dem Buch beigelegten CD-ROM.

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

Hinweis

Bei vielen Distributionen werden Sie Python 3.0 mit einem anderen Befehl, beispielsweise `python3`, starten müssen, da diese Python 2.x und 3.0 parallel installieren.

3.1.3 Mac OS X

Sie finden die Mac OS X-Installationsdatei von Python 3.0 auf der dem Buch beigelegten CD-ROM.

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

3.2 Tippen, kompilieren, testen

In diesem Abschnitt sollen die Arbeitsabläufe besprochen werden, die nötig sind, um ein Python-Programm zu erstellen und auszuführen. Ganz allgemein sollten Sie sich darauf einstellen, dass wir in einem Großteil des Buchs ausschließlich sogenannte *Konsolenanwendungen* in Python schreiben werden. Eine Konsolenanwendung hat eine rein textbasierte Schnittstelle zum Benutzer und läuft in der Konsole des jeweiligen Betriebssystems ab.

Grundsätzlich besteht ein Python-Programm aus einer oder mehreren Programmdateien. Diese Programmdateien haben die Dateierendung `.py` und enthalten den Python-Quelltext. Dabei handelt es sich im Prinzip um nichts anderes als um Textdateien. Programmdateien können also mit einem normalen Texteditor bearbeitet werden.

Nachdem eine Programmdatei geschrieben worden ist, besteht der nächste logische Schritt darin, sie auszuführen. Wenn Sie IDLE verwenden, kann die Programmdatei bequem über den Menüpunkt `RUN • RUN MODULE` ausgeführt werden. Sollten Sie einen Editor verwenden, der keine vergleichbare Funktion unterstützt, müssen Sie in einer Kommandozeile in das Verzeichnis der Programmdatei wechseln und, abhängig von Ihrem Betriebssystem, verschiedene Kommandos ausführen.

Unter Windows reicht es, den Namen der Programmdatei einzugeben und mit  zu bestätigen. Im folgenden Beispiel soll die Programmdatei *programm.py* im Ordner `C:\Ordner` ausgeführt werden. Dazu müssen Sie ein Konsolenfenster unter `START • PROGRAMME • ZUBEHÖR • EINGABEAUFFORDERUNG` starten.

Bei »Dies schreibt Ihnen Ihr Python-Programm« handelt es sich um eine Ausgabe des Python-Programms in der Datei *programm.py*, die beweist, dass das Python-Programm tatsächlich ausgeführt wurde.

```

C:\>cd Ordner
C:\Ordner>dir
Volume in Laufwerk C: hat keine Bezeichnung.
Volumeseriennummer: 5C99-CBR7

Verzeichnis von C:\Ordner

24.09.2007  16:48    <DIR>          .
24.09.2007  16:48    <DIR>          ..
24.09.2007  16:49                49 programm.py
               1 Datei(en)               49 Bytes
               2 Verzeichnis(se), 2.388.004.864 Bytes frei

C:\Ordner>programm.py
Dies schreibt Ihnen Ihr Python-Programm
C:\Ordner>

```

Abbildung 3.3 Ausführen eines Python-Programms unter Windows

Hinweis

Unter Windows ist es auch möglich, ein Python-Programm durch einen Doppelklick auf die jeweilige Programmdatei auszuführen. Das hat aber gegenüber der soeben besprochenen Methode den Nachteil, dass sich das Konsolenfenster sofort nach Beenden des Programms schließt und die Ausgaben des Programms somit nicht erkennbar sind.

Unter Unix-ähnlichen Betriebssystemen wie Linux oder Mac OS X wechseln Sie ebenfalls in das Verzeichnis, in dem die Programmdatei liegt, und starten dann den Python-Interpreter mit dem Kommando `python`, gefolgt von dem Namen der auszuführenden Programmdatei. Im folgenden Beispiel soll die Programmdatei *programm.py* unter Linux ausgeführt werden, die sich im Verzeichnis */home/user/ordner* befindet.

```

[user@USER ~]$ cd /home/user/ordner
[user@USER ordner]$ ls -al
Insgesamt 24
drwxr-xr-x  2 user users 4096 25. Sep 03:14 .
drwx--x--x 88 user users 4096 26. Sep 13:12 ..
-rw-r--r--  1 user users 110 25. Sep 04:00 programm.py
[peter@P ordner]$ python programm.py
Dies schreibt Ihnen Ihr Python-Programm
[peter@P ordner]$

```

Abbildung 3.4 Ausführen eines Python-Programms unter Linux

Bitte beachten Sie den Hinweis aus 3.1.2, der besagt, dass das Kommando, mit dem Sie Python 3.0 starten, je nach Distribution von dem hier demonstrierten `python` abweichen kann.

3.2.1 Shebang

Unter einem Unix-ähnlichen Betriebssystem wie beispielsweise Linux können Python-Programmdateien mithilfe eines sogenannten *Shebangs*, auch *Magic Line* genannt, direkt ausführbar gemacht werden. Dazu muss die erste Zeile der Programmdatei in der Regel folgendermaßen lauten:

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

In diesem Fall wird das Betriebssystem dazu angehalten, diese Programmdatei immer mit dem Python-Interpreter auszuführen. Unter anderen Betriebssystemen, beispielsweise Windows, wird die Shebang-Zeile ignoriert.

Beachten Sie, dass der Python-Interpreter auf Ihrem System in einem anderen Verzeichnis als dem hier angegebenen installiert sein könnte. Allgemein ist daher folgende Shebang-Zeile besser, da sie vom tatsächlichen Installationsort Pythons unabhängig ist:

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

Beachten Sie, dass das Executable-Flag der Programmdatei gesetzt werden muss, bevor die Datei tatsächlich ausführbar ist. Das geschieht mit dem Befehl

```
chmod +x dateiname
```

Die in diesem Buch gezeigten Beispiele enthalten aus Gründen der Übersicht keine Shebang-Zeile. Das bedeutet aber ausdrücklich nicht, dass vom Einsatz einer Shebang-Zeile grundsätzlich abzuraten wäre.

3.2.2 Interne Abläufe

Bislang haben Sie einen ungefähren Begriff davon, was Python ausmacht und wo die Stärken der Programmiersprache liegen. Außerdem wurde das theoretische Grundwissen zum Erstellen und Ausführen einer Python-Programmdatei vermittelt. Doch in den vorherigen Abschnitten sind Begriffe wie »Compiler« oder »Interpreter« gefallen, ohne tatsächlich erklärt worden zu sein. In diesem Abschnitt möchten wir uns daher den internen Vorgängen widmen, die beim Ausführen einer Python-Programmdatei ablaufen. Die Grafik in Abbildung 3.5 veranschaulicht, was beim Ausführen einer Programmdatei namens *programm.py* geschieht.

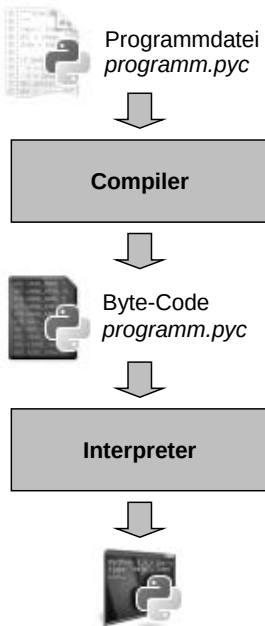


Abbildung 3.5 Kompilieren und Interpretieren einer Programmdatei

Wenn die Programmdatei *programm.py* wie zu Beginn des Kapitels beschrieben ausgeführt wird, passiert sie zunächst den sogenannten *Compiler*. Ein Compiler ist ein allgemeiner Begriff der Informatik und bezeichnet ein Programm, das von einer formalen Sprache in eine andere übersetzt. In Falle von Python übersetzt der Compiler von der Sprache Python in den sogenannten *Byte-Code*. Dabei steht es dem Compiler frei, den generierten Byte-Code im Arbeitsspeicher zu behalten oder als *programm.pyc* auf der Festplatte zu speichern.

Beachten Sie, dass das vom Compiler generierte Kompilat, im Gegensatz zu beispielsweise C- oder C++-Kompilaten, nicht direkt auf dem Prozessor ausgeführt werden kann. Zur Ausführung des Byte-Codes wird eine weitere Abstraktionsschicht, der sogenannte *Interpreter*, benötigt. Der Interpreter, häufig auch *virtuelle Maschine* (engl. *virtual machine*) genannt, liest den vom Compiler erzeugten Byte-Code ein und führt ihn aus.

Dieses Prinzip einer interpretierten Programmiersprache hat verschiedene Vorteile. So kann derselbe Python-Code beispielsweise unmodifiziert auf allen Plattformen ausgeführt werden, für die ein Python-Interpreter existiert. Allerdings laufen Programme interpretierter Programmiersprachen aufgrund des zwischengeschalteten Interpreters auch immer langsamer als ein vergleichbares C-Programm, das direkt auf dem Prozessor ausgeführt wird.

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

4 Der interaktive Modus

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

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

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

```
Python 3.0 (r30:67503, Dec 7 2008, 04:54:04)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Nach der Eingabeaufforderung (`>>>`) kann beliebiger Python-Code eingegeben werden. Bei Zeilen, die nicht mit `>>>` beginnen, handelt es sich um Ausgaben des Interpreters. Zur Bedienung des interaktiven Modus ist noch zu sagen, dass er über eine *History-Funktion* verfügt. Das heißt, dass Sie über die `↑`- und `↓`-Tasten alte Eingaben bequem wieder hervorholen können und nicht erneut eingeben müssen.

Wir beginnen mit der Einführung von konstanten Werten. Dabei unterscheiden wir zunächst einmal drei Typen von Werten: ganze Zahlen, Gleitkommazahlen und Zeichenketten. Es gibt dabei bestimmte Regeln, nach denen man einen Zahlenwert oder eine Zeichenkette zu schreiben hat, damit diese vom Interpreter erkannt werden. Eine solche Schreibweise nennt man *Literal*.

4.1 Ganze Zahlen

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

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

Das Literal für eine ganze Zahl besteht dabei aus den Ziffern 0 bis 9. Zudem kann ein positives oder negatives Vorzeichen vorangestellt werden. Eine Zahl ohne Vorzeichen wird stets als positiv angenommen.

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

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

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

```
>>> ((21 - 3) * 9 + 8) / 4
42.5
```

Hier zeigt sich, dass der Interpreter die gewohnten mathematischen Rechengesetze anwendet und das erwartete Ergebnis ausgibt. Bei dem Ergebnis der Berechnung handelt es sich korrekterweise nicht um eine ganze Zahl, sondern um eine sogenannte Gleitkommazahl, von denen der nächste Abschnitt handeln wird.¹

¹ Diese so natürliche Eigenschaft unterscheidet Python bereits von vielen anderen Programmiersprachen. Seit Python 3.0 ist das Ergebnis einer Division stets eine Gleitkommazahl, zuvor wurde bei zwei ganzzahligen Operanden eine sogenannte Integer-Division, also eine ganzzahlige Division, durchgeführt. Dies kann durchaus ein gewünschtes Verhalten sein, führt aber gerade bei Programmieranfängern häufig zu Verwirrung und wurde deshalb in Python 3.0 abgeschafft. Wenn Sie eine Integer-Division in Python 3 durchführen möchten, müssen Sie den Operator `//` verwenden:

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

4.2 Gleitkommazahlen

Das Literal für eine Gleitkommazahl besteht aus einem Vorkommaanteil, einem Dezimalpunkt und einem Nachkommaanteil. Wie schon bei den ganzen Zahlen ist es möglich, ein Vorzeichen anzugeben:

```
>>> 0.5
0.5
>>> -123.456
-123.456
>>> +1.337
1.337
```

Beachten Sie, dass es sich bei dem Dezimaltrennzeichen um einen Punkt handeln muss. Die in Deutschland übliche Schreibweise mit einem Komma ist nicht zulässig. Gleitkommazahlen lassen sich ebenso intuitiv in Termen verwenden wie die ganzen Zahlen:

```
>>> 1.5 / 2.1
0.7142857142857143
```

Soviel zunächst zu ganzen Zahlen und Gleitkommazahlen. Zu einem späteren Zeitpunkt werden wir auf diese grundlegenden Datentypen zurückkommen und sie in aller Ausführlichkeit behandeln. Doch nun zu einem weiteren wichtigen Datentyp, den Zeichenketten.

4.3 Zeichenketten

Neben den Zahlen sind Zeichenketten, auch *Strings* genannt, von entscheidender Bedeutung. Strings ermöglichen es, Text vom Benutzer einzulesen, zu speichern, zu bearbeiten oder auszugeben.

Um einen konstanten String zu erzeugen, wird der zugehörige Text in doppelte Hochkommata geschrieben:

```
>>> "Hallo Welt"
'Hallo Welt'
>>> "abc123"
'abc123'
```

Dass der Interpreter den Wert des Strings in einfachen Hochkommata ausgibt, sollte Sie im Moment nicht weiter stören; wir werden zu gegebener Zeit darauf zurückkommen.

Ähnlich wie bei Ganz- und Gleitkommazahlen gibt es auch Operatoren für Strings. So fügt der Operator `+` beispielsweise zwei Strings zusammen:

```
>>> "Hallo" + " " + "Welt"
'Hallo Welt'
```

Abgesehen davon kann ein String unter Verwendung des Operators `*` mit einer ganzen Zahl multipliziert werden:

```
>>> "Hallo" * 3
'HalloHalloHallo'
>>> 3 * "Hallo"
'HalloHalloHallo'
```

Die Operatoren `-` und `/`, die wir für die Ganz- und Gleitkommazahlen eingeführt haben, sind für Strings nicht definiert.

4.4 Variablen

Es ist in Python möglich, einer Zahl oder Zeichenkette einen Namen zu geben. Dazu wird der Name auf der linken und das entsprechende Literal auf der rechten Seite eines Gleichheitszeichens geschrieben. Eine solche Operation wird *Zuweisung* genannt.

```
>>> name = 0.5
>>> var123 = 12
>>> string = "Hallo Welt!"
```

Die mit den Namen verknüpften Werte können später ausgegeben oder in Berechnungen verwendet werden, indem der Name anstelle des jeweiligen Werts eingegeben wird:

```
>>> name
0.5
>>> 2 * name
1.0
>>> (var123 + var123) / 3
8
>>> var123 + name
12.5
```

Es ist genauso möglich, dem Ergebnis einer Berechnung einen Namen zu geben:

```
>>> a = 1 + 2
>>> b = var123 / 4
```

Dabei wird immer zuerst die Seite rechts vom Gleichheitszeichen ausgewertet. So wird beispielsweise bei der Anweisung $a = 1 + 2$ stets zuerst das Ergebnis von $1 + 2$ bestimmt, bevor dem entstandenen Wert ein Name zugewiesen wird.

Ein Variablenname, auch *Bezeichner* genannt, darf seit Python Version 3.0 aus nahezu beliebigen Buchstaben und dem Unterstrich (`_`) bestehen. Nach mindestens einem führenden Buchstaben oder Unterstrich dürfen auch Ziffern verwendet werden. Beachten Sie, dass auch Umlaute und spezielle Buchstaben anderer Sprachen erlaubt sind, wie folgendes Beispiel zeigt:

```
>>> äöüßée = 123
>>> äöüßée
123
```

Solche Freiheiten, was Bezeichner angeht, finden sich in anderen Programmiersprachen so gut wie nie. Nicht zuletzt deshalb empfehlen wir, sich auf das englische Alphabet zu beschränken. Die fehlenden Umlaute und das β fallen auch bei deutschen Bezeichnern kaum ins Gewicht und wirken im Quellcode eher verwirrend als natürlich.

Bestimmte sogenannte *Schlüsselwörter* sind in Python für die Sprache selbst reserviert und dürfen nicht als Bezeichner verwendet werden. Eine Übersicht über alle reservierten Wörter finden Sie im Anhang.

Zum Schluss möchten wir noch einen weiteren Begriff einführen: Alles, was mit numerischen Literalen – also Ganz- oder Gleitkommazahlen, Variablen und den bisher vorgestellten Operatoren – formuliert werden kann, wird als *arithmetischer Ausdruck* bezeichnet. Ein solcher Ausdruck könnte also so aussehen:

```
(a * a + b) / 12
```

Alle bisher eingeführten Operatoren `+`, `-`, `*` und `/` werden folgerichtig als *arithmetische Operatoren* bezeichnet.

Beachten Sie bei der Verwendung von Variablen, dass Python *case sensitive* ist. Dies bedeutet, dass bei Bezeichnern zwischen Groß- und Kleinschreibung unterschieden wird. In der Praxis heißt das, dass die Bezeichner `otto` und `Otto` nicht identisch sind, sondern durchaus zwei verschiedene Werte haben können.

4.5 Logische Ausdrücke

Es ist möglich, Zahlen miteinander zu vergleichen:

```
>>> 3 < 4
True
```

Hier wird getestet, ob 3 kleiner ist als 4. Auf solche Vergleiche antwortet der Interpreter mit einem *Wahrheitswert*, also mit `True` (dt. »wahr«) oder `False` (dt. »falsch«). Ein Vergleich wird mithilfe eines sogenannten *Vergleichsoperators*, in diesem Fall `<`, durchgeführt. Die folgende Liste führt alle relevanten Vergleichsoperatoren auf:

Vergleich	Bedeutung
<code>3 == 4</code>	Ist 3 gleich 4? Beachten Sie das doppelte Gleichheitszeichen, das den Vergleich von einer Zuweisung unterscheidet.
<code>3 != 4</code>	Ist 3 ungleich 4?
<code>3 < 4</code>	Ist 3 kleiner als 4?
<code>3 > 4</code>	Ist 3 größer als 4?
<code>3 <= 4</code>	Ist 3 kleiner oder gleich 4?
<code>3 >= 4</code>	Ist 3 größer oder gleich 4?

Tabelle 4.1 Vergleiche in Python

Allgemein kann für 3 und 4 ein beliebiger arithmetischer Ausdruck eingesetzt werden. Wenn zwei arithmetische Ausdrücke durch einen der obigen Operatoren miteinander verglichen werden, so erzeugt man einen sogenannten *logischen Ausdruck*. Ein solcher könnte also auch folgendermaßen aussehen:

```
(a - 7) < (b * b + 6.5)
```

Neben den bereits eingeführten arithmetischen Operatoren gibt es drei logische Operatoren, mit denen Sie das Ergebnis eines logischen Ausdrucks verändern oder zwei logische Ausdrücke miteinander verknüpfen können.

Der Operator `not` kehrt das Ergebnis eines Vergleiches um, macht also aus `True` `False` und aus `False` `True`. Der Ausdruck `not (3 < 4)` wäre also das Gleiche wie `3 >= 4`:

```
>>> not (3 < 4)
False
>>> not (4 < 3)
True
```

Der Operator `and` bekommt zwei logische Ausdrücke als Operanden und ergibt nur dann `True`, wenn sowohl der erste Ausdruck als auch der zweite `True` ergeben haben. Er entspricht damit der umgangssprachlichen »Und«-Verknüpfung zweier Satzteile. Im Beispiel kann dies so aussehen:

```
>>> (3 < 4) and (5 < 6)
True
```

```
>>> (3 < 4) and (4 < 3)
False
```

Der Operator `or` entspricht dem umgangssprachlichen »oder«. Er bekommt zwei logische Ausdrücke als Operanden und ergibt nur dann `False`, wenn sowohl der erste Ausdruck als auch der zweite `False` ergeben haben. Der Operator ergibt also `True`, wenn mindestens einer seiner Operanden `True` ergeben hat:

```
>>> (3 < 4) or (5 < 6)
True
>>> (3 < 4) or (4 < 3)
True
>>> (5 > 6) or (4 < 3)
False
```

Wir haben der Einfachheit halber hier nur Zahlen miteinander verglichen. Selbstverständlich ergibt ein solcher Vergleich nur dann einen Sinn, wenn komplexere arithmetische Ausdrücke miteinander verglichen werden. Durch die vergleichenden Operatoren und die drei sogenannten *booleschen Operatoren* `not`, `and` und `or` können schon sehr komplexe Vergleiche erstellt werden.

Beachten Sie, dass bei allen Beispielen aus Gründen der Übersicht Klammern gesetzt wurden. Durch Prioritätsregelungen der Operatoren untereinander sind diese überflüssig. Das bedeutet, dass jedes hier vorgestellte Beispiel auch ohne Klammern wie erwartet funktionieren würde. Trotzdem ist es gerade am Anfang sehr sinnvoll, durch Klammerung die Zugehörigkeiten visuell eindeutig zu gestalten.

4.6 Bildschirmausgaben

Auch wenn wir hin und wieder auf den interaktiven Modus zurückgreifen werden, ist es natürlich unser Ziel, möglichst schnell echte Python-Programme zu schreiben. Es ist eine Besonderheit des interaktiven Modus, dass der Wert eines eingegebenen Ausdrucks automatisch ausgegeben wird. In einem normalen Programm müssen Bildschirmausgaben dagegen vom Programmierer erzeugt werden. Um den Wert einer Variablen auszugeben, wird in Python der Befehl² `print` verwendet:

2 Beachten Sie, dass der Begriff »Befehl« an dieser Stelle etwas schwammig ist, denn bei `print` handelt es sich seit Python 3.0 nicht mehr um ein Schlüsselwort, sondern um eine Funktion. Aus diesem Grund dürfen auch die Klammern um den auszugebenden Wert nicht weggelassen werden. Was eine Funktion genau ist, werden wir in Kapitel 10 »Funktionen« ausführlich behandeln.

```
>>> print(1.2)
1.2
```

Beachten Sie, dass mittels `print`, im Gegensatz zur automatischen Ausgabe des interaktiven Modus, nur der Wert an sich ausgegeben wird. So wird bei der automatischen Ausgabe der Wert eines Strings in Hochkommata geschrieben, während dies bei `print` nicht der Fall ist:

```
>>> "Hallo Welt"
'Hallo Welt'
>>> print("Hallo Welt")
Hallo Welt
```

Auch hier ist es problemlos möglich, statt eines konstanten Wertes einen Variablennamen zu verwenden:

```
>>> var = 9
>>> print(var)
9
```

Oder Sie geben das Ergebnis eines Ausdrucks direkt aus:

```
>>> print(-3 * 4)
-12
```

Außerdem ermöglicht `print` es, mehrere Variablen oder Konstanten in einer Zeile auszugeben. Dazu werden die Werte durch Kommata getrennt angegeben. Jedes Komma wird bei der Ausgabe durch ein Leerzeichen ersetzt:

```
>>> print(-3, 12, "Python rockt")
-3 12 Python rockt
```

Das ist insbesondere dann hilfreich, wenn Sie nicht nur einzelne Werte, sondern auch einen kurzen erklärenden Text dazu ausgeben möchten. So etwas könnte folgendermaßen erreicht werden:

```
>>> var = 9
>>> print("Die magische Zahl ist:", var)
Die magische Zahl ist: 9
```

Abschließend ist noch zu sagen, dass `print` nach jeder Ausgabe einen Zeilenvorschub ausgibt. Es wird also stets in eine neue Zeile geschrieben.

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

5 Grundlegendes zu Python-Programmen

5.1 Grundstruktur eines Python-Programms

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

Um Ihnen ein Gefühl für die Sprache Python zu vermitteln, möchten wir zunächst einen Überblick über ihre Syntax geben. Dazu ist zu sagen, dass Python dem Programmierer sehr genaue Vorgaben macht, wie er seinen Quellcode zu strukturieren hat. Obwohl erfahrene Programmierer darin eine Einschränkung sehen mögen, kommt diese Eigenschaft gerade Programmierneulingen zugute, denn unstrukturierter und unübersichtlicher Code ist eine der größten Fehlerquellen in der Programmierung. Grundsätzlich besteht ein Python-Programm aus einzelnen *Anweisungen*, die im einfachsten Fall genau eine Zeile im Quelltext einnehmen. Folgende Anweisung gibt beispielsweise einen Text auf dem Bildschirm aus:

```
print("Hallo Welt")
```

Einige Anweisungen lassen sich in einen *Anweisungskopf* und einen *Anweisungskörper* unterteilen, wobei der Körper weitere Anweisungen enthalten kann:

```
Anweisungskopf:  
| Anweisung  
| :  
| Anweisung
```

Abbildung 5.1 Struktur einer mehrzeiligen Anweisung

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


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

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

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

Achtung!

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

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

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

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

Beachten Sie, dass Sie, auch wenn eine Zeile mit `...` beginnt, die aktuelle Einrückungstiefe berücksichtigen müssen. Der Interpreter kann das Ende des Anweisungskörpers nicht automatisch erkennen, da dieser beliebig viele Anweisungen

enthalten kann. Deswegen muss ein Anweisungskörper im interaktiven Modus durch Drücken der -Taste beendet werden.

5.2 Das erste Programm

Als Einstieg in die Programmierung mit Python bieten wir hier ein kleines Beispielprogramm, das Spiel Zahlenraten. Die Spielidee ist folgende:

Der Spieler soll eine im Programm festgelegte Zahl erraten. Dazu stehen ihm beliebig viele Versuche zur Verfügung. Nach jedem Versuch informiert ihn das Programm darüber, ob die geratene Zahl zu groß, zu klein oder genau richtig gewesen ist. Sobald der Spieler die Zahl erraten hat, gibt das Programm die Anzahl der Versuche aus und wird beendet. Aus Sicht des Spielers soll das Ganze folgendermaßen aussehen:

```
Raten Sie: 42
Zu klein
Raten Sie: 10000
Zu gross
Raten Sie: 999
Zu klein
Raten Sie: 1337
Super, Sie haben es in 4 Versuchen geschafft!
```

Kommen wir vom Ablaufprotokoll zur konkreten Implementierung in Python:

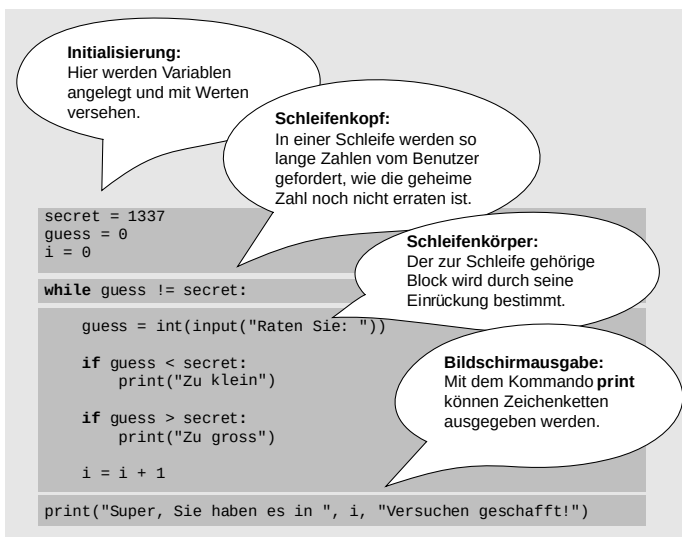


Abbildung 5.2 Zahlenraten, ein einfaches Beispiel

Jetzt möchten wir die einzelnen Bereiche des Programms noch einmal ausführlich diskutieren.

Initialisierung

Bei der Initialisierung werden die für das Spiel benötigten Variablen angelegt. Python unterscheidet zwischen verschiedenen *Variablentypen*, wie Zeichenketten, Ganz- oder Fließkommazahlen. Der Typ einer Variablen wird zur Laufzeit des Programms anhand des ihr zugewiesenen Wertes bestimmt. Es ist also nicht nötig, einen Variablentyp explizit anzugeben. Eine Variable kann im Laufe des Programms ihren Typ ändern.

In unserem Spiel werden Variablen für die gesuchte Zahl (`secret`), die Benutzereingabe (`guess`) und den Versuchszähler (`i`) angelegt und mit Anfangswerten versehen. Dadurch, dass `guess` und `secret` zu Beginn des Programms verschiedene Werte haben, ist sichergestellt, dass die Schleife anläuft.

Schleifenkopf

Eine `while`-Schleife wird eingeleitet. Eine `while`-Schleife läuft so lange, wie die im Schleifenkopf genannte Bedingung (`guess != secret`) erfüllt ist, also in diesem Fall, bis die Variablen `guess` und `secret` den gleichen Wert haben. Aus Benutzersicht bedeutet dies: Die Schleife läuft so lange, bis die Benutzereingabe mit der gespeicherten Zahl übereinstimmt.

Den zum Schleifenkopf gehörigen Schleifenkörper erkennt man daran, dass die nachfolgenden Zeilen um eine Stufe weiter eingerückt wurden. Sobald die Einrückung wieder um einen Schritt nach links geht, endet der Schleifenkörper.

Schleifenkörper

In der ersten Zeile des Schleifenkörpers wird eine vom Spieler eingegebene Zahl eingelesen und in der Variablen `guess` gespeichert. Dabei wird mithilfe von `input("Raten Sie: ")` die Eingabe des Benutzers eingelesen und mit `int` in eine ganze Zahl konvertiert. Diese Konvertierung ist wichtig, da Benutzereingaben generell als String eingelesen werden. In unserem Fall möchten wir die Eingabe jedoch als ganze Zahl weiterverwenden. Der String `"Raten Sie: "` wird vor der Eingabe ausgegeben und dient dazu, den Benutzer zur Eingabe der Zahl aufzufordern.

Nach dem Einlesen wird einzeln geprüft, ob die eingegebene Zahl `guess` größer oder kleiner als die gesuchte Zahl `secret` ist, und mittels `print` eine entsprechende Meldung ausgegeben. Schlussendlich wird der Versuchszähler `i` um eins erhöht.

Nach dem Hochzählen des Versuchszählers endet der Schleifenkörper, da die nächste Zeile nicht mehr unter dem Schleifenkopf eingerückt ist.

Bildschirmausgabe

Die letzte Programmzeile gehört nicht mehr zum Schleifenkörper. Das bedeutet, dass sie erst ausgeführt wird, wenn die Schleife vollständig durchlaufen, das Spiel also gewonnen ist. In diesem Fall werden eine Erfolgsmeldung sowie die Anzahl der benötigten Versuche ausgegeben. Das Spiel ist beendet.

Erstellen Sie jetzt Ihr erstes Python-Programm, indem Sie den Programmcode in eine Datei namens *spiel.py* schreiben und ausführen. Ändern Sie den Startwert von `guess`, und spielen Sie das Spiel.

5.3 Kommentare

Sie können sich sicherlich vorstellen, dass es nicht das Ziel ist, Programme zu schreiben, die auf eine Postkarte passen würden. Mit der Zeit wird der Quelltext Ihrer Programme umfangreicher und komplexer werden. Irgendwann ist der Zeitpunkt erreicht, da bloßes Gedächtnistraining nicht mehr ausreicht, um die Übersicht zu bewahren. Spätestens dann kommen Kommentare ins Spiel.

Ein *Kommentar* ist ein kleiner Text, der eine bestimmte Stelle des Quellcodes kurz erläutert und auf Probleme, offene Aufgaben oder Ähnliches hinweist. Ein Kommentar wird vom Interpreter einfach ignoriert, ändert also am Ablauf des Programms selbst nichts.

Die einfachste Möglichkeit, einen Kommentar zu verfassen, ist der sogenannte *Zeilenkommentar*. Diese Art des Kommentars wird mit dem `#`-Zeichen begonnen und endet mit dem Ende der Zeile:

```
# Ein Beispiel mit Kommentaren
print("Hallo Welt!") # Simple Hallo-Welt-Ausgabe
```

Für längere Kommentare bietet sich ein *Blockkommentar* an. Ein Blockkommentar beginnt und endet mit drei aufeinanderfolgenden Anführungszeichen (`"""`):

```
""" Dies ist ein Blockkommentar,
er kann sich über mehrere Zeilen erstrecken. """
```

Kommentare sollten nur gesetzt werden, wenn sie zum Verständnis des Quelltextes beitragen oder sonstige wertvolle Informationen enthalten. Jede noch so unwichtige Zeile zu kommentieren führt dazu, dass man den Wald vor lauter Bäumen nicht mehr sieht.

5.4 Der Fehlerfall

Vielleicht haben Sie bereits ein wenig mit dem Beispielprogramm aus Abschnitt 5.2 gespielt und sind dabei auf eine solche oder ähnliche Ausgabe des Interpreters gestoßen:

```
File "spiel.py", line 8
    if guess < secret
                ^
SyntaxError: invalid syntax
```

Es handelt sich dabei um eine Fehlermeldung, die in diesem Fall auf einen Syntaxfehler im Programm hinweist. Können Sie erkennen, welcher Fehler hier vorliegt? Richtig, es fehlt der Doppelpunkt am Ende der Zeile.

Python stellt bei der Ausgabe einer Fehlermeldung wichtige Informationen bereit, die bei der Fehlersuche hilfreich sind:

- ▶ Die erste Zeile der Fehlermeldung gibt Aufschluss darüber, in welcher Zeile innerhalb welcher Datei der Fehler aufgetreten ist. In diesem Fall handelt es sich um die Zeile 8 in der Datei *spiel.py*.
- ▶ Der mittlere Teil zeigt den betroffenen Ausschnitt des Quellcodes, wobei die genaue Stelle, auf die sich die Meldung bezieht, mit einem kleinen Pfeil markiert ist. Wichtig ist, dass dies die Stelle ist, an der der Interpreter den Fehler erstmalig feststellen konnte. Das ist nicht unbedingt gleichbedeutend mit der Stelle, an der der Fehler gemacht wurde.
- ▶ Die letzte Zeile spezifiziert den Typ der Fehlermeldung, in diesem Fall einen Syntax Error. Dies sind die am häufigsten auftretenden Fehlermeldungen. Sie zeigen an, dass der Compiler das Programm aufgrund eines formalen Fehlers nicht weiter übersetzen konnte.

Neben dem Syntaxfehler gibt es eine ganze Reihe weiterer Fehlertypen, die hier nicht alle im Detail besprochen werden sollen. Wir möchten jedoch noch auf den `IndentationError` (dt. »Einrückungsfehler«) hinweisen, da er gerade bei Python-Anfängern häufig auftritt. Versuchen Sie dazu einmal, folgendes Programm auszuführen:

```
i = 10
if i == 10:
print("Falsch eingerueckt")
```

Sie sehen, dass die letzte Zeile eigentlich einen Schritt weiter eingerückt sein müsste. So, wie das Programm jetzt geschrieben wurde, hat die `if`-Anweisung

keinen Anweisungskörper. Das ist nicht zulässig, und es tritt ein `IndentationError` auf:

```
File "indent.py", line 3
    print("Falsch eingerueckt")
    ^
```

`IndentationError: expected an indented block`

Nachdem wir uns mit diesen Grundlagen vertraut gemacht haben, kommen wir zu einem wichtigen Sprachelement aller modernen Programmiersprachen, den Kontrollstrukturen.

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

6 Kontrollstrukturen

Unter einer *Kontrollstruktur* versteht man ein Konstrukt, mit dessen Hilfe sich der Ablauf eines Programms steuern lässt. Dabei unterscheidet man in Python zwei Arten von Kontrollstrukturen: *Schleifen* und *Fallunterscheidungen*.

Schleifen dienen dazu, einen Codeblock mehrmals auszuführen. Fallunterscheidungen hingegen knüpfen einen Codeblock an eine Bedingung, so dass er nur ausgeführt wird, wenn die Bedingung erfüllt ist. Wie und in welchem Umfang diese zwei Typen unterstützt werden, ist von Programmiersprache zu Programmiersprache verschieden. Python kennt jeweils zwei Unterarten, die wir hier behandeln werden.

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

6.1 Fallunterscheidungen

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

6.1.1 if, elif, else

Die einfachste Möglichkeit der Fallunterscheidung ist die *if-Anweisung*. Eine *if-Anweisung* besteht aus einem Anweisungskopf, der eine Bedingung enthält, und aus einem Codeblock als Anweisungskörper:


```

if Bedingung:
    | Anweisung
    | :
    | Anweisung

```

Abbildung 6.1 Struktur einer if-Anweisung

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

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

```

if x == 1:
    print("x hat den Wert 1")

```

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

Selbstverständlich können Sie auch andere vergleichende Operatoren oder einen komplexeren logischen Ausdruck verwenden und mehr als eine Anweisung in den Körper schreiben:

```

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

```

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

```

if x == 1:
    print("x hat den Wert 1")
if x == 2:
    print("x hat den Wert 2")

```

Dies ist aus Sicht des Interpreters eine ineffiziente Art, das Ziel zu erreichen, denn beide Bedingungen werden in jedem Fall ausgewertet und überprüft. Jedoch bräuhnte die zweite Fallunterscheidung nicht mehr in Betracht gezogen zu wer-

den, wenn die Bedingung der ersten bereits `True` ergeben hat. Die Variable `x` kann unter keinen Umständen sowohl den Wert 1 als auch 2 haben. Um solche Fälle aus Sicht des Interpreters performanter und aus Sicht des Programmierers übersichtlicher zu machen, kann eine `if`-Anweisung um einen oder mehrere sogenannte `elif`-Zweige (»`elif`« ist ein Kürzel für »`else if`«) erweitert werden.

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

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

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

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

```
if Bedingung:
    | Anweisung
    | :
    | Anweisung
elif Bedingung:
    | Anweisung
    | :
    | Anweisung
elif Bedingung:
    | Anweisung
    | :
    | Anweisung
```

Abbildung 6.2 Struktur einer `if`-Anweisung mit `elif`-Zweigen

Im Quelltext könnte dies folgendermaßen aussehen:

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

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

```

if Bedingung:
    | Anweisung
    | :
    | Anweisung
else:
    | Anweisung
    | :
    | Anweisung

```

Abbildung 6.3 Struktur einer `if`-Anweisung mit `else`-Zweig

Konkret im Quelltext kann dies so aussehen:

```

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

```

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

Hinweis

Sollten Sie bereits eine Programmiersprache wie C oder Java beherrschen, so wird Sie interessieren, dass in Python kein Pendant zur `switch/case`-Kontrollstruktur dieser Sprachen existiert. Das Verhalten dieser Kontrollstruktur kann trotzdem durch eine Kaskade von `if/elif/else`-Zweigen nachgebildet werden.

Abschließend soll Abbildung 6.4 den Aufbau einer `if`-Anweisung noch einmal zusammenfassend und übersichtlich darstellen:

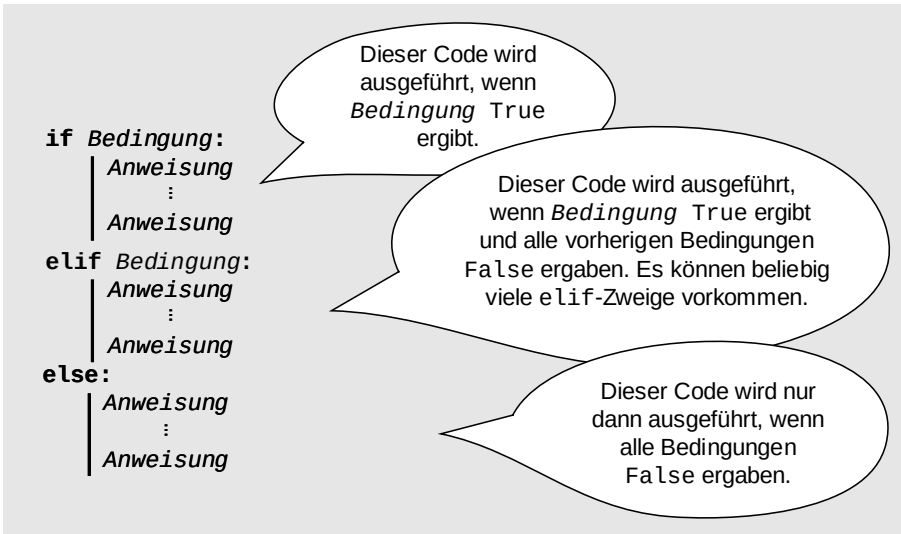


Abbildung 6.4 Aufbau einer if-Anweisung

6.1.2 Conditional Expressions

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

```

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

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

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

```

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

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

```

A if Bedingung else B
  
```

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

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

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

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

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

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

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

6.2 Schleifen

Eine sogenannte *Schleife* ermöglicht es ganz allgemein, einen Codeblock, den sogenannten *Schleifenkörper*, mehrmals hintereinander auszuführen. Python unterscheidet zwei Typen von Schleifen: eine `while`-Schleife als sehr simples Konstrukt und eine `for`-Schleife zum Durchlaufen komplexerer Datentypen.

6.2.1 While-Schleife

Die `while`-Schleife haben wir bereits in unserem Spiel »Zahlenraten« verwendet. Sie dient dazu, einen Codeblock so lange auszuführen, wie eine bestimmte Bedingung erfüllt ist. In unserem ersten Programm aus Abschnitt 5.2 wurde mithilfe

einer `while`-Schleife so lange eine neue Zahl vom Spieler eingelesen, bis die eingegebene Zahl mit der gesuchten Zahl übereinstimmt.

Grundsätzlich besteht eine `while`-Schleife aus einem Schleifenkopf, in dem die Bedingung steht, sowie einem Schleifenkörper, der dem auszuführenden Codeblock entspricht. Beachten Sie, dass die Schleife läuft, *solange* die Bedingung erfüllt ist, und nicht, *bis* sie erfüllt ist.

```
while Bedingung:
    Anweisung
    :
    Anweisung
```

Abbildung 6.5 Struktur einer `while`-Schleife

Das folgende Beispiel ist ein etwas verknappter Ausschnitt des »Zahlenraten«-Spiels und soll die Verwendung der `while`-Schleife veranschaulichen:

```
secret = 1337
guess = 0
while guess != secret:
    guess = int(input("Raten Sie: "))
```

Das Schlüsselwort `while` leitet den Schleifenkopf ein, und wird von der gewünschten Bedingung und einem Doppelpunkt gefolgt. In den nächsten Zeilen folgt, um eine Stufe weiter eingerückt, der Schleifenkörper. Dort wird eine Zahl vom Benutzer eingelesen und mit dem Namen `guess` versehen. Dieser Prozess läuft so lange, bis die im Schleifenkopf genannte Bedingung erfüllt ist, bis also die Eingabe des Benutzers (`guess`) mit der geheimen Zahl (`secret`) übereinstimmt.

Ähnlich wie eine `if`-Anweisung kann eine `while`-Schleife um einen `else`-Zweig erweitert werden. Der Codeblock, der zu diesem Zweig gehört, wird genau einmal ausgeführt, nämlich dann, wenn die Schleife vollständig abgearbeitet wurde, also die Bedingung zum ersten Mal `False` ergibt:

```
while Bedingung:
    Anweisung
    :
    Anweisung
else:
    Anweisung
    :
    Anweisung
```

Abbildung 6.6 Struktur einer `while`-Schleife mit `else`-Zweig

Betrachten wir dies an einem konkreten Beispiel:

```
secret = 1337
guess = 0
while guess != secret:
    guess = int(input("Raten Sie: "))
else:
    print("Sie haben es geschafft!")
```

Aus Benutzersicht bedeutet dies, dass die Erfolgsmeldung ausgegeben wird, wenn die richtige Zahl geraten wurde:

```
Raten Sie: 100
Raten Sie: 200
Raten Sie: 1337
Sie haben es geschafft!
```

Momentan scheint dieser `else`-Zweig überflüssig, da der gleiche Effekt durch folgenden Code erreicht werden kann:

```
secret = 1337
guess = 0
while guess != secret:
    guess = int(input("Raten Sie: "))
print("Sie haben es geschafft!")
```

Die beiden Beispiele sind in diesem Anwendungsfall völlig äquivalent zu verwenden. Dies ist nicht immer der Fall. Dass der `else`-Zweig einer `while`-Schleife durchaus seine Berechtigung hat, werden Sie im nächsten Abschnitt sehen.

6.2.2 Vorzeitiger Abbruch einer Schleife

Stellen Sie sich einmal vor, wir wollten das Beispiel, das wir im vorherigen Abschnitt eingeführt haben, dahingehend erweitern, dass das Spiel durch Eingabe einer `0` beendet werden kann. Dies ist mit Ihrem bisherigen Kenntnisstand zwar möglich, jedoch nur über Umwege zu erreichen. Was wirklich fehlt, ist eine Möglichkeit, eine Schleife in besonderen Fällen vorzeitig zu beenden. Genau dies erreichen Sie mit der sogenannten `break`-Anweisung:

```
secret = 1337
guess = 0
while guess != secret:
    guess = int(input("Raten Sie: "))
    if guess == 0:
        print("Das Spiel wird beendet")
        break
```

```
else:
    print("Sie haben es geschafft!")
```

Das Beispiel wurde durch eine `if`-Anweisung erweitert. Direkt nachdem eine Zahl vom Spieler eingegeben und mit dem Namen `guess` versehen wurde, wird geprüft, ob es sich bei der Eingabe um eine 0 handelt (`guess == 0`). Sollte dies der Fall sein, wird eine entsprechende Meldung ausgegeben und die `while`-Schleife mit `break` beendet.

In Kombination mit `break` zeigt sich auch die eigentliche Bedeutung des `else`-Zweigs einer Schleife. Der `else`-Zweig wird nur ausgeführt, wenn die Schleife vollständig durchlaufen wurde, und nicht, wenn sie durch `break` vorzeitig beendet wurde. Das Ablaufprotokoll gibt uns hier recht:

```
Raten Sie: 100
Raten Sie: 200
Raten Sie: 0
Das Spiel wird beendet
```

6.2.3 Vorzeitiger Abbruch eines Schleifendurchlaufs

Wir haben mit `break` bereits eine Möglichkeit vorgestellt, den Ablauf einer Schleife zu beeinflussen. Die sogenannte `continue`-Anweisung bricht im Gegensatz zu `break` jedoch nicht die gesamte Schleife ab, sondern nur den aktuellen Schleifendurchlauf. Um dies zu veranschaulichen, betrachten wir das folgende Beispiel, das bisher noch ohne `continue`-Anweisung auskommt:

```
while True:
    zahl = int(input("Geben Sie eine Zahl ein: "))
    ergebnis = 1
    while zahl > 0:
        ergebnis = ergebnis * zahl
        zahl = zahl - 1
    print("Ergebnis: ", ergebnis)
```

Zur Erklärung des Beispiels: In einer Endlosschleife – also einer `while`-Schleife, deren Bedingung unter allen Umständen erfüllt ist (`while True`) –, wird eine Zahl eingelesen und die Variable `ergebnis` mit 1 initialisiert. In einer darauf folgenden weiteren `while`-Schleife wird `ergebnis` so lange mit `zahl` multipliziert, wie die Bedingung `zahl > 0` erfüllt ist. Zudem wird in jedem Durchlauf der inneren Schleife der Wert von `zahl` um 1 verringert.

Nachdem die innere Schleife durchlaufen ist, wird die Variable `ergebnis` ausgegeben. Wie Sie vermutlich bereits erkannt haben, berechnet das Beispielprogramm die Fakultät einer jeden eingegebenen Zahl:


```
Geben Sie eine Zahl ein: 4
Ergebnis: 24
Geben Sie eine Zahl ein: 5
Ergebnis: 120
Geben Sie eine Zahl ein: 6
Ergebnis: 720
```

Allerdings erlaubt der obige Code auch eine solche Eingabe:

```
Geben Sie eine Zahl ein: -10
Ergebnis: 1
```

Durch die Eingabe einer negativen Zahl ist die Bedingung der inneren Schleife (`zahl > 0`) von vornherein `False`, die Schleife wird also gar nicht erst ausgeführt. Aus diesem Grund wird sofort der Wert von `ergebnis` ausgegeben, der in diesem Fall 1 ist.

Das ist allerdings nicht ganz das, was in diesem Fall erwartet würde. Bei einer negativen Zahl handelt es sich um eine ungültige Eingabe. Idealerweise sollte das Programm also bei Eingabe einer ungültigen Zahl die Berechnung abbrechen und kein Ergebnis anzeigen. Eben dies wird durch Verwendung einer `continue`-Anweisung erreicht:

```
while True:
    zahl = int(input("Geben Sie eine Zahl ein: "))
    if zahl < 0:
        print("Negative Zahlen sind nicht erlaubt")
        continue
    ergebnis = 1
    while zahl > 0:
        ergebnis = ergebnis * zahl
        zahl = zahl - 1
    print("Ergebnis: ", ergebnis)
```

Direkt nachdem die Eingabe des Benutzers eingelesen wurde, wird in einer `if`-Abfrage überprüft, ob es sich um eine negative Zahl handelt (`zahl < 0`). Sollte das der Fall sein, so wird mit `print` eine entsprechende Fehlermeldung ausgegeben und der aktuelle Schleifendurchlauf mit `continue` abgebrochen. Das heißt, dass alle Codezeilen, die zur Schleife gehören und hinter `continue` stehen, erst im nächsten Schleifendurchlauf wieder interpretiert werden. Aus Benutzersicht bedeutet das, dass nach Eingabe einer negativen Zahl kein Ergebnis, sondern eine Fehlermeldung ausgegeben wird. Danach wird zur Eingabe der nächsten Zahl aufgefordert:

```
Geben Sie eine Zahl ein: 4
Ergebnis: 24
```

```
Geben Sie eine Zahl ein: 5
Ergebnis: 120
Geben Sie eine Zahl ein: -10
Negative Zahlen sind nicht erlaubt
Geben Sie eine Zahl ein: -100
Negative Zahlen sind nicht erlaubt
```

Rückblickend möchten wir an dieser Stelle noch einmal den Unterschied zwischen `break` und `continue` herausarbeiten:

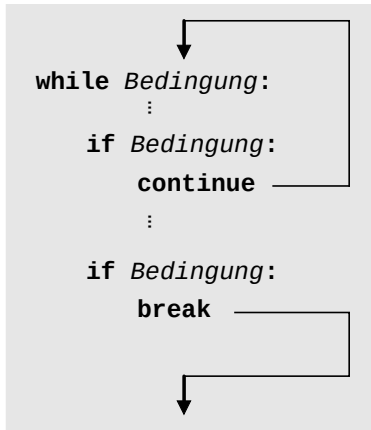


Abbildung 6.7 Eine Schleife mit `break` und `continue`

Während `break` die Schleife vollständig abbricht, beendet `continue` nur den aktuellen Schleifendurchlauf, die Schleife an sich läuft aber weiter.

6.2.4 For-Schleife

Neben der bisher behandelten `while`-Schleife existiert in Python ein weiteres Schleifenkonstrukt, die sogenannte `for`-Schleife. Eine `for`-Schleife kann im einfachsten Fall als *Zählschleife* verwendet werden. Das ist eine Schleife, die es dem Programmierer ermöglicht, festzulegen, wie oft ein Codeblock erneut ausgeführt werden soll. Die Anzahl der bisherigen Schleifendurchläufe steht im Codeblock als Variable, dem sogenannten *Schleifenzähler*, zur Verfügung:

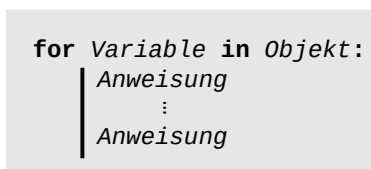


Abbildung 6.8 Struktur einer `for`-Schleife

Was genau für *Objekt* eingesetzt werden kann, werden wir im Folgenden erklären. Konkret im Quelltext sieht eine `for`-Schleife beispielsweise so aus:

```
for i in range(5):
    print(i)
```

In diesem Beispiel werden alle ganzen Zahlen von 0 bis einschließlich 4 ausgegeben. `range` kann dabei nicht nur ein Limit setzen, sondern allgemein in drei Varianten verwendet werden:

- ▶ `range(stop)`
- ▶ `range(start, stop)`
- ▶ `range(start, stop, step)`

Der Platzhalter *start* steht dabei für die Zahl, mit der begonnen wird. Die Schleife wird beendet, sobald *stop* erreicht wurde. Wichtig ist zu wissen, dass der Schleifenzähler selbst niemals den Wert *stop* erreicht, er bleibt stets kleiner. In jedem Schleifendurchlauf wird der Schleifenzähler um *step* erhöht. Sowohl *start* als auch *stop* und *step* müssen ganze Zahlen sein. Wenn alle Werte angegeben sind, sieht die `for`-Schleife folgendermaßen aus:

```
for i in range(1, 10, 2):
    print(i)
```

Die Zählvariable *i* beginnt jetzt mit dem Wert 1; die Schleife wird ausgeführt, solange *i* kleiner ist als 10, und in jedem Schleifendurchlauf wird *i* um 2 erhöht. Damit gibt die Schleife die Werte 1, 3, 5, 7 und 9 auf dem Bildschirm aus.

Eine `for`-Schleife kann nicht nur in positiver Richtung verwendet werden, es ist auch möglich, herunterzuzählen:

```
for i in range(10, 1, -2):
    print(i)
```

In diesem Fall wird *i* zu Beginn der Schleife auf den Wert 10 gesetzt und in jedem Durchlauf um 2 verringert. Die Schleife läuft, solange *i* größer ist als 1, und gibt die Werte 10, 8, 6, 4 und 2 auf dem Bildschirm aus.

Damit bietet sich die `for`-Schleife geradezu an, um das Beispiel des letzten Abschnitts zur Berechnung der Fakultät einer Zahl zu überarbeiten. Es ist gleichzeitig ein Beispiel dafür, dass `while`- und `for`-Schleifen wie selbstverständlich ineinander verschachtelt werden können:

```
while True:
    zahl = int(input("Geben Sie eine Zahl ein: "))
    if zahl < 0:
```

```

        print("Negative Zahlen sind nicht erlaubt")
        continue
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis = ergebnis * i
    print("Ergebnis: ", ergebnis)

```

Nachdem eine Eingabe durch den Benutzer erfolgt ist und auf ihr Vorzeichen hin überprüft wurde, wird eine `for`-Schleife eingeleitet. Der Schleifenzähler `i` der Schleife beginnt mit dem Wert 2. Die Schleife läuft, solange `i` kleiner als `zahl+1` ist: Der höchstmögliche Wert von `i` ist also `zahl`. In jedem Schleifendurchlauf wird dann die Variable `ergebnis` mit `i` multipliziert.

Es wurde bereits angedeutet, dass eine Zählschleife nur eine mögliche Verwendung der `for`-Schleife darstellt. Ganz allgemein durchläuft eine `for`-Schleife ein sogenanntes *iterierbares Objekt*. Ohne näher ins Detail gehen zu wollen, ist zu sagen, dass ein solches Objekt in der Regel eine Art Container für eine Reihe verschiedener Werte darstellt. Sie werden im Laufe dieses Buchs viele solcher Objekte kennenlernen, und wir werden dabei jedes Mal auf die Verwendung der `for`-Schleife zurückkommen.

Eines dieser iterierbaren Objekte haben Sie jedoch bereits kennengelernt: Ein String kann ganz allgemein als ein Container für eine Reihe von Buchstaben betrachtet und als solcher auch mithilfe der `for`-Schleife durchlaufen werden. Der Vorgang wird auch *Iterieren* genannt. Es wird »über einen String iteriert«:

```

for c in "Hallo Welt":
    print(c)

```

Die Ausgabe dieses Beispiels lautet:

```

H
a
l
l
o

W
e
l
t

```

Der Namenswechsel der Schleifenvariable von `i` nach `c` hat übrigens keine syntaktische Bedeutung, sondern eher eine assoziative: `i` kann als Abkürzung für »integer« (dt. »ganze Zahl«) und `c` für »character« (dt. »Buchstabe«) angesehen werden.

Dass jeder Buchstabe in eine neue Zeile geschrieben wurde, hat nichts mit der Schleife zu tun, sondern ist ein normales Verhalten der `print`-Anweisung.

Abschließend ist noch zu sagen, dass eine `for`-Schleife genauso über einen `else`-Zweig verfügen kann wie eine `while`-Schleife. Auch bei einer `for`-Schleife ist ein `else`-Zweig nur in Kombination mit `break` sinnvoll.

```
for Variable in Objekt:
    Anweisung
    :
    Anweisung
else:
    Anweisung
    :
    Anweisung
```

Abbildung 6.9 Struktur einer `for`-Schleife mit `else`-Zweig

Konkret:

```
for c in "abc":
    print(c)
else:
    print("abc")
```

Dies führt zu folgender Ausgabe:

```
a
b
c
abc
```

Hinweis

Die `for`-Schleife, wie sie in Python existiert, ist kein Pendant des gleichnamigen Schleifenkonstrukts aus C oder Java. Sie ist eher mit der `foreach`-Schleife aus PHP oder Perl vergleichbar.¹

¹ Die `for`-Schleife, wie sie in C existiert, ist ein mächtiges Konstrukt. Sie kann durch die `for`-Schleife aus Python, allein in Kombination mit `range`, nicht ersetzt werden. Es ist in Python allerdings möglich, sogenannte Generatorfunktionen zu erstellen, die die Einsatzgebiete der `for`-Schleife erheblich erweitern. Näheres zu Generatorfunktionen folgt in Abschnitt Generatoren.

6.3 Die pass-Anweisung

Während der Entwicklung eines Programms kommt es vor, dass eine Kontrollstruktur vorerst nur teilweise implementiert wird. Der Programmierer erstellt einen Anweisungskopf, fügt aber keinen Anweisungskörper an, da er sich vielleicht zuerst um andere, wichtigere Dinge kümmern möchte. Ein in der Luft hängender Anweisungskopf ohne entsprechenden Körper ist aber ein Syntaxfehler.

Zu diesem Zweck existiert die `pass`-Anweisung. Es ist eine Anweisung, die gar nichts macht. Sie könnte folgendermaßen angewendet werden:

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

In diesem Fall ist im Körper der `if`-Anweisung nur `pass` zu finden. Sollte `x` also den Wert 1 haben, passiert schlicht und einfach nichts.

Die `pass`-Anweisung hat den Zweck, Syntaxfehler in vorläufigen Programmversionen zu vermeiden. Fertige Programme enthalten in der Regel keine `pass`-Anweisungen.

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

7 Das Laufzeitmodell

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

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

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

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

```
>>> 2674 / a  
2.0
```

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

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

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

```
>>> 12345  
12345
```

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

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

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

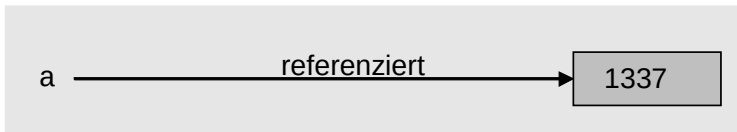


Abbildung 7.1 Schema der Referenz-Instanz-Beziehung

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

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

Grafisch veranschaulicht sieht das Ergebnis so aus:



Abbildung 7.2 Zwei Referenzen auf dieselbe Instanz

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

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

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

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

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

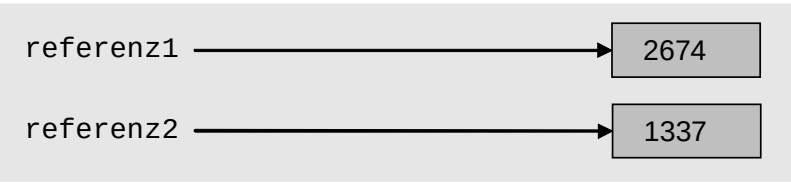


Abbildung 7.3 Die beiden Referenzen sind voneinander unabhängig.

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

7.1 Die Struktur von Instanzen

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



Abbildung 7.4 Eine Instanz mit ihren drei Eigenschaften

Datentyp

Der Datentyp dient bei der Erzeugung der Instanz als Bauplan und legt fest, welche Werte die Instanz annehmen darf. So erlaubt der Datentyp `int` beispielsweise das Speichern einer ganzen Zahl. Strings lassen sich mit dem Datentyp `str` verwalten. Im folgenden Beispiel wird gezeigt, wie sich die Datentypen verschiedener Instanzen mithilfe von `type` herausfinden lassen:¹

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

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

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

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

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

Es ist also falsch, zu sagen: »`zuerst_ein_string` hat den Typ `str`.« Korrekt ist: »`zuerst_ein_string` referenziert momentan eine Instanz des Typs `str`.«

1 Bei `type` handelt es sich um eine sogenannte Funktion. Was genau das bedeutet, ist an dieser Stelle noch nicht wichtig. Wir werden uns in Kapitel Funktionen eingehend mit Funktionen beschäftigen und dort auch auf `type` zurückkommen.

Wert

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

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

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

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

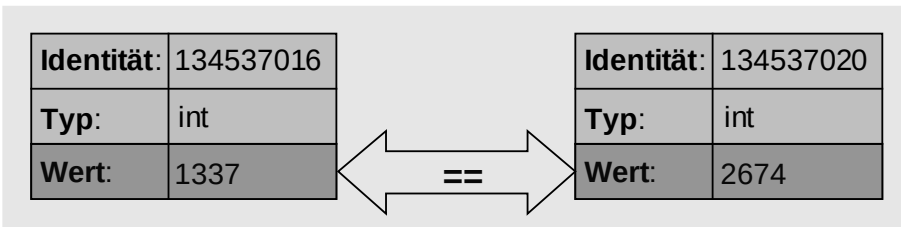


Abbildung 7.5 Wertevergleich zweier Instanzen (in diesem Fall False)

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

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

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

Zahlen und Zeichenketten haben strukturell wenig gemeinsam, da es sich bei Zahlen um einzelne Werte handelt, während bei Zeichenketten mehrere Buchstaben zu einer Einheit zusammengefasst werden.

Aus diesem Grund liefert der Operator `==` für den Vergleich zwischen Strings und Zahlen immer `False`, auch wenn die Werte für einen Menschen gleich aussehen:

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

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

Identität

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

```
>>> id(1337)
134537016
>>> v1 = "Hallo Welt"
>>> id(v1)
3082572528
```

Identitäten werden immer dann wichtig, wenn man prüfen möchte, ob es sich um eine ganz bestimmte Instanz handelt und nicht nur um eine mit dem gleichen Typ und Wert:²

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

2 Es ist möglich, dass Sie im folgenden Beispiel auf Ihrem Rechner eine andere Ausgabe erhalten als hier abgedruckt. Der Unterschied zwischen Wert und Identität bleibt aber trotzdem bestehen. Er lässt sich nur unter Umständen an dieser Stelle nicht praktisch aufzeigen, da die bis hierher eingeführten Datentypen so einfach sind, dass Python entscheiden kann, ob wirklich eine neue Instanz erzeugt wird oder nicht. Im Abschnitt »Mutable vs. immutable Datentypen« (Mutable vs. immutable Datentypen) wird ausführlich auf dieses Thema eingegangen.

```
>>> id(v1) == id(v2)
True
```

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

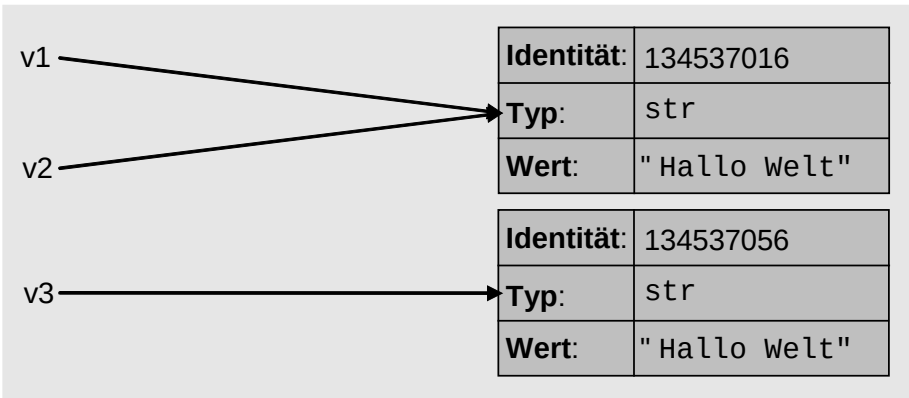


Abbildung 7.6 Drei Referenzen, zwei Instanzen

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

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

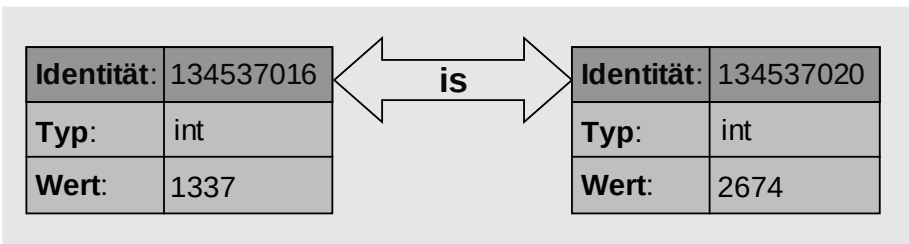


Abbildung 7.7 Identitätenvergleich zweier Instanzen

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

7.2 Referenzen und Instanzen freigeben

Während eines Programmlaufs werden in der Regel sehr viele Instanzen angelegt, die aber nicht alle die ganze Zeit benötigt werden. Betrachten wir einmal den folgenden fiktiven Programmanfang:

```
willkommen = "Herzlich willkommen im Beispielprogramm"
print(willkommen)
# Hier würde es jetzt mit dem restlichen Programm weitergehen
```

Es ist leicht ersichtlich, dass die von `willkommen` referenzierte Instanz nach der Begrüßung nicht mehr gebraucht wird und somit während der restlichen Programmlaufzeit sinnlos Speicher verschwendet. Wünschenswert wäre also eine Möglichkeit, nicht mehr benötigte Instanzen auf Anfrage entfernen zu können.

Python lässt den Programmierer den Speicher nicht direkt verwalten, sondern übernimmt dies für ihn. Als Folge davon können wir bestehende Instanzen nicht manuell löschen, sondern müssen uns auf einen Automatismus verlassen, die sogenannte *Garbage Collection*.³

Trotzdem gibt es eine Form der Einflussnahme:

Instanzen, auf die keine Referenzen mehr verweisen, werden von Python als nicht mehr benötigt eingestuft und dementsprechend wieder freigegeben. Wollen wir also eine Instanz entfernen, müssen wir nur die dazugehörigen Referenzen freigeben. Für diesen Zweck gibt es in Python die `del`-Anweisung. Nach ihrer Freigabe existiert die Referenz nicht mehr, und ein versuchter Zugriff führt zu einem `NameError`:

```
>>> v1 = 1337
>>> v1
1337
>>> del v1
>>> v1

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'v1' is not defined
```

3 Die *Garbage Collection* (dt. *Müllabfuhr*) ist ein System, das nicht mehr benötigte Datenobjekte entfernt und den dazugehörigen Speicher wieder freigibt. Sie arbeitet für den Programmierer unsichtbar im Hintergrund.

Für technisch Interessierte: Pythons *Garbage Collection* ist durch ein *Reference-Counting*-System implementiert, das durch einen Algorithmus zur Erkennung zyklischer Referenzen ergänzt wird.

Möchte man mehrere Instanzen auf einmal freigeben, trennt man sie einfach durch Kommata voneinander ab:

```
>>> v1 = 1337
>>> v2 = 2674
>>> v3 = 4011
>>> del v1, v2, v3
>>> v1
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'v1' is not defined
```

Um zu erkennen, wann für eine Instanz keine Referenzen mehr existieren, speichert Python intern für jede Instanz einen Zähler, den sogenannten *Referenzzähler* (engl. *reference count*). Für frisch erzeugte Instanzen hat er den Wert null. Immer wenn eine neue Referenz auf eine Instanz erzeugt wird, erhöht sich der Referenzzähler der Instanz um eins, und immer, wenn eine Referenz freigegeben wird, wird er um eins verringert. Damit gibt der Referenzzähler einer Instanz stets die aktuelle Anzahl von Referenzen an, die auf die Instanz verweisen. Erreicht der Zähler den Wert null, gibt es für die Instanz keine Referenz mehr. Da Instanzen für den Programmierer nur über Referenzen zugänglich sind, ist der Zugriff auf eine solche Instanz nicht mehr möglich – sie kann gelöscht werden.

7.3 Mutable vs. immutable Datentypen

Vielleicht sind Sie beim Ausprobieren des gerade Beschriebenen schon auf den folgenden Scheinwiderspruch gestoßen:

```
>>> a = 1
>>> b = 1
>>> id(a)
9656320
>>> id(b)
9656320
>>> a is b
True
```

Warum referenzieren *a* und *b* dieselbe Ganzzahl-Instanz, wie es der Identitätenvergleich zeigt, obwohl wir in den ersten beiden Zeilen ausdrücklich zwei Instanzen mit dem Wert 1 erzeugt haben?

Um diese Frage zu beantworten, müssen wir wissen, das Python grundlegend zwischen zwei Arten von Datentypen unterscheidet: zwischen *mutable* (dt. »ver-

änderlichen«) Datentypen und *immutable* (dt. »unveränderlichen«) Datentypen. Wie die Namen schon sagen, besteht der Unterschied zwischen den beiden Arten darin, ob sich der Wert einer Instanz zur Laufzeit ändern kann, ob sie also veränderbar ist. Instanzen eines mutable Typs sind dazu in der Lage, nach ihrer Erzeugung andere Werte anzunehmen, während dies bei immutable Datentypen nicht der Fall ist.

Wenn sich der Wert einer Instanz aber nicht ändern kann, ergibt es auch keinen Sinn, mehrere immutable Instanzen des gleichen Werts im Speicher zu verwalten, weil im Optimalfall genau eine Instanz ausreicht, auf die dann alle entsprechenden Referenzen verweisen. Wie Sie sich nun sicherlich denken, handelt es sich bei Ganzzahlen eben um so einen immutable Datentyp, und Python hat aus Optimierungsgründen bei beiden Einsen auf dieselbe Instanz verweisen lassen. Auch Strings sind immutable.⁴

Es ist allerdings nicht so, dass es immer nur genau eine Instanz zu jedem benötigten Wert eines unveränderlichen Datentyps gibt, obwohl dies theoretisch möglich wäre. Der Grund dafür liegt in der Optimierung:

Wird eine neue Instanz eines immutable Typs vom Programm angefordert, gibt es für Python zwei Möglichkeiten: Entweder wird eine neue Instanz im Speicher erstellt oder eine vorhandene ein weiteres Mal referenziert. Eine neue Instanz im Speicher zu erzeugen, »kostet« Python Rechenzeit und Speicherplatz. Python muss Speicher anfordern und diesen mit den entsprechenden Informationen füllen. Eine bestehende Instanz ein weiteres Mal zu referenzieren, ist um ein Vielfaches »billiger«, da sowohl das Bereitstellen als auch das Befüllen des Speichers entfallen und stattdessen nur ein Referenzzähler erhöht und eine Speicheradresse kopiert werden muss. Das stimmt aber nur dann, wenn der Interpreter schon weiß, an welcher Stelle im Speicher eine Instanz mit dem gleichen Wert wie die neu angeforderte Instanz liegt. Je »länger« der Wert der neuen Instanz ist und je mehr Instanzen es bereits gibt, desto aufwendiger gestaltet sich die Suche nach einer bereits bestehenden passenden Instanz. Ab einem gewissen Punkt ist es dann nicht mehr effizient, eine bereits existierende Instanz erneut zu referenzieren, weil die Suche mehr Rechenzeit kostet als das Erstellen einer neuen Instanz. Python entscheidet unabhängig vom Programmierer, welchen der beiden Wege es beschreitet. Beispielsweise haben wir im letzten Abschnitt die Arbeitsweise von `id` mit dem String `"Hallo Welt"` verdeutlicht und festgestellt, dass sich die Identitäten der beiden Instanzen unterscheiden: Python hat in diesem Fall aus

4 Das bedeutet natürlich nicht, dass Strings und Ganzzahlen aus Sicht des Programmierers unveränderlich sind. Es wird nur bei jeder Manipulation eines immutable Datentyps eine neue Instanz des Datentyps erzeugt, anstatt die alte zu verändern.

den oben genannten Optimierungsgründen zwei Instanzen des Strings erstellt, obwohl dies nicht nötig gewesen wäre.

Bei den mutable, also den veränderlichen Datentypen sieht es anders aus: Weil Python damit rechnen muss, dass sich der Wert einer solchen Instanz nachträglich ändern wird, ist das obige System, nach Möglichkeit bereits vorhandene Instanzen erneut zu referenzieren, nicht sinnvoll. Hier kann man sich also darauf verlassen, dass immer eine neue Instanz erzeugt wird.

Weil wir bisher noch keinen veränderbaren Datentyp eingeführt haben, muss an dieser Stelle auf ein Beispiel verzichtet werden.

Wir werden im Folgenden bei der Einführung neuer Datentypen angeben, zu welcher der beiden Kategorien sie gehören.

»Alles ist Zahl.«
– Pythagoras

8 Basisdatentypen

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

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

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

8.1 Operatoren

Den Begriff des Operators kennen Sie aus der Mathematik, wo er ein Formelzeichen beschreibt, das für eine bestimmte Rechenoperation steht. In Python können Sie Operatoren beispielsweise verwenden, um zwei numerische Werte zu einem arithmetischen Ausdruck zu verbinden:

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

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

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

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

Betrachten Sie einmal folgende Ausdrücke:

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

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

```
a * b + c
```

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

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

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

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

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

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

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

```
a < b < c
```

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

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

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

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

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

ausgewertet zu:

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

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

8.2 Das Nichts – NoneType

Beginnen wir mit dem einfachsten Datentyp überhaupt: dem Nichts. Der dazugehörige Basisdatentyp wird *NoneType* genannt. Es drängt sich natürlich die Frage auf, wieso es eines Datentyps bedarf, der einzig und allein dazu da ist, »nichts« zu repräsentieren. Nun, es ist eigentlich nur konsequent. Stellen Sie sich einmal folgende Situation vor: Sie implementieren ein Verfahren, bei dem jede reelle Zahl

ein mögliches Ergebnis ist. Allerdings kann es in einigen Fällen vorkommen, dass die Berechnung nicht durchführbar ist. Welcher Wert soll als Ergebnis zurückgegeben werden? Richtig: »Nichts«. Auch dass das »Nichts« in Python ein eigener Datentyp ist, hat durchaus seine Berechtigung, denn dadurch kann man Variablen explizit auf den Wert »Nichts« testen.

Kommen wir zur konkreten Verwendung des Datentyps: Es gibt nur eine einzige Instanz des »Nichts« namens `None`. Dies ist eine Konstante, die Sie jederzeit im Quelltext verwenden können:

```
>>> ref = None
>>> ref
>>> print(ref)
None
```

Im Beispiel wurde eine Referenz namens `ref` auf `None` angelegt. Dass `None` tatsächlich dem »Nichts« entspricht, merken wir in der zweiten Zeile: Wir versuchen, `ref` vom Interpreter ausgeben zu lassen, und erhalten tatsächlich kein Ergebnis. Um den Wert dennoch auf dem Bildschirm ausgeben zu können, müssen wir uns des Schlüsselwortes `print` bedienen.

Es wurde bereits gesagt, dass `None` die einzige Instanz des »Nichts« ist. Diese Besonderheit können wir uns zunutze machen, um sehr effizient zu überprüfen, ob eine Referenz auf `None` verweist oder nicht:

```
if ref is None:
    print("ref ist None")
```

Mit dem Schlüsselwort `is` wird überprüft, ob die von `ref` referenzierte Instanz mit `None` identisch ist. Diese Art, einen Wert auf `None` zu testen, kann vom Interpreter schneller ausgeführt werden als der wertbezogene Vergleich mit dem Operator `==`, der selbstverständlich auch möglich ist. Beachten Sie, dass diese beiden Operationen nur in diesem Fall und auch hier nur vordergründig äquivalent sind: Mit `==` werden zwei Werte und mit `is` zwei Identitäten auf Gleichheit geprüft.

8.3 Numerische Datentypen

Die numerischen Datentypen sind eine Kategorie, zu der vier Basisdatentypen gehören: `int` zum Speichern von ganzen Zahlen, `float` für Gleitkommazahlen, `complex` für komplexe Zahlen und `bool` für boolesche Werte. Alle numerischen Datentypen sind immutable, also unveränderlich. Beachten Sie, dass dies nicht bedeutet, dass es keine Operatoren gibt, um Zahlen zu verändern, sondern vielmehr, dass nach jeder Veränderung eine neue Instanz des jeweiligen Datentyps

erzeugt werden muss. Aus Sicht des Programmierers besteht also zunächst kaum ein Unterschied.

Für alle numerischen Datentypen sind folgende Operatoren definiert:

Operator	Ergebnis
$x + y$	Summe von x und y
$x - y$	Differenz von x und y
$x * y$	Produkt von x und y
x / y	Quotient von x und y
$x \% y$	Rest beim Teilen von x durch y (außer bei <code>complex</code>)
$+x$	positives Vorzeichen, lässt x unverändert
$-x$	negatives Vorzeichen – Vorzeichenwechsel bei x
$x ** y$	x hoch y
$x // y$	abgerundeter Quotient von x und y (außer bei <code>complex</code>)

Tabelle 8.1 Gemeinsame Operatoren numerischer Datentypen

Hinweis
Sollten Sie bereits eine C-ähnliche Programmiersprache beherrschen, wundern Sie sich zu Recht, denn in Python gibt es keinen Operator für Inkrementierungen ($x++$) oder Dekrementierungen ($x--$).

Neben diesen grundlegenden Operatoren existiert in Python eine Reihe zusätzlicher Operatoren. Oftmals möchte man beispielsweise die Summe von x und y berechnen und das Ergebnis in x speichern, x also um y erhöhen. Dazu ist mit den obigen Operatoren folgende Anweisung nötig:

`x = x + y`

Für solche Fälle gibt es in Python sogenannte *erweiterte Zuweisungen* (engl. *augmented assignments*), die als eine Art Abkürzung für die obige Anweisung angesehen werden können.

Operator	Entsprechung
$x += y$	<code>x = x + y</code>
$x -= y$	<code>x = x - y</code>
$x *= y$	<code>x = x * y</code>
$x /= y$	<code>x = x / y</code>

Tabelle 8.2 Gemeinsame Operatoren numerischer Datentypen

Operator	Entsprechung
<code>x %= y</code>	<code>x = x % y</code>
<code>x **= y</code>	<code>x = x ** y</code>
<code>x /= y</code>	<code>x = x / y</code>

Tabelle 8.2 Gemeinsame Operatoren numerischer Datentypen (Forts.)

Wichtig ist, dass Sie hier für `y` einen beliebigen arithmetischen Ausdruck einsetzen können, während `x` ein Ausdruck sein muss, der auch als Ziel einer normalen Zuweisung eingesetzt werden könnte.

Für die Datentypen `int`, `float` und `bool` sind außerdem *vergleichende Operatoren* definiert. Da komplexe Zahlen prinzipiell nicht sinnvoll anzuordnen sind, lässt der Datentyp `complex` nur die Verwendung der ersten drei Operatoren zu:

Operator	Ergebnis
<code>==</code>	wahr, wenn <code>x</code> und <code>y</code> gleich sind
<code>!=</code>	wahr, wenn <code>x</code> und <code>y</code> verschieden sind
<code><</code>	wahr, wenn <code>x</code> kleiner ist als <code>y</code> (außer bei <code>complex</code>)
<code><=</code>	wahr, wenn <code>x</code> kleiner oder gleich <code>y</code> ist (außer bei <code>complex</code>)
<code>></code>	wahr, wenn <code>x</code> größer ist als <code>y</code> (außer bei <code>complex</code>)
<code>>=</code>	wahr, wenn <code>x</code> größer oder gleich <code>y</code> ist (außer bei <code>complex</code>)

Tabelle 8.3 Gemeinsame Operatoren numerischer Datentypen

Jeder dieser vergleichenden Operatoren liefert als Ergebnis einen Wahrheitswert. Ein solcher Wert wird zum Beispiel als Bedingung einer `if`-Anweisung erwartet. Die Operatoren könnten also folgendermaßen verwendet werden:

```
if x < 4:
    print("x ist kleiner als 4")
```

Sie können beliebig viele der vergleichenden Operatoren zu einer Reihe verkettet. Das obere Beispiel ist genau genommen nur ein Spezialfall dieser Regel, mit lediglich zwei Operanden. Die Bedeutung einer solchen Verkettung entspricht der mathematischen Sichtweise und ist am folgenden Beispiel zu erkennen:

```
if 2 < x < 4:
    print("x liegt zwischen 2 und 4")
```

Mehr zu booleschen Werten folgt in Abschnitt 8.3.3.

Numerische Datentypen können ineinander umgeformt werden. Dabei können je nach Umformung Informationen verlorengehen. Als Beispiel betrachten wir einige Konvertierungen im interaktiven Modus:

```
>>> float(33)
33.0
>>> int(33.5)
33
>>> bool(12)
True
>>> complex(True)
(1+0j)
```

Allgemein wird zunächst der Name des Datentyps geschrieben, in den konvertiert werden soll, gefolgt von dem zu konvertierenden Wert in Klammern. Statt eines konkreten Literals kann auch eine Referenz eingesetzt bzw. eine Referenz mit dem entstehenden Wert verknüpft werden:

```
>>> var1 = 12.5
>>> int(var1)
12
>>> var2 = int(40.25)
>>> var2
40
```

So viel zur allgemeinen Einführung in die numerischen Datentypen. Die folgenden Abschnitte werden jeden dieser Datentypen im Detail behandeln.

8.3.1 Ganzzahlen – int

Für den Raum der ganzen Zahlen gibt es in Python den Datentyp `int`. Im Gegensatz zu vielen anderen Programmiersprachen unterliegt dieser Datentyp in seinem Wertebereich keinen prinzipiellen Grenzen, was den Umgang mit großen ganzen Zahlen in Python sehr komfortabel macht.¹

Wir haben bereits viel mit ganzen Zahlen gearbeitet, so dass die Verwendung von `int` eigentlich keiner Demonstration mehr bedarf. Der Vollständigkeit halber dennoch ein kleines Beispiel:

¹ Dies ist eine Neuerung in Python 3.0. Zuvor existierten zwei Datentypen für ganze Zahlen: `int` für den begrenzten Zahlenraum von -2^{31} bis $2^{31}-1$ (auf 32-Bit-Systemen) sowie `long` mit einem unbegrenzten Wertebereich. Eine `int`-Instanz wurde jedoch auch schon in älteren Python-Versionen automatisch nach `long` konvertiert, wenn der Zahlenraum von `int` gesprengt wurde, so dass die Python-Entwickler keinen Sinn mehr darin sahen, die beiden Datentypen zu trennen.

```
>>> i = 1234
>>> i
1234
>>> p = int(5678)
>>> p
5678
```

Zahlensysteme

Ganze Zahlen können in Python in mehreren Zahlensystemen geschrieben werden:

- Zahlen, die, wie im obigen Beispiel, ohne ein spezielles Präfix geschrieben sind, werden im *Dezimalsystem* (Basis 10) interpretiert. Zu beachten ist, dass einer solchen Zahl keine führenden Nullen vorangestellt werden dürfen:

```
v_dez = 1337
```

- Das Präfix `0o` (»Null-o«) kennzeichnet eine Zahl, die im *Oktalsystem* (Basis 8) geschrieben wurde. Die Verwendung des Oktalsystems ist ein Relikt aus älteren Zeiten und wird heute kaum noch benötigt. Beachten Sie, dass hier nur Ziffern von 0 bis 7 erlaubt sind:

```
v_okt = 0o2471
```

Das kleine »o« im Präfix kann auch durch ein großes »O« ersetzt werden. Wir empfehlen hier jedoch, stets ein kleines »o« zu verwenden, da das große »O« in vielen Schriftarten von der »0« kaum zu unterscheiden ist.²

- Die nächste und weitaus gebräuchlichere Variante ist das *Hexadezimalsystem* (Basis 16), das durch das Präfix `0x` bzw. `0X` gekennzeichnet wird. Die Zahl selbst darf aus den Ziffern 0–9 und den Buchstaben A–F bzw. a–f gebildet werden:

```
v_hex = 0x5A3F
```

- Neben dem Hexadezimalsystem ist in der Informatik das *Dualsystem* (Basis 2) von entscheidender Bedeutung. Seit Version 3.0 unterstützt Python ein eigenes Literal für Dualzahlen. Diese werden analog zu den vorangegangenen Literalen durch das Präfix `0b` eingeleitet:

```
v_bin = 0b1101
```

Beachten Sie, dass Sie im Dualsystem nur die Ziffern 0 und 1 verwenden dürfen.

2 Bis zu Version 3.0 wurde in Python, wie beispielsweise in C auch, die »O« als Präfix für Oktalzahlen verwendet.

Für alle diese Literale ist die Verwendung eines negativen Vorzeichens möglich:

```
>>> -1234
-1234
>>> -0o777
-511
>>> -0xFF
-255
>>> -0b1010101
-85
```

Vielleicht möchten Sie sich nicht auf diese vier Zahlensysteme beschränken, die von Python explizit unterstützt werden, sondern ein exotischeres verwenden. Natürlich gibt es in Python nicht für jedes mögliche Zahlensystem ein eigenes Literal. Stattdessen können Sie sich folgender Schreibweise bedienen:

```
v_6 = int("54425", 6)
```

Es handelt sich um eine alternative Methode, eine Instanz des Datentyps `int` zu erzeugen und mit einem Anfangswert zu versehen. Dazu werden in den Klammern ein String, der den gewünschten Initialwert in dem gewählten Zahlensystem enthält, sowie die Basis dieses Zahlensystems als ganze Zahl geschrieben. Beide Werte müssen durch ein Komma getrennt werden. Im Beispiel wurde das Sechssystem verwendet.

Python unterstützt Zahlensysteme mit einer Basis von 2 bis 36. Wenn ein Zahlensystem mehr als zehn verschiedene Ziffern zur Darstellung einer Zahl benötigt, werden zusätzlich zu den Ziffern 0 bis 9 die Buchstaben A bis Z des englischen Alphabets verwendet.

`v_6` hat jetzt den Wert 7505 (im Dezimalsystem).

Beachten Sie, dass es sich bei den Zahlensystemen nur um eine alternative Schreibweise des gleichen Wertes handelt. Der Datentyp `int` springt beispielsweise nicht in eine Art Hexadezimalmodus, sobald er einen solchen Wert enthält. Ein Zahlensystem ist nur bei Wertzuweisungen oder -ausgaben von Bedeutung. Standardmäßig werden alle Zahlen im Dezimalsystem ausgegeben:

```
>>> v1 = 0xFF
>>> v2 = 0o777
>>> v1
255
>>> v2
511
```

Wir werden später, im Zusammenhang mit Strings, darauf zurückkommen, wie sich Zahlen in anderen Zahlensystemen ausgeben lassen.

Bit-Operationen

Wie bereits gesagt, hat das Dualsystem, oder auch Binärsystem, in der Informatik eine große Bedeutung. Für den Datentyp `int` sind daher einige zusätzliche Operatoren definiert, die sich explizit auf die binäre Darstellung der Zahl beziehen:

Operator	Ergebnis
<code>x & y</code>	bitweises UND von <code>x</code> und <code>y</code> (AND)
<code>x y</code>	bitweises nicht ausschließendes ODER von <code>x</code> und <code>y</code> (OR)
<code>x ^ y</code>	bitweises ausschließendes ODER von <code>x</code> und <code>y</code> (XOR)
<code>~x</code>	bitweises Komplement von <code>x</code>
<code>x << n</code>	Bitverschiebung um <code>n</code> Stellen nach links
<code>x >> n</code>	Bitverschiebung um <code>n</code> Stellen nach rechts

Tabelle 8.4 Bit-Operatoren der Datentypen `int` und `long`

Auch hier sind erweiterte Zuweisungen mithilfe der folgenden Operatoren möglich:

Operator	Entsprechung
<code>x &= y</code>	<code>x = x & y</code>
<code>x = y</code>	<code>x = x y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>
<code>x <<= n</code>	<code>x = x << n</code>
<code>x >>= n</code>	<code>x = x >> n</code>

Tabelle 8.5 Bit-Operatoren der Datentypen `int` und `long`

Da vielleicht nicht jedem unmittelbar klar ist, was die einzelnen Operationen bewirken, möchten wir sie im Folgenden im Detail besprechen.

Das *bitweise UND* zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verknüpft werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine 1, wo beide der jeweiligen Bits der Operanden 1 sind, und sonst eine 0. Dies veranschaulicht Abbildung 8.1:

Dual								Dezimal	
	0	1	1	0	1	0	1	0	106
&	0	0	0	0	1	1	0	0	12
	0	0	0	0	1	0	0	0	8

Abbildung 8.1 Bitweises UND

Im interaktiven Modus von Python probieren wir aus, ob das bitweise UND mit den in der Grafik gewählten Operanden tatsächlich das erwartete Ergebnis zurückgibt:

```
>>> 106 & 12
8
```

Diese Prüfung des Ergebnisses werden wir nicht für jede Operation einzeln durchführen. Um allerdings mit den bitweisen Operatoren vertrauter zu werden, lohnt es sich, hier ein wenig zu experimentieren.

Das *bitweise ODER* zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verglichen werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine 1, wo mindestens eines der jeweiligen Bits der Operanden 1 ist. Abbildung 8.2 veranschaulicht dies.

Dual								Dezimal	
	0	1	1	0	1	0	1	0	106
I	0	0	0	0	1	1	0	0	12
	0	1	1	0	1	1	1	0	110

Abbildung 8.2 Bitweises nicht ausschließendes ODER

Das *bitweise ausschließende ODER* (auch *exklusives ODER*) zweier Zahlen wird gebildet, indem beide Zahlen in ihrer Binärdarstellung Bit für Bit miteinander verglichen werden. Die resultierende Zahl hat in ihrer Binärdarstellung genau da eine 1, wo sich die jeweiligen Bits der Operanden voneinander unterscheiden, und eine 0, wo sie gleich sind. Dies zeigt Abbildung 8.3.

Dual								Dezimal	
0 1 1 0 1 0 1 0								106	
^	0 0 0 0 1 1 0 0							12	
0 1 1 0 0 1 1 0								102	

Abbildung 8.3 Bitweises exklusives ODER

Das *bitweise Komplement* bildet das sogenannte Einerkomplement einer Dualzahl, das der Negation aller vorkommenden Bits entspricht. In Python ist dies auf Bit-Ebene nicht möglich, da eine ganze Zahl in ihrer Länge unbegrenzt ist und das Komplement immer in einem abgeschlossenen Zahlenraum gebildet werden muss. Deswegen wird die eigentliche Bit-Operation zur arithmetischen Operation und ist folgendermaßen definiert:

$$\sim x = -x - 1^3$$

Bei der *Bitverschiebung* wird die Bitfolge in der binären Darstellung des ersten Operanden um die durch den zweiten Operanden gegebene Anzahl Stellen nach links bzw. rechts verschoben. Die entstandene Lücke wird mit Nullen gefüllt.

Abbildung 8.4 und Abbildung 8.5 veranschaulichen eine Verschiebung um zwei Stellen nach links bzw. nach rechts.

Dual								Dezimal	
0 1 1 0 1 0 1 1								107	
0 1 1 0 1 0 1 1 0 0								428	

n = 2

Abbildung 8.4 Bitverschiebung um zwei Stellen nach links

3 Das ist sinnvoll, da man zur Darstellung negativer Zahlen in abgeschlossenen Zahlenräumen das sogenannte Zweierkomplement verwendet. Dieses erhält man, indem man zum Einerkomplement 1 addiert.

Also: $-x$ = Zweierkomplement von $x = \sim x + 1$

Daraus folgt: $\sim x = -x - 1$

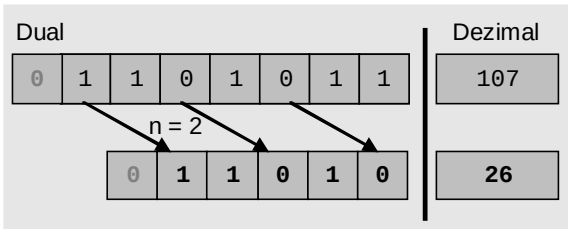


Abbildung 8.5 Bitverschiebung um zwei Stellen nach rechts

Die in der Bitdarstellung entstehenden Lücken auf der rechten bzw. linken Seite werden mit Nullen aufgefüllt.

Beachten Sie, dass auch die Bitverschiebung in Python arithmetisch implementiert ist. Ein Shift um x Stellen nach rechts entspricht einer ganzzahligen Division mit 2^x . Ein Shift um x Stellen nach links entspricht einer Multiplikation mit 2^x . Diese Regeln werden insbesondere auch bei einem Bitshift auf einer negativen Zahl angewandt, bei der das obige Modell nicht ganz stimmig ist.

8.3.2 Gleitkommazahlen – float

Zu Beginn dieses Teils des Buches sind wir bereits oberflächlich auf Gleitkommazahlen eingegangen, was wir hier ein wenig vertiefen möchten. Zum Speichern einer Gleitkommazahl mit begrenzter Genauigkeit wird der Datentyp `float` verwendet.

Wie bereits besprochen wurde, sieht eine Gleitkommazahl im einfachsten Fall folgendermaßen aus:

```
v = 3.141
```

Python unterstützt außerdem eine Notation, die es ermöglicht, die Exponentialschreibweise zu verwenden:

```
v = 3.141e-12
```

Durch ein kleines oder großes e wird die *Mantisse* (3.141) vom *Exponenten* (-12) getrennt. Übertragen in die mathematische Schreibweise, entspricht $3.141e-12$ $3.141 \cdot 10^{-12}$. Beachten Sie, dass sowohl die Mantisse als auch der Exponent im Dezimalsystem anzugeben sind. Andere Zahlensysteme sind nicht vorgesehen, was die gefahrlose Verwendung von führenden Nullen ermöglicht:

```
v = 03.141e-0012
```

Es gibt noch weitere Varianten, eine gültige Gleitkommazahl zu definieren. Es handelt sich dabei um Spezialfälle der obigen Notation, weswegen sie etwas exo-

tisch wirken. Sie sollen der Vollständigkeit halber trotzdem erwähnt werden. Pythons interaktiver Modus gibt nach jeder Eingabe ihren Wert aus. Das machen wir uns zunutze und lassen zu jedem Spezialfall den normal formatierten Wert automatisch ausgeben:

```
>>> -3.
-3.0
>>> .001
0.001
>>> 3e2
300.0
```

Eventuell haben Sie gerade schon etwas mit den Gleitkommazahlen experimentiert und sind dabei auf einen vermeintlichen Fehler des Interpreters gestoßen:

```
>>> 0.9
0.90000000000000002
```

Aufgrund der Begrenztheit von `float` können reelle Zahlen nicht unendlich präzise gespeichert werden. Stattdessen werden sie mit einer bestimmten Genauigkeit angenähert. In diesem Fall konnte keine präzisere Annäherung an die `0.9` gefunden werden. Es ist unter Verwendung der Basisdatentypen nicht möglich, mit beliebig genauen Dezimalzahlen zu rechnen. Dazu muss die Standardbibliothek bemüht werden, was wir zu gegebener Zeit behandeln werden.⁴

Gleitkommazahlen können nicht beliebig genau gespeichert werden. Das impliziert auch, dass es sowohl eine Ober- als auch eine Untergrenze für diesen Datentyp geben muss. Und tatsächlich können Gleitkommazahlen, die in ihrer Größe ein bestimmtes Limit überschreiten, in Python nicht mehr dargestellt werden. Wenn das Limit überschritten wird, wird die Zahl als `inf` gespeichert, bzw. als `-inf`, wenn das untere Limit unterschritten wurde. Es kommt also zu keinem Fehler, und es ist immer noch möglich, eine übergroße Zahl mit anderen zu vergleichen:

```
>>> 3.0e999
inf
>>> -3.0e999
-inf
>>> 3.0e999 < 12.0
False
>>> 3.0e999 > 12.0
True
```

4 Dabei handelt es sich um das Modul `decimal`, das in Abschnitt »Präzise Dezimalzahlen – `decimal`« behandelt wird.

```
>>> 3.0e999 == 3.0e9999999999999
True
```

Es ist zwar möglich, zwei unendlich große Gleitkommazahlen miteinander zu vergleichen, jedoch lässt sich nur bedingt mit ihnen rechnen. Dazu folgendes Beispiel:

```
>>> 3.0e999 + 1.5e999999
inf
>>> 3.0e999 - 1.5e999999
nan
>>> 3.0e999 * 1.5e999999
inf
>>> 3.0e999 / 1.5e999999
nan
```

Zwei unendlich große Gleitkommazahlen lassen sich problemlos addieren oder multiplizieren. Das Ergebnis ist in beiden Fällen wieder `inf`. Ein Problem gibt es aber, wenn versucht wird, zwei solche Zahlen zu subtrahieren bzw. zu dividieren. Da diese Rechenoperationen nicht sinnvoll sind, ergeben sie `nan`. Der Status `nan` ist vom Typ her ähnlich wie `inf`, bedeutet jedoch »not a number«, also so viel wie »nicht berechenbar«.

Beachten Sie, dass weder `inf` noch `nan` eine Konstante ist, die Sie selbst in einem Python-Programm verwenden könnten.

8.3.3 Boolesche Werte – bool

Eine Instanz des Datentyps `bool` kann nur zwei verschiedene Werte annehmen: »wahr« oder »falsch« oder, um innerhalb der Python-Syntax zu bleiben, `True` bzw. `False`. Deshalb ist es auf den ersten Blick absurd, `bool` den numerischen Datentypen unterzuordnen. Python sieht hier jedoch `True` analog zur 1 und `False` analog zur 0, so dass sich mit booleschen Werten genauso rechnen lässt wie beispielsweise schon mit den ganzen Zahlen. Bei den Namen `True` und `False` handelt es sich um Konstanten, die im Quelltext verwendet werden können. Zu beachten ist besonders, dass die Konstanten mit einem Großbuchstaben beginnen:

```
v1 = True
v2 = False
```

Logische Operatoren

Ein oder mehrere boolesche Werte lassen sich mithilfe von bestimmten Operatoren zu einem booleschen Ausdruck kombinieren. Ein solcher Ausdruck resultiert, wenn er ausgewertet wurde, wieder in einem booleschen Wert, also in `True` oder

False. Bevor es zu theoretisch wird, folgt hier zunächst die Tabelle der sogenannten *logischen Operatoren*, und darunter sehen Sie weitere Erklärungen mit konkreten Beispielen.

Operator	Ergebnis
not x	logische Negierung von x
x and y	logisches UND zwischen x und y
x or y	logisches (nicht ausschließendes) ODER zwischen x und y

Tabelle 8.6 Logische Operatoren des Datentyps bool

Die *logische Negierung* eines booleschen Wertes ist schnell erklärt: Der entsprechende Operator `not` macht `True` zu `False` und `False` zu `True`. In einem konkreten Beispiel würde das folgendermaßen aussehen:

```
if not x:
    print("x ist False")
else:
    print("x ist True")
```

Das *logische UND* zwischen zwei Wahrheitswerten ergibt nur dann `True`, wenn beide Operanden bereits `True` sind. In der folgenden Tabelle sind alle möglichen Fälle aufgelistet:

x	y	Ausdruck: a and b
True	True	True
False	True	False
True	False	False
False	False	False

Tabelle 8.7 Mögliche Fälle des logischen UNDS

In einem konkreten Beispiel würde die Anwendung des logischen UNDS so aussehen:

```
if x and y:
    print("x und y sind True")
```

Das *logische ODER* zwischen zwei Wahrheitswerten ergibt genau dann eine wahre Aussage, wenn mindestens einer der beiden Operanden wahr ist. Es handelt sich demnach um ein nicht ausschließendes ODER. Ein Operator für ein logisches ausschließendes (exklusives) ODER existiert in Python nicht. Folgende Tabelle listet alle möglichen Fälle auf:

x	y	Ausdruck: a or b
True	True	True
False	True	True
True	False	True
False	False	False

Tabelle 8.8 Mögliche Fälle des logischen ODERs

Ein logisches ODER könnte folgendermaßen implementiert werden:

```
if x or y:
    print("x oder y ist True")
```

Selbstverständlich können Sie all diese Operatoren miteinander kombinieren und in einem komplexen Ausdruck verwenden. Das könnte etwa folgendermaßen aussehen:

```
if x and y or y and z and not x:
    print("Holla die Waldfee")
```

Wir möchten diesen Ausdruck hier nicht im Einzelnen besprechen. Es sei nur gesagt, dass der Einsatz von Klammern den erwarteten Effekt hat, nämlich dass umklammerte Ausdrücke zuerst ausgewertet werden. Die folgende Tabelle zeigt den Wahrheitswert des Ausdrucks auf, und zwar in Abhängigkeit von den drei Parametern x, y und z:

x	y	z	Ausdruck: x and y or y and z and not x
True	True	True	True
False	True	True	True
True	False	True	False
True	True	False	True
False	False	True	False
False	True	False	False
True	False	False	False
False	False	False	False

Tabelle 8.9 Mögliche Ergebnisse des Ausdrucks

Zu Beginn des Abschnitts über numerische Datentypen haben wir einige vergleichende Operatoren eingeführt, die eine Wahrheitsaussage in Form eines booleschen Wertes ergeben. Das folgende Beispiel zeigt, dass diese ganz selbstverständlich zusammen mit den logischen Operatoren verwendet werden können:

```
if x > y or (y > z and x != 0):
    print("Mein lieber Schwan")
```

In diesem Fall muss es sich bei `x`, `y` und `z` um Variablen der Typen `int`, `float` oder auch `bool` handeln.

Wahrheitswerte anderer Datentypen

In Python lassen sich Instanzen eines jeden Basisdatentyps in einen booleschen Wert überführen. Dies ist eine sinnvolle Eigenschaft, da sich eine Instanz der Basisdatentypen häufig in zwei Stadien befinden kann: »leer« und »nicht leer«. Oftmals möchte man beispielsweise testen, ob ein String Buchstaben enthält oder nicht. Da ein String in einen booleschen Wert konvertiert werden kann, wird ein solcher Test sehr einfach durch logische Operatoren möglich:

```
>>> not ""
True
>>> not "abc"
False
```

Durch Verwendung eines logischen Operators wird der Operand automatisch als Wahrheitswert interpretiert.

Für jeden Basisdatentyp wurde ein bestimmter Wert als `False` definiert. Alle davon abweichenden Werte sind `True`. Die folgende Tabelle listet für jeden Datentyp den entsprechenden `False`-Wert auf. Einige der Datentypen wurden noch nicht eingeführt, woran Sie sich an dieser Stelle jedoch nicht weiter stören sollten.

Basisdatentyp	False-Wert	Beschreibung
<code>NoneType</code>	<code>None</code>	der Wert <code>None</code>
Numerische Datentypen		
<code>int</code> , <code>long</code>	<code>0</code>	der Wert <code>Null</code>
<code>float</code>	<code>0.0</code>	der Wert <code>Null</code>
<code>complex</code>	<code>0 + 0j</code>	der Wert <code>Null</code>
Sequentielle Datentypen		
<code>str</code>	<code>""</code>	ein leerer String
<code>list</code>	<code>[]</code>	eine leere Liste
<code>tuple</code>	<code>()</code>	ein leeres Tupel

Tabelle 8.10 Wahrheitswerte anderer Datentypen

Basisdatentyp	False-Wert	Beschreibung
Assoziative Datentypen		
dict	{ }	ein leeres Dictionary
Mengen		
set, frozenset	set(), frozenset()	eine leere Menge

Tabelle 8.10 Wahrheitswerte anderer Datentypen (Forts.)

Alle anderen Werte ergeben `True`.

Betrachten wir die Konvertierung eines Wertes in einen Wahrheitswert anhand einiger Gleitkommazahlen:

```
>>> bool(0.0)
False
>>> bool(0.0e12)
False
>>> bool(1.0)
True
>>> bool(123.456)
True
```

Auswertung logischer Operatoren

Python wertet logische Ausdrücke grundsätzlich von links nach rechts aus, also im folgenden Beispiel zuerst `a` und dann `b`:

```
if a or b:
    print("a oder b sind True")
```

Es wird aber nicht garantiert, dass jeder Teil des Ausdrucks tatsächlich ausgewertet wird. Aus Optimierungsgründen bricht Python die Auswertung des Ausdrucks sofort ab, wenn das Ergebnis feststeht. Wenn im obigen Beispiel also `a` bereits den Wert `True` hat, ist der Wert von `b` nicht weiter von Belang; `b` würde dann nicht mehr ausgewertet. Dieses Detail scheint unwichtig, kann aber zu schwer auffindbaren Fehlern führen.

Zu Beginn dieses Kapitels wurde gesagt, dass ein boolescher Ausdruck stets einen booleschen Wert ergibt, wenn er ausgewertet wurde. Das ist nicht ganz korrekt, denn auch hier wurde die Arbeitsweise des Interpreters in einer Weise optimiert, über die man Bescheid wissen sollte. Deutlich wird dies an folgendem Beispiel aus dem interaktiven Modus:

```
>>> 0 or 1
1
```

Nach dem, was wir bisher besprochen haben, sollte das Ergebnis des Ausdrucks `True` sein, was mitnichten der Fall ist. Stattdessen gibt Python hier den ersten Operanden mit dem Wahrheitswert `True` zurück. Das ist um einiges effizienter, da keine neue Instanz erzeugt werden muss, und hat in vielen Fällen trotzdem den erwünschten Effekt, denn der zurückgegebene Wert wird problemlos automatisch in den Wahrheitswert `True` überführt. Die Auswertung der beiden Operatoren `or` und `and` läuft dabei folgendermaßen ab:

- ▶ Das logische ODER (`or`) nimmt den Wert des ersten Operanden an, der den Wahrheitswert `True` besitzt, oder – wenn es einen solchen nicht gibt – den Wert des letzten Operanden.
- ▶ Das logische UND (`and`) nimmt den Wert des ersten Operanden an, der den Wahrheitswert `False` besitzt, oder – wenn es einen solchen nicht gibt – den Wert des letzten Operanden.

Diese Details haben dabei auch durchaus ihren unterhaltsamen Wert:

```
>>> "Python" or "Java"
'Python'
```

8.3.4 Komplexe Zahlen – complex

Überraschenderweise findet sich ein Datentyp zur Speicherung komplexer Zahlen unter Pythons Basisdatentypen. In vielen Programmiersprachen würden komplexe Zahlen eher eine Randnotiz in der Standardbibliothek darstellen oder ganz außen vor bleiben. Sollten Sie nicht mit komplexen Zahlen vertraut sein, können Sie dieses Kapitel gefahrlos überspringen. Es wird nichts behandelt, was für das weitere Erlernen von Python vorausgesetzt würde.

Komplexe Zahlen bestehen aus einem Realteil und einem Imaginärteil, der aus einer reellen Zahl besteht, die mit der imaginären Einheit j multipliziert wird. Das in der Mathematik eigentlich übliche Symbol der imaginären Einheit ist i . Python hält sich hier an die Notationen der Elektrotechnik. Die imaginäre Einheit j kann als Lösung der Gleichung

$$j^2 = -1$$

verstanden werden. Im folgenden Beispiel weisen wir einer komplexen Zahl den Namen `v` zu:

```
v = 4j
```

Wenn man, wie im Beispiel, nur einen Imaginärteil angibt, wird der Realteil automatisch als 0 angenommen. Um den Realteil festzulegen, wird dieser zum Imaginärteil addiert. Die beiden folgenden Schreibweisen sind äquivalent:

```
v1 = 3 + 4j
v2 = 4j + 3
```

Statt des kleinen `j` ist auch ein großes `J` als Literal für den Imaginärteil einer komplexen Zahl zulässig. Entscheiden Sie hier ganz nach Ihren Vorlieben, welche der beiden Möglichkeiten Sie verwenden möchten.

Sowohl der Real- als auch der Imaginärteil kann eine beliebige reelle Zahl sein, also Instanzen der Typen `int` oder `float`. Folgende Schreibweise ist demnach auch korrekt:

```
v3 = 3.4 + 4e2j
```

Zu Beginn des Abschnitts über numerische Datentypen wurde bereits angedeutet, dass sich komplexe Zahlen von den anderen numerischen Datentypen unterscheiden. Da für komplexe Zahlen keine mathematische Reihenfolge definiert ist, können Instanzen des Datentyps `complex` nur auf Gleichheit oder Ungleichheit überprüft werden. Die Menge der vergleichenden Operatoren ist also auf `==` und `!=` beschränkt.

Des Weiteren sind sowohl der Modulo-Operator `%` als auch der Operator `//` für eine ganzzahlige Division im Komplexen zwar formal möglich, haben jedoch keinen mathematischen Sinn. Deswegen ist ihre Verwendung mit komplexen Operanden seit Python 3.0 nicht mehr möglich.

Der Datentyp `complex` besitzt zwei sogenannte *Attribute*, die das Arbeiten mit ihm erheblich erleichtern. Es kommt zum Beispiel vor, dass man Berechnungen nur mit dem Realteil oder nur mit dem Imaginärteil der gespeicherten Zahl anstellen möchte. Um einen der beiden Teile zu isolieren, erlaubt Python folgende Notationen, die hier exemplarisch an einer Referenz auf eine komplexe Zahl namens `x` gezeigt werden:

Attribut	Beschreibung
<code>x.real</code>	Realteil von <code>x</code> als reelle Zahl (<code>float</code>)
<code>x.imag</code>	Imaginärteil von <code>x</code> als reelle Zahl (<code>float</code>)

Tabelle 8.11 Attribute des Datentyps `complex`

Diese können im Code ganz selbstverständlich verwendet werden:

```
>>> c = 23 + 4j
>>> c.real
23.0
>>> c.imag
4.0
```


Wir werden im Zusammenhang mit objektorientierter Programmierung in Abschnitt 12 darauf zurückkommen und näher darauf eingehen, was ein Attribut genau ist.

Außer über seine zwei Attribute verfügt der Datentyp `complex` über eine sogenannte *Methode*, die in der Tabelle exemplarisch für eine Referenz auf eine komplexe Zahl namens `x` erklärt wird.

Methode	Beschreibung
<code>x.conjugate()</code>	Liefert die zu <code>x</code> konjugiert komplexe Zahl.

Tabelle 8.12 Methoden des Datentyps `complex`

Im Quelltext kann eine Methode ähnlich einfach verwendet werden wie ein Attribut:

```
>>> c = 23 + 4j
>>> c.conjugate()
(23-4j)
```

Das Ergebnis von `conjugate` ist wieder eine komplexe Zahl, der selbstverständlich ein Name zugewiesen werden kann. Außerdem verfügt natürlich auch das Ergebnis über eine Methode `conjugate`:

```
>>> c = 23 + 4j
>>> c2 = c.conjugate()
>>> c2
(23-4j)

>>> c3 = c2.conjugate()
>>> c3
(23+4j)
```

Näheres zur Verwendung von Methoden erfahren Sie im nächsten Abschnitt.

8.4 Methoden und Parameter

Die bisher behandelten numerischen Datentypen waren sehr einfach aufgebaut: Ihre Werte ließen sich mit einer Zahl oder – bei `complex` – mit zwei Zahlen beschreiben, und der Umgang mit ihnen beschränkte sich auf Rechen-, Bit- und Vergleichsoperationen.

Im Folgenden werden wir uns mit umfassenderen Datentypen beschäftigen, für die es Operationen gibt, die nicht durch solche Operatoren abgebildet werden

können. Um diese Funktionalität trotzdem zu ermöglichen, bedient man sich sogenannter *Methoden*. Methoden beziehen sich immer auf Instanzen bestimmter Datentypen und werden durch einen sogenannten *Methodenaufruf* verwendet. Der Aufruf einer Methode sieht folgendermaßen aus:

```
referenz.methode()
```

Das bedeutet dann: »Führe die von `methode` definierten Operationen mit der Instanz aus, auf die `referenz` verweist.« Welche Methoden für eine Instanz verfügbar sind, hängt von ihrem Datentyp ab.

Viele Methoden benötigen neben der Instanz weitere Informationen, um zu funktionieren. Hierfür gibt es sogenannte *Parameter*, die durch Kommata getrennt in die Klammern am Ende des Methodenaufrufs geschrieben werden:

```
referenz.methode(parameter1, parameter2)
```

Als Parameter können formal sowohl Referenzen als auch Literale verwendet werden:

```
var = 12
referenz.methode(var, "Hallo Welt!")
```

Es gibt auch *optionale Parameter*, die nur bei Bedarf übergeben werden müssen. Wenn wir Methoden mit solchen Parametern einführen, werden diese in der Parameterliste durch eckige Klammern gekennzeichnet:

```
referenz.methode(param1, param2[, param3])
```

In diesem Beispiel wären `param1` und `param2` reguläre, d. h. erforderliche Parameter, und `param3` wäre ein optionaler Parameter. Die Methode könnte also mit zwei verschiedenen Konfigurationen aufgerufen werden:

```
referenz.methode(1, 2, 3)
referenz.methode(1, 2)
```

Bei dem ersten Aufruf wäre der Wert 3 für den optionalen Parameter `param3` übergeben worden, während er beim zweiten ausgelassen wurde.

Bei den bisher besprochenen Parameterübergaben war immer die Position eines Übergabewertes entscheidend dafür, für welchen formalen Parameter er eingesetzt wurde. Im letzten Beispiel stand die 1 als Übergabewert an erster Stelle und wurde dadurch mit dem ersten Parameter der Liste, `param1`, verknüpft. Gleiches gilt für die 2 und `param2`.

Man kann einer Methode Parameter auch als sogenannte *Schlüsselwortparameter* (engl. *keyword arguments*) übergeben. Schlüsselwortparameter werden direkt mit dem formalen Parameternamen verknüpft, und ihre Reihenfolge in der Liste

spielt keine Rolle mehr. Um einen Wert als Schlüsselwortparameter zu übergeben, weist man dem Parameternamen innerhalb des Aufrufs den zu übergebenden Wert mithilfe des Gleichheitszeichens zu. Die beiden folgenden Methodenaufrufe sind demnach vollkommen gleichwertig:

```
referenz.methode(1, 2, 3)
referenz.methode(param2=2, param1=1, param3=3)
```

Wie Sie sehen, spielt es dabei keine Rolle, ob es sich bei solchen Übergaben um optionale oder um erforderliche Parameter handelt. Man kann auch positions- und schlüsselwortbezogene Parameter mischen, wobei allerdings alle Schlüsselwortparameter am Ende der Parameterliste stehen müssen. Damit ist der nachstehende Aufruf äquivalent zu den beiden vorhergehenden:

```
referenz.methode(1, param3=3, param2=2)
```

`param1` wurde als positionsbezogener Parameter übergeben, während `param2` und `param3` als Schlüsselwortparameter übergeben wurden.

Welche der beiden Übergabemethoden man in der Praxis bevorzugt, ist größtenteils Geschmackssache. Schlüsselwortparameter haben den Vorteil, dass man nicht an die Reihenfolge der Parameter in der Funktionsdefinition gebunden ist. Deshalb bleiben solche Aufrufe auch dann noch korrekt, wenn sich die Reihenfolge der Parameterliste ändert. Außerdem sieht man schon an der Stelle des Aufrufs anhand des Parameternamens, wofür der übergebene Wert innerhalb der Funktion benutzt wird. Dadurch kann man die Lesbarkeit eines Programms verbessern. Demgegenüber ist die Übergabe positionsbezogener Parameter mit weniger Schreibaufwand verbunden, weil nicht immer der Parametername mit angegeben werden muss. Wenn sich die Namen der formalen Parameter in der Funktionsdefinition ändern, funktionieren positionsbezogene Übergaben auch ohne Änderung, wohingegen Übergaben mit Schlüsselwortparametern angepasst werden müssen. Es ist in der Regel so, dass positionsbezogene Parameter häufiger verwendet werden, was wahrscheinlich an dem geringeren Schreibaufwand liegt.

Die meisten Methoden erzeugen ein Ergebnis, das uns als Aufrufendem zur Verfügung steht. Beispielsweise verfügen Zeichenketten über eine Methode `lower`, mit deren Hilfe Sie einen neuen String erzeugen, in dem alle Großbuchstaben des Ursprungsstrings in Kleinbuchstaben konvertiert wurden:

```
>>> s = "DaS sIeHt AbEr KoMiSch aUs"
>>> s.lower()
'das sieht aber komisch aus'
```

Bei diesem sogenannten *Rückgabewert* der Methode handelt es sich um eine neue Instanz, in diesem Fall um eine String-Instanz, die wir wie gewohnt mit Referenzen versehen und anschließend weiterverwenden können:

```
>>> s = "DaS sIeHt AbEr KoMiSch aUs"
>>> low = s.lower()
>>> low.upper()
'DAS SIEHT ABER KOMISCH AUS'
```

Die Referenz `low` zeigt auf die von `s.lower()` zurückgegebene String-Instanz mit dem Wert `'das sieht aber komisch aus'`. Wie alle Strings besitzt diese ihrerseits eine Methode `upper`, die alle Kleinbuchstaben in Großbuchstaben umwandelt und von uns mit `low.upper()` aufgerufen wird.

Neben diesen Methoden, die immer an einen bestimmten Datentyp gebunden sind, existieren Operationen, die global und damit unabhängig von bestimmten Typen zur Verfügung stehen. Sie werden *Built-in Functions* (dt. »eingebaute Funktionen«) genannt und sind fast genauso zu verwenden wie Methoden, außer dass ihnen keine Referenz auf eine Instanz vorangestellt werden muss, und auch der Punkt entfällt. Die Instanz, auf die sich die Operation bezieht, wird in der Regel als Parameter übergeben:

```
builtin_name(referenz)
```

Sie haben schon solche Funktionen kennengelernt, wie zum Beispiel `type` und `id` in Abschnitt 7.1, »Die Struktur von Instanzen«:

```
>>> t = type(1337)
>>> t
<class 'int'>
```

8.5 Sequentielle Datentypen

Unter *sequentiiellen Datentypen* wird eine Klasse von Datentypen zusammengefasst, die Folgen von gleichartigen oder verschiedenen *Elementen* verwalten. Die in sequentiellen Datentypen gespeicherten Elemente haben eine definierte Reihenfolge, und man kann über eindeutige Indizes auf sie zugreifen.

Python stellt im Wesentlichen die folgenden vier sequentiellen Typen zu Verfügung: `str`, `bytes`, `list` und `tuple`.

Die ersten beiden sequentiellen Datentypen, `str` und `bytes` ermöglichen in Python die Arbeit mit Zeichenketten, also Folgen von Buchstaben, wobei je nach Anwendungsfall einer von ihnen besser geeignet ist. Instanzen des Typs `bytes`

speichern Folgen von Bytes, weshalb er sich besonders zum Speichern binärer Datenströme eignet. Der Datentyp `str` ist für die Speicherung von Text-Strings konzipiert und speichert Folgen von Zeichen in einem speziellen Unicode-Format, das auch die komfortable Verwaltung von speziellen Sonderzeichen wie den deutschen Umlauten oder dem Eurozeichen ermöglicht.⁵

Beide Datentypen sind `immutable`, ihr Wert kann sich nach der Instantiierung also nicht mehr verändern. Trotzdem können Sie komfortabel mit Strings arbeiten. Bei Änderungen wird nur nicht der Ursprungsstring verändert, sondern stets ein neuer String erzeugt.

Die Typen `list` und `tuple` können Folgen beliebiger Instanzen speichern. Der wesentliche Unterschied zwischen den beiden fast identischen Datentypen ist, dass eine Liste nach ihrer Erzeugung verändert werden kann, während ein Tupel keine Änderung des Anfangsinhalts zulässt: `list` ist ein `mutable`, `tuple` ein `immutable` Datentyp.

Für jede Instanz eines sequentiellen Datentyps gibt es einen Grundstock von Operatoren und Methoden, der immer verfügbar ist. Der Einfachheit halber werden wir diesen allgemein am Beispiel von `str`-Instanzen einführen und erst in den folgenden Abschnitten Besonderheiten bezüglich der einzelnen Datentypen aufzeigen.

Für alle sequentiellen Datentypen sind folgende Operationen definiert (`s` und `t` sind hierbei Instanzen desselben sequentiellen Datentyps; `i`, `j`, `k` und `n` sind Ganzzahlen; `x` ist eine Referenz auf eine beliebige Instanz):

Notation	Beschreibung
<code>x in s</code>	Prüft, ob <code>x</code> in <code>s</code> enthalten ist. Das Ergebnis ist eine <code>bool</code> -Instanz.
<code>x not in s</code>	Prüft, ob <code>x</code> nicht in <code>s</code> enthalten ist. Das Ergebnis ist eine <code>bool</code> -Instanz. Gleichwertig mit <code>not x in s</code> .
<code>s + t</code>	Das Ergebnis ist eine neue Sequenz, die die Verkettung von <code>s</code> und <code>t</code> enthält.

Tabelle 8.13 Methoden der sequentiellen Datentypen

5 Dies ist eine der großen Neuerungen in Python 3.0. In früheren Python-Versionen gab es die beiden Datentypen `str` und `unicode`, wobei `str` dem jetzigen `bytes` und `unicode` dem jetzigen `str` entsprach. Da vorwiegend der alte Datentyp `str` zum Speichern von Strings genutzt wurde, gab es einige Stolpersteine, wenn man Sonderzeichen mit Python-Programmen verarbeiten wollte. Durch die neue Typaufteilung ist der Umgang mit Zeichenketten wesentlich komfortabler und weniger fehleranfällig geworden.

Notation	Beschreibung
<code>s += t</code>	Erzeugt die Verkettung von <code>s</code> und <code>t</code> und weist sie <code>s</code> zu.
<code>s * n</code> oder <code>n * s</code>	Liefert eine neue Sequenz, die die Verkettung von <code>n</code> Kopien von <code>s</code> enthält.
<code>s *= n</code>	Erzeugt das Produkt <code>s * n</code> und weist es <code>s</code> zu.
<code>s[i]</code>	Liefert das <code>i</code> -te Element von <code>s</code> .
<code>s[i:j]</code>	Liefert den Ausschnitt aus <code>s</code> von <code>i</code> bis <code>j</code> .
<code>s[i:j:k]</code>	Liefert den Ausschnitt aus <code>s</code> von <code>i</code> bis <code>j</code> , wobei nur jedes <code>k</code> -te Element beachtet wird.
<code>len(s)</code>	Gibt eine Ganzzahl zurück, die die Anzahl der Elemente von <code>s</code> angibt.
<code>min(s)</code>	Liefert das kleinste Element von <code>s</code> , sofern eine Ordnungsrelation für die Elemente definiert ist.
<code>max(s)</code>	Liefert das größte Element von <code>s</code> , sofern eine Ordnungsrelation für die Elemente definiert ist.

Tabelle 8.13 Methoden der sequentiellen Datentypen (Forts.)

Wie bereits bekannt ist, lässt sich ein neuer String erzeugen, indem man seinen Inhalt in doppelte Hochkommata schreibt:

```
>>> s = "Dies ist unser Teststring"
```

Ist ein Element vorhanden?

Mithilfe von `in` lässt sich ermitteln, ob ein bestimmtes Element in einer Sequenz enthalten ist. Da die Elemente eines Strings Buchstaben sind, können wir mit dem Operator prüfen, ob ein bestimmter Buchstabe in einem String vorkommt. Als Ergebnis wird ein Wahrheitswert geliefert: `True`, wenn das Element vorhanden ist, und `False`, wenn es nicht vorhanden ist. Buchstaben können Sie in Python durch Strings der Länge eins abbilden:

```
>>> s = "Dies ist unser Teststring"
>>> "u" in s
True
>>> if "j" in s:
...     print("Juhuu, mein Lieblingsbuchstabe ist enthalten")
... else:
...     print("Ich mag diesen String nicht...")
Ich mag diesen String nicht...
```

Um das Gegenteil – also ob ein Element nicht in einer Sequenz enthalten ist – zu prüfen, dient der `not in`-Operator. Seine Verwendung entspricht der des

`in`-Operators, mit dem einzigen Unterschied, dass er das negierte Ergebnis produziert:

```
>>> "a" in "Besuch beim Zahnarzt"
True
>>> "a" not in "Besuch beim Zahnarzt"
False
```

Sie werden sich an dieser Stelle zu Recht fragen, warum für diesen Zweck ein eigener Operator definiert worden ist, wo man doch mit `not` jeden booleschen Wert negieren kann. Folgende Überprüfungen sind vollkommen gleichwertig:

```
>>> "n" not in "Python ist toll"
False
>>> not "n" in "Python ist toll"
False
```

Der Grund für diese scheinbar überflüssige Definition liegt in der besseren Lesbarkeit. `x not in s` liest sich im Gegensatz zu `not x in s` genau wie ein englischer Satz, während die andere Form unnötig kompliziert zu lesen ist.⁶

Verkettung von Sequenzen

Es kommt häufig vor, dass man mehrere Sequenzen aneinanderhängen möchte, um mit dem Ergebnis weiterzuarbeiten. Beispielsweise könnte man den Vor- und den Nachnamen eines Benutzers zu seinem gesamten Namen zusammenfügen, um ihn dann persönlich zu begrüßen. Für solche Zwecke dient der `+`-Operator, der aus zwei Sequenzen eine neue erzeugt, indem er die beiden verkettet:

```
>>> vorname = "Heinz"
>>> nachname = "Meier"
>>> name = vorname + " " + nachname
>>> name
'Heinz Meier'
```

Eine weitere Möglichkeit, Strings zu verketten, bietet der Operator `+=` für erweiterte Zuweisungen:

```
>>> s = "Musik"
>>> s += "lautsprecher"
>>> s
'Musiklautsprecher'
```

6 Zusätzlich muss man für die Interpretation von `not x in s` die Priorität der beiden Operatoren `not` bzw. `in` kennen. Wenn der `not`-Operator stärker bindet, würde der Ausdruck wie `(not x) in s` ausgewertet. Hat `in` eine höhere Priorität, wäre der Ausdruck wie `not (x in s)` zu behandeln. Tatsächlich bindet `in` stärker als `not`, womit letztere Deutung die richtige ist.

Wiederholung von Sequenzen

Sie können in Python das Produkt einer Sequenz *s* mit einer Ganzzahl *n* bilden: *n * s* oder *s * n*. Das Ergebnis ist eine neue Sequenz, die *n* Kopien von *s* hintereinander enthält:

```
>>> 3 * "abc"
'abccabccabc'
>>> "xyz" * 5
'xyzxyzxyzxyzxyz'
```

Genau wie bei der Verkettung gibt es auch hier einen Operator für die erweiterte Zuweisung: **=*:

```
>>> weihnachtsmann = "ho"
>>> weihnachtsmann *= 3
>>> weihnachtsmann
'hohoho'
```

Zugriff auf bestimmte Elemente einer Sequenz

Wie eingangs erwähnt wurde, stellen Sequenzen Folgen von Elementen dar. Da diese Elemente in einer bestimmten Reihenfolge gespeichert werden – beispielsweise wäre ein String, bei dem die Reihenfolge der Buchstaben willkürlich ist, wenig sinnvoll –, kann man jedem Element der Sequenz eine ganze Zahl, den sogenannten *Index*, zuweisen. Dafür werden alle Elemente der Sequenz fortlaufend von vorn nach hinten durchnummeriert, wobei das erste Element den Index 0 bekommt.

Mit dem `[]`-Operator kann man auf ein bestimmtes Element der Sequenz zugreifen, indem man den entsprechenden Index in die eckigen Klammern schreibt:

```
>>> alphabet = "abcdefghijklmnopqrstuvwxyz"
>>> alphabet[9]
'j'
>>> alphabet[1]
'b'
```

Um komfortabel auf das letzte oder das *x*-te Element von hinten zugreifen zu können, gibt es eine weitere Indizierung der Elemente von hinten nach vorn. Das letzte Element erhält dabei als Index -1, das vorletzte -2 und so weiter:

```
>>> name = "Python"
>>> name[-2]
'o'
```


Versucht man, mit einem Index auf ein nicht vorhandenes Element zuzugreifen, wird dies mit einem `IndexError` quittiert:

```
>>> zukurz = "Ich bin zu kurz"
>>> zukurz[1337]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Neben dem Zugriff auf einzelne Elemente der Sequenz ist es mit dem `[]`-Operator auch möglich, ganze Teilsequenzen auszulesen. Dies erreicht man dadurch, dass man den Anfang und das Ende der gewünschten Teilfolge durch einen Doppelpunkt getrennt in die eckigen Klammern schreibt. Der Anfang ist dabei der Index des ersten Elements der gewünschten Teilfolge, und das Ende ist der Index des ersten Elements, das nicht mehr in der Teilfolge enthalten sein soll.

Um im folgenden Beispiel die Zeichenfolge "WICHTIG" aus dem String zu extrahieren, geben wir den Index des großen "W" und den des ersten "s" nach "WICHTIG" an:

```
>>> s = "schrottschrottWICHTIGschrottschrott"
>>> s[14]
'W'
>>> s[21]
's'
>>> s[14:21]
'WICHTIG'
```

Es ist auch möglich, bei diesem sogenannten *Slicing* (dt. »Abschneiden«) positive und negative Indizes zu mischen. Beispielsweise ermittelt der folgende Code-Abschnitt eine Teilfolge ohne das erste und letzte Element der Ursprungssequenz:

```
>>> string = "ameisen"
>>> string[1:-1]
'meise'
```

Aus Bequemlichkeitsgründen können die Indizes weggelassen werden, was dazu führt, dass der maximal bzw. minimal mögliche Wert angenommen wird. Entfällt der Startindex, wird das nullte als erstes Element der Teilsequenz angenommen, und verzichtet man auf den Endindex, werden alle Buchstaben bis zum Ende kopiert. Möchten wir zum Beispiel die ersten fünf Buchstaben eines Strings oder alle ab dem fünften Zeichen ermitteln, geht das folgendermaßen:

```
>>> s = "abcdefghijklmnopqrstuvwxyz"
>>> s[:5]
'abcde'
```

```
>>> s[5:]
'fghijklmnopqrstuvwxyz'
```

Wenn man beide Indizes ausspart (`s[:]`), lässt sich auch eine echte Kopie der Sequenz erzeugen, weil dann alle Elemente vom ersten bis zum letzten kopiert werden. Beachten Sie bitte die unterschiedlichen Ergebnisse der beiden folgenden Code-Ausschnitte:

```
>>> s1 = "Kopier mich!"
>>> s2 = s1
>>> s1 == s2
True
>>> s1 is s2
True
```

Wie erwartet verweisen `s1` und `s2` auf dieselbe Instanz, sind also identisch. Anders sieht es bei dem nächsten Beispiel aus, bei dem eine echte Kopie von "Kopier mich!" im Speicher erzeugt wird. Dies zeigt sich beim Identitätsvergleich mit `is`:

```
>>> s1 = "Kopier mich!"
>>> s2 = s1[:]
>>> s1 == s2
True
>>> s1 is s2
False
```

Slicing bietet noch flexiblere Möglichkeiten, wenn man nicht eine ganze Teilsequenz, sondern nur bestimmte Elemente dieses Teils extrahieren möchte. Mit der *Schrittweite* (hier engl. *step*) lässt sich angeben, wie die Indizes vom Beginn bis zum Ende einer Teilsequenz gezählt werden sollen. Die Schrittweite wird, durch einen weiteren Doppelpunkt abgetrennt, nach der hinteren Grenze angegeben. Eine Schrittweite von 2 sorgt beispielsweise dafür, dass nur jedes zweite Element kopiert wird:

```
>>> ziffern = "0123456789"
>>> ziffern[1:10:2]
'13579'
```

Die Zeichenfolge, die ab dem ersten Element (Achtung: Die Zählweise beginnt bei 0) jedes zweite Element von `ziffern` enthält, ergibt einen neuen String mit den ungeraden Ziffern. Auch bei dieser erweiterten Notation können die Grenzündizes entfallen. Der folgende Code ist also zum vorherigen Beispiel äquivalent:

```
>>> ziffern = "0123456789"
>>> ziffern[1::2]
'13579'
```

Eine negative Schrittweite bewirkt ein Rückwärtszählen vom Start- zum Endindex, wobei in diesem Fall der Startindex auf ein weiter hinten liegendes Element der Sequenz als der Endindex verweisen muss. Mit einer Schrittweite von -1 lässt sich sehr elegant eine Sequenz »umdrehen«:

```
>>> name = "ytnoM Python"
>>> name[4::-1]
'Monty'
>>> name[::-1]
'nohtyP Monty'
```

Bei negativen Schrittweiten vertauschen sich Anfang und Ende der Sequenz. Deshalb wird in dem Beispiel `name[4::-1]` nicht alles vom vierten bis zum letzten Zeichen, sondern der Teil vom vierten bis zum ersten Zeichen ausgelesen.

Wichtig für den Umgang mit dem Slicing ist die Tatsache, dass zu große oder zu kleine Indizes nicht zu einem `IndexError` führen, wie es beim Zugriff auf einzelne Elemente der Fall ist. Zu große Indizes werden intern durch den maximal möglichen, zu kleine durch den minimal möglichen Index ersetzt. Liegen beide Indizes außerhalb des gültigen Bereichs oder ist der Startindex bei positiver Schrittweite größer als der Endindex, wird eine leere Sequenz zurückgegeben:

```
>>> s = "Viel weniger als 1337 Zeichen"
>>> s[5:1337]
'weniger als 1337 Zeichen'
>>> s[-100:100]
'Viel weniger als 1337 Zeichen'
>>> s[1337:2674]
''
>>> s[10:4]
''
```

Länge einer Sequenz

Als *Länge einer Sequenz* ist in Python die Anzahl ihrer Elemente definiert. Sie ist eine ganze Zahl größer oder gleich null und lässt sich mit der Built-in Function `len` ermitteln:

```
>>> string = "Wie lang bin ich wohl?"
>>> len(string)
22
```

Das kleinste und das größte Element einer Sequenz

Eine sehr häufige Aufgabe innerhalb eines Programms besteht darin, das kleinste beziehungsweise größte Element einer Sequenz zu ermitteln. Aus diesem Grund existieren in Python die Funktionen `min` und `max`, wobei `min` das kleinste und `max` das größte Element zurückgibt. Allerdings sind diese beiden Funktionen nur dann sinnvoll, wenn eine *Ordnungsrelation* für die Elemente der Sequenz existiert (in Abschnitt 8.3.4 über komplexe Zahlen wird zum Beispiel der Datentyp `complex` ohne Ordnungsrelation beschrieben). Für Buchstaben wird ihre Position im Alphabet als Ordnungsrelation benutzt, solange es sich nur um Großbuchstaben oder nur um Kleinbuchstaben handelt. Beim Vergleichen von Groß- und Kleinbuchstaben untereinander gelten Kleinbuchstaben immer als größer⁷ – "a" ist also kleiner als "z" und größer als "A":

```
>>> max("wer gewinnt wohl")
'w'
>>> min("zeichenkette")
'c'
```

8.5.1 Listen – list

In diesem Abschnitt werden Sie den ersten veränderbaren (mutable) Datentyp, die *Liste*, kennenlernen. Anders als bei dem sequentiellen Datentyp `str`, der nur gleichartige Elemente, die Buchstaben, speichern kann, sind Listen für die Verwaltung beliebiger Instanzen auch unterschiedlicher Datentypen geeignet. Eine Liste kann also durchaus Zahlen, Strings oder auch weitere Listen als Elemente enthalten, wodurch sie sehr flexibel anwendbar ist.

Eine neue Liste lässt sich dadurch erzeugen, dass man eine Aufzählung ihrer Elemente in eckige Klammern `[]` schreibt:

```
>>> l = [1, 0.5, "String", 2]
```

Die Liste `l` enthält nun zwei Ganzzahlen, eine Gleitkommazahl und einen String.

Da es sich bei dem Listentyp, der innerhalb von Python den Namen `list` hat, um einen sequentiellen Datentyp handelt, können alle im letzten Abschnitt beschriebenen Methoden und Verfahren auf ihn angewandt werden.

7 Falls Sie sich über dieses merkwürdige Verhalten wundern: Die Reihenfolge im Alphabet beschreibt nur einen Teilaspekt der Ordnungsrelation für einzelne Zeichen. Sonderzeichen wie beispielsweise das Leerzeichen lassen sich damit nicht sinnvoll einordnen. Sie werden im Abschnitt über Strings die Hintergründe hierzu kennenlernen.

Allerdings kann sich der Inhalt einer Liste auch nach ihrer Erzeugung ändern, weshalb eine Reihe weiterer Operatoren und Methoden für sie verfügbar sind:

Operator	Wirkung
<code>s[i] = x</code>	Das Element von <code>s</code> mit dem Index <code>i</code> wird durch <code>x</code> ersetzt.
<code>s[i:j] = t</code>	Der Teil <code>s[i:j]</code> wird durch <code>t</code> ersetzt. Dabei muss <code>t</code> iterierbar sein.
<code>s[i:j:k] = t</code>	Die Elemente von <code>s[i:j:k]</code> werden durch die von <code>t</code> ersetzt.
<code>del s[i]</code>	Das <code>i</code> -te Element von <code>s</code> wird entfernt.
<code>del s[i:j]</code>	Der Teil <code>s[i:j]</code> wird aus <code>s</code> entfernt. Das ist äquivalent zu <code>s[i:j] = []</code> .
<code>del s[i:j:k]</code>	Die Elemente der Teilfolge <code>s[i:j:k]</code> werden aus <code>s</code> entfernt.

Tabelle 8.14 Operatoren für den Datentyp `list`

Wir werden diese Operatoren der Reihe nach mit kleinen Beispielen erklären.

Verändern eines Wertes innerhalb der Liste

Sie können kann Elemente einer Liste durch andere ersetzen, wenn Sie ihren Index kennen:

```
>>> s = [1, 2, 3, 4, 5, 6, 7]
>>> s[3] = 1337
>>> s
[1, 2, 3, 1337, 5, 6, 7]
```

Diese Methode eignet sich allerdings nicht, um mehr Elemente in die Liste einzufügen. Es können nur bereits bestehende Elemente ersetzt werden, und die Länge der Liste bleibt unverändert.

Ersetzen von Teillisten und Einfügen neuer Elemente

Es ist möglich, eine ganze Teilliste durch andere Elemente zu ersetzen. Dazu schreiben Sie den zu ersetzenden Teil der Liste wie beim Slicing auf, wobei er aber auf der linken Seite einer Zuweisung stehen muss:

```
>>> einkaufen = ["Brot", "Eier", "Milch", "Fisch", "Mehl"]
>>> einkaufen[1:3] = ["Wasser", "Wurst"]
>>> einkaufen
['Brot', 'Wasser', 'Wurst', 'Fisch', 'Mehl']
```

Die Liste, die eingefügt werden soll, kann auch mehr oder weniger Elemente als der zu ersetzende Teil haben und sogar ganz leer sein.

Man kann wie beim Slicing auch eine Schrittweite angeben, um beispielsweise nur jedes dritte Element der Teilsequenz zu ersetzen. Im nachstehenden Beispiel wird jedes dritte Element der Teilsequenz `s[2:11]` durch das entsprechende Element aus `["A", "B", "C"]` ersetzt:

```
>>> s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> s[2:9:3] = ["A", "B", "C"]
>>> s
[0, 1, 'A', 3, 4, 'B', 6, 7, 'C', 9, 10]
```

Wird eine Schrittweite angegeben, muss die Sequenz auf der rechten Seite der Zuweisung genauso viele Elemente wie die Teilsequenz auf der linken Seite haben. Ist das nicht der Fall, wird ein `ValueError` erzeugt.

Elemente und Teillisten löschen

Um einen einzelnen Wert aus einer Liste zu entfernen, dient der `del`-Operator:

```
>>> s = [26, 7, 1987]
>>> del s[0]
>>> s
[7, 1987]
```

Auf diese Weise lassen sich auch ganze Teillisten entfernen:

```
>>> s = [9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> del s[3:6]
>>> s
[9, 8, 7, 3, 2, 1]
```

Für das Entfernen von Teilen einer Liste wird auch die Schrittfolge der Slicing-Notation unterstützt. Im folgenden Beispiel werden damit alle Elemente mit geradem Index entfernt (Achtung: "a" hat den Index 0):

```
>>> s = ["a","b","c","d","e","f","g","h","i","j"]
>>> del s[::2]
>>> s
['b', 'd', 'f', 'h', 'j']
```

Nachdem nun die Operatoren für Listen behandelt worden sind, wenden wir uns den Methoden einer Liste zu. In der Tabelle sind `s` und `t` Listen, `i`, `j` und `k` sind Ganzzahlen, und `x` ist eine beliebige Instanz:

Methode	Wirkung
<code>s.append(x)</code>	Hängt <code>x</code> ans Ende von <code>s</code> an.
<code>s.extend(t)</code>	Hängt alle Elemente von <code>t</code> ans Ende von <code>s</code> an.
<code>s.count(x)</code>	Gibt an, wie oft das Element <code>x</code> in <code>s</code> vorkommt.
<code>s.index(x[, i[, j]])</code>	Gibt den Index <code>k</code> des ersten Vorkommens von <code>x</code> im Bereich <code>i ≤ k < j</code> zurück.
<code>s.insert(i, x)</code>	Fügt <code>x</code> an der Stelle <code>i</code> in <code>s</code> ein. Anschließend hat <code>s[i]</code> den Wert von <code>x</code> , wobei alle folgenden Elemente um eine Stelle nach hinten aufrücken.
<code>s.pop([i])</code>	Gibt das <code>i</code> -te Element von <code>s</code> zurück und entfernt es aus <code>s</code> . Ist <code>i</code> nicht angegeben, wird das letzte Element genommen.
<code>s.remove(x)</code>	Entfernt das erste Vorkommen von <code>x</code> aus der Sequenz <code>s</code> .
<code>s.reverse()</code>	Kehrt die Reihenfolge der Elemente in <code>s</code> um.
<code>s.sort([key[, reverse]])</code>	Sortiert <code>s</code> .

Tabelle 8.15 Methoden von list-Instanzen

s.append(x)

Mit `append` erweitern Sie eine Liste am Ende um ein weiteres Element:

```
>>> s = ["Nach mir soll noch ein String stehen"]
>>> s.append("Hier ist er")
>>> s
['Nach mir soll noch ein String stehen', 'Hier ist er']
```

s.extend(t)

Um an eine Liste mehrere Elemente anzuhängen, dient die Methode `extend`, die ein iterierbares Objekt – beispielsweise eine andere Liste – als Parameter `t` erwartet. Im Ergebnis werden alle Elemente von `t` an die Liste `s` angehängt:

```
>>> s = [1, 2, 3]
>>> s.extend([4, 5, 6])
>>> s
[1, 2, 3, 4, 5, 6]
```

s.count(x)

Man kann mit `count` ermitteln, wie oft ein bestimmtes Element `x` in einer Liste enthalten ist:

```
>>> s = [1, 2, 2, 3, 2]
>>> s.count(2)
3
```

s.index(x[, i[, j]])

Mit `index` ermitteln Sie die Position eines Elements in einer Liste:

```
>>> ziffern = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ziffern.index(3)
2
```

Um die Suche auf einen Teilbereich der Liste einzuschränken, dienen die Parameter *i* und *j*, wobei *i* den ersten Index der gewünschten Teilfolge und *j* den ersten Index hinter der gewünschten Teilfolge angibt:

```
>>> [1, 22, 333, 4444, 333, 22, 1].index(1, 3, 7)
6
```

Ist das Element *x* nicht in *s* oder in der angegebenen Teilfolge enthalten, führt `index` zu einem `ValueError`:

```
>>> s = [2.5, 2.6, 2.7, 2.8]
>>> s.index(2.4)
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    s.index(2.4)
ValueError: list.index(x): x not in list
```

s.insert(i, x)

Mit `insert` kann man an beliebiger Stelle ein neues Element in eine Liste einfügen. Der erste Parameter *i* gibt den gewünschten Index des neuen Elements, der zweite, *x*, das Element selbst an:

```
>>> erst_mit_loch = [1, 2, 3, 5, 6, 7, 8]
>>> erst_mit_loch.insert(3, 4)
>>> erst_mit_loch
[1, 2, 3, 4, 5, 6, 7, 8]
```

Ist der Index *i* zu klein, wird *x* am Anfang von *s* eingefügt; ist er zu groß, wird er wie bei `append` am Ende angehängt.

s.pop([i])

Das Gegenstück zu `insert` ist `pop`. Mit dieser Methode kann man ein beliebiges Element anhand seines Index aus einer Liste entfernen. Ist der optionale Parameter nicht angegeben, so wird das letzte Element der Liste entfernt. Das entfernte Element wird von `pop` zurückgegeben:

```
>>> s = ["H", "a", "l", "l", "o"]
>>> s.pop()
'o'
```



```
>>> s.pop(0)
'H'
>>> s
['a', 'l', 'l']
```

Wird versucht, einen ungültigen Index zu übergeben oder ein Element aus einer leeren Liste zu entfernen, wird ein `IndexError` erzeugt.

s.remove(x)

Möchten Sie ein Element mit einem bestimmten Wert aus einer Liste entfernen, egal welchen Index es hat, können Sie die Methode `remove` bemühen. Sie entfernt das erste Element der Liste, das den gleichen Wert wie `x` hat.

```
>>> s = ["H", "u", "h", "u"]
>>> s.remove("u")
>>> s
['H', 'h', 'u']
```

Der Versuch, ein nicht vorhandenes Element zu entfernen, führt zu einem `ValueError`.

s.reverse()

Mit `reverse` kehren Sie die Reihenfolge der Elemente einer Liste um:

```
>>> s = [1, 2, 3]
>>> s.reverse()
>>> s
[3, 2, 1]
```

Im Unterschied zu der Slice-Notation `s[::-1]` geschieht die Umkehrung »in place«. Es wird also keine neue `list`-Instanz erzeugt, sondern die alte verändert. Da dies weniger Rechenzeit und Speicher kostet, ist `reverse` der Slice-Notation vorzuziehen, wenn Sie nicht unbedingt eine neue Liste brauchen.

s.sort([key[, reverse]])

Die komplexeste Methode des `list`-Datentyps ist `sort`, die eine Liste nach bestimmten Kriterien sortiert. Rufen Sie die Methode ohne Parameter auf, benutzt Python die normalen Vergleichsoperatoren zum Sortieren:

```
>>> l = [4, 2, 7, 3, 6, 1, 9, 5, 8]
>>> l.sort()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Enthält eine Liste Elemente, für die keine Ordnungsrelation definiert ist, wie zum Beispiel `complex`, führt der Aufruf von `sort` ohne Parameter zu einem `TypeError`:

```
>>> lst = [5 + 13j, 1 + 4j, 6 + 2j]
>>> lst.sort()
Traceback (most recent call last):
  File "<pyshe11#18>", line 1, in <module>
    lst.sort()
TypeError: no ordering relation is defined for complex numbers
```

Um eine Liste nach bestimmten Kriterien zu sortieren, dient der Parameter *key*. Die Methode `sort` erwartet im Parameter *key* eine Funktion, die vor jedem Vergleich für beide Operanden aufgerufen wird und deshalb ihrerseits einen Parameter erwartet. Im Ergebnis werden dann nicht die Operanden direkt verglichen, sondern stattdessen die entsprechenden Rückgabewerte der übergebenen Funktion.

Wir wollen eine Liste von Namen nach ihrer Länge sortieren. Zu diesem Zweck benutzen wir die Built-in Function `len`, die jedem Namen seine Länge zuordnet. In der Praxis sieht das dann folgendermaßen aus:

```
>>> l = ["Katharina", "Peter", "Jan", "Florian", "Paula"]
>>> l.sort(key=len)
>>> l
['Jan', 'Peter', 'Paula', 'Florian', 'Katharina']
```

Natürlich können Sie auch komplexere Funktionen als die `len`-Built-in übergeben. Wie Sie Ihre eigenen Funktionen definieren, um sie beispielsweise mit `sort` zu verwenden, lernen Sie in Kapitel 10, »Funktionen«.

Der letzte Parameter, *reverse*, erwartet für die Übergabe einen booleschen Wert, der angibt, ob die Reihenfolge der Sortierung umgekehrt werden soll:

```
>>> l = [4, 2, 7, 3, 6, 1, 9, 5, 8]
>>> l.sort(reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Es bleibt noch anzumerken, dass `sort` eine Funktion ist, die ausschließlich Schlüsselwortparameter akzeptiert. Versuchen Sie trotzdem, positionsbezogene Parameter zu übergeben, führt dies zu einem Fehler.

In folgendem Beispiel versuchen wir wieder, die Namensliste nach Länge zu sortieren. Allerdings verwenden wir diesmal einen positionsbezogenen Parameter für die Übergabe von `len`:

```
>>> l = ["Katharina", "Peter", "Jan", "Florian", "Paula"]
>>> l.sort(len)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must use keyword argument for key function
```

Stabile Sortiervverfahren

Eine wichtige Eigenschaft von `sort` ist, dass es sich um eine *stabile Sortierung* handelt. Stabile Sortiervverfahren zeichnen sich dadurch aus, dass sie beim Sortieren die relative Position gleichwertiger Elemente nicht vertauschen. Stellen Sie sich einmal vor, Sie hätten folgende Namensliste:

Vorname	Nachname
Natalie	Schmidt
Mathias	Schwarz
Florian	Kroll
Ricarda	Schmidt
Helmut	Schmidt
Peter	Kaiser

Tabelle 8.16 Fiktive Namensliste

Nun ist es Ihre Aufgabe, diese Liste alphabetisch nach den Nachnamen zu sortieren. Gruppen mit gleichem Nachnamen sollen nach den jeweiligen Vornamen sortiert werden. Um dieses Problem zu lösen, können Sie die Liste im ersten Schritt nach den Vornamen sortieren, was zu folgender Anordnung führt:

Vorname	Nachname
Florian	Kroll
Helmut	Schmidt
Mathias	Schwarz
Natalie	Schmidt
Peter	Kaiser
Ricarda	Schmidt

Tabelle 8.17 Nach Vornamen sortierte Namensliste

Im Resultat interessieren uns jetzt nur die Positionen der drei Personen, deren Nachname »Schmidt« ist. Würden Sie einfach alle anderen Namen streichen, wären die Schmidts richtig sortiert, weil ihre relative Position durch den ersten Sortierlauf korrekt hergestellt wurde. Nun kommt die Stabilität der `sort`-Methode zum Tragen, weil dadurch bei einem erneuten Sortierdurchgang nach den Nachnamen diese relative Ordnung nicht zerstört wird. Das Ergebnis sähe am Ende so aus:

Vorname	Nachname
Peter	Kaiser
Florian	Kroll
Helmut	Schmidt
Natalie	Schmidt
Ricarda	Schmidt
Mathias	Schwarz

Tabelle 8.18 Vollständig sortierte Namensliste

Wäre `sort` nicht stabil, so gäbe es keine Garantie dafür, dass Helmut vor Natalie und Ricarda eingeordnet wird.

Wie Sie sehen, ist die `sort`-Methode extrem flexibel und mächtig. Bei Ihrer Arbeit mit Python werden Sie höchstwahrscheinlich niemals etwas anderes zum Sortieren Ihrer Daten verwenden.

Weitere Eigenschaften von Listen

Im Zusammenhang mit Pythons `list`-Datentyp ergeben sich ein paar Besonderheiten, die nicht unmittelbar ersichtlich sind.

Zum einen ist `list` ein veränderbarer Datentyp, und deshalb betreffen Änderungen an einer `list`-Instanz immer alle Referenzen, die auf sie verweisen. Betrachten wir einmal das folgende Beispiel, in dem der unveränderliche Datentyp `str` mit `list` verglichen wird:

```
>>> a = "Hallo "
>>> b = a
>>> b += "Welt"
>>> b
'Hallo Welt'
>>> a
'Hallo '
```

Dieses Beispiel erzeugt einfach eine `str`-Instanz mit dem Wert `"Hallo "` und lässt die beiden Referenzen `a` und `b` auf sie verweisen. Anschließend wird mit dem Operator `+=` an den String, auf den `b` verweist, `"Welt"` angehängt. Wie die Ausgaben zeigen und wie wir es auch erwartet haben, wird eine neue Instanz mit dem Wert `"Hallo Welt"` erzeugt und `b` zugewiesen; `a` bleibt davon unberührt.

Übertragen wir das obige Beispiel auf Listen, ergibt sich ein wichtiger Unterschied:

```
>>> a = [1337]
>>> b = a
>>> b += [2674]
>>> b
[1337, 2674]
>>> a
[1337, 2674]
```

Strukturell gleicht der Code dem `str`-Beispiel, nur ist diesmal der verwendete Datentyp nicht `str`, sondern `list`. Der interessante Teil ist die Ausgabe am Ende, laut der `a` und `b` denselben Wert haben, obwohl die Operation nur auf `b` durchgeführt wurde. Tatsächlich verweisen `a` und `b` auf dieselbe Instanz, wovon Sie sich mithilfe des `is`-Operators überzeugen können:

```
>>> a is b
True
```

Diese sogenannten *Seiteneffekte*⁸ sollten Sie bei der Arbeit mit Listen im Hinterkopf behalten. Wenn Sie sichergehen möchten, dass die Originalliste nicht verändert wird, legen Sie mithilfe von Slicing eine echte Kopie an:

```
>>> a = [1337]
>>> b = a[:]
>>> b += [2674]
>>> b
[1337, 2674]
>>> a
[1337]
```

In diesem Beispiel wurde die von `a` referenzierte Liste kopiert und so vor indirekten Manipulationen über `b` geschützt. Sie müssen in solchen Fällen die Performance gegen den Schutz vor Seiteneffekten abwägen, da die Kopien der Listen im Speicher erzeugt werden müssen. Das kostet insbesondere bei langen Listen Rechenzeit und Speicherplatz und kann somit das Programm ausbremsen.

Im Zusammenhang mit Seiteneffekten sind auch die Elemente einer Liste interessant: Eine Liste speichert keine Instanzen an sich, sondern nur Referenzen auf sie. Das macht Listen einerseits flexibler und performanter, andererseits aber auch anfällig für Seiteneffekte. Schauen wir uns einmal das folgende – auf den ersten Blick merkwürdig anmutende – Beispiel an:

⁸ Seiteneffekte werden im Zusammenhang mit Funktionen in Abschnitt Seiteneffekte eine wichtige Rolle spielen.

```
>>> a = [[]]
>>> a = 4 * a
>>> a
[[], [], [], []]
>>> a[0].append(10)
>>> a
[[10], [10], [10], [10]]
```

Zu Beginn referenziert `a` eine Liste, in der eine weitere, leere Liste enthalten ist. Bei der anschließenden Multiplikation mit dem Faktor 4 wird die innere leere Liste nicht kopiert, sondern nur weitere drei Male referenziert. In der Ausgabe sehen wir also viermal dieselbe Liste. Wenn man das verstanden hat, ist es offensichtlich, warum die dem ersten Element von `a` angehängte 10 auch den anderen drei Listen hinzugefügt wird: Es handelt sich einfach um dieselbe Liste.

Es ist auch durchaus möglich, dass eine Liste sich selbst als Element enthält:

```
>>> a = []
>>> a.append(a)
```

Das Resultat ist eine unendlich tiefe Verschachtelung, da jede Liste wiederum sich selbst als Element enthält. Da nur Referenzen gespeichert werden müssen, verbraucht diese unendliche Verschachtelung nur sehr wenig Speicher und nicht, wie man zunächst vermuten könnte, unendlich viel. Trotzdem bergen solche Verschachtelungen die Gefahr von Endlosschleifen, wenn man die enthaltenen Daten verarbeiten möchte. Stellen Sie sich beispielsweise einmal vor, Sie wollten eine solche Liste auf dem Bildschirm ausgeben. Das würde zu unendlich vielen öffnenden und schließenden Klammern führen und somit den Computer lahmlegen. Trotzdem ist es möglich, solche Listen mit `print` auszugeben. Python überprüft selbstständig, ob eine Liste sich selbst enthält, und gibt dann anstelle von weiteren Verschachtelungen drei Punkte `...` aus:

```
>>> a = []
>>> a.append(a)
>>> print(a)
[...]
```

Bitte beachten Sie, dass die Schreibweise mit den drei Punkten kein gültiger Python-Code ist, um in sich selbst verschachtelte Listen zu erzeugen.

Wenn Sie selbst mit Listen arbeiten, die rekursiv sein könnten, sollten Sie Ihre Programme mit Abfragen ausrüsten, um Verschachtelungen von Listen mit sich selbst zu erkennen, damit das Programm bei der Verarbeitung nicht in einer endlosen Schleife stecken bleiben kann.

8.5.2 Unveränderliche Listen – tuple

Der Datentyp `list` ist sehr flexibel und wird häufig verwendet. Seine Mächtigkeit und Flexibilität hat aber auch den Nachteil, dass dafür relativ viel Rechenleistung und Speicher benötigt wird. Oft wird gar nicht die Flexibilität einer Liste benötigt, sondern nur ihre Fähigkeit, Referenzen auf beliebige Instanzen zu speichern. Deshalb existiert in Python neben `list` der Datentyp `tuple`, der im Gegensatz zu `list` immutable ist.

Der Datentyp `tuple` bringt keinen Mehrwert in Bezug auf Funktionalität, denn Listen können alles, was `tuple` leistet. Tatsächlich steht für `tuple`-Instanzen nur der Grundstock an Operationen für sequentielle Datentypen bereit.

Zum Erzeugen neuer `tuple`-Instanzen dienen die runden Klammern, die – wie bei den Listen – durch Kommata getrennt die Elemente des Tupels enthalten:

```
>>> a = (1, 2, 3, 4, 5)
>>> a[3]
4
```

Ein leeres Tupel wird durch zwei runde Klammern `()` ohne Inhalt definiert. Eine Besonderheit ergibt sich für Tupel mit nur einem Element. Würde man versuchen, ein Tupel mit nur einem Element auf die oben beschriebene Weise zu erzeugen, wäre das Literal unter Umständen nicht eindeutig:

```
>>> kein_tuple = (2)
>>> type(kein_tuple)
<class 'int'>
```

Mit `(2)` wird keine neue `tuple`-Instanz erzeugt, weil die Klammer in diesem Kontext schon für die Verwendung in Rechenoperationen für Ganzzahlen verwendet wird. Das Problem wird umgangen, indem in Literalen für Tupel mit nur einem Element diesem Element ein Komma nachgestellt werden muss:

```
>>> ein_tuple = (2,)
>>> type(ein_tuple)
<class 'tuple'>
```

Tuple Packing und Tuple Unpacking

Es ist möglich, die umschließenden Klammern bei einer `tuple`-Definition entfallen zu lassen. Trotzdem werden die durch Kommata getrennten Referenzen zu einem `tuple` zusammengefasst, was man *Tuple Packing* nennt:

```
>>> datum = 26, 7, 1987
>>> datum
(26, 7, 1987)
```

Umgekehrt ist es möglich, die Werte eines Tupels wieder zu entpacken:

```
>>> datum = 26, 7, 1987
>>> (tag, monat, jahr) = datum
>>> tag
26
>>> monat
7
>>> jahr
1987
```

Dieses Verfahren heißt *Tuple Unpacking*, und auch hier können die umschließenden Klammern entfallen. Durch Kombination von *Tuple Packing* und *Tuple Unpacking* ist es sehr elegant möglich, die Werte zweier Variablen ohne Hilfsvariable zu tauschen oder mehrere Zuweisungen in einer Zeile zusammenzufassen:

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Richtig angewandt kann die Nutzung dieses Features zur Lesbarkeit von Programmen beitragen, da das technische Detail der Zwischenspeicherung von Daten hinter die eigentliche Absicht, die Werte zu tauschen, zurücktritt.

Immutable heißt nicht zwingend unveränderlich!

Auch wenn `tuple`-Instanzen immutable sind, können sich die Werte der enthaltenen Elemente auch nach der Erzeugung ändern. Bei der Erzeugung eines neuen Tupels werden die Referenzen festgelegt, die es speichern soll. Verweist eine solche Referenz auf eine Instanz eines mutable Datentyps, beispielsweise eine Liste, so kann sich dessen Wert trotzdem ändern:

```
>>> a = ([],)
>>> a[0].append("Und sie dreht sich doch!")
>>> a
(['Und sie dreht sich doch!'],)
```

Die Unveränderlichkeit eines Tupels bezieht sich also nur auf die enthaltenen Referenzen und ausdrücklich nicht auf die dahinterstehenden Instanzen.

Dass Tupel immutable sind, ist also keine Garantie dafür, dass sich die Elemente nach der Erzeugung des Tupels nicht mehr verändern.

8.5.3 Strings – str, bytes

Dieser Abschnitt behandelt Pythons Umgang mit Zeichenketten und insbesondere die Eigenschaften der dafür bereitgestellten Datentypen `str` und `bytes`.

Wie Sie im vorhergehenden Kapitel gelernt haben, handelt es sich bei Strings um Folgen von *Zeichen*. Dies bedeutet, dass alle Operationen für sequentielle Typen für sie verfügbar sind.

Wir werden uns bis auf weiteres nur mit `str`-Instanzen beschäftigen, weil sich der Umgang mit `str` nicht wesentlich von dem mit `bytes` unterscheidet. Trotzdem haben beide Datentypen ihre Daseinsberechtigung, weil `str` für das Speichern von Textdaten und `bytes` für die Speicherung von Binärdaten gedacht ist.

Um neue `str`-Instanzen zu erzeugen, gibt es folgende Literale:

```
>>> string1 = "Ich wurde mit doppelten Hochkommata definiert"
>>> string2 = 'Ich wurde mit einfachen Hochkommata definiert'
```

Der gewünschte Inhalt des Strings wird zwischen die Hochkommata geschrieben, darf allerdings keine Zeilenvorschübe enthalten (im folgenden Beispiel wurde am Ende der ersten Zeile  gedrückt):

```
>>> s = "Erste Zeile
      File "<stdin>", line 1
        s = "Erste Zeile
          ^
SyntaxError: EOL while scanning string literal
```

String-Konstanten, die sich auch über mehrere Zeilen erstrecken können, werden durch `"""` bzw. `'''` eingefasst:

```
>>> string3 = """Erste Zeile!
Ui, noch eine Zeile"""
```

Stehen zwei String-Literale unmittelbar oder durch Leerzeichen getrennt hintereinander, werden sie von Python zu einem String verbunden:

```
>>> string = "Erster Teil" "Zweiter Teil"
>>> string
Erster TeilZweiter Teil
```

Wie Sie im Beispiel sehen, sind die Leerzeichen zwischen den Literalen bei der Verkettung nicht mehr vorhanden.

Diese Art der Verkettung eignet sich sehr gut, um lange oder unübersichtliche Strings auf mehrere Programmzeilen aufzuteilen, ohne dass die Zeilenvorschübe und Leerzeichen im Resultat gespeichert werden, wie es bei Strings mit `"""` oder

''' der Fall wäre. Um diese Aufteilung zu erreichen, schreibt man die String-Teile in runde Klammern:

```
>>> a = ("Gestern Abend ging die Party schon gut ab. "
...      "Aber heute geht die Party RICHTIG ab! "
...      "Und der Ramin ist hundertprozentig auch am "
...      "Start!")
>>> a
'Gestern Abend ging die Party schon gut ab. Aber heute geht die
Party RICHTIG ab! Und der Ramin ist hundertprozentig auch am Start!'
```

Wie Sie sehen, wurde der String so gespeichert, als ob er in einer einzigen Zeile definiert worden wäre.

Die Erzeugung von `bytes`-Instanzen funktioniert genauso wie die oben beschriebene Erzeugung von `str`-Instanzen. Der einzige Unterschied ist, dass Sie dem Stringliteral ein kleines `b` voranstellen müssen, um einen `bytes`-String zu erhalten:

```
>>> string1 = b"Ich bin bytes!"
>>> string1
b'Ich bin bytes!'
>>> type(string1)
<class 'bytes'>
```

Die anderen Arten der Stringerzeugung funktionieren für `bytes` funktionieren analog.

Steuerzeichen

Es gibt besondere Textelemente, die den Textfluss steuern und sich auf dem Bildschirm nicht als einzelne Zeichen darstellen lassen. Zu diesen sogenannten *Steuerzeichen* zählen unter anderem der Zeilenvorschub, der Tabulator oder der Rückschritt (von engl. *backspace*). Die Darstellung solcher Zeichen innerhalb von String-Literalen erfolgt mittels spezieller Zeichenfolgen, sogenannter *Escape-Sequenzen*. Escape-Sequenzen werden von einem Backslash `\` eingeleitet, der von der Kennung des gewünschten Sonderzeichens gefolgt wird. Die Escape-Sequenz `"\n"` steht beispielsweise für einen Zeilenumbruch:

```
>>> a = "Erste Zeile\nZweite Zeile"
>>> a
'Erste Zeile\nZweite Zeile'
>>> print(a)
Erste Zeile
Zweite Zeile
```

Beachten Sie bitte den Unterschied zwischen der Ausgabe mit `print` und der ohne `print` im interaktiven Modus: Die `print`-Anweisung setzt die Steuerzeichen in ihre Bildschirmdarstellung um (bei `"\n"` wird zum Beispiel eine neue Zeile begonnen), wohingegen die Ausgabe ohne `print` ein String-Literal mit den Escape-Sequenzen der Sonderzeichen auf dem Bildschirm anzeigt.

Für Steuerzeichen gibt es in Python die folgenden Escape-Sequenzen:

Escape-Sequenz	Bedeutung
<code>\a</code>	Bell (BEL) erzeugt einen Signalton.
<code>\b</code>	Backspace (BS) setzt die Ausgabeposition um ein Zeichen zurück.
<code>\f</code>	Formfeed (FF) erzeugt einen Seitenvorschub.
<code>\n</code>	Linefeed (LF) setzt die Ausgabeposition in die nächste Zeile.
<code>\r</code>	Carriage Return (CR) setzt die Ausgabeposition an den Anfang der nächsten Zeile.
<code>\t</code>	Horizontal Tab (TAB) hat die gleiche Bedeutung wie die Tabulator-taste.
<code>\v</code>	Vertikaler Tabulator (VT); dient zur vertikalen Einrückung.
<code>\"</code>	doppeltes Hochkomma
<code>\'</code>	einfaches Hochkomma
<code>\\</code>	Backslash, der wirklich als solcher in dem String erscheinen soll

Tabelle 8.19 Escape-Sequenzen für Steuerzeichen

Allerdings stammen Steuerzeichen aus der Zeit, als die Ausgaben hauptsächlich über Drucker erfolgten. Deshalb haben einige dieser Zeichen heute nur noch eine geringe praktische Bedeutung.

Die Escape-Sequenzen für einfache und doppelte Hochkommata sind notwendig, weil Python diese Zeichen als Begrenzung für String-Literale verwendet. Soll die Art von Hochkomma, die für die Begrenzung eines Strings verwendet wurde, innerhalb dieses Strings als Zeichen vorkommen, muss dort das entsprechende Hochkomma als Escape-Sequenz angegeben werden:

```
>>> a = "Das folgende Hochkomma muss nicht kodiert werden ' "
```

```
>>> b = "Dieses doppelte Hochkomma schon \" "
```

```
>>> c = 'Das gilt auch in Strings mit einfachen Hochkommata " '
```

```
>>> d = 'Hier muss eine Escape-Sequenz benutzt werden \' '
```

Im Abschnitt »Zeichensätze und Sonderzeichen« werden wir auf Escape-Sequenzen zurückkommen, um damit beliebige Sonderzeichen wie Umlaute oder das €-Zeichen zu kodieren.

Das automatische Ersetzen von Escape-Sequenzen ist manchmal lästig, insbesondere dann, wenn sehr viele Backslashes in einem String vorkommen sollen. Für diesen Zweck gibt es in Python die Präfixe `r` oder `R`, die einem String-Literal vorangestellt werden können. Diese Präfixe markieren das Literal als einen sogenannten *Raw-String* (dt. »roh«), was dazu führt, dass alle Backslashes eins zu eins in den Resultat-String übernommen werden:

```
>>> "Ein \tString mit \\ vielen \nEscape-Sequenzen\t"
'Ein \tString mit \\ vielen \nEscape-Sequenzen\t'
>>> r"Ein \tString mit \\ vielen \nEscape-Sequenzen\t"
'Ein \\tString mit \\\\ vielen \\nEscape-Sequenzen\\t'
>>> print(r"Ein \tString mit \\ vielen \nEscape-Sequenzen\t")
Ein \tString mit \\ vielen \nEscape-Sequenzen\t
```

Wie Sie an den doppelten Backslashes im Literal des Resultats und der Ausgabe mit `print` sehen können, wurden die Escape-Sequenzen nicht interpretiert.

Stringmethoden

String-Instanzen verfügen zusätzlich zu den Methoden für sequentielle Datentypen über weitere Methoden, die den Umgang mit Zeichenketten vereinfachen. Aufgrund der großen Anzahl der String-Methoden gibt es statt der zusammenfassenden Tabelle aller Methoden mehrere Kategorien, die einzeln erklärt werden.

Wenn wir im Folgenden von *Whitespaces* sprechen, sind alle Arten von Zeichen zwischen den Wörtern gemeint, die nicht als eigenes Zeichen angezeigt werden. Whitespaces sind folgende Zeichen: das Leerzeichen, der Zeilenvorschub, der vertikale und horizontale Tabulator, Linefeed, Formfeed und Carriage Return.

Trennen von Strings

Um Strings nach bestimmten Regeln in mehrere Teile zu zerlegen, dienen folgende Methoden:

- ▶ `s.split([sep[, maxsplit]])`
- ▶ `s.rsplit([sep[, maxsplit]])`
- ▶ `s.splitlines([keepends])`
- ▶ `s.partition(sep)`
- ▶ `s.rpartition(sep)`

Die Methoden `split` und `rsplit` zerteilen einen String in seine Wörter und geben diese als Liste zurück. Dabei gibt der Parameter `sep` die Zeichenfolge an, die die Wörter trennt. Mit `maxsplit` kann die Anzahl der Trennungen begrenzt werden. Geben Sie `maxsplit` nicht an, wird der String so oft zerteilt, wie `sep` in ihm

vorkommt. Ein gegebenenfalls verbleibender Rest wird als String in die resultierende Liste eingefügt. `split` beginnt mit dem Teilen am Anfang des Strings, während `rsplit` am Ende anfängt:

```
>>> s = "1-2-3-4-5-6-7-8-9-10"
>>> s.split("-")
['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
>>> s.split("-", 5)
['1', '2', '3', '4', '5', '6-7-8-9-10']
>>> s.rsplit("-", 5)
['1-2-3-4-5', '6', '7', '8', '9', '10']
```

Folgen mehrere Trennzeichen aufeinander, werden sie nicht zusammengefasst, sondern es wird jedes Mal erneut getrennt:

```
>>> s = "1--2-3"
>>> s.split("-")
['1', '', '', '2', '3']
```

Wird `sep` nicht angegeben, verhalten sich die beiden Methoden anders. Zuerst werden alle Whitespaces am Anfang und am Ende des Strings entfernt, und anschließend wird der String anhand von Whitespaces zerteilt, wobei dieses Mal aufeinanderfolgende Trennzeichen zu einem zusammengefasst werden:

```
>>> s = " Irgendein \t\t Satz  mit \n\r\t Whitespaces"
>>> s.split()
['Irgendein', 'Satz', 'mit', 'Whitespaces']
```

Der Aufruf von `split` ganz ohne Parameter ist sehr nützlich, um einen Text-String in seine Wörter zu spalten, auch wenn diese nicht nur durch Leerzeichen voneinander getrennt sind.

Die Methode `splitlines` spaltet einen String in seine einzelnen Zeilen auf und gibt eine Liste zurück, die die Zeilen enthält. Dabei werden Unix-Zeilenvorschübe `"\n"`, Windows-Zeilenvorschübe `"\r\n"` und Mac-Zeilenvorschübe `"\r"` als Trennzeichen benutzt:

```
>>> s = "Unix\nWindows\r\nMac\rLetzte Zeile"
>>> s.splitlines()
['Unix', 'Windows', 'Mac', 'Letzte Zeile']
```

Sollen die trennenden Zeilenvorschübe an den Enden der Zeilen erhalten bleiben, muss für den optionalen Parameter *keepends* der Wert `True` übergeben werden.

Die Methode `partition` zerteilt einen String an der ersten Stelle, an der der übergebene Trennstring *sep* auftritt, und gibt ein Tupel zurück, das aus dem Teil vor

dem Trennstring, dem Trennstring selbst und dem Teil danach besteht. Die Methode `rpartition` arbeitet genauso, nimmt aber das letzte Vorkommen von `sep` im Ursprungsstring als Trennstelle:

```
>>> s = "www.galileo-computing.de"
>>> s.partition(".")
('www', '.', 'galileo-computing.de')
>>> s.rpartition(".")
('www.galileo-computing', '.', 'de')
```

Suchen von Teilstrings

Um die Position und die Anzahl der Vorkommen eines Strings in einem anderen String zu ermitteln oder Teile eines Strings zu ersetzen, existieren folgende Methoden:

- ▶ `s.find(sub[, start[, end]])`
- ▶ `s.rfind(sub[, start[, end]])`
- ▶ `s.index(sub[, start[, end]])`
- ▶ `s.rindex(sub[, start[, end]])`
- ▶ `s.count(sub[, start[, end]])`

Die optionalen Parameter `start` und `end` der fünf Methoden dienen dazu, den Suchbereich einzugrenzen. Geben Sie `start` bzw. `end` an, wird nur der Teilstring `s[start:end]` betrachtet.

Hinweis

Zur Erinnerung: Beim Slicing eines Strings `s` mit `s[start:end]` wird ein Teilstring erzeugt, der das Element `s[end]` nicht mehr enthält.

Um herauszufinden, ob ein bestimmter String in einem anderen vorkommt und, wenn ja, wo, bietet Python die Methoden `find` und `index` mit ihren Gegenstücken `rfind` und `rindex` an. `find` gibt den Index des ersten Vorkommens von `sub` in `s` zurück, `rfind` entsprechend den Index des letzten Vorkommens. Ist `sub` nicht in `s` enthalten, geben `find` und `rfind` den Wert `-1` zurück:

```
>>> s = "Mal sehen, wo das 'e' in diesem String vorkommt"
>>> s.find("e")
5
>>> s.rfind("e")
29
```

Die Methoden `index` und `rindex` arbeiten auf die gleiche Weise, erzeugen aber einen `ValueError`, wenn `sub` nicht in `s` enthalten ist:

```
>>> s = "Dieser String wird gleich durchsucht"
>>> s.index("wird")
14
>>> s.index("nicht vorhanden")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    s.index("nicht vorhanden")
ValueError: substring not found
```

Der Grund für diese fast identischen Methoden liegt darin, dass sich Fehlermeldungen unter Umständen eleganter handhaben lassen als ungültige Rückgabewerte.⁹

Wie oft ein Teilstring in einem anderen enthalten ist, lässt sich mit `count` ermitteln:

```
>>> "Fischers Fritze fischt frische Fische".count("sch")
4
```

Ersetzen von Teilstrings

Mit den folgenden Methoden lassen sich bestimmte Teile oder Buchstaben von Strings durch andere ersetzen:

- ▶ `s.replace(old, new[, count])`
- ▶ `s.lower()`
- ▶ `s.upper()`
- ▶ `s.swapcase()`
- ▶ `s.capitalize()`
- ▶ `s.title()`
- ▶ `s.expandtabs([tabsize])`

Die Methode `replace` gibt einen String zurück, in dem alle Vorkommen von *old* durch *new* ersetzt wurden:

```
>>> falsch = "Python ist nicht super!"
>>> richtig = falsch.replace("nicht", "richtig")
>>> richtig
'Python ist richtig super!'
```

Mit dem Parameter *count* kann die Anzahl der Ersetzungen begrenzt werden:

⁹ Sie können die Details in Abschnitt 13.1, »Exception Handling«, nachlesen.

```
>>> s = "Bitte nur die ersten vier e ersetzen"
>>> s.replace("e", "E", 4)
'Bitte nur die ErstEn vier e ersetzen'
```

Die Methode `lower` ersetzt alle Großbuchstaben eines Strings durch die entsprechenden Kleinbuchstaben und gibt den Ergebnis-String zurück:

```
>>> s = "ERST GANZ GROSS UND DANN GANZ KLEIN!"
>>> s.lower()
'erst ganz gross und dann ganz klein!'
```

Mit `upper` erreichen Sie genau den umgekehrten Effekt.

Die Methode `swapcase` ändert die Groß- bzw. Kleinschreibung aller Buchstaben eines Strings, indem sie alle Großbuchstaben durch die entsprechenden Kleinbuchstaben und umgekehrt ersetzt:

```
>>> s = "wENN MAN IM dEUTSCHEN ALLE wORTE SO SCHRIEBE..."
>>> s.swapcase()
'Wenn man im Deutschen alle Worte so schriebe...'
```

Die Methode `capitalize` gibt eine Kopie des Ursprungsstrings zurück, wobei das erste Zeichen – sofern möglich – in einen Großbuchstaben umgewandelt wurde:

```
>>> s = "alles klein... noch ;)"
>>> s.capitalize()
'Alles klein... noch ;)'
```

Die Methode `title` erzeugt einen String, bei dem alle Wörter groß-, aber ihre restlichen Buchstaben kleingeschrieben sind, wie dies im Englischen bei Titeln üblich ist:

```
>>> s = "nOch BIn iCH eHeR weNiGeR aLS TiTeL gEeiGNeT"
>>> s.title()
'Noch Bin Ich Eher Weniger Als Titel Geeignet'
```

Mit `expandtabs` können Sie alle Tabulator-Zeichen ("`\t`") eines Strings durch Leerzeichen ersetzen lassen. Der optionale Parameter `tabsize` gibt dabei an, wie viele Leerzeichen für einen Tabulator eingefügt werden sollen. Ist `tabsize` nicht angegeben, werden acht Leerzeichen verwendet:

```
>>> s = "\tHier kann Quellcode stehen\n\t\
tEine Ebene weiter unten""
>>> print(s.expandtabs(4))
    Hier kann Quellcode stehen
    Eine Ebene weiter unten
```


Entfernen bestimmter Zeichen am Anfang oder am Ende von Strings

Die `strip`-Methoden ermöglichen es, unerwünschte Zeichen am Anfang oder am Ende eines Strings zu entfernen:

- ▶ `s.strip([chars])`
- ▶ `s.lstrip([chars])`
- ▶ `s.rstrip([chars])`

Die Methode `strip` entfernt unerwünschte Zeichen auf beiden Seiten des Strings. `lstrip` entfernt nur die Zeichen auf der linken Seite und `rstrip` nur die Zeichen auf der rechten.

Für den optionalen Parameter `chars` können Sie einen String übergeben, der die Zeichen enthält, die entfernt werden sollen. Geben Sie `chars` nicht an, werden alle Whitespaces gelöscht:

```
>>> s = "    \t\n \rUmgeben von Whitespaces    \t\t\r"
>>> s.strip()
'Umgeben von Whitespaces'
>>> s.lstrip()
'Umgeben von Whitespaces    \t\t\r'
>>> s.rstrip()
'    \t\n \rUmgeben von Whitespaces'
```

Um beispielsweise alle umgebenden Ziffern zu entfernen, könnten Sie so vorgehen:

```
>>> ziffern = "0123456789"
>>> s = "3674784673546Versteckt zwischen Zahlen3425923935"
>>> s.strip(ziffern)
'Versteckt zwischen Zahlen'
```

Ausrichten von Strings

Die folgenden Methoden erzeugen einen String mit einer bestimmten Länge und richten den Ursprungsstring darin auf eine bestimmte Weise aus:

- ▶ `s.center(width[, fillchar])`
- ▶ `s.ljust(width[, fillchar])`
- ▶ `s.rjust(width[, fillchar])`
- ▶ `s.zfill(width)`

Mit dem Parameter `width` geben Sie die gewünschte Länge des neuen Strings an. Ist die Länge von `s` größer als `width`, wird eine Kopie von `s` zurückgegeben. Die Methode `center` zentriert `s` im neuen String, `ljust` richtet `s` links aus, `rjust` rich-

tet *s* rechts aus. Der optionale Parameter *fillchar* der drei ersten Methoden muss ein String der Länge eins sein und gibt das Zeichen an, das zum Auffüllen bis zur übergebenen Länge verwendet werden soll. Standardmäßig werden Leerzeichen zum Füllen benutzt:

```
>>> s = "Richte mich aus"
>>> s.center(50)
'                Richte mich aus                '
>>> s.ljust(50)
'Richte mich aus                                '
>>> s.rjust(50, "-")
'-----Richte mich aus'
```

Die Methode *zfill* ist ein Spezialfall von *rjust* und für Strings gedacht, die numerische Werte enthalten. *zfill* erzeugt einen String der Länge *width*, in dem der Ursprungsstring rechts ausgerichtet ist und die linke Seite mit Nullen aufgefüllt wurde:

```
>>> "13.37".zfill(20)
'000000000000000013.37'
```

String-Tests

Die folgenden Methoden geben einen Wahrheitswert zurück, der aussagt, ob der Inhalt des Strings eine bestimmte Eigenschaft hat. Mit *islower* beispielsweise prüfen Sie, ob alle Buchstaben in *s* Kleinbuchstaben sind.

Methode	Beschreibung
<i>s.isalnum()</i>	True, wenn alle Zeichen in <i>s</i> Buchstaben oder Ziffern sind
<i>s.isalpha()</i>	True, wenn alle Zeichen in <i>s</i> Buchstaben sind
<i>s.isdigit()</i>	True, wenn alle Zeichen in <i>s</i> Ziffern sind
<i>s.islower()</i>	True, wenn alle Buchstaben in <i>s</i> Kleinbuchstaben sind
<i>s.isupper()</i>	True, wenn alle Buchstaben in <i>s</i> Großbuchstaben sind
<i>s.isspace()</i>	True, wenn alle Zeichen in <i>s</i> Whitespaces sind
<i>s.istitle()</i>	True, wenn alle Wörter in <i>s</i> großgeschrieben sind

Tabelle 8.20 Methoden für einfache String-Tests

Da sich diese Methoden alle sehr ähneln, soll ein Beispiel an dieser Stelle ausreichen:

```
>>> s = "1234abcd"
>>> s.isdigit()
False
```

```
>>> s.isalpha()
False
>>> s.isalnum()
True
```

Um zu prüfen, ob ein String mit einer bestimmten Zeichenkette beginnt oder endet, dienen die Methoden `startswith` bzw. `endswith`:

- ▶ `s.startswith(prefix[, start[, end]])`
- ▶ `s.endswith(suffix[, start[, end]])`

Die optionalen Parameter `start` und `end` begrenzen dabei – wie schon bei den Suchen-und-Ersetzen-Methoden – die Abfrage auf einen bestimmten Bereich von `s`:

```
>>> s = "www.galileo-computing.de"
>>> s.startswith("www.")
True
>>> s.endswith(".de")
True
>>> s.startswith("galileo", 4)
True
```

Verkettung von Elementen in sequentiellen Datentypen

Eine häufige Aufgabe ist es, eine Liste von Strings mit einem Trennzeichen zu verketteten. Beispielsweise könnte man die Namen in einer Kontaktliste seines Instant-Messengers durch Kommata getrennt ausgeben wollen. Für diesen Zweck stellt Python die Methode `join` zur Verfügung:

- ▶ `s.join(seq)`

Der Parameter `seq` ist dabei ein beliebiges iterierbares Objekt, dessen Elemente alle Strings sein müssen. Die Elemente von `seq` werden mit `s` als Trennzeichen verkettet. Kommen wir auf unser Kontaktlistenbeispiel zurück:

```
>>> kontakt_liste = ["Fix", "Foxy", "Lupo", "Dr. Knox"]
>>> ", ".join(kontakt_liste)
'Fix, Foxy, Lupo, Dr. Knox'
```

Wird für `seq` ein String übergeben, so ist das Ergebnis die Verkettung aller Buchstaben, jeweils durch `s` voneinander getrennt:

```
>>> satz = "Stoiber-Satz"
>>> "...ehm...".join(satz)
'S...ehm...t...ehm...o...ehm...i...ehm...b...ehm...e...ehm...r...ehm
...-...ehm...S...ehm...a...ehm...t...ehm...z'
```

Formatierung

Oft möchte man seine Bildschirmausgaben auf bestimmte Weise anpassen. Um beispielsweise eine dreispaltige Tabelle von Zahlen anzuzeigen, müssen abhängig von der Länge der Zahlen Leerzeichen eingefügt werden, damit die einzelnen Spalten untereinander angezeigt werden. Eine Anpassung der Ausgabe ist auch nötig, wenn Sie einen Geldbetrag ausgeben möchten, der in einer `float`-Instanz gespeichert ist, die mehr als zwei Nachkommastellen besitzt.

Für die Lösung solcher Probleme gibt es seit Python 3.0 die `format`-Methode des Datentyps `str`.¹⁰ Mithilfe von `format` können Sie in einem String Platzhalter durch bestimmte Werte ersetzen lassen. Diese Platzhalter sind durch geschweifte Klammern eingefasst und können sowohl Zahlen als auch Zeichenketten sein. Im folgenden Beispiel lassen wir die Platzhalter `{0}` und `{1}` durch zwei Zahlen ersetzen:

```
>>> "Es ist {0}:{1} Uhr".format(13, 37)
'Es ist 13:37 Uhr'
```

Wenn Zahlen als Platzhalter verwendet werden, müssen sie fortlaufend bei 0 beginnend durchnummeriert sein. Sie werden dann der Reihe nach durch die Parameter ersetzt, die der `format`-Methode übergeben wurden – der erste Parameter ersetzt `{0}`, der zweite Parameter ersetzt `{1}` und so fort.

Wie bereits erwähnt, können auch Namen als Platzhalter verwendet werden. In diesem Fall müssen Sie die Werte als Schlüsselwortparameter an die `format`-Methode übergeben:

```
>>> "Es ist {stunde}:{minute} Uhr".format(stunde=13, minute=37)
'Es ist 13:37 Uhr'
```

Als Namen für die Platzhalter kommen dabei alle Zeichenketten infrage, die auch als Variablenname in Python verwendet werden können. Insbesondere sollten Ihre Platzhalternamen nicht mit Ziffern beginnen, da sonst versucht würde, sie als Ganzzahlen zu interpretieren.

¹⁰ `format` löst den Formatierungsoperator `%` ab. Da der Operator `%` als veraltet eingestuft ist und deshalb in zukünftigen Python-Versionen nicht mehr vorhanden sein wird, sollten Sie in Ihren Programmen nur noch die Methode `format` für die Stringformatierung verwenden.

Um Python-Programmierern den Umstieg zu erleichtern, wurde die `format`-Methode auch schon in Python 2.6 hinzugefügt. Sie können also auch dann `format` einsetzen, falls Ihre Programme mit Python 2.6 funktionieren sollen.

Es ist auch möglich, nummerierte Platzhalter mit symbolischen Platzhaltern zu mischen:

```
>>> "Es ist {stunde}:{0} Uhr".format(37, stunde=13)
'Es ist 13:37 Uhr'
```

Statt der Zahlenwerte, die in den bisherigen Beispielen verwendet wurden, können Sie allgemein beliebige Objekte als Werte verwenden, sofern sie in einen String konvertiert werden können.¹¹ Im folgenden Code-Schnipsel werden verschiedene Datentypen an die `format`-Methode übergeben:

```
>>> "Liste: {0}, String: {string}, Komplexe Zahl: {1}".format(
    [1,2], 13 + 37j, string="Hallo Welt")
'Liste: [1, 2], String: Hallo Welt, Komplexe Zahl: (13+37j)'
```

Möchten Sie verhindern, dass eine geschweifte Klammer als Anfang eines Platzhalters interpretiert wird, setzen Sie zwei Klammern hintereinander. Im Ergebnis werden diese doppelten Klammern durch einfache ersetzt:

```
>>> "Nicht formatiert: {{KeinName}}. Formatiert: {test}.".format(
    test="nur ein Test")
'Nicht formatiert: {KeinName}. Formatiert: nur ein Test.'
```

Zugriff auf Member und Elemente

Neben dem einfachen Ersetzen von Platzhaltern ist es auch möglich, in dem Format-String auf Attribute des übergebenen Wertes zuzugreifen. Dazu schreiben Sie das gewünschte Attribut durch einen Punkt abgetrennt hinter den Namen des Platzhalters, genau wie dies beim normalen Attributzugriff in Python funktioniert.

Das folgende Beispiel gibt auf diese Weise den Imaginär- und Realteil einer komplexen Zahl aus:

```
>>> c = 15 + 20j
>>> "Realteil: {0.real}, Imaginaerteil: {0.imag}".format(c)
'Realteil: 15.0, Imaginaerteil: 20.0'
```

Wie Sie sehen, funktioniert der Attributzugriff auch bei nummerierten Platzhaltern.

Neben dem Zugriff auf Attribute des zu formatierenden Wertes kann auch der `[]`-Operator verwendet werden. Damit können beispielsweise gezielt Elemente einer Liste ausgegeben werden:

¹¹ Näheres dazu, wie diese Konvertierung intern abläuft und beeinflusst werden kann, finden Sie in Abschnitt 12.3, »Magic Members«.

```
>>> l = ["Ich bin der Erste!", "Nein, ich bin der Erste!"]
>> "{liste[1]}. {liste[0]}".format(liste=l)
'Nein, ich bin der Erste!. Ich bin der Erste!'
```

Auch wenn wir zu diesem Zeitpunkt keine weiteren Datentypen kennengelernt haben, die den `[]`-Operator unterstützen, ist die Anwendung in Format-Strings nicht auf sequentielle Datentypen beschränkt. Insbesondere ist diese Art von Zugriff bei Dictionarys interessant, die wir in Abschnitt 8.6, »Mappings«, behandeln werden.¹²

Formatierung der Ausgabe

Bisher haben wir mithilfe von `format` nur Platzhalter durch bestimmte Werte ersetzt, ohne dabei festzulegen, nach welchen Regeln die Ersetzung vorgenommen wird. Um dies zu erreichen, gibt es die sogenannten *Formatangaben* (engl. *format specifier*), die von dem Namen des Platzhalters durch einen Doppelpunkt getrennt angegeben werden. Um beispielsweise eine Gleitkommazahl auf zwei Nachkommastellen gerundet auszugeben, benutzt man die Formatangabe `.2f`:

```
>>> "Betrag: {0:.2f} Euro".format(13.37690)
'Betrag: 13.38 Euro'
```

Die Wirkung der Formatangaben hängt von dem Datentyp ab, der als Wert für den jeweiligen Platzhalter übergeben wird. Wir werden im Folgenden die Formatierungsmöglichkeiten für Pythons eingebaute Datentypen unter die Lupe nehmen.

Beachten Sie, dass sämtliche Formatierungsangaben optional und unabhängig voneinander sind. Sie können deshalb auch einzeln auftreten.

Bevor wir uns mit den Formatierungsmöglichkeiten im Detail beschäftigen, möchten wir Ihnen kurz den prinzipiellen Aufbau einer Formatangabe zeigen:

```
[[fill]][align][sign][#][0][minimumwidth][.precision][type]
```

Die eckigen Klammern bedeuten dabei, dass es sich bei ihrem Inhalt um optionale Angaben handelt. Im Folgenden werden alle dieser Felder einzeln diskutiert.

Minimale Breite festlegen – `minimumwidth`

Wird als Formatangabe eine einfache Ganzzahl verwendet, so legt sie die minimale Breite fest, die der ersetzte Wert einnehmen soll. Möchten wir beispiels-

¹² Bitte beachten Sie, dass die Schlüssel des Dictionarys nicht in Hochkommata eingeschlossen werden, auch wenn es sich dabei um Strings handeln sollte. Dies führt dazu, dass beispielsweise der Schlüssel `":-]"` nicht in einem Format-String verwendet werden kann.

weise eine Tabelle mit Namen ausgeben und sicherstellen, dass alles bündig untereinandersteht, erreichen wir dies folgendermaßen:

```
f = "{0:15} | {1:15}"
print(f.format("Vorname", "Nachname"))
print(f.format("Florian", "Kroll"))
print(f.format("Ramin", "Shirazi-Nejad"))
print(f.format("Sven", "Bisdorff"))
print(f.format("Kaddah", "Hotzenplotz"))
```

In diesem Miniprogramm formatieren wir die beiden Platzhalter 0 und 1 mit einer Breite von jeweils 15 Zeichen. Die Ausgabe sieht damit folgendermaßen aus:

Vorname	Nachname
Florian	Kroll
Ramin	Shirazi-Nejad
Sven	Bisdorff
Kaddah	Hotzenplotz

Sollte ein Wert länger sein als die minimale Breite, so wird die Breite des eingefügten Wertes an den Wert angepasst und nicht etwa abgeschnitten:

```
>>> "{lang:1}".format(lang="Ich bin laenger als ein Zeichen!")
'Ich bin laenger als ein Zeichen!'
```

Wie bereits gesagt, sind sämtliche Formatierungsangaben optional. Dies gilt insbesondere für die minimale Breite. Wenn also im Folgenden davon gesprochen wird, dass eine Angabe zwischen zwei anderen steht oder Ähnliches, so soll damit nur deutlich gemacht werden, wie die einzelnen Formatierungsangaben relativ zueinander stehen müssen, wenn sie denn angegeben sind.

Ausrichtung bestimmen – align

Wenn Sie die minimale Breite eines Feldes angeben, können Sie die Ausrichtung des Wertes bestimmen, falls er – wie im es im obigen Beispiel der Fall war – nicht die gesamte Breite ausfüllt.

Um beispielsweise einen Geldbetrag wie üblich rechts auszurichten, setzen Sie vor die minimale Breite ein >-Zeichen:

```
>>> "Endpreis: {sum:>5} Euro".format(sum=443)
'Endpreis:   443 Euro'
```

Insgesamt gibt es vier Ausrichtungsarten, die in der folgenden Tabelle aufgeführt sind.

Zeichen	Bedeutung
<	Der Wert wird linksbündig in den reservierten Platz eingefügt. Dies ist das Standardverhalten, sofern keine Ausrichtung angegeben ist.
>	Der Wert wird rechtsbündig in den reservierten Platz eingefügt.
=	Sorgt dafür, dass bei numerischen Werten das Vorzeichen immer am Anfang des eingefügten Wertes steht und erst danach eine Ausrichtung nach rechts erfolgt. Diese Angabe ist ausschließlich bei numerischen Werten sinnvoll und führt bei anderen Datentypen zu einem <code>ValueError</code> . Ein Beispiel zu dieser Ausrichtungsart wird weiter unten bei der Vorzeichenbehandlung gegeben.
^	Der Wert wird zentriert in den reservierten Platz eingefügt.

Tabelle 8.21 Ausrichtungsarten

Beachten Sie, dass eine Ausrichtungsangabe keinen Effekt hat, wenn der eingefügte Wert genauso lang wie oder länger als die minimale Breite ist.

Füllzeichen – fill

Vor der Ausrichtungsangabe kann das Zeichen festgelegt werden, mit dem die überschüssigen Zeichen beim Ausrichten aufgefüllt werden sollen. Standardmäßig wird dafür das Leerzeichen verwendet. Es kann aber jedes beliebige Zeichen eingesetzt werden:

```
>>> "{text:~^25}".format(text="Hallo Welt")
'-----Hallo Welt-----'
```

Hier wurde der String "Hallo Welt" zentriert von Minuszeichen umgeben eingefügt.

Behandlung von Vorzeichen – sign

Zwischen der Angabe für die minimale Breite und der Ausrichtungsangabe können Sie festlegen, wie mit dem Vorzeichen eines numerischen Wertes verfahren werden soll. Die drei möglichen Formatierungszeichen zeigt folgende Tabelle.

Zeichen	Bedeutung
+	Sowohl bei positiven als auch bei negativen Zahlenwerten wird ein Vorzeichen angegeben.
-	Nur bei negativen Zahlen wird das Vorzeichen angegeben. Dies ist das Standardverhalten.

Tabelle 8.22 Vorzeichenbehandlungsarten

Zeichen	Bedeutung
(Leerzeichen)	Mit dem Leerzeichen sorgen Sie dafür, dass bei positiven Zahlenwerten anstelle eines Vorzeichens eine Leerstelle eingefügt wird. Negative Zahlen erhalten bei dieser Einstellung ein Minus als Vorzeichen.

Tabelle 8.22 Vorzeichenbehandlungsarten (Forts.)

Wir demonstrieren die Behandlung von Vorzeichen an ein paar einfachen Beispielen:

```
>>> "Kosten: {0:+}".format(135)
'Kosten: +135'
>>> "Kosten: {0:+}".format(-135)
'Kosten: -135'
>>> "Kosten: {0:-}".format(135)
'Kosten: 135'
>>> "Kosten: {0: }".format(135)
'Kosten:  135'
>>> "Kosten: {0: }".format(-135)
'Kosten: -135'
```

Wie schon erwähnt, ist die Ausrichtungsangabe = erst bei der Verwendung mit Vorzeichen sinnvoll:

```
>>> "Kosten: {0:==+10}".format(-135)
'Kosten: -      135'
```

Wie Sie sehen, wird in dem obigen Beispiel das Minuszeichen am Anfang des reservierten Platzes eingefügt und erst danach die Zahl 135 nach rechts ausgerichtet.

Typenangaben – type

Um bei Zahlenwerten die Ausgabe weiter anpassen zu können, gibt es verschiedene Ausgabetypen, die ganz am Ende der Formatangabe eingefügt werden. Beispielsweise werden mit der Typangabe `b` Ganzzahlen in Binärschreibweise ausgegeben:

```
>>> "Lustige Bits: {0:b}".format(109)
'Lustige Bits: 1101101'
```

Insgesamt bietet Python für Ganzzahlen acht mögliche Typangaben, die nachfolgend tabellarisch aufgelistet sind.

Zeichen	Bedeutung
b	Die Zahl wird in Binärdarstellung ausgegeben.
c	Die Zahl wird als Unicode-Zeichen interpretiert. Näheres zum Thema Unicode finden Sie im Abschnitt »Zeichensätze und Sonderzeichen« weiter unten.
d	Die Zahl wird in Dezimaldarstellung ausgegeben.
o	Die Zahl wird in Oktaldarstellung ausgegeben.
x	Die Zahl wird in Hexadezimaldarstellung ausgegeben, wobei für die Ziffern a bis f Kleinbuchstaben verwendet werden.
X	Wie x, aber mit Großbuchstaben für die Ziffern von a bis f.
n	Wie d, aber es wird versucht, das für die Region übliche Zeichen zur Trennung von Zahlen zu verwenden (zum Beispiel Tausendertrennung durch einen Punkt).
(Keine Angabe)	Wird kein Typ angegeben, wird das Verhalten von d benutzt.

Tabelle 8.23 Ausgabetypen von Ganzzahlen

Es gibt noch einen alternativen Modus für die Ausgabe von Ganzzahlen, den Sie aktivieren, indem Sie zwischen die minimale Breite und das Vorzeichen eine Raute # schreiben. In diesem Modus werden die Ausgaben in Zahlensystemen mit anderer Basis als 10 durch entsprechende Präfixe gekennzeichnet:

```
>>> "{0:#b} vs. {0:b}".format(109, 109)
'0b1101101 vs. 1101101'
>>> "{0:#o} vs. {0:o}".format(109, 109)
'0o155 vs. 155'
>>> "{0:#x} vs. {0:x}".format(109, 109)
'0x6d vs. 6d'
```

Auch für Gleitkommazahlen existieren diverse Ausgabetypen, die folgende Tabelle auflistet.

Zeichen	Bedeutung
e	Die Zahl wird in wissenschaftlicher Schreibweise ausgegeben, wobei ein kleines »e« zur Trennung von Mantisse und Exponent verwendet wird.
E	Wie e, nur mit großem »E« als Trennzeichen.
f	Die Zahl wird als Dezimalzahl mit Dezimalpunkt ausgegeben.

Tabelle 8.24 Ausgabetypen für Gleitkommazahlen

Zeichen	Bedeutung
g	Die Zahl wird, wenn sie nicht zu lang ist, wie bei f ausgegeben. Für zu lange Zahlen wird automatisch der e-Typ verwendet.
G	Wie g, nur dass für zu lange Zahlen der E-Typ verwendet wird.
n	Wie g, aber es wird versucht, ein an die Region angepasstes Trennzeichen zu verwenden.
%	Der Zahlenwert wird zuerst mit hundert multipliziert und dann von einem Prozentzeichen gefolgt ausgegeben.
(Keine Angabe)	Wie g, aber es wird mindestens eine Nachkommastelle angegeben.

Tabelle 8.24 Ausgabetypen für Gleitkommazahlen (Forts.)

Das nachstehende Beispiel veranschaulicht die Formatierungen für Gleitkommazahlen:

```
>>> "{zahl:e}".format(zahl=123.456)
'1.234560e+02'
>>> "{zahl:f}".format(zahl=123.456)
'123.456000'
>>> "{zahl:n}".format(zahl=123.456)
'123.456'
>>> "{zahl:%}".format(zahl=0.75)
'75.000000 %'
```

Genauigkeit bei Gleitkommazahlen – precision

Es ist außerdem möglich, die Anzahl der Nachkommastellen bei der Ausgabe von Gleitkommazahlen festzulegen. Dazu schreiben Sie die gewünschte Anzahl durch einen Punkt abgetrennt zwischen die minimale Länge und den Ausgabetypp, wie wir es schon in unserem Einleitungsbeispiel gemacht haben:

```
>>> "Betrag: {0:.2f} Euro".format(13.37690)
'Betrag: 13.38 Euro'
```

Die überschüssigen Nachkommastellen werden bei der Formatierung nicht abgeschnitten, sondern gerundet.

Beachten Sie, dass in diesem Beispiel die minimale Länge nicht angegeben wurde und dass deshalb die Formatangabe mit einem Punkt beginnt.

Als letzte Formatierungsmöglichkeit kann eine 0 direkt vor der minimalen Breite eingefügt werden. Diese Null bewirkt, dass der überschüssige Platz mit Nullen aufgefüllt und das Vorzeichen am Anfang des reservierten Platzes eingefügt wird.

Damit ist dieser Modus gleichwertig mit der Ausrichtungsart = und dem Füllzeichen 0:

```
>>> "Es gilt {z1:05} = {z2:0=5}.".format(z1=23, z2=23)
'Es gilt 00023 = 00023.'
```

Zeichensätze und Sonderzeichen

Bisher haben wir uns der Einfachheit halber nur mit Strings beschäftigt, die keine Sonderzeichen (wie Umlaute oder das €-Zeichen) enthalten. Die Besonderheiten beim Umgang mit solchen Zeichen liegen zum Teil an der geschichtlichen Entwicklung der Zeichenkodierung. Deshalb werden wir diese im Folgenden kurz umreißen.

Zuerst müssen wir eine Vorstellung davon entwickeln, wie ein Computer intern mit Zeichenketten umgeht. Generell lässt sich sagen, dass der Computer eigentlich überhaupt keine Zeichen kennt, da sich in seinem Speicher nur Zahlen befinden. Um trotzdem Bildschirmausgaben zu produzieren oder andere Operationen mit Zeichen durchzuführen, hat man deshalb Übersetzungstabellen, die sogenannten *Codepages* (dt. *Zeichensatztabellen*), definiert, die jedem Buchstaben eine bestimmte Zahl zuordnen. Der bekannteste und wichtigste Zeichensatz ist durch die *ASCII-Tabelle*¹³ festgelegt.

Durch diese Zuordnung werden neben den Buchstaben und Ziffern auch Satz- und einige Sonderzeichen abgebildet. Außerdem existieren nicht druckbare Steuerzeichen, wie der Tabulator oder der Zeilenvorschub. Die ASCII-Tabelle ist eine 7-Bit-Zeichenkodierung, was bedeutet, dass von jedem Buchstaben 7 Bit Speicherplatz belegt werden. Es können also $2^7 = 128$ verschiedene Zeichen abgebildet werden. Die Definition des ASCII-Zeichensatzes orientiert sich am Alphabet der englischen Sprache, das insbesondere keine Umlaute wie »ä« oder »ü« enthält. Um auch solche Sonderzeichen in Strings abspeichern zu können, erweiterte man den ASCII-Code, indem man den Speicherplatz für ein Zeichen um ein Bit auf $2^8 = 256$ Möglichkeiten erhöhte, was 128 Plätze für weitere Sonderzeichen bot. Welche Interpretation konkret für diese weiteren Plätze verwendet wird, hängt von der verwendeten Codepage ab und unterscheidet sich in der Regel zwischen verschiedenen Plattformen.

Pythons `bytes`-Datentyp implementiert einen solchen 8-Bit-String und ist im Prinzip nichts anderes als eine Kette von Bytes. Um den Zahlenwert eines Zeichens zu ermitteln, gibt es in Python die Built-in Function `ord`, die als einzigen Parameter einen String der Länge eins erwartet:

¹³ »American Standard Code for Information Interchange« (dt. »Amerikanische Standardcodierung für den Informationsaustausch«)

```
>>> ord("j")
106
>>> ord("[")
91
```

Umgekehrt liefert die Built-in Function `chr` das zu einem Byte gehörige Zeichen:

```
>>> chr(106)
'j'
>>> chr(91)
'['
```

Die Beispiele oben beziehen sich nur auf Zeichen mit Ordnungszahlen, die kleiner als 128 sind und damit noch im ASCII-Bereich liegen. Interessanter ist das folgende Beispiel:

```
>>> ord("ä")
228
```

Auf dem Computer, der dieses Beispiel ausgeführt hat, läuft eine Version von Microsoft Windows für Westeuropa, die standardmäßig eine Codepage mit dem Namen »Windows-1252« verwendet. »Windows-1252« bildet alle wichtigen Zeichen für Westeuropa, das Eurozeichen inbegriffen, ab. Wenn Sie das Beispiel ausführen und eine andere Zahl als 228 auf dem Bildschirm sehen, liegt das einfach daran, dass Ihr Computer eine andere Codepage als »Windows-1252« verwendet.

Wir haben uns bereits während der Einführung zu Strings mit Escape-Sequenzen beschäftigt. In Bezug auf Sonderzeichen spielen sie eine zentrale Rolle:

```
>>> '\xdcberpr\xfcfung der \xc4nderungen'
'Überprüfung der Änderungen'
```

Was auf den ersten Blick kryptisch erscheint, hat eine einfache Struktur: Wie Sie bereits wissen, wird durch den Backslash `\` innerhalb von String-Literalen eine Escape-Sequenz eingeleitet. Die Escape-Sequenz mit der Kennung `x` ermöglicht es, einzelne Bytes in `str`-Instanzen direkt zu kodieren. Sie erwartet eine zweistellige Hexadezimalzahl als Parameter, die direkt hinter das `x` geschrieben wird. Der Wert dieses Parameters gibt den Zahlenwert des Bytes an, im Beispiel also `0xdc` = 220 ("Ü"), `0xfc` = 252 ("ü") und `0xc4` = 196 ("Ä"). Diese Zahlen hat Python der aktuellen Codepage entnommen, in der sie genau den angegebenen Zeichen entsprechen:

```
>>> print(chr(220), chr(252), chr(196))
Ü ü Ä
```

Diese Kodierung von Sonderzeichen hat den Vorteil, dass der Quelltext nur aus normalen ASCII-Zeichen besteht und beim Abspeichern und Verteilen nicht mehr auf die verwendete Codepage geachtet werden muss.

Allerdings bringt eine solche Kodierung zwei wichtige Nachteile mit sich: Zum einen ist die Anzahl möglicher Zeichen auf 256 begrenzt, und zum anderen muss jemand, der einen so kodierten String verarbeiten will, wissen, welche Codepage verwendet wurde, weil sich viele Codepages widersprechen. Den zweiten Nachteil kann man eher als Schönheitsfehler betrachten, da eine einfache Lösung darin besteht, einfach zu jedem String die verwendete Kodierung mit anzugeben.

Ein wirklicher Mangel ist dagegen die Begrenzung der Zeichenanzahl. Stellen Sie sich einen String vor, der eine Ausarbeitung über Autoren aus verschiedenen Sprachräumen mit Originalzitaten enthält: Sie würden aufgrund der vielen verschiedenen Alphabete sehr schnell an die Grenze der 8-Bit-Kodierung stoßen und könnten das Werk nicht digitalisieren. Oder stellen Sie sich vor, Sie wollen einen Text in chinesischer Sprache kodieren, was durch die über 10.000 Schriftzeichen unmöglich würde.

Ein naheliegender Lösungsansatz für dieses Problem bestand darin, den Speicherplatz pro Zeichen zu erhöhen, was aber neue Nachteile mit sich brachte. Verwendet man beispielsweise 16 Bits für jedes einzelne Zeichen, ist die Anzahl der Zeichen immer noch auf 65.536 begrenzt, und man muss davon ausgehen, dass die Sprachen sich weiterentwickeln werden und somit auch diese Anzahl einmal nicht mehr ausreichen wird.¹⁴ Außerdem würde sich im 16-Bit-Beispiel der Speicherplatzbedarf für einen String verdoppeln, weil für jedes Zeichen doppelt so viele Bits wie bei erweiterter ASCII-Kodierung verwendet würden, und das, obwohl ein Großteil aller Texte hauptsächlich aus einer kleinen Teilmenge aller vorhandenen Zeichen besteht. Die einfache Speicherplatzerhöhung für jedes einzelne Zeichen ist also keine wirkliche Lösung, denn das Problem wird irgendwann wieder auftreten, wenn die neu gesetzte Schranke erneut überschritten wird. Außerdem wird unnötig Speicherplatz vergeudet.

Eine langfristige Lösung für das Kodierungsproblem wurde schließlich durch den Standard namens *Unicode* erarbeitet, der variable Kodierungslängen für einzelne Zeichen vorsieht. Im Prinzip ist Unicode eine riesige Tabelle, die jedem bekannten Zeichen eine Zahl, den sogenannten *Codepoint*, zuweist. Diese Tabelle wird vom *Unicode Consortium*, einer gemeinnützigen Institution, gepflegt und ständig

¹⁴ Es ist tatsächlich so, dass 16 Bit schon heute nicht mehr ausreichen, um alle Zeichen der menschlichen Sprache zu kodieren.

erweitert. Codepoints werden in der Regel als »U+x« geschrieben, wobei *x* die hexadezimale Repräsentation des Codepoints ist. Das wirklich Neue an Unicode ist das Verfahren *UTF* (Unicode Transformation Format), das Codepoints durch Byte-Folgen unterschiedlicher Länge darstellen kann. Es gibt verschiedene dieser Transformationsformate, aber das wichtigste und am weitesten verbreitete ist UTF-8. UTF-8 verwendet bis zu 7 Byte, um ein einzelnes Zeichen zu kodieren, wobei die tatsächliche Länge von der Häufigkeit des Zeichens in Texten abhängt. So lassen sich zum Beispiel alle Zeichen des ASCII-Standards mit jeweils einem Byte kodieren, das zusätzlich den gleichen Zahlenwert wie die entsprechende ASCII-Kodierung des Zeichens hat. Durch dieses Vorgehen wird erreicht, dass jeder mit ASCII kodierte String auch gültiger UTF-8-Code ist: UTF-8 ist zu ASCII abwärtskompatibel. Wie das technisch genau realisiert worden ist, soll uns an dieser Stelle nicht weiter beschäftigen, sondern uns interessiert in erster Linie, wie wir Unicode mit Python nutzen können.

Seit Python 3.0 ist der Umgang mit Unicode wesentlich komfortabler geworden, da eine klare Trennung zwischen Binärdaten (Datentyp `bytes`) und Textdaten (Datentyp `str`) eingeführt wurde. Sie müssen sich deshalb nicht mehr so intensiv wie früher mit der Kodierung von Zeichen befassen. Dennoch gibt es Situationen, in denen Sie direkt mit der Zeichenkodierung in Berührung kommen.

Wie wir bereits im Beispiel am Anfang gesehen haben, können wir Sonderzeichen in String-Literalen durch Escape-Sequenzen kodieren. Wir haben dabei Escape-Sequenzen verwendet, die mit `\x` beginnen. Diese Sequenzen sind allerdings nur für Zeichen geeignet, die einen der ersten 256 Codepoints verwenden. Für beliebige Sonderzeichen, wie zum Beispiel das Euro-Symbol € (Codepoint 8364), gibt es Escape-Sequenzen, die mit `\u` eingeleitet werden:

```
>>> s = "\u20ac"
>>> print(s)
€
```

Der neue Datentyp `str` eignet sich wunderbar für die Arbeit mit Text-Strings in Python-Programmen und vereinfacht dabei den Umgang mit internationalen Schriftzeichen enorm. Allerdings gibt es einige Besonderheiten, die bei der Verwendung des neuen `str` beachtet werden müssen.¹⁵ Unicode abstrahiert von Bytes zu Zeichen, was für den Programmierer angenehmer ist, auf Maschinenebene aber den Nachteil mit sich bringt, dass solche Strings nicht einfach in Byte-Ketten gespeichert werden können. Möchten Sie aber beispielsweise Daten auf

¹⁵ Vor allem, wenn Sie den Umgang mit 8-Bit-Strings gewohnt sind, ist hier Vorsicht geboten.

der Festplatte ablegen, sie über das Netzwerk versenden oder mit anderen Programmen austauschen, sind Sie auf die Gegebenheiten der Maschine und damit auch die Byte-Ketten beschränkt. Es muss also Möglichkeiten geben, aus einem abstrakten `str`-String eine konkrete Byte-Folge, also ein `bytes`-Objekt, zu erzeugen und umgekehrt. `str`-Instanzen haben eine Methode `encode`, die als Parameter den Namen der gewünschten Kodierung enthält, zum Beispiel `"utf8"`. Das Ergebnis dieser Umwandlung ist eine `bytes`-Instanz, die die Repräsentation des Strings in der übergebenen Kodierung enthält. Um aus einer kodierten `bytes`-Instanz wieder ein `str`-Objekt zu machen, verwenden wir die Methode `decode`. Sie erwartet als Parameter den Namen der Kodierungsvorschrift, die beim Erzeugen des Strings verwendet wurde:

```
>>> textstring = "Überprüfung der Änderungen; \u20ac"
>>> textstring
'Überprüfung der Änderungen in €'
>>> utf8bytes = textstring.encode("utf8")
>>> utf8bytes
b'\xc3\x9cberpr\xc3\xbcfung der \xc3\x84nderungen; \xe2\x82\xac'
>>> t = utf8bytes.decode("utf8")
>>> t
'Überprüfung der Änderungen; €'
```

Im Beispiel erzeugen wir zuerst die `str`-Instanz `textstring`, die neben drei direkt eingegebenen Sonderzeichen auch ein maskiertes Eurozeichen enthält. Anschließend nutzen wir die Methode `encode`, um die UTF-8-Repräsentation von `textstring` zu ermitteln und mit der Referenz `utf8bytes` zu verknüpfen. In der Ausgabe von `utf8bytes` sehen wir, dass für die Kodierung der Umlaute zwei und für die des Eurozeichens sogar drei Bytes verwendet wurden. Am Ende erhalten wir eine neue `str`-Instanz, die den gleichen Inhalt hat wie `textstring`, indem wir `utf8bytes` mithilfe von `decode` als UTF-8-String interpretieren.

Innerhalb eines einzelnen Programms ist es wenig sinnvoll, `str`-Strings erst zu kodieren und dann wieder zu dekodieren, da man intern sehr bequem mit ihnen arbeiten kann. Wichtig wird die Kodierung erst, wenn Sie die enthaltenen Daten senden oder speichern möchten, wobei der Kommunikationskanal oder das Speichermedium nur mit Bytes arbeiten kann.

Folgendes Schema veranschaulicht den Transfer von Unicode mithilfe von Kodierung und Dekodierung:

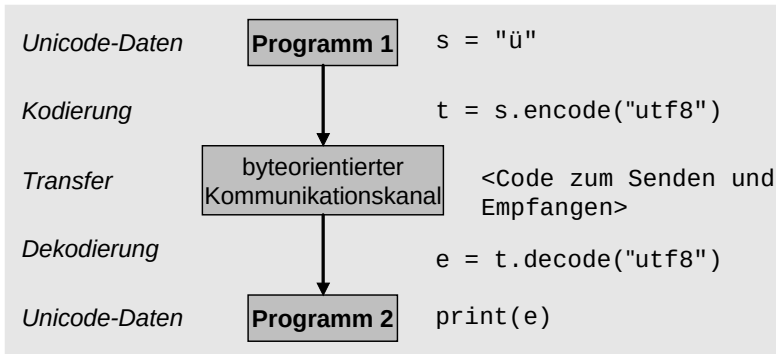


Abbildung 8.6 Schematische Darstellung eines Unicode-Transfers

Angenommen, Programm 1 erzeugt einen String `s`, der zum Beispiel ein »ü« enthält. Nun soll diese Zeichenkette über eine Netzwerkverbindung, die nur Bytefolgen übertragen kann, an Programm 2 gesendet werden. Dazu wird `s` zuerst in sein UTF-8-Äquivalent überführt und dann – wie genau ist hier nicht wichtig – über das Netzwerk an Programm 2 gesendet, wo es wieder dekodiert und anschließend verwendet werden kann.

Als Faustregeln für den Umgang mit den Datentypen `str` und `bytes` können Sie sich Folgendes merken:

1. Benutzen Sie `bytes` ausschließlich für Binärdaten.
2. Verwenden Sie für alle Textdaten, die das Programm verwendet, `str`-Instanzen.
3. Kodieren Sie `str`-Daten beim Speichern oder beim Datenversand zu anderen Programmen.
4. Gewinnen Sie beim Lesen und Empfangen der Daten mit dem entsprechenden Dekodierungsverfahren wieder die `str`-Instanzen zurück.

Wenn Sie diese Regeln konsequent einhalten, kann das Programm mit beliebigen Sonderzeichen umgehen, ohne dass besondere Anpassungen notwendig werden. Dadurch wird nicht nur die Übersetzung, sondern auch der allgemeine Umgang mit Textdaten vereinfacht, weil sich der Programmierer nicht mehr mit den Beschränkungen der Maschine beschäftigen muss. Er muss nur dafür Sorge tragen, dass die Schnittstellen nach außen enkodierte Daten bereitstellen.

Codecs

Bis jetzt sind wir nur mit den beiden Kodierungsverfahren »Windows-1252« und »UTF-8« in Berührung gekommen. Es gibt neben diesen beiden noch eine ganze

Reihe weiterer Verfahren, von denen Python viele von Haus aus unterstützt. Jede dieser Kodierungen hat in Python einen String als Namen, den Sie der `encode`-Methode übergeben können. Die folgende Tabelle zeigt exemplarisch ein paar dieser Namen.

Name in Python	Eigenschaften
"ascii"	Kodierung mithilfe der ASCII-Tabelle; englisches Alphabet, englische Ziffern, Satzzeichen und Steuerzeichen; ein Byte pro Zeichen.
"utf8"	Kodierung für alle Unicode-Codepoints; abwärtskompatibel mit ASCII; variable Anzahl Bytes pro Zeichen
"cp1252"	Kodierung für Westeuropa, die von Windows verwendet wird; zusätzlich zu den ASCII-Zeichen Unterstützung für europäische Sonderzeichen, insbesondere das Eurozeichen; abwärtskompatibel mit ASCII; ein Byte pro Zeichen

Tabelle 8.25 Drei der von Python unterstützten Encodings

Wenn Sie nun versuchen, einen `unicode`-String mit einem Kodierungsverfahren zu enkodieren, das nicht für alle in dem String enthaltenen Zeichen geeignet ist, führt dies zu einem Fehler (U+03a9 ist der Codepoint des großen Omega Ω):

```
>>> s = "\u03a9"
>>> print(s)
?
>>> s.encode("cp1252")
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    t = s.encode("cp1252")
  File "C:\Python30\lib\encodings\cp1252.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_table)
UnicodeEncodeError: 'charmap' codec can't encode character
'\u03a9' in position 0: character maps to <undefined>
```

Wie aus dem Beispiel ersichtlich ist, unterstützt »Windows-1252« das Omega-Zeichen nicht, weshalb das Enkodieren mit einer Fehlermeldung quittiert wird. Es ergibt sich ein Problem, wenn Sie mit Kodierungen arbeiten, die nicht jedes beliebige Zeichen verarbeiten können: Sie können nie sicher sein, dass die beispielsweise vom Benutzer eingegebenen Daten unterstützt werden, und laufen deshalb Gefahr, bei der Verarbeitung sein Programm abstürzen zu lassen. Um dieses Problem zu umgehen, bieten die Methoden `encode` und `decode` einen optionalen Parameter namens `errors` an, der die Vorgehensweise in solchen Fehlerfällen definiert. Für `errors` können die folgenden Werte übergeben werden:

Wert	Bedeutung
"strict"	Standardeinstellung. Jedes nicht kodierbare Zeichen führt zu einem Fehler.
"ignore"	Nicht kodierbare Zeichen werden ignoriert.
"replace"	Nicht kodierbare Zeichen werden durch einen Platzhalter ersetzt: beim Enkodieren durch das Fragezeichen "?", beim Dekodieren durch das Unicode-Zeichen U+FFFD.
"xmlcharrefreplace"	Nicht kodierbare Zeichen werden durch ihre XML-Entität ersetzt. ¹⁶ (Nur bei <code>encode</code> möglich.)
"backslashreplace"	Nicht kodierbare Zeichen werden durch eine Escape-Sequenz ersetzt. (Nur bei <code>encode</code> möglich.)

Tabelle 8.26 Werte für den `errors`-Parameter von `encode` und `decode`

Wir betrachten das letzte Beispiel mit anderen Werten für `errors`:

```
>>> s = "\u03a9"  
>>> print(s)  
?  
>>> s.encode("cp1252", "replace")  
b'?'  
>>> s.encode("cp1252", "xmlcharrefreplace")  
b'&#937;'  
>>> s.encode("cp1252", "backslashreplace")  
b'\\u03a9'
```

Damit es erst gar nicht nötig wird, Kodierungsprobleme durch diese Hilfsmittel zu umgehen, sollten Sie nach Möglichkeit immer zu allgemeinen Kodierungsverfahren wie UTF-8 greifen.

Encoding-Deklaration

Damit Sonderzeichen nicht nur innerhalb von Strings, sondern auch in Kommentaren geschrieben werden dürfen, muss im Kopf einer Python-Programmdatei eine sogenannte *Encoding-Deklaration* stehen. Dies ist eine Zeile, die das Encoding kennzeichnet, in dem die Programmdatei gespeichert wurde.

Das ist nur dann wichtig, wenn Sie in der Programmdatei Buchstaben oder Zeichen verwendet haben, die nicht im englischen Alphabet enthalten sind.¹⁷

¹⁶ Dabei handelt es sich um spezielle Formatierungen zur Darstellung von Sonderzeichen in XML-Dateien. Näheres zu XML-Dateien erfahren Sie in Abschnitt XML.

¹⁷ Oder Sie speichern Ihre Programme UTF-8-kodiert, was seit Python 3.0 Standard ist.

Ein Encoding ermöglicht es dem Python-Interpreter dann, diese Zeichen korrekt zuzuordnen. Eine Encoding-Deklaration sieht folgendermaßen aus und steht in der Regel direkt unter der Shebang-Zeile¹⁸ bzw. in der ersten Zeile der Programmdatei:

```
# -*- coding: cp1252 -*-
```

In diesem Fall wurde das Windows-Encoding `cp1252` verwendet.

Beachten Sie, dass aus Gründen der Übersichtlichkeit in keinem Beispielprogramm des Buchs eine Encoding-Deklaration enthalten ist. Das bedeutet aber ausdrücklich nicht, dass der Einsatz einer Encoding-Deklaration grundsätzlich falsch wäre.

Die in diesem Buch vorgestellten Beispielprogramme enthalten nicht nur keine Encoding-Deklaration, sondern sind auch ohne sie lauffähig.

8.6 Mappings

Die Kategorie *Mappings* (dt. »Zuordnungen«) enthält Datentypen, die eine Zuordnung zwischen verschiedenen Objekten herstellen.

8.6.1 Dictionary – dict

Der einzige Datentyp der Kategorie Mappings ist das *Dictionary*, wofür in Python der Name `dict` verwendet wird. Der Name des Datentyps gibt dabei schon einen guten Hinweis darauf, was sich dahinter verbirgt: Ein Dictionary enthält beliebig viele *Schlüssel-Wert-Paare* (engl. *key/value pairs*), wobei der Schlüssel nicht unbedingt, wie bei einer Liste, eine ganze Zahl sein muss. Vielleicht ist Ihnen dieser Datentyp schon von einer anderen Programmiersprache her bekannt, wo er als *assoziatives Array* (u. a. in PHP), *Map* (u. a. in C++) oder *Hash* (u. a. in Perl) bezeichnet wird. Der Datentyp `dict` ist mutable, also veränderlich.

Im folgenden Beispiel wird erklärt, wie ein `dict` mit mehreren Schlüssel-Wert-Paaren innerhalb von geschweiften Klammern erzeugt wird. Außerdem wird die Assoziation mit einem Wörterbuch ersichtlich:

```
woerterbuch = {"Germany" : "Deutschland", "Spain" : "Spanien"}
```

In diesem Fall wurde ein `dict` mit zwei Einträgen angelegt, die durch ein Komma getrennt werden. Beim ersten wurde dem Schlüssel "Germany" der Wert

¹⁸ Die Bedeutung einer Shebang-Zeile wurde in Abschnitt 3.2.1, »Shebang«, geklärt.

"Deutschland" zugewiesen. Schlüssel und Wert werden durch einen Doppelpunkt voneinander getrennt. Beachten Sie, dass Sie nicht gezwungen sind, alle Paare in eine Zeile zu schreiben. Innerhalb der geschweiften Klammern können Sie Ihren Quellcode beliebig formatieren:

```
woerterbuch = {
    "Germany" : "Deutschland",
    "Spain" : "Spanien",
    "France" : "Frankreich"
}
```

Hinter dem letzten Schlüssel-Wert-Paar kann ein weiteres Komma stehen, es wird aber nicht benötigt. Jeder Schlüssel muss im Dictionary eindeutig sein, es darf also kein zweiter Schlüssel mit demselben Namen existieren. Formal ist Folgendes zwar möglich, es bewirkt aber nur, dass das erste Schlüssel-Wert-Paar überschrieben wird.

```
d = {
    "Germany" : "Deutschland",
    "Germany" : "Pusemuckel"
}
```

Im Gegensatz dazu brauchen die Werte eines Dictionarys nicht eindeutig zu sein, dürfen also ruhig mehrfach vorkommen:

```
d = {
    "Germany" : "Deutschland",
    "Allemagne" : "Deutschland"
}
```

In den bisherigen Beispielen waren bei allen Paaren sowohl der Schlüssel als auch der Wert ein String. Das muss nicht unbedingt sein:

```
mapping = {
    0 : 1,
    "abc" : 0.5,
    1.2e22 : [1,2,3,4],
    (1,3,3,7) : "def"
}
```

In einem Dictionary können beliebige Instanzen, seien sie mutable oder immutable, als Werte verwendet werden. Bei dem Schlüssel ist zu beachten, dass nur Instanzen unveränderlicher (immutable) Datentypen verwendet werden dürfen. Dabei handelt es sich um alle bisher besprochenen Datentypen mit Ausnahme der Listen und der Dictionarys selbst. Versuchen wir beispielsweise, ein Dictio-

nary zu erstellen, in dem eine Liste als Schlüssel verwendet wird, so meldet sich der Interpreter mit einem entsprechenden Fehler:

```
>>> d = {[1,2,3] : "abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Diese Beschränkung rührt daher, dass die Schlüssel eines Dictionarys anhand eines aus ihrem Wert errechneten *Hash-Werts* verwaltet werden. Prinzipiell lässt sich aus jedem Objekt ein Hash-Wert berechnen; bei veränderlichen Objekten ist dies jedoch wenig sinnvoll, da sich der Hash-Wert bei Veränderung des Objekts ebenfalls ändern würde. Eine solche Veränderung würde beispielsweise die Schlüsselverwaltung eines Dictionarys zerstören. Aus diesem Grund sind veränderliche Objekte »unhashable«, wie obige Fehlermeldung besagt.

Bei einem Dictionary handelt es sich um ein iterierbares Objekt. Es ist daher möglich, ein Dictionary in einer `for`-Schleife zu durchlaufen. Dabei wird nicht über das komplette Dictionary iteriert, sondern nur über alle Schlüssel. Im folgenden Beispiel durchlaufen wir alle Schlüssel unseres Wörterbuchs und geben sie mit `print` aus:

```
for key in woerterbuch:
    print(key)
```

Die Ausgabe des Codes sieht erwartungsgemäß folgendermaßen aus:

```
Germany
Spain
France
```

Selbstverständlich kann in einer solchen Schleife auch auf die Werte des Dictionarys zugegriffen werden. Dazu bedient man sich des Zugriffsoperators, den wir im Folgenden unter anderem behandeln werden. Beachten Sie, dass Sie die Größe des Dictionarys nicht verändern dürfen, während es in einer Schleife durchlaufen wird. Die Größe des Dictionarys würde zum Beispiel durch das Hinzufügen oder Löschen eines Schlüssel-Wert-Paares beeinflusst. Sollten Sie es dennoch versuchen, bekommen Sie folgende Fehlermeldung angezeigt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

Diese Beschränkung gilt ausschließlich für Operationen, die die Größe des Dictionarys beeinflussen, also beispielsweise das Hinzufügen und Entfernen von Ein-

tragen. Sollten Sie in einer Schleife lediglich den korrelierenden Wert eines Schlüssels ändern, tritt keinerlei Fehler auf.

Operatoren

Bisher haben Sie gelernt, was ein Dictionary ist und wie es erzeugt wird. Außerdem sind wir auf einige Besonderheiten eingegangen. Nachfolgend besprechen wir die für Dictionarys verfügbaren Operatoren.

Operator	Beschreibung
<code>len(s)</code>	Liefert die Anzahl aller im Dictionary <code>s</code> enthaltenen Elemente.
<code>d[k]</code>	Zugriff auf den Wert mit dem Schlüssel <code>k</code>
<code>del d[k]</code>	Löschen des Schlüssels <code>k</code> und seines Wertes
<code>k in d</code>	True, wenn sich der Schlüssel <code>k</code> in <code>d</code> befindet
<code>k not in d</code>	True, wenn sich der Schlüssel <code>k</code> nicht in <code>d</code> befindet

Tabelle 8.27 Operatoren eines Dictionarys

Nachfolgend besprechen wir die Operatoren eines Dictionarys im Detail. Die meisten der Operatoren werden anhand des Dictionarys `woerterbuch` erklärt, das wir zu Beginn dieses Abschnitts eingeführt haben.

Länge eines Dictionarys

Um die Länge eines Dictionarys zu bestimmen, wird die eingebaute Funktion `len` verwendet. Die Länge entspricht dabei der Anzahl von Schlüssel-Wert-Paaren:

```
>>> len(woerterbuch)
3
```

Zugriff auf einen Wert

Um in einem Dictionary auf einen Wert zuzugreifen, schreiben Sie den entsprechenden Schlüssel in eckigen Klammern hinter den Namen des Dictionarys. Bei dem im zweiten Beispiel angelegten Wörterbuch könnte ein solcher Zugriff folgendermaßen aussehen:

```
>>> woerterbuch["Germany"]
'Deutschland'
```

Dabei erfolgt der Zugriff, indem Werte miteinander verglichen werden und nicht Identitäten. Das liegt daran, dass die Schlüssel eines Dictionarys intern durch ihren Hash-Wert repräsentiert werden, der ausschließlich anhand des Wertes einer Instanz gebildet wird. In der Praxis bedeutet dies, dass beispielsweise die Zugriffe `d[1]` und `d[1.0]` äquivalent sind.

Zu guter Letzt werfen wir noch einen Blick darauf, was passiert, wenn auf einen Wert zugegriffen werden soll, der nicht existiert. Der Interpreter antwortet mit einer Fehlermeldung:

```
>>> d = {}
>>> d[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100
```

Löschen eines Schlüssel-Wert-Paares

Um in einem Dictionary einen Eintrag zu löschen, kann das Schlüsselwort `del` in Kombination mit dem Zugriffoperator verwendet werden. Im folgenden Beispiel wird der Eintrag `"Germany" : "Deutschland"` aus dem Dictionary entfernt werden.

```
del woerterbuch["Germany"]
```

Das Dictionary selbst existiert auch dann noch, wenn es durch Löschen des letzten Eintrags leer geworden ist.

Auf bestimmte Schlüssel testen

Um ein Dictionary auf bestimmte Schlüssel zu testen, werden die Operatoren `in` und `not in` verwendet. Sie prüfen, ob sich ein Schlüssel im Dictionary befindet oder nicht, und geben das entsprechende Ergebnis als Wahrheitswert zurück:

```
>>> "France" in woerterbuch
True
>>> "Spain" not in woerterbuch
False
```

Methoden

Neben den Operatoren ist für Dictionaries eine ganze Reihe von Methoden definiert, die die Arbeit mit Dictionaries erleichtern.

Methode	Beschreibung
<code>d.clear()</code>	Löscht den Inhalt des Dictionaries <code>d</code> . Das Dictionary selbst bleibt bestehen.
<code>d.copy()</code>	Erzeugt eine Kopie von <code>d</code> . Beachten Sie, dass nur das Dictionary selbst kopiert wird. Alle Werte bleiben Referenzen auf dieselben Instanzen.

Tabelle 8.28 Methoden eines Dictionarys

Methode	Beschreibung
<code>d.items()</code>	Erlaubt es, in einer <code>for</code> -Schleife alle Schlüssel-Wert-Paare von <code>d</code> zu durchlaufen.
<code>d.keys()</code>	Erlaubt es, in einer <code>for</code> -Schleife alle Schlüssel von <code>d</code> zu durchlaufen.
<code>d.values()</code>	Erlaubt es, in einer <code>for</code> -Schleife alle Werte von <code>d</code> zu durchlaufen. ¹⁹
<code>d.update(d2)</code>	Fügt ein Dictionary <code>d2</code> zu <code>d</code> hinzu und überschreibt gegebenenfalls die Werte von bereits vorhandenen Schlüsseln.
<code>d.fromkeys(seq[, value])</code>	Erstellt ein neues Dictionary mit den Werten der Liste <code>seq</code> als Schlüssel und setzt jeden Wert initial auf <code>value</code> . Beachten Sie, dass diese Methode nichts am Dictionary <code>d</code> ändert.
<code>d.get(k[, x])</code>	Liefert <code>d[k]</code> , wenn der Schlüssel <code>k</code> vorhanden ist, ansonsten <code>x</code> .
<code>d.setdefault(k[, x])</code>	Das Gegenteil von <code>get</code> . Setzt <code>d[k] = x</code> , wenn der Schlüssel <code>k</code> nicht vorhanden ist.
<code>d.pop(k)</code>	Gibt den zum Schlüssel <code>key</code> gehörigen Wert zurück und löscht das Schlüssel-Wert-Paar aus dem Dictionary.
<code>d.popitem()</code>	Gibt ein willkürliches Schlüssel-Wert-Paar von <code>d</code> zurück und entfernt es aus dem Dictionary.

Tabelle 8.28 Methoden eines Dictionarys (Forts.)

Jetzt möchten wir alle Methoden detailliert und jeweils mit einem kurzen Beispiel im interaktiven Modus erläutern. Alle Beispiele werden dabei in folgendem Kontext erklärt:

```
>>> d = {"k1" : "v1", "k2": "v2", "k3": "v3"}
```

Es ist also in jedem Beispiel ein Dictionary `d` mit drei Schlüssel-Wert-Paaren vorhanden. In den Beispielen werden wir das Dictionary verändern und uns vom Interpreter seinen Wert ausgeben lassen. Die Ausgabe des unveränderten Dictionarys sieht folgendermaßen aus:

¹⁹ Vor Python 3.0 gaben die Methoden `items`, `keys` und `values` jeweils eine Liste mit den gewünschten Einträgen zurück. Inzwischen wird aus Effizienzgründen ein Iterator-Objekt zurückgegeben. Ein solches Iterator-Objekt lässt sich wie eine Liste in einer `for`-Schleife durchlaufen.

Sollten Sie unbedingt eine Liste benötigen, verwenden Sie `list(d.items())`, bzw. Analoges für die Methoden `keys` und `values`.

```
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}
```

Sie können dabei jedes Beispiel für sich betrachten und von diesen Grundvoraussetzungen ausgehen. Änderungen, die in einem Beispiel an dem Dictionary *d* durchgeführt werden, wirken sich nicht auf die Folgebeispiele aus.

d.clear()

Die Methode `clear` löscht alle Schlüssel-Wert-Paare von *d*. Sie hat dabei nicht den gleichen Effekt wie `del d`, da das Dictionary selbst nicht gelöscht, sondern nur geleert wird:

```
>>> d.clear()
>>> d
{}
```

d.copy()

Die Methode `copy` erzeugt eine Kopie des Dictionarys *d*. Beachten Sie, dass zwar das Dictionary selbst kopiert wird, es sich bei den Werten aber nach wie vor um Referenzen auf dieselben Objekte handelt.

```
>>> e = d.copy()
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}
```

d.items()

Die Methode `items` erlaubt es, in einer `for`-Schleife über alle Schlüssel-Wert-Paare zu iterieren. Das könnte zum Beispiel folgendermaßen aussehen:

```
for paar in d.items():
    print(paar)
```

In jedem Schleifendurchlauf enthält die Variable `paar` das jeweilige Schlüssel-Wert-Paar als Tupel. Dementsprechend sieht die Ausgabe des Beispiels so aus:

```
('k3', 'v3')
('k2', 'v2')
('k1', 'v1')
```

d.keys()

Die Methode `keys` erlaubt es, ähnlich wie `items`, in einer `for`-Schleife alle Schlüssel zu durchlaufen. Im folgenden Beispiel werden alle im Dictionary *d* vorhandenen Schlüssel mit `print` ausgegeben:

```
for key in d.keys():
    print(key)
```

Wir haben eingangs gesagt, dass es keiner speziellen Methode bedarf, um alle Schlüssel eines Dictionarys zu durchlaufen. Die Methode `keys` kann problemlos durch folgenden Code umgangen werden:

```
for key in d:
    print(key)
```

Beide Beispiele sind äquivalent und erzeugen folgende Ausgabe:

```
k3
k2
k1
```

d.values()

Die Methode `values` verhält sich ähnlich wie `keys`, mit dem Unterschied, dass sie es ermöglicht, alle Werte zu durchlaufen:

```
for value in d.values():
    print(value)
```

Das Beispiel erzeugt folgende Ausgabe:

```
v3
v2
v1
```

d.update(d2)

Die Methode `update` erweitert das Dictionary `d` um die Schlüssel und Werte des Dictionarys `d2`, das der Methode als Parameter übergeben wird:

```
>>> d.update({"k4" : "v4"})
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1', 'k4': 'v4'}
```

Sollten beide Dictionarys über einen gleichen Schlüssel verfügen, so wird der mit diesem Schlüssel verbundene Wert in `d` mit dem aus `d2` überschrieben:

```
>>> d.update({"k1" : "python rulez"})
{'k3': 'v3', 'k2': 'v2', 'k1': 'python rulez'}
```

d.fromkeys(seq[, value])

Die Methode `fromkeys` erzeugt ein neues Dictionary und verwendet dabei die Einträge der Liste `seq` als Schlüssel. Der Parameter `value` ist optional. Sollte er jedoch angegeben werden, so wird er als Wert eines jeden Schlüssel-Wert-Paars verwendet:

```
>>> d.fromkeys([1,2,3], "python")
{1: 'python', 2: 'python', 3: 'python'}
```

Wird der Parameter *value* ausgelassen, so wird stets `None` als Wert eingetragen:

```
>>> d.fromkeys([1,2,3])
{1: None, 2: None, 3: None}
```

d.get(k[, x])

Die Methode `get` ermöglicht den Zugriff auf einen Wert des Dictionarys. Im Gegensatz zum Zugriffoperator wird aber keine Exception erzeugt, wenn der Schlüssel nicht vorhanden ist. Stattdessen wird in diesem Fall der optionale Parameter *x* zurückgegeben. Sollte *x* nicht angegeben worden sein, so wird er als `None` angenommen. Die Methode `get` kann also als Ersatz für folgenden Code gesehen werden:

```
if k in d:
    wert = d[k]
else:
    wert = x
```

Die Methode `get` kann folgendermaßen verwendet werden:

```
>>> d.get("k2", 1337)
'v2'
>>> d.get("k5", 1337)
1337
```

d.setdefault(k[, x])

Die Methode `setdefault` fügt das Schlüssel-Wert-Paar $\{k : x\}$ zum Dictionary *d* hinzu, sollte der Schlüssel *k* nicht vorhanden sein:

```
>>> d.setdefault("k2", 1337)
'v2'
>>> d.setdefault("k5", 1337)
1337
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1', 'k5': 1337}
```

d.pop(k)

Die Methode `pop` löscht das Schlüssel-Wert-Paar mit dem Schlüssel *k* aus dem Dictionary und gibt den Wert dieses Pairs zurück:

```
>>> d.pop("k1")
'v1'
>>> d.pop("k3")
'v3'
>>> d
{'k2': 'v2'}
```

d.popitem()

Die Methode `popitem` gibt ein willkürliches Schlüssel-Wert-Paar als Tupel zurück und entfernt es aus dem Dictionary. Beachten Sie, dass das zurückgegebene Paar zwar willkürlich, aber nicht zufällig ist:

```
>>> d.popitem()
('k3', 'v3')
>>> d
{'k2': 'v2', 'k1': 'v1'}
```

Sollte *d* leer sein, so wird eine entsprechende Exception erzeugt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

8.7 Mengen

Eine *Menge* (engl. *set*) ist eine ungeordnete Ansammlung von Elementen. Jedes Element kann sich dabei nur einmal in der Menge befinden. In Python gibt es zur Darstellung von Mengen zwei Basisdatentypen: `set` für eine veränderliche Menge sowie `frozenset` für eine unveränderliche Menge – `set` ist demnach `mutable`, `frozenset` `immutable`.

Eine leere Instanz der Datentypen `set` und `frozenset` wird folgendermaßen erzeugt:

```
s = set()
fs = frozenset()
```

Wenn die Menge bereits zum Zeitpunkt der Instantiierung Elemente enthalten soll, so können Sie sich seit Python 3.0 eines speziellen Literals für Mengen bedienen:

```
>>> s = {1, 2, 3, 99, -7}
>>> s
{3, -7, 2, 99, 1}
```

Ganz wie in der Mathematik werden die Elemente, die die Menge enthalten soll, durch Kommata getrennt in geschweifte Klammern geschrieben. Diese Schreibweise bringt ein Problem mit sich: Da die geschweiften Klammern bereits für Dictionaries verwendet werden, ist es mit diesem Literal nicht möglich, eine leere

Menge zu erzeugen – {} instantiiert stets ein leeres Dictionary. Leere Mengen müssen also wie oben gezeigt über set() instantiiert werden.

Bei einer Menge handelt es sich um ein iterierbares Objekt, das problemlos in einer for-Schleife durchlaufen werden kann. Dazu folgendes Beispiel:

```
menge = {1, 100, "a", 0.5}
for element in menge:
    print(element)
```

Dieser Code erzeugt folgende Ausgabe:

```
a
1
100
0.5
```

Operatoren

Die Datentypen set und frozenset verfügen über eine gemeinsame Schnittstelle, die im Folgenden näher erläutert werden soll. Wir möchten damit beginnen, alle gemeinsamen Operatoren zu behandeln. Der Einfachheit halber werden wir uns bei der Beschreibung der Operatoren ausschließlich auf den Datentyp set beziehen. Dennoch können sie und auch die Methoden, die später beschrieben werden, für frozenset genauso verwendet werden.

Operator	Beschreibung
len(s)	Liefert die Anzahl aller im Set s enthaltenen Elemente.
x in s	True, wenn x im Set s enthalten ist, andernfalls False
x not in s	True, wenn x nicht im Set s enthalten ist, andernfalls False
s <= t	True, wenn es sich bei der Menge s um eine Teilmenge der Menge t handelt, andernfalls False
s < t	True, wenn es sich bei der Menge s um eine echte Teilmenge ²⁰ der Menge t handelt, andernfalls False
s >= t	True, wenn es sich bei der Menge t um eine Teilmenge der Menge s handelt, andernfalls False
s > t	True, wenn es sich bei der Menge t um eine echte Teilmenge der Menge s handelt, andernfalls False

Tabelle 8.29 Operatoren der Datentypen set und frozenset

20 Eine Menge T wird »echte Teilmenge« einer zweiten Menge M genannt, wenn T Teilmenge von M ist und weniger Elemente als M enthält.

Operator	Beschreibung
$s \mid t$	Erzeugt ein neues Set, das alle Elemente von s und t enthält. Diese Operation bildet also die Vereinigungsmenge zweier Mengen.
$s \& t$	Erzeugt ein neues Set, das die Objekte enthält, die sowohl Element der Menge s als auch Element der Menge t sind. Diese Operation bildet also die Schnittmenge zweier Sets.
$s - t$	Erzeugt ein neues Set mit allen Elementen von s , außer denen, die auch in t enthalten sind. Diese Operation erzeugt also die Differenz zweier Mengen.
$s \wedge t$	Erzeugt ein neues Set, das alle Objekte enthält, die entweder in s oder in t vorkommen, nicht aber in beiden. Diese Operation bildet also die symmetrische Differenz zweier Mengen.

Tabelle 8.29 Operatoren der Datentypen set und frozenset (Forts.)

Für einige dieser Operatoren existieren auch erweiterte Zuweisungen. Beachten Sie, dass es diese Operatoren auch für den Datentyp `frozenset` gibt. Sie verändern aber keineswegs die Menge selbst, sondern erzeugen in diesem Fall eine neue `frozenset`-Instanz, die das Ergebnis der Operation enthält und von nun an von s referenziert wird.

Operator	Entsprechung
$s \mid= t$	$s = s \mid t$
$s \&= t$	$s = s \& t$
$s -= t$	$s = s - t$
$s \wedge= t$	$s = s \wedge t$

Tabelle 8.30 Operatoren des Datentyps set

Im Folgenden werden alle Operatoren anhand von Beispielen anschaulich beschrieben. Die Beispiele sind dabei in diesem Kontext zu sehen:

```
>>> s = {0,1,2,3,4,5,6,7,8,9}
>>> t = {6,7,8,9,10,11,12,13,14,15}
```

Es existieren also zwei Mengen namens s und t , die aus Gründen der Übersichtlichkeit jeweils ausschließlich über numerische Elemente verfügen. Die Mengen überschneiden sich in einem gewissen Bereich. Grafisch kann die Ausgangssituation wie in Abbildung 8.7 veranschaulicht werden. Der dunkelgraue Bereich entspricht der Schnittmenge von s und t .

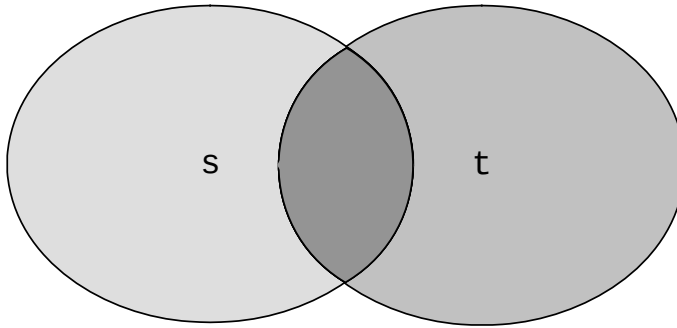


Abbildung 8.7 Die Ausgangssituation

Anzahl der Elemente

Um die Anzahl der Elemente zu bestimmen, die in einer Menge enthalten sind, wird – wie schon bei den sequentiellen Datentypen sowie dem Dictionary – die eingebaute Funktion `len` verwendet:

```
>>> len(s)
10
```

Ist ein Element im Set enthalten?

Zum Test, ob ein Element in einem Set enthalten ist, dient der Operator `in`. Zudem kann sein Gegenstück `not in` verwendet werden, um das Gegenteil zu prüfen:

```
>>> 10 in s
False
>>> 10 not in t
False
```

Handelt es sich um eine Teilmenge?

Um zu testen, ob es sich bei einem Set um eine Teilmenge eines anderen Sets handelt, werden die Operatoren `<=` und `>=` sowie `<` und `>` für echte Teilmengen, verwendet:

```
>>> u = {4,5,6}
>>> u <= s
True
>>> u < s
True
>>> u >= s
False
>>> u <= t
False
```


Beachten Sie den Unterschied zwischen *Teilmenge* ($\leq$, $\geq$) und *echter Teilmenge* ($<$, $>$) an folgendem Beispiel:

```
>>> m = {1,2,3}
>>> n = {1,2,3}
>>> m <= n
True
>>> m < n
False
```

Vereinigung zweier Mengen

Um zwei Mengen zusammenzufügen, bietet Python den Operator `|`. Er erzeugt ein neues Set, das alle Elemente enthält, die in `s` oder in `t` enthalten sind:

```
>>> s | t
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

Abbildung 8.8 veranschaulicht dies.

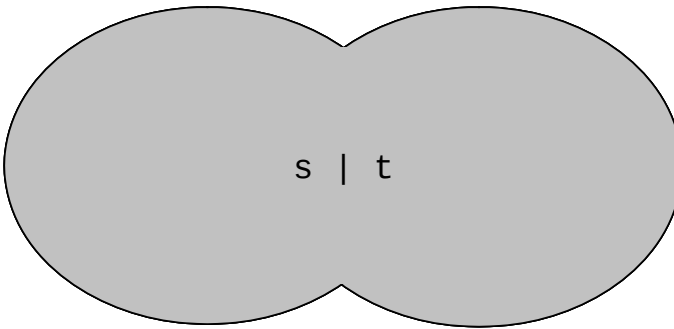


Abbildung 8.8 Vereinigungsmenge von `s` und `t`

Schnittmenge

Um die Schnittmenge zweier Mengen zu bestimmen, wird der Operator `&` verwendet. Er erzeugt ein neues Set, das alle Elemente enthält, die sowohl im ersten als auch im zweiten Operanden enthalten sind.

```
>>> s & t
{8, 9, 6, 7}
```

Auch die Auswirkungen dieses Operators veranschaulichen wir (Abbildung 8.9):

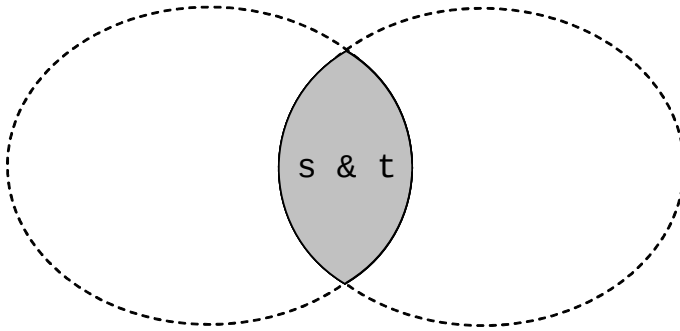


Abbildung 8.9 Schnittmenge von s und t

Differenz zweier Mengen

Um die Differenz zweier Mengen zu bestimmen, wird der Operator `-` verwendet. Es wird ein neues Set erzeugt, das alle Elemente des ersten Operanden enthält, die nicht zugleich im zweiten Operanden enthalten sind:

```
>>> s - t
{0, 1, 2, 3, 4, 5}
>>> t - s
{10, 11, 12, 13, 14, 15}
```

Grafisch ist dies in Abbildung 8.10 dargestellt.

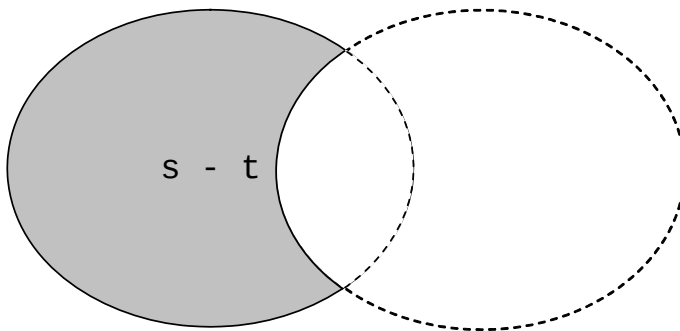


Abbildung 8.10 Differenz von s und t

Symmetrische Differenz zweier Mengen

Um die symmetrische Differenz zweier Mengen zu bestimmen, nutzen Sie den Operator `^`. Er erzeugt ein neues Set, das alle Elemente enthält, die entweder im ersten oder im zweiten Operanden vorkommen, nicht aber in beiden gleichzeitig:

```
>>> s ^ t
{0, 1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15}
```

Gönnen wir uns einen letzten Blick auf unsere Grafik in Abbildung 8.11:

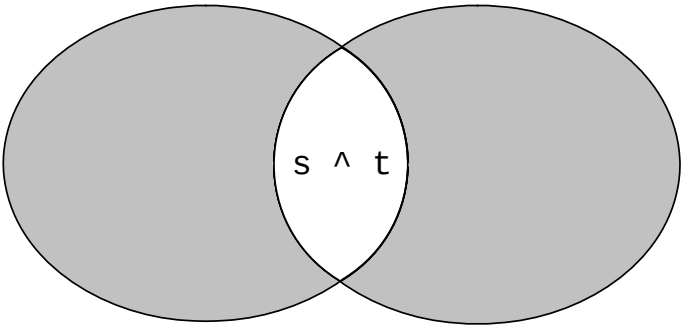


Abbildung 8.11 Symmetrische Differenz von s und t

Methoden

Die Datentypen `set` und `frozenset` verfügen über eine recht überschaubare Liste von Methoden, die in ihrem Zweck sogar größtenteils gleichbedeutend mit einem der bereits diskutierten Operatoren sind. Sie haben dennoch ihre Daseinsberechtigung, da sie aufgrund ihres Namens im Quelltext selbsterklärend sind – ganz im Gegensatz zu einem Operator, dessen Sinn sich erst nach intensiver Beschäftigung mit `set` und `frozenset` erschließt:

Methode	Beschreibung
<code>s.issubset(t)</code>	Äquivalent zu $s \leq t$
<code>s.issuperset(t)</code>	Äquivalent zu $s \geq t$
<code>s.isdisjoint(t)</code>	Prüft, ob die Mengen s und t disjunkt sind, das heißt, ob sie eine leere Schnittmenge haben.
<code>s.union(t)</code>	Äquivalent zu $s \mid t$
<code>s.intersection(t)</code>	Äquivalent zu $s \& t$
<code>s.difference(t)</code>	Äquivalent zu $s - t$
<code>s.symmetric_difference(t)</code>	Äquivalent zu $s \wedge t$
<code>s.copy()</code>	Erzeugt eine Kopie des Sets s .

Tabelle 8.31 Methoden der Datentypen `set` und `frozenset`

`s.copy()`

Eine Kopie eines Sets erzeugt die Methode `copy`:

```
>>> m = s.copy()
>>> m
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> m is s
False
>>> m == s
True
```

Wichtig ist, dass nur das Set selbst kopiert wird. Bei den enthaltenen Elementen handelt es sich sowohl in der ursprünglichen Menge als auch in der Kopie um Referenzen auf dieselben Objekte. Dies ist Ihnen bereits aus Abschnitt 8.5.1, »Listen – list«, geläufig.

8.7.1 Mengen – set

Das `set` bietet, als Datentyp für veränderliche Mengen, einige Methoden, die über den eben besprochenen Grundbestand hinausgehen. Beachten Sie, dass alle hier eingeführten Methoden nicht für `frozenset` verfügbar sind.

Methode	Beschreibung
<code>s.update(t)</code>	Äquivalent zu <code>s = t</code>
<code>s.intersection_update(t)</code>	Äquivalent zu <code>s &= t</code>
<code>s.difference_update(t)</code>	Äquivalent zu <code>s -= t</code>
<code>s.symmetric_difference_update(t)</code>	Äquivalent zu <code>s ^= t</code>
<code>s.add(e)</code>	Fügt das Objekt <code>e</code> als Element in das Set <code>s</code> ein.
<code>s.remove(e)</code>	Löscht das Element <code>e</code> aus dem Set <code>s</code> . Sollte <code>e</code> nicht vorhanden sein, wird eine Exception erzeugt.
<code>s.discard(e)</code>	Löscht das Element <code>e</code> aus dem Set <code>s</code> . Sollte <code>e</code> nicht vorhanden sein, wird dies ignoriert.
<code>s.clear()</code>	Löscht alle Elemente des Sets <code>s</code> , jedoch nicht das Set selbst.

Tabelle 8.32 Methoden des Datentyps `set`

Diese Methoden möchten wir nachfolgend anhand einiger Beispiele erläutern. Die Beispiele sind dabei in diesem Kontext zu sehen:

```
>>> s = {1,2,3,4,5}
>>> s
{1, 2, 3, 4, 5}
```

s.add(e)

Die Methode `add` fügt ein Element `e` in das Set `s` ein:

```
>>> s.add(6)
>>> s
{1, 2, 3, 4, 5, 6}
```

Sollte e bereits im Set vorhanden sein, so wird dies ignoriert.

s.remove(e)

Die Methode `remove` löscht das Element e aus dem Set s :

```
>>> s.remove(5)
>>> s
{1, 2, 3, 4, 6}
```

Sollte das zu löschende Element nicht im Set vorhanden sein, so wird eine Fehlermeldung erzeugt:

```
>>> s.remove(17)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 17
```

s.discard(e)

Die Methode `discard` löscht ein Element e aus dem Set s . Der einzige Unterschied zur Methode `remove` besteht darin, dass keine Fehlermeldung erzeugt wird, wenn e nicht in s vorhanden ist:

```
>>> s.discard(5)
>>> s
{1, 2, 3, 4}
>>> s.discard(17)
>>> s
{1, 2, 3, 4}
```

s.clear()

Die Methode `clear` entfernt alle Elemente aus dem Set s . Das Set selbst bleibt nach dem Aufruf von `clear` jedoch weiterhin vorhanden:

```
>>> s.clear()
>>> s
set()
```

8.7.2 Unveränderliche Mengen – frozenset

Da es sich bei einem `frozenset` lediglich um eine Version des `set` handelt, die nach dem Erstellen nicht mehr verändert werden darf, wurden alle Operatoren

und Methoden bereits im Rahmen der Grundfunktionalität zu Beginn des Abschnitts erklärt.

Beachten Sie jedoch, dass ein `frozenset` nicht wie ein `set` mithilfe von geschweiften Klammern instantiiert werden kann. Die Instantiierung eines Frozensets geschieht stets folgendermaßen:

```
>>> fs_leer = frozenset()
>>> fs_voll = frozenset({1,2,3,4})
>>> fs_leer
frozenset()
>>> fs_voll
frozenset({1, 2, 3, 4})
```

Bei dem Aufruf von `frozenset` kann ein iterierbares Objekt, beispielsweise ein `set`, übergeben werden, dessen Elemente in das Frozenset eingetragen werden sollen.

Beachten Sie, dass ein `frozenset` nicht nur selbst unveränderlich ist, sondern auch nur unveränderliche Elemente enthalten darf:

```
>>> frozenset([1, 2, 3, 4])
frozenset([1, 2, 3, 4])
>>> frozenset([[1, 2], [3, 4]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Welche Vorteile bietet nun das explizite Behandeln einer Menge als unveränderlich? Nun, neben gewissen Vorteilen in puncto Geschwindigkeit und Speichereffizienz kommt, wir erinnern uns, als Schlüssel eines Dictionarys nur ein unveränderliches Objekt in Frage. Innerhalb eines Dictionarys kann also ein `frozenset` sowohl als Schlüssel als auch als Wert verwendet werden. Das möchten wir im folgenden Beispiel veranschaulichen:

```
>>> d = {frozenset({1,2,3,4}) : "Hello World"}
>>> d
{frozenset({1, 2, 3, 4}): 'Hello World'}
```

Im Gegensatz dazu passiert Folgendes, wenn Sie versuchen, ein `set` als Schlüssel zu verwenden:

```
>>> d = {{1,2,3,4} : "Hello World"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Mit dem `set` haben wir den letzten Basisdatentyp behandelt. Freuen Sie sich nun darauf, das Gelernte anzuwenden. Im nächsten Kapitel werden wir über die verschiedenen Wege sprechen, wie ein Programm mit dem Benutzer interagieren kann.

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

9 Dateien

Nachdem wir Sie in die grundlegenden Sprachelemente von Python eingeführt haben, wartet hier das erste praxisorientierte Kapitel auf Sie. Bisher können Sie Instanzen diverser Datentypen erstellen und mit ihnen arbeiten. Darüber hinaus wissen Sie bereits, wie der Programmfluss durch Kontrollstrukturen beeinflusst werden kann. Es ist an der Zeit, all dieses Wissen sinnvoll zu verwenden und Sie in die Lage zu versetzen, komplexere Programme zu schreiben. Dieses Kapitel widmet sich dem Lesen und Schreiben von Dateien. Dies sollte zum Standardrepertoire eines jeden Programmierers gehören – sei es, um Daten abzuspeichern, die später wiederverwendet werden sollen, oder um eine Loggdatei zu führen, die den Programmablauf protokolliert.

Bevor wir das Lesen und Schreiben von Dateien in Python behandeln, werden wir uns ganz allgemein mit Datenströmen befassen.

9.1 Datenströme

Unter einem *Datenstrom* (engl. *data stream*) versteht man eine kontinuierliche Folge von Daten. Dabei werden zwei Typen unterschieden: Von eingehenden Datenströmen (engl. *downstreams*) können Daten gelesen und in ausgehende Datenströme (engl. *upstreams*) geschrieben werden. Bildschirmausgaben, Tastatureingaben sowie Dateien und sogar Netzwerkverbindungen werden als Datenstrom betrachtet.

Es gibt zwei Standarddatenströme, die Sie, ohne es zu wissen, bereits verwendet haben: Sowohl die Ausgabe eines Strings auf dem Bildschirm als auch eine Benutzereingabe sind nichts anderes als Operationen auf den Standardeingabe- bzw. -ausgabeströmen `stdin` und `stdout`.

Einige Betriebssysteme, darunter vor allem Windows, erlauben es, Datenströme im Text- und Binärmodus zu öffnen. Der Unterschied besteht darin, dass im Textmodus bestimmte Steuerzeichen berücksichtigt werden. So wird ein im Textmodus geöffneter Strom beispielsweise nur bis zum ersten Auftreten des sogenannten EOF-Zeichens gelesen, das das Ende einer Datei (engl. *end of file*) signalisiert. Im Binärmodus hingegen wird der vollständige Inhalt des Datenstroms eingelesen.

Als letzte Unterscheidung gibt es Datenströme, in denen man sich beliebig positionieren kann, und solche, in denen das nicht geht. Eine Datei stellt zum Beispiel einen Datenstrom dar, in dem die Schreib-/Leseposition beliebig festgelegt werden kann. Ein Beispiel für einen Datenstrom, in dem das nicht funktioniert, wäre der Standardeingabestrom (`stdin`) oder eine Netzwerkverbindung.

9.2 Daten aus einer Datei auslesen

Wir beginnen damit, Daten aus einer Datei auszulesen. Dazu müssen wir lesend auf diese Datei zugreifen. Bei der Testdatei, die wir in diesem Beispiel verwenden werden, handelt es sich um ein Wörterbuch, das in jeder Zeile ein englisches Wort und, durch ein Leerzeichen davon getrennt, seine deutsche Übersetzung enthält. Die Datei soll *woerterbuch.txt* heißen:

```
Spain Spanien
Germany Deutschland
Sweden Schweden
France Frankreich
Italy Italien
```

Im Programm würden wir die Daten, die in dieser Datei stehen, gerne so aufbereiten, dass wir später in einem Dictionary bequem auf sie zugreifen können. Als kleine Zugabe werden wir das Programm noch dahingehend erweitern, dass der Benutzer das Programm nach der Übersetzung eines englischen Begriffes fragen kann.

Zunächst einmal muss die Datei zum Lesen geöffnet werden. Dazu verwenden wir die Built-in Function `open`. Diese gibt ein sogenanntes *Dateiobjekt* zurück:

```
fobj = open("woerterbuch.txt", "r")
```

Nachdem `open` aufgerufen wurde, können mit dem Dateiobjekt Daten aus der Datei gelesen werden. Nachdem das Lesen der Datei beendet worden ist, muss sie explizit durch Aufrufen der Methode `close` geschlossen werden:

```
fobj.close()
```

Als erster Parameter von `open` übergeben wir einen String, der den Dateinamen enthält. Beachten Sie, dass hier sowohl relative als auch absolute Dateinamen erlaubt sind. In diesem Fall handelt es sich um einen relativen Dateinamen, die Datei muss sich also im selben Verzeichnis wie das Programm befinden. Der zweite Parameter ist ebenfalls ein String und spezifiziert den Modus, in dem die Datei geöffnet werden soll, wobei `"r"` für »read« steht und bedeutet, dass die Datei zum Lesen geöffnet wird. Das von der Funktion zurückgegebene Dateiojekt verknüpfen wir mit der Referenz `fobj`. Sollte die Datei nicht vorhanden sein, wird ein `IOError` erzeugt:

```
Traceback (most recent call last):
  File "woerterbuch.py", line 1, in <module>
    fobj = open("woerterbuch.txt", "r")
IOError: [Errno 2] No such file or directory: 'woerterbuch.txt'
```

Wenn ein Dateiojekt nicht mehr benötigt wird, muss es durch Aufruf der Methode `close` geschlossen werden. Nach Aufruf dieser Methode können keine weiteren Daten mehr aus dem Dateiojekt gelesen werden.

Im nächsten Schritt möchten wir die Datei zeilenweise auslesen. Dies ist relativ einfach, da das Dateiojekt zeilenweise iterierbar ist. Wir können also die altbekannte `for`-Schleife verwenden:

```
fobj = open("woerterbuch.txt", "r")
for line in fobj:
    print(line)
fobj.close()
```

In der `for`-Schleife iterieren wir zeilenweise über das Dateiojekt, wobei `line` jeweils den Inhalt der aktuellen Zeile referenziert. Momentan wird jede Zeile im Schleifenkörper lediglich ausgegeben. Wir möchten jedoch im Programm ein Dictionary aufbauen, das nach dem Einlesen der Datei die englischen Begriffe als Schlüssel und den jeweiligen deutschen Begriff als Wert enthält.

Dazu legen wir zunächst ein leeres Dictionary an:

```
woerter = {}
```

Dann wird die Datei *woerterbuch.txt* zum Lesen geöffnet und in einer Schleife über alle Zeilen der Datei iteriert:

```
fobj = open("woerterbuch.txt", "r")
for line in fobj:
    zuordnung = line.split(" ")
    woerter[zuordnung[0]] = zuordnung[1]
fobj.close()
```

Im Schleifenkörper verwenden wir nun die Methode `split` eines Strings, um die aktuell eingelesene Zeile in zwei Teile einer Liste aufzubrechen: in den Teil links vom Leerzeichen, also das englische Wort, und in den Teil rechts vom Leerzeichen, also das deutsche Wort. In der nächsten Zeile des Schleifenkörpers wird dann ein neuer Eintrag im Dictionary angelegt, mit dem Schlüssel `zuordnung[0]` (dem englischen Wort) und dem Wert `zuordnung[1]` (dem deutschen Wort).

Verändern Sie einmal den obigen Code dahingehend, dass nach dem Schließen des Dateiobjekts das erzeugte Dictionary mit `print` ausgegeben wird. Diese Ausgabe wird etwa so aussehen:

```
{'Italy': 'Italien', 'Sweden': 'Schweden\n', 'Germany': 'Deutschland\n', 'Spain': 'Spanien\n', 'France': 'Frankreich\n'}
```

Sie sehen, dass hinter jedem Wert ein `\n`, also die Escape-Sequenz für einen Zeilenumbruch, steht. Das liegt daran, dass ein Zeilenumbruch in Python als Buchstabe und damit als Teil des Dateiinhaltes angesehen wird. Deswegen wird jede Zeile einer Datei vollständig, also inklusive eines möglichen Zeilenumbruchs am Ende, eingelesen. Der Zeilenumbruch wird natürlich nur eingelesen, wenn er wirklich vorhanden ist. Das bedeutet, dass die letzte Zeile (in diesem Fall `Italy Italien`) ohne Zeilenumbruch am Ende eingelesen wird.

Den Zeilenumbruch möchten wir im endgültigen Dictionary nicht wiederfinden. Aus diesem Grund rufen wir in jedem Schleifendurchlauf die `strip`-Methode des Strings `line` auf. Diese entfernt alle Whitespace-Zeichen, unter anderem also einen Zeilenumbruch, am Anfang und Ende des Strings.

```
woerter = {}
fobj = open("woerterbuch.txt", "r")
for line in fobj:
    line = line.strip()
    zuordnung = line.split(" ")
    woerter[zuordnung[0]] = zuordnung[1]
fobj.close()
```

Damit ist der Inhalt der Datei vollständig in ein Dictionary überführt worden. Als kleine Zugabe haben wir uns vorgenommen, es dem Benutzer zu ermöglichen, Anfragen an das Programm zu senden. Im Ablaufprotokoll soll das folgendermaßen aussehen:

```
Geben Sie ein Wort ein: Germany
Das deutsche Wort lautet: Deutschland
```

```
Geben Sie ein Wort ein: Italy
Das deutsche Wort lautet: Italien
Geben Sie ein Wort ein: Greece
Das Wort ist unbekannt
```

Im Programm lesen wir in einer Endlosschleife Anfragen vom Benutzer ein. Mit dem `in`-Operator prüfen wir, ob das eingelesene Wort als Schlüssel im Dictionary vorhanden ist. Ist das der Fall, so wird die entsprechende deutsche Übersetzung ausgegeben. Sollte das eingegebene Wort nicht vorhanden sein, so wird eine Fehlermeldung ausgegeben.

```
woerter = {}

fobj = open("woerterbuch.txt", "r")
for line in fobj:
    line = line.strip()
    zuordnung = line.split(" ")
    woerter[zuordnung[0]] = zuordnung[1]
fobj.close()

while True:
    wort = input("Geben Sie ein Wort ein: ")
    if wort in woerter:
        print("Das deutsche Wort lautet:", woerter[word])
    else:
        print("Das Wort ist unbekannt")
```

Das hier vorgestellte Beispielprogramm ist weit davon entfernt, perfekt zu sein, jedoch zeigt es sehr schön, wie Dateiobjekte und auch Dictionaries sinnvoll eingesetzt werden können. Fühlen Sie sich dazu ermutigt, das Programm zu erweitern. Sie könnten es dem Benutzer beispielsweise ermöglichen, das Programm ordnungsgemäß zu beenden, Übersetzungen in beide Richtungen anbieten oder das Verwenden mehrerer Quelldateien erlauben.

Hinweis

Sie werden in Abschnitt 13.8 die `with`-Anweisung kennenlernen, mit deren Hilfe sich das Öffnen und Schließen einer Datei eleganter schreiben lässt:

```
with open("woerterbuch.txt", "r") as fobj:
    # Ihre Dateioperationen auf fobj
```

Der Vorteil ist, dass das Dateiobjekt nicht mehr explizit geschlossen werden muss. Wählen Sie hier ganz nach Ihren Vorlieben, welche Schreibweise Ihnen besser gefällt.

9.3 Daten in eine Datei schreiben

Im letzten Abschnitt haben wir uns dem Lesen von Dateien gewidmet. Dass es auch andersherum geht, soll in diesem Kapitel das Thema sein. Um eine Datei zum Schreiben zu öffnen, verwenden wir ebenfalls die Built-in Function `open`. Sie erinnern sich, dass diese Funktion einen Modus als zweiten Parameter erwartet, der im letzten Abschnitt `"r"` für »read« sein musste. Analog dazu muss `"w"` (für »write«) angegeben werden, wenn die Datei zum Schreiben geöffnet werden soll. Sollte die gewünschte Datei bereits vorhanden sein, so wird sie geleert. Nicht vorhandene Dateien werden erstellt.

```
fobj = open("ausgabe.txt", "w")
```

Nachdem alle Daten in die Datei geschrieben wurden, muss das Dateiobjekt durch Aufruf der Methode `close` geschlossen werden:

```
fobj.close()
```

Das Schreiben eines Strings in die geöffnete Datei erfolgt durch Aufruf der Methode `write` des Dateiobjekts. Das folgende Beispielprogramm versteht sich als Gegenstück zu dem im vorherigen Abschnitt. Wir gehen davon aus, dass `woerter` ein Dictionary referenziert, das englische Begriffe als Schlüssel und die deutschen Übersetzungen als Werte enthält, beispielsweise ein solches:

```
woerter = {"Germany" : "Deutschland",
           "Spain"   : "Spanien",
           "Greece"  : "Griechenland"}
```

Es handelt sich also genau um ein Dictionary, wie es von dem Beispielprogramm des letzten Abschnitts erzeugt wurde.

```
fobj = open("ausgabe.txt", "w")
for engl in woerter:
    fobj.write(engl + " " + woerter[engl] + "\n")
fobj.close()
```

Zunächst öffnen wir eine Datei namens *ausgabe.txt* zum Schreiben. Danach werden alle Schlüssel des Dictionarys `woerter` durchlaufen. In jedem Schleifendurchlauf wird mit `fobj.write` ein entsprechend formatierter String in die Datei geschrieben. Beachten Sie, dass Sie beim Schreiben einer Datei explizit durch Ausgabe eines `\n` in eine neue Zeile springen müssen.

Die von diesem Beispiel geschriebene Datei kann problemlos durch das Beispielprogramm aus dem letzten Abschnitt wieder eingelesen werden.

Hinweis

Um Sonderzeichen innerhalb einer Textdatei verwenden zu können, wird die Datei, wie Sie es bereits von Sonderzeichen in Strings her kennen, in einer bestimmten Kodierung gespeichert.

Um solche kodiert gespeicherten Dateien komfortabel lesen oder schreiben zu können, müssen Sie der Built-in Function `open` das Encoding der Datei übergeben. Näheres dazu erfahren Sie im nächsten Abschnitt.

9.4 Verwendung des Dateiobjekts

Das Dateiojekt besitzt, wie beispielsweise die komplexeren Datentypen auch, Methoden und Attribute. Einige von ihnen haben wir in den beiden vorherigen Abschnitten bereits besprochen. Wir möchten auf das Dateiojekt bezogene Attribute, Methoden und Built-in Functions noch einmal ausführlich erklären. Dazu gehen wir zunächst auf die Built-in Function `open` ein:

`open(filename[, mode[, buffering[, encoding[, errors[, newline[, closefd]]]])])`

Die Built-in Function `open` öffnet eine Datei und gibt das erzeugte Dateiojekt zurück. Mithilfe dieses Dateiobjekts können Sie nachher die gewünschten Operationen an der Datei durchführen.

Die ersten beiden Parameter haben wir in den vorherigen Abschnitten bereits besprochen. Dabei handelt es sich um den Dateinamen bzw. den Pfad zur zu öffnenden Datei (*filename*) und um den Modus (*mode*), in dem die Datei zu öffnen ist. Für den Parameter *mode* muss ein String übergeben werden, wobei alle gültigen Werte und ihre Bedeutung in der folgenden Tabelle aufgelistet sind:

Modus	Beschreibung
"r"	Die Datei wird ausschließlich zum Lesen geöffnet (r für »read«).
"w"	Die Datei wird ausschließlich zum Schreiben geöffnet. Eine eventuell bestehende Datei gleichen Namens wird überschrieben (w steht für »write«).
"a"	Die Datei wird ausschließlich zum Schreiben geöffnet. Eine eventuell bestehende Datei gleichen Namens wird nicht überschrieben, sondern erweitert (a steht für »append«).
"r+", "w+", "a+"	Die Datei wird zum Lesen und Schreiben geöffnet. Beachten Sie, dass "w+" eine eventuell bestehende Datei gleichen Namens leert.

Tabelle 9.1 Dateimodi

Modus	Beschreibung
"rb", "wb", "ab", "r+b", "w+b", "a+b"	Die Datei wird im Binärmodus geöffnet. Beachten Sie, dass in diesem Fall <code>bytes</code> -Instanzen statt <code>Strings</code> verwendet werden müssen (b steht für »binary«).

Tabelle 9.1 Dateimodi (Forts.)

Der Parameter *mode* ist optional und wird als "r" angenommen, wenn er weggelassen wird.

Über den vierten, optionalen Parameter *encoding* kann das Encoding festgelegt werden, in dem die Datei gelesen bzw. geschrieben werden soll. Die Angabe eines Encodings ergibt beim Öffnen einer Datei im Binärmodus keinen Sinn und sollte in diesem Fall weggelassen werden. Der fünfte Parameter *errors* bestimmt, wie mit Fehlern bei der Kodierung von Zeichen im angegebenen Encoding verfahren werden soll. Wird für *errors* "ignore" übergeben, werden diese schlicht ignoriert. Bei einem Wert von "strict" wird eine `ValueError`-Exception geworfen.¹

Die Parameter *buffer* steuert die interne Puffergröße, und *newline* legt die Zeichen fest, die beim Lesen oder Schreiben der Datei als Newline-Zeichen erkannt bzw. verwendet werden sollen. Diese und auch der letzte Parameter *closefd* sind sehr speziell, weswegen sie hier keine weitere Rolle spielen sollen. Weitere Informationen zu ihnen finden Sie in der Python-Dokumentation.

In der nun folgenden Tabelle möchten wir einen Überblick über die Methoden des von `open` zurückgegebenen Dateiobjekts geben. Dabei sei *f* stets ein erfolgreich erzeugtes Dateiobjekt.

Methode	Beschreibung
<code>f.close()</code>	Schließt ein bestehendes Dateiobjekt. Beachten Sie, dass danach keine Lese- oder Schreiboperationen mehr durchgeführt werden dürfen.
<code>f.flush()</code>	Verfügt, dass anstehende Schreiboperationen sofort ausgeführt werden.
<code>f.fileno()</code>	Gibt den Deskriptor der geöffneten Datei als ganze Zahl zurück.

Tabelle 9.2 Methoden eines Dateiobjekts

¹ Näheres zu den Parametern *encoding* und *errors* erfahren Sie in Abschnitt 8.5.3, »Strings – str, bytes«, im Teil über Codecs.

Methode	Beschreibung
<code>f.isatty()</code>	True, wenn das Dateiobjekt auf einem Datenstrom geöffnet wurde, der nicht an beliebiger Stelle geschrieben oder gelesen werden kann
<code>f.next()</code>	Liest die nächste Zeile der Datei ein und gibt sie als String zurück.
<code>f.read([size])</code>	Liest <code>size</code> Bytes der Datei ein, oder weniger, wenn vorher das Ende der Datei erreicht wurde. Sollte <code>size</code> nicht angegeben sein, so wird die Datei vollständig eingelesen. Die Daten werden als String zurückgegeben.
<code>f.readline([size])</code>	Liest eine Zeile der Datei ein. Durch Angabe von <code>size</code> lässt sich die Anzahl der zu lesenden Bytes begrenzen.
<code>f.readlines([sizehint])</code>	Liest alle Zeilen und gibt sie in Form einer Liste von Strings zurück. Sollte <code>sizehint</code> angegeben sein, so wird nur gelesen, bis ungefähr <code>sizehint</code> Bytes gelesen wurden. ²
<code>f.seek(offset[, whence])</code>	Setzt die aktuelle Schreib-/Leseposition in der Datei auf <code>offset</code> . Eine ausführliche Beschreibung von <code>f.seek</code> finden Sie am Ende des Kapitels.
<code>f.tell()</code>	Liefert die aktuelle Schreib-/Leseposition in der Datei.
<code>f.truncate([size])</code>	Löscht in der Datei alle Daten, die hinter der aktuellen Schreib-/Leseposition bzw. – sofern angegeben – hinter <code>size</code> stehen.
<code>f.write(str)</code>	Schreibt den String <code>str</code> in die Datei.
<code>f.writelines(sequence)</code>	Schreibt mehrere Zeilen in die Datei. <code>sequence</code> muss eine Liste von Strings sein.

Tabelle 9.2 Methoden eines Dateiobjekts (Forts.)

Darüber hinaus enthält das Dateiobjekt folgende Attribute:

Attribut	Beschreibung
<code>f.closed</code>	True, wenn die Datei geschlossen ist, andernfalls False
<code>f.encoding</code>	Enthält das Encoding, das genutzt wird, um eine Datei im Textmodus zu schreiben bzw. zu lesen. Ein Wert von <code>None</code> bedeutet, dass der Systemdefault verwendet wird.

Tabelle 9.3 Attribute eines Dateiobjekts

² In diesem Zusammenhang bedeutet »ungefähr«, dass die Anzahl der zu lesenden Bytes möglicherweise zu einer internen Puffergröße aufgerundet wird.

Attribut	Beschreibung
<code>f.errors</code>	Beschreibt das Verhalten des Dateiobjekts bei einem Encoding-Fehler. Dabei sind dieselben Werte wie beim Parameter <code>errors</code> der Funktion <code>open</code> möglich.
<code>f.mode</code>	Enthält den Modus, der beim Öffnen der Datei angegeben wurde.
<code>f.name</code>	Enthält den Namen der geöffneten Datei.
<code>f.newlines</code>	Dieses Attribut enthält alle Typen von Newline-Zeichen, die bisher vorgekommen sind, da diese von System zu System sehr verschieden sind.

Tabelle 9.3 Attribute eines Dateiobjekts (Forts.)

Viele der oben beschriebenen Methoden sind durch vorangegangene Beispiele oder den erklärenden Text ausreichend beschrieben. Wir möchten uns trotzdem noch einmal eingehend mit der Methode `seek` befassen:

`f.seek(offset[, whence])`

Setzt die Schreib-/Leseposition innerhalb der Datei. Beachten Sie, dass diese Methode je nach Modus, in dem die Datei geöffnet wurde, keine Auswirkung hat (Modus "a") oder dass die Schreibposition vor der nächsten Ausgabe zurückgesetzt werden kann (Modus "a+").

Sollte die Datei im Binärmodus geöffnet worden sein, wird der Parameter *offset* in Bytes vom Dateianfang aus gezählt. Diese Interpretation von *offset* lässt sich durch den optionalen Parameter *whence* beeinflussen:

Wert von <i>whence</i>	Interpretation von <i>offset</i>
0	Anzahl Bytes relativ zum Dateianfang
1	Anzahl Bytes relativ zur aktuellen Schreib-/Leseposition
2	Anzahl Bytes relativ zum Dateiende

Tabelle 9.4 Der Parameter *whence*

Beachten Sie, dass Sie `seek` nicht so unbeschwert verwenden können, wenn die Datei im Textmodus geöffnet wurde. Hier sollten als *offset* nur Rückgabewerte der Methode `tell` verwendet werden. Abweichende Werte können zu undefiniertem Verhalten führen.

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

10 Funktionen

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

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

Python ist, wie viele andere Programmiersprachen auch, eine *funktionale Sprache*¹. Das bedeutet, dass Ihnen ein Hilfsmittel zur Seite gestellt wird, mit dem Sie Ihr Programm in Unterprogramme unterteilen können. Ein solches Unterprogramm wird *Funktion* genannt. Dadurch wird das Problem der dramatisch abnehmenden Übersichtlichkeit gelöst, denn Funktionen ermöglichen es Ihnen, gewisse Teile des Quellcodes zu kapseln, zu gruppieren oder von anderen Teilen abzugrenzen. Des Weiteren kann eine Funktion an beliebigen Stellen des Quellcodes beliebig oft aufgerufen werden, was es dem Programmierer in der Regel ermöglicht, Quellcode ohne Codedopplungen zu schreiben.

Damit eine Funktion korrekt arbeiten kann, müssen bei ihrem Aufruf möglicherweise Informationen übertragen werden. So sollte eine Funktion, die beispiels-

1 Beachten Sie, dass es einen Unterschied zwischen *funktionalen* und *rein funktionalen* Programmiersprachen gibt. Als Vertreter rein funktionaler Programmiersprachen kann beispielsweise Haskell angesehen werden.

weise die Fakultät einer ganzen Zahl berechnet, wissen, von welcher Zahl die Fakultät zu berechnen ist. Dazu können beim Aufruf sogenannte *Parameter* übergeben werden. Zudem sollte eine Funktion dem aufrufenden, übergeordneten Programm das Ergebnis der Berechnung mitteilen können. Dazu verfügt jede Funktion über einen sogenannten *Rückgabewert*.

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

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

```
ergebnis = range(0, 10, 2)
```

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

Nachdem die Funktion abgearbeitet wurde, wird ihr Ergebnis zurückgegeben. Sie können sich bildlich vorstellen, dass der Funktionsaufruf, wie er im Quelltext steht, durch den Rückgabewert ersetzt wird. Im obigen Beispiel haben wir dem Rückgabewert von `range` direkt einen Namen zugewiesen und können ihn fortan über `ergebnis` referenzieren. So können wir beispielsweise in einer `for`-Schleife über das Ergebnis des `range`-Aufrufs iterieren:

```
>>> for i in ergebnis:
...     print(i)
...
0
2
4
6
8
```

Es ist auch möglich, das Ergebnis des `range`-Aufrufs mit `list` in eine Liste zu überführen:

```
>>> liste = list(ergebnis)
>>> liste
[0, 2, 4, 6, 8]
>>> liste[3]
6
```

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

10.1 Schreiben einer Funktion

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

- ▶ Eine Funktion muss einen *Namen* haben, über den sie in anderen Teilen des Programms aufgerufen werden kann. Die Zusammensetzung des Funktionsnamens erfolgt nach denselben Regeln wie die Namensgebung einer Referenz.
- ▶ Eine Funktion muss eine *Schnittstelle* haben, über die Informationen vom aufrufenden Programmteil in den Kontext der Funktion übertragen werden. Eine Schnittstelle kann aus beliebig vielen (unter Umständen auch keinen) Parametern bestehen. Funktionsintern wird jedem dieser Parameter ein Name gegeben. Sie lassen sich dann wie Referenzen im Funktionskörper verwenden.
- ▶ Eine Funktion muss einen *Wert* zurückgeben. Jede Funktion gibt automatisch `None` zurück, wenn der Rückgabewert nicht ausdrücklich angegeben wurde.

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

```
def Funktionsname(prm_1, ..., prm_n):
    | Anweisung
    | :
    | Anweisung
```

Abbildung 10.1 Definition einer Funktion

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

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

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

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

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

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

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

Sie sehen, dass der Quellcode sehr schön in zwei Komponenten aufgeteilt wurde: zum einen in die Funktionsdefinition oben und zum anderen in das auszuführende Hauptprogramm unten. Das Hauptprogramm besteht aus einer Endlosschleife, in der im Wesentlichen die Funktion `fak` mit der eingegebenen Zahl als Parameter aufgerufen wird.

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

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

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

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

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

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

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

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

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

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

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

Selbstverständlich können Sie in Ihrem Quelltext mehrere eigene Funktionen definieren und aufrufen. Das folgende Beispiel soll bei Eingabe einer negativen Zahl keine Fehlermeldung, sondern die Fakultät des Betrages dieser Zahl ausgeben:

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

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

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

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

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

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

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

10.2 Funktionsparameter

Wir haben bereits oberflächlich besprochen, was Funktionsparameter sind und wie sie verwendet werden können, doch das ist bei Weitem noch nicht die ganze Wahrheit. In diesem Abschnitt sollen drei Techniken eingeführt werden, die die Verwendung von Funktionsparametern bequemer oder eleganter machen. Alle drei Techniken sind mehr oder weniger speziell und somit nicht für alle Einsatzgebiete von Funktionen geeignet.

10.2.1 Optionale Parameter

Zu Beginn dieses Kapitels wurde die Verwendung einer Funktion anhand der Built-in Function `range` erklärt. Erinnern Sie sich noch daran, als `range` im Zusammenhang mit der `for`-Schleife eingeführt wurde? Wenn ja, dann wissen Sie sicherlich noch, dass unter anderem der letzte der drei Parameter optional war. Das bedeutet zunächst einmal, dass dieser Parameter beim Funktionsaufruf weggelassen werden kann. Ein optionaler Parameter muss funktionsintern mit einem Wert vorbelegt sein, üblicherweise einem Standardwert, der in einem Großteil der Funktionsaufrufe ausreichend ist. Bei der Funktion `range` regelt der dritte Parameter die Schrittweite und ist mit 1 vorbelegt. Folgende Aufrufe von `range` sind also äquivalent:

- ▶ `range(2, 10, 1)`
- ▶ `range(2, 10)`

Dies ist ein interessantes Sprachmerkmal von Python, denn oftmals hat eine Funktion ein Standardverhalten, das sich durch zusätzliche Parameter an spezielle Gegebenheiten anpassen lassen soll. In den überwiegenden Fällen, in denen das Standardverhalten jedoch genügt, wäre es umständlich, trotzdem die für diesen Aufruf völlig überflüssigen Parameter anzugeben. Deswegen sind vordefinierte Parameterwerte oft eine sinnvolle Ergänzung der eigenen Funktionsschnittstelle.

Um einen Funktionsparameter mit einem Defaultwert vorzubelegen, wird dieser Wert bei der Funktionsdefinition zusammen mit einem Gleichheitszeichen hinter den Parameternamen geschrieben. Die folgende Funktion soll, je nach Anwendung, die Summe von zwei, drei oder vier ganzen Zahlen berechnen und das Ergebnis zurückgeben. Dabei soll der Programmierer beim Aufruf der Funktion nur so viele Zahlen angeben müssen, wie er benötigt:

```
def summe(a, b, c=0, d=0):
    return a + b + c + d
```

Um eine Addition durchzuführen, müssen mindestens zwei Parameter übergeben worden sein. Die anderen beiden werden mit 0 vorbelegt. Sollten sie beim Funktionsaufruf nicht explizit angegeben werden, so fließen sie nicht in die Addition ein. Die Funktion könnte folgendermaßen aufgerufen werden:

```
summe(1, 2)
summe(1, 2, 3)
summe(1, 2, 3, 4)
```

Beachten Sie, dass optionale Parameter nur am Ende einer Funktionsschnittstelle stehen dürfen. Das heißt, dass auf einen optionalen kein nicht-optionaler Parameter mehr folgen darf. Diese Einschränkung ist wichtig, damit alle angegebenen Parameter eindeutig zuzuordnen sind.

10.2.2 Schlüsselwortparameter

Neben den bislang verwendeten sogenannten *Positional Arguments* (Positionsparameter) gibt es in Python eine weitere Möglichkeit, Parameter zu übergeben. Solche Parameter werden *Keyword Arguments* (Schlüsselwortparameter) genannt. Es handelt sich dabei lediglich um eine weitere Technik, Parameter beim Funktionsaufruf zu übergeben. An der Funktionsdefinition ändert sich nichts. Betrachten wir dazu unsere Summenfunktion, die wir im vorangegangenen Abschnitt geschrieben haben:

```
def summe(a, b, c=0, d=0):
    return a + b + c + d
```

Diese Funktion kann auch folgendermaßen aufgerufen werden:

```
summe(d=1, b=3, c=2, a=1)
```

Dazu werden im Funktionsaufruf die Parameter, wie bei einer Zuweisung, auf den gewünschten Wert gesetzt. Da bei der Übergabe der jeweilige Parametername angegeben werden muss, ist die Zuordnung unter allen Umständen eindeutig. Das erlaubt es dem Programmierer, Schlüsselwortparameter in beliebiger Reihenfolge anzugeben.

Es ist möglich, beide Formen der Parameterübergabe zu kombinieren. Dabei ist zu beachten, dass keine Positional Arguments auf Keyword Arguments folgen dürfen, Letztere also immer am Ende des Funktionsaufrufs stehen müssen.

```
summe(1, 2, c=10, d=11)
```

Beachten Sie außerdem, dass nur solche Parameter als Keyword Arguments übergeben werden dürfen, die im selben Funktionsaufruf nicht bereits als Positional Argument übergeben wurden.

Zum Schluss möchten wir noch anmerken, dass optionale Parameter auch unter Verwendung von Keyword Arguments wie erwartet funktionieren.

10.2.3 Beliebige Anzahl von Parametern

Für beide Formen der Parameterübergabe (Positional und Keyword) gibt es eine Notation, die es einer Funktion ermöglicht, beliebig viele Parameter entgegenzunehmen. Bleiben wir zunächst einmal bei den Positional Arguments. Betrachten Sie dazu folgende Funktionsdefinition:

```
def funktion(a, b, *weitere):
    print("Feste Parameter:", a, b)
    print("Weitere Parameter:", weitere)
```

Zunächst einmal werden ganz klassisch zwei Parameter `a` und `b` festgelegt und zusätzlich ein dritter namens `weitere`. Wichtig ist der Stern vor seinem Namen. Bei einem Aufruf dieser Funktion würden `a` und `b`, wie Sie das bereits kennen, die ersten beiden übergebenen Instanzen referenzieren. Interessant ist, dass `weitere` fortan ein Tupel referenziert, das alle zusätzlich übergebenen Instanzen enthält. Anschaulich wird dies, wenn wir folgende Funktionsaufrufe betrachten:

```
funktion(1, 2)
funktion(1, 2, "Hallo Welt", 42, [1,2,3,4])
```

Die Ausgabe der Funktion im Falle des ersten Aufrufs wäre:

```
Feste Parameter: 1 2
Weitere Parameter: ()
```

Der Parameter `weitere` referenziert also ein leeres Tupel. Im Falle des zweiten Aufrufs sähe die Ausgabe folgendermaßen aus:

```
Feste Parameter: 1 2
Weitere Parameter: ('Hallo Welt', 42, [1, 2, 3, 4])
```

Der Parameter `weitere` referenziert nun ein Tupel, in dem alle über `a` und `b` hinausgehenden Instanzen in der Reihenfolge enthalten sind, wie sie übergeben wurden.

Diese Art, einer Funktion das Entgegennehmen beliebig vieler Parameter zu ermöglichen, funktioniert ebenso für Keyword Arguments. Der Unterschied besteht darin, dass der Parameter, der alle weiteren Instanzen enthalten soll, in der Funktionsdefinition mit zwei Sternen geschrieben werden muss, sowie darin, dass er später kein Tupel, sondern ein Dictionary referenziert. Dieses Dictionary enthält den jeweiligen Parameternamen als Schlüssel und die übergebene Instanz als Wert. Betrachten Sie dazu folgende Funktionsdefinition:

```
def funktion(a, b, **weitere):
    print("Feste Parameter:", a, b)
    print("Weitere Parameter:", weitere)
```

und diese beiden dazu passenden Funktionsaufrufe:

```
funktion(1, 2)
funktion(1, 2, johannes="ernesti", peter="kaiser")
```

Die Ausgabe nach dem ersten Funktionsaufruf sähe folgendermaßen aus:

```
Feste Parameter: 1 2
Weitere Parameter: {}
```

Der Parameter `weitere` referenziert also ein leeres Dictionary. Nach dem zweiten Aufruf sähe die Ausgabe so aus:

```
Feste Parameter: 1 2
Weitere Parameter: {'johannes': 'ernesti', 'peter': 'kaiser'}
```

Beide Techniken können zusammen verwendet werden, wie folgende Funktionsdefinition zeigt:

```
def funktion(*positional, **keyword):
    print(positional)
    print(keyword)
```

Der Funktionsaufruf

```
funktion(1, 2, 3, 4, hallo="welt", key="word")
```

gibt diese Werte aus:

```
(1, 2, 3, 4)
{'hallo': 'welt', 'key': 'word'}
```

Sie sehen, dass `positional` ein Tupel mit allen Positions- und `keyword` ein Dictionary mit allen Schlüsselwortparametern referenziert.

Entpacken einer Parameterliste

Wenn eine Funktion beliebige Parameter erwartet, kommen diese funktionsintern gesammelt entweder in Form eines Tupels (Positional Arguments) oder eines Dictionarys (Keyword Arguments) an. Gelegentlich möchte man die in diesem Tupel bzw. Dictionary enthaltenen Parameter an eine andere Funktion weiterreichen. Dabei soll aber jedes Element des Tupels bzw. jedes Schlüssel-Wert-Paar des Dictionarys beim Aufruf der zweiten Funktion als eigenständiger Parameter übergeben werden. Dieser Vorgang wird *Entpacken* eines Tupels oder eines Dictionarys genannt.

Das Entpacken eines Tupels soll an einem Beispiel verdeutlicht werden. Dazu definieren wir zwei Funktionen, `f1` und `f2`, wobei `f1` über eine feste Schnittstelle verfügt, während `f2` beliebig viele Positional Arguments akzeptiert. Die Funktion `f2` soll die ihr übergebenen Parameter, auf die sie in Form eines Tupels zugreifen kann, entpacken und an die Funktion `f1` weiterreichen.

```
def f1(a, b, c, d):
    print("Parameter:", a, b, c, d)

def f2(*prm):
    f1(*prm)
```

Zur Funktion `f1` muss nicht viel gesagt werden: Sie erwartet vier Parameter und gibt diese auf dem Bildschirm aus. Viel interessanter ist die Funktion `f2`, die eine beliebige Anzahl Positionsparameter erwartet und diese im Funktionskörper an die Funktion `f1` weiterreichen soll. Das Tupel `prm`, das die der Funktion `f2` übergebenen Parameter enthält, kann im Funktionsaufruf von `f1` durch ein vorangestelltes Sternchen (*) entpackt werden. Wenn das Tupel vier Elemente enthält, kommen diese in Form der Parameter `a`, `b`, `c` und `d` bei `f1` an. Sollte das Tupel weniger oder mehr Elemente enthalten, verursacht dies einen Fehler. So gibt `f1` bei einem Funktionsaufruf von

```
f2(1, 2, 3, 4)
```

den Text

```
Parameter: 1 2 3 4
```

auf dem Bildschirm aus.

Analog dazu kann ein Dictionary mit zwei vorangestellten Sternchen entpackt werden:

```
def f1(a, b, c, d):
    print("Parameter:", a, b, c, d)
```

```
def f2(**prm):
    f1(**prm)
```

In diesem Fall führt der Funktionsaufruf

```
f2(a=5, b=6, c=7, d=8)
```

zur erwarteten Bildschirmausgabe:

```
Parameter: 5 6 7 8
```

Beachten Sie allgemein, dass die hier vorgestellte Syntax nur innerhalb eines Funktionsaufrufs verwendet werden darf und außerhalb dessen zu einem Fehler führt.

10.2.4 Seiteneffekte

Bisher haben wir diese Thematik geschickt umschifft, doch Sie sollten immer im Hinterkopf behalten, dass sogenannte *Seiteneffekte* (engl. *side effects*) immer dann auftreten können, wenn eine Instanz eines mutable Datentyps, also zum Beispiel einer Liste oder eines Dictionarys, als Funktionsparameter übergeben wird.

Um dies verstehen zu können, müssen wir zunächst allgemein darüber sprechen, auf welchen Wegen Funktionsparameter übergeben werden. In der Programmierung unterscheidet man dabei grob zwei Arten:

- ▶ Bei einem *Call-by-Value* wird funktionsintern mit Kopien der als Parameter übergebenen Instanzen gearbeitet. Das hat den Vorteil, dass eine Funktion keine ungewollten Änderungen im Hauptprogramm bewirken kann, erzeugt jedoch unter Umständen einen erheblichen Overhead, da auch größere Instanzen wie Listen oder Dictionarys bei jedem Funktionsaufruf kopiert werden müssten.
- ▶ Das gegensätzliche Prinzip wird *Call-by-Reference* genannt und bedeutet, dass funktionsintern mit Referenzen auf die im Hauptprogramm befindlichen Instanzen gearbeitet wird. Der Vorteil dieser Methode liegt auf der Hand: Es müssen keine Instanzen kopiert werden, und ein Funktionsaufruf wird dementsprechend performant. Der größte Nachteil der Referenzparameter ist, dass innerhalb einer Funktion eine übergebene Instanz so verändert werden kann, dass sich dies auch im Hauptprogramm auswirkt. Solche Änderungen sind vom Programmierer meist nicht erwünscht und werden als Seiteneffekte bezeichnet.

In Python werden Funktionsparameter grundsätzlich »by Reference« übergeben. Betrachten Sie dazu folgendes Beispiel, das sich zunächst auf unveränderliche Datentypen wie `int` oder `float` beschränkt:

```

>>> def f(a, b):
...     print(id(a))
...     print(id(b))
...
>>> p = 1
>>> q = 2
>>> id(p)
134537016
>>> id(q)
134537004
>>> f(p, q)
134537016
134537004

```

Im interaktiven Modus definieren wir zuerst eine Funktion, die zwei Parameter `a` und `b` erwartet und deren jeweilige Identität ausgibt. Nachfolgend werden zwei Referenzen `p` und `q` angelegt, die je eine Instanz des Datentyps `int` referenzieren. Dann lassen wir uns die Identitäten der beiden Referenzen ausgeben und rufen die angelegte Funktion `f` auf. Sie sehen, dass die ausgegebenen Identitäten gleich sind. Es handelt sich also sowohl bei `p` und `q` als auch bei `a` und `b` im Funktionskörper um Referenzen auf dieselben Instanzen.

Trotzdem ist die Verwendung eines immutable Datentyps grundsätzlich frei von Seiteneffekten, da dieser bei Veränderung automatisch kopiert wird und alte Referenzen davon nicht berührt werden. Sollten wir also beispielsweise `a` im Funktionskörper um eins erhöhen, so werden nachher `a` und `p` verschiedene Instanzen referenzieren. Dies ermöglicht es uns, mit unveränderlichen Parametern umzugehen, als wären sie »by Value« übergeben worden.

Diese Sicherheit können uns mutable Datentypen nicht geben. Dazu folgendes Beispiel:

```

def f(liste):
    liste[0] = 42
    liste += [5,6,7,8,9]

zahlen = [1,2,3,4]

print(zahlen)
f(zahlen)
print(zahlen)

```

Zunächst wird eine Funktion definiert, die eine Liste als Parameter erwartet und diese im Funktionskörper verändert. Im Hauptprogramm wird eine Liste angelegt

und ausgegeben. Danach wird die Funktion aufgerufen und die Liste erneut ausgegeben. Die Ausgabe des Beispiels sieht folgendermaßen aus:

```
[1, 2, 3, 4]
[42, 2, 3, 4, 5, 6, 7, 8, 9]
```

Es ist zu erkennen, dass sich die Änderungen nicht allein auf den Kontext der Funktion beschränken, sondern sich auch im Hauptprogramm auswirken. Wenn eine Funktion nicht nur lesend auf eine Instanz eines veränderlichen Datentyps zugreifen muss und Seiteneffekte nicht ausdrücklich erwünscht sind, sollten Sie innerhalb der Funktion oder bei der Parameterübergabe eine Kopie der Instanz erzeugen. Das könnte in Bezug auf das obige Beispiel so aussehen:

```
f(zahlen[:])2
```

Neben den bisher besprochenen Referenzparametern existiert eine weitere, seltenere Form von Seiteneffekten, die auftritt, wenn ein veränderlicher Datentyp als Defaultwert eines Parameters verwendet wird:

```
>>> def f(a=[1,2,3]):
...     a += [4,5]
...     print(a)
...
>>> f()
[1, 2, 3, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5, 4, 5]
```

Wir definieren im interaktiven Modus eine Funktion, die einen einzigen Parameter erwartet, der mit einer Liste vorbelegt ist. Im Funktionskörper wird diese Liste um zwei Elemente vergrößert und ausgegeben. Nach mehrmaligem Aufrufen der Funktion ist zu erkennen, dass es sich bei dem Defaultwert augenscheinlich immer um dieselbe Instanz gehandelt hat.

Das liegt daran, dass eine Instanz, die als Defaultwert genutzt wird, nur einmalig und nicht bei jedem Funktionsaufruf neu erzeugt wird. Grundsätzlich sollten Sie

2 Sie erinnern sich, dass beim Slicen einer Liste stets eine Kopie derselben erzeugt wird. Im Beispiel wurde das Slicing genutzt, um eine vollständige Kopie der Liste zu erzeugen, indem weder ein Start- noch ein Endindex angegeben wurde.

also darauf verzichten, Instanzen unveränderlicher Datentypen als Defaultwert zu verwenden. Schreiben Sie Ihre Funktionen stattdessen folgendermaßen:

```
def f(a=None):
    if a is None:
        a = [1,2,3]
```

Selbstverständlich können Sie statt `None` eine Instanz eines beliebigen anderen immutable Datentypen verwenden, ohne dass Seiteneffekte auftreten.

10.3 Lokale Funktionen

Es ist möglich, sogenannte *lokale Funktionen* zu definieren. Das sind Funktionen, die im lokalen Namensraum einer anderen Funktion angelegt werden und nur dort gültig sind. Das folgende Beispiel zeigt eine solche Funktion:

```
def globale_funktion(n):
    def lokale_funktion(n):
        return n**2

    return lokale_funktion(n)
```

Innerhalb der globalen Funktion `globale_funktion` wurde eine lokale Funktion namens `lokale_funktion` definiert. Beachten Sie, dass der jeweilige Parameter `n` trotz des gleichen Namens nicht zwangsläufig denselben Wert referenziert. Die lokale Funktion kann im Namensraum der globalen Funktion völlig selbstverständlich wie jede andere Funktion auch aufgerufen werden.

Da sie einen eigenen Namensraum besitzt, hat die lokale Funktion keinen Zugriff auf lokale Referenzen der globalen Funktion. Um dennoch einige ausgewählte Referenzen an die lokale Funktion durchzuschleusen, bedient man sich eines Tricks mit vorbelegten Funktionsparametern:

```
def globale_funktion(n):
    def lokale_funktion(n=n):
        return n**2

    return lokale_funktion()
```

Wie Sie sehen, muss der lokalen Funktion der Parameter `n` beim Aufruf nicht mehr explizit übergeben werden. Er wird vielmehr implizit in Form eines vorbelegten Parameters übergeben.

10.4 Anonyme Funktionen

Mithilfe des Schlüsselwortes `lambda` können kleine, anonyme Funktionen erstellt werden. Solche Funktionen werden üblicherweise für häufig auftretende Berechnungen verwendet, um sich alle Vorteile einer echten Funktion zu erhalten, diese gleichzeitig aber nicht aufwendig definieren zu müssen. Eine anonyme Funktion wird zum Beispiel folgendermaßen erzeugt:

```
f = lambda x: x * 3 + 7
```

Auf das Schlüsselwort `lambda` folgen eine Parameterliste und ein Doppelpunkt. Hinter dem Doppelpunkt muss ein beliebiger arithmetischer oder logischer Ausdruck stehen, der nach seiner Auswertung im Rückgabewert der Funktion mündet. Beachten Sie, dass die Beschränkung auf einen arithmetischen Ausdruck zwar die Verwendung von Kontrollstrukturen ausschließt, nicht aber die Verwendung einer Conditional Expression.

Eine `lambda`-Form ergibt ein Funktionsobjekt und kann, wie im Beispiel geschehen, referenziert werden. Der Aufruf der Funktion läuft wie gewohnt ab:

```
r = f(10)
```

Der Rückgabewert wäre in diesem Fall 37. Betrachten wir noch ein etwas komplexeres Beispiel einer anonymen Funktion mit drei Parametern:

```
f = lambda x, y, z: (x - y) * z
```

Beachten Sie, dass Sie im »Funktionskörper« keine Kontrollstrukturen verwenden dürfen, es muss ein rein arithmetischer Ausdruck sein. Jede `lambda`-Form kann ebenso durch eine »echte« Funktion ersetzt werden. Das entsprechende Gegenstück zum obigen Beispiel sähe so aus:

```
def f(x, y, z):
    return (x - y) * z
```

Anonyme Funktionen können auch aufgerufen werden, ohne sie vorher referenzieren zu müssen. Dazu muss der `lambda`-Ausdruck in Klammern gesetzt werden:

```
(lambda x, y, z: (x - y) * z)(1, 2, 3)
```

10.5 Namensräume

Bisher wurde ein Funktionskörper als abgekapselter Bereich betrachtet, der ausschließlich über Parameter bzw. den Rückgabewert Informationen mit dem Hauptprogramm austauschen kann. Das ist zunächst auch gar keine schlechte

Sichtweise, denn so hält man seine Schnittstelle »sauber«. In manchen Situationen ist es aber sinnvoll, eine Funktion über ihren lokalen Namensraum hinaus wirken zu lassen, was in diesem Kapitel thematisiert werden soll.

10.5.1 Zugriff auf globale Variablen – global

Zunächst einmal müssen zwei Begriffe unterschieden werden. Wenn wir uns im Kontext einer Funktion, also im Funktionskörper befinden, dann können wir dort selbstverständlich Referenzen und Instanzen erzeugen und verwenden. Diese haben jedoch nur unmittelbar in der Funktion selbst Gültigkeit. Sie existieren im sogenannten *lokalen Namensraum*. Im Gegensatz dazu existieren Referenzen des Hauptprogramms im *globalen Namensraum*.

Begrifflich wird auch zwischen *globalen Referenzen* und *lokalen Referenzen* unterschieden. Dazu folgendes Beispiel:

```
def f():
    a = "lokaler String"
b = "globaler String"
```

Wie stark zwischen globalem und lokalem Namensraum unterschieden wird, zeigt das folgende Beispiel:

```
def f(a):
    print(a)

a = 10
f(100)
```

In diesem Beispiel existiert sowohl im globalen als auch im lokalen Namensraum eine Referenz namens *a*. Im globalen Namensraum referenziert sie die ganze Zahl 10 und im lokalen Namensraum der Funktion den übergebenen Parameter, in diesem Fall die ganze Zahl 100. Es ist wichtig zu verstehen, dass diese beiden Referenzen nichts miteinander zu tun haben, da sie in verschiedenen Namensräumen existieren.

Im lokalen Namensraum des Funktionskörpers kann jederzeit lesend auf eine globale Referenz zugegriffen werden, solange keine lokale Referenz gleichen Namens existiert:

```
def f():
    print(s)

s = "globaler String"
f()
```

Sobald versucht wird, schreibend auf eine globale Referenz zuzugreifen, wird stattdessen eine entsprechende lokale Referenz erzeugt:

```
def f():
    s = "lokaler String"
    print(s)

s = "globaler String"
f()
print(s)
```

Die Ausgabe dieses Beispiels lautet:

```
lokaler String
globaler String
```

Eine Funktion kann dennoch, mithilfe der `global`-Anweisung, schreibend auf eine globale Referenz zugreifen. Dazu muss im Funktionskörper das Schlüsselwort `global`, gefolgt von einer oder mehreren globalen Referenzen, geschrieben werden:

```
def f():
    global s
    s = "lokaler String"
    print(s)

s = "globaler String"
f()
print(s)
```

Die Ausgabe des Beispiels lautet:

```
lokaler String
lokaler String
```

Im Funktionskörper von `f` wird `s` explizit als globale Referenz gekennzeichnet und kann fortan als solche verwendet werden.

10.5.2 Zugriff auf übergeordnete Namensräume – `nonlocal`

Im vorherigen Abschnitt wurde von den zwei existierenden Namensräumen, dem globalen und dem lokalen, gesprochen. Diese Unterteilung ist richtig, unterschlägt aber einen interessanten Fall, denn laut Abschnitt 10.3, »Lokale Funktionen«, dürfen auch lokale Funktionen innerhalb von Funktionen definiert werden. Lokale Funktionen bringen natürlich wieder ihren eigenen lokalen Namensraum im lokalen Namensraum der übergeordneten Funktion mit. Bei ver-

verschachtelten Funktionsdefinitionen kann man die Welt der Namensräume also nicht so banal in die lokale und die globale Ebene unterteilen. Dennoch stellt sich auch hier die Frage, wie eine lokale Funktion auf Referenzen zugreifen kann, die im lokalen Namensraum der übergeordneten Funktion liegen.

Das Schlüsselwort `global` hilft dabei nicht weiter, denn es erlaubt nur den Zugriff auf den äußersten, globalen Namensraum. Für diesen Zweck existiert seit Python 3.0 das Schlüsselwort `nonlocal`. Betrachten wir dazu einmal folgendes Beispiel:

```
def funktion1():
    def funktion2():
        nonlocal res
        res += 1

    res = 1
    funktion2()
    print(res)
```

Innerhalb der Funktion `funktion1` wurde eine lokale Funktion `funktion2` definiert, die die Referenz `res` aus dem lokalen Namensraum von `funktion1` inkrementieren soll. Dazu muss `res` innerhalb von `funktion2` als `nonlocal` gekennzeichnet werden. Die Schreibweise lehnt sich an den Zugriff auf Referenzen aus dem globalen Namensraum via `global` an.

Nachdem `funktion2` definiert wurde, wird `res` im lokalen Namensraum von `funktion1` definiert und mit dem Wert 1 verknüpft. Schließlich wird die lokale Funktion `funktion2` aufgerufen und der Wert von `res` ausgegeben. Im Beispiel gäbe `funktion1` den Wert 2 aus.

Das Schlüsselwort `nonlocal` lässt sich auch bei mehreren ineinander verschachtelten Funktionen verwenden, wie folgende Erweiterung unseres Beispiels zeigt:

```
def funktion1():
    def funktion2():
        def funktion3():
            nonlocal res
            res += 1

        nonlocal res
        funktion3()
        res += 1

    res = 1
    funktion2()
    print(res)
```

Nun wurde eine zusätzliche lokale Funktion im lokalen Namensraum von `funktion2` definiert. Auch aus dem lokalen Namensraum von `funktion3` heraus lässt sich `res` mithilfe von `nonlocal` inkrementieren. Die Funktion `funktion1` gäbe in diesem Beispiel den Wert 3 aus.

Allgemein funktioniert `nonlocal` bei tieferen Funktionsverschachtelungen so, dass es in der Hierarchie der Namensräume aufsteigt und die erste Referenz mit dem angegebenen Namen in den Namensraum des `nonlocal`-Schlüsselworts einbindet.

10.6 Rekursion

Python erlaubt es dem Programmierer, sogenannte *rekursive Funktionen* zu schreiben. Das sind Funktionen, die sich selbst aufrufen. Die aufgerufene Funktion ruft sich erneut selbst auf. Das geht so weiter, bis eine Abbruchbedingung diese – sonst endlose – Rekursion beendet. Die Anzahl der verschachtelten Funktionsaufrufe wird *Rekursionstiefe* genannt und ist von der Laufzeitumgebung auf einen bestimmten Wert begrenzt.

Jede rekursive Funktion kann, unter Umständen mit viel Aufwand, in eine iterative umgeformt werden. Eine iterative Funktion ruft sich selbst nicht auf, sondern löst das Problem allein durch Einsatz von Kontrollstrukturen, speziell Schleifen. Eine rekursive Funktion ist oft eleganter und kürzer als ihr iteratives Ebenbild, in der Regel aber auch langsamer.

Im folgenden Beispiel wurde eine rekursive Funktion zur Berechnung der Fakultät einer ganzen Zahl geschrieben:

```
def fak(n):
    if n > 0:
        return fak(n - 1) * n
    else:
        return 1
```

Es soll nicht Sinn und Zweck dieses Abschnitts sein, vollständig in die Thematik der Rekursion einzuführen. Stattdessen möchten wir hier nur einen kurzen Überblick geben. Sollten Sie das Beispiel nicht auf Anhieb verstehen, seien Sie nicht entmutigt, denn es lässt sich auch ohne Rekursion passabel in Python programmieren. Trotzdem sollten Sie nicht leichtfertig über die Rekursion hinwegsehen, denn es handelt sich dabei um einen höchst interessanten Weg, sehr elegante Programme zu schreiben.

10.7 Vordefinierte Funktionen

Es war im Laufe des Buches schon oft von sogenannten *Built-in Functions* die Rede. Das sind vordefinierte Funktionen, die dem Programmierer jederzeit zur Verfügung stehen. Üblicherweise handelt es sich dabei um Hilfsfunktionen, die das Programmieren in Python erheblich erleichtern. Sie kennen bereits die Built-in Functions `len` und `range`. Im Folgenden werden alle bisher relevanten Built-in Functions ausführlich beschrieben. Im Anhang finden Sie eine vollständige tabellarische Übersicht.

abs(x)

Die Funktion `abs` berechnet den Betrag von `x`. Der Parameter `x` muss dabei ein numerischer Wert sein, also eine Instanz der Datentypen `int`, `float`, `bool` oder `complex`.

```
>>> abs(1)
1
>>> abs(-12.34)
12.34
>>> abs(3 + 4j)
5.0
```

all(iterable)

Die Funktion `all` gibt immer dann `True` zurück, wenn alle Elemente des als Parameter übergebenen iterierbaren Objekts, also beispielsweise einer Liste oder eines Tupels, den Wahrheitswert `True` ergeben. Sie wird folgendermaßen verwendet:

```
>>> all([True, True, False])
False
>>> all([True, True, True])
True
```

any(iterable)

Die Funktion `any` arbeitet ähnlich wie `all`. Sie gibt immer dann `True` zurück, wenn mindestens ein Element des als Parameter übergebenen iterierbaren Objekts, also zum Beispiel einer Liste oder eines Tupels, den Wahrheitswert `True` ergibt. Sie wird folgendermaßen verwendet:

```
>>> any([True, False, False])
True
>>> any([False, False, False])
False
```

ascii(object)

Die Funktion `ascii` gibt eine lesbare Entsprechung der Instanz *object* in Form eines Strings zurück. Im Gegensatz zu der für denselben Zweck existierenden Built-in Function `repr` enthält der von `ascii` zurückgegebene String ausschließlich Zeichen des ASCII-Zeichensatzes:

```
>>> ascii(range(0, 10))
'range(0, 10)'
>>> ascii("Püthon")
"'P\\xfcthon'"
>>> repr("Püthon")
"'Püthon'"
```

bin(x)

Gibt einen String zurück, der die für *x* übergebene ganze Zahl in ihrer Binärdarstellung enthält:

```
>>> bin(123)
'0b1111011'
>>> bin(-12)
'-0b1100'
>>> bin(0)
'0b0'
```

bool([x])

Gibt den Wahrheitswert der Instanz *x* zurück. Wenn kein Parameter übergeben wurde, gibt die Funktion `bool` den booleschen Wert `False` zurück.

bytearray([arg[, encoding[, errors]])]

Erzeugt eine Instanz des Datentyps `bytearray`, der eine Sequenz von Byte-Werten darstellt, also ganzen Zahlen im Zahlenbereich von 0 bis 255. Beachten Sie, dass `bytearray` im Gegensatz zu `bytes` ein veränderlicher Datentyp ist.

Der Parameter *arg* wird zum Initialisieren des Byte-Arrays verwendet und kann verschiedene Bedeutungen haben:

Wenn für *arg* ein String übergeben wird, wird dieser mithilfe der Parameter *encoding* und *errors* in eine Byte-Folge kodiert und dann zur Initialisierung des Byte-Arrays verwendet. Die Parameter *encoding* und *errors* haben die gleiche Bedeutung wie bei der Built-in Function `str`.

Wenn für *arg* eine ganze Zahl übergeben wird, wird ein Byte-Array der Länge *arg* angelegt und mit Nullen gefüllt.

Wenn für *arg* ein iterierbares Objekt, beispielsweise eine Liste, übergeben wird, wird das Byte-Array mit den Elementen gefüllt, über die *arg* iteriert. Beachten Sie, dass es sich dabei um ganze Zahlen aus dem Zahlenbereich von 0 bis 255 handeln muss.

Außerdem kann für *arg* eine beliebige Instanz eines Datentyps übergeben werden, der das sogenannte Buffer-Protokoll unterstützt. Das sind beispielsweise die Datentypen `bytes` und `bytearray` selbst.

```
>>> bytearray("äöü", "utf-8")
bytearray(b'\xc3\xa4\xc3\xb6\xc3\xbc')
>>> bytearray([1,2,3,4])
bytearray(b'\x01\x02\x03\x04')
>>> bytearray(10)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
```

bytes([arg[, encoding[, errors]])

Erzeugt eine Instanz des Datentyps `bytes`, der, wie der Datentyp `bytearray`, eine Folge von Byte-Werten speichert. Im Gegensatz zu `bytearray` handelt es sich aber um einen unveränderlichen Datentyp, weswegen wir auch von einem bytes-String sprechen.

Die Parameter *args*, *encoding* und *errors* werden wie bei der Built-in Function `bytearray` zur Initialisierung der Byte-Folge verwendet:

```
>>> bytes(10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> bytes([1,2,3])
b'\x01\x02\x03'
>>> bytes("äöü", "utf-8")
b'\xc3\xa4\xc3\xb6\xc3\xbc'
```

chr(i)

Die Funktion `chr` gibt einen String der Länge 1 zurück, der das Zeichen mit dem Unicode-Code *i* enthält:

```
>>> chr(65)
'A'
>>> chr(33)
'!'
>>> chr(8364)
'€'
```


complex([real[, imag]])

Dies erzeugt eine Instanz des Datentyps `complex` zur Speicherung einer komplexen Zahl. Die erzeugte Instanz hat den komplexen Wert $real + imag * j$. Fehlende Parameter werden als 0 angenommen.

Außerdem ist es möglich, der Funktion `complex` einen String zu übergeben, der das Literal einer komplexen Zahl enthält. In diesem Fall darf jedoch kein weiterer Parameter angegeben werden.

```
>>> complex(1, 3)
(1+3j)
>>> complex(1.2, 3.5)
(1.2+3.5j)
>>> complex("3+4j")
(3+4j)
>>> complex("3")
(3+0j)
```

Beachten Sie, dass ein eventuell übergebener String keine Leerzeichen um den `+`-Operator enthalten darf:

```
>>> complex("3 + 4j")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: complex() arg is a malformed string
```

Leerzeichen am Anfang oder Ende des Strings sind aber kein Problem.

dict([source])

Erzeugt eine Instanz des Datentyps `dict`. Wenn kein Parameter übergeben wird, wird ein leeres Dictionary erstellt. Durch einen der folgenden Aufrufe ist es möglich, das Dictionary beim Erzeugen mit Werten zu füllen:

- Wenn *source* ein Dictionary ist, werden die Schlüssel und Werte dieses Dictionarys in das neue übernommen. Beachten Sie, dass dabei keine Kopien der Werte entstehen, sondern diese weiterhin dieselben Instanzen referenzieren.

```
>>> dict({"a" : 1, "b" : 2})
{'a': 1, 'b': 2}
```

- Alternativ kann *source* eine Liste von Tupeln sein, wobei jedes Tupel zwei Elemente enthalten kann: Den Schlüssel und den damit assoziierten Wert. Die Liste muss die Struktur `[("a", 1), ("b", 2)]` haben:

```
>>> dict [("a", 1), ("b", 2)]
{'a': 1, 'b': 2}
```

- Zudem erlaubt es `dict`, Schlüssel und Werte als Keyword Arguments zu übergeben. Der Parametername wird dabei in einen String geschrieben und als Schlüssel verwendet. Beachten Sie, dass Sie damit bei der Namensgebung den Beschränkungen eines Bezeichners unterworfen sind:

```
>>> dict(a=1, b=2)
{'a': 1, 'b': 2}
```

divmod(a, b)

Die Funktion `divmod` gibt folgendes Tupel zurück: $(a//b, a\%b)$. Mit Ausnahme von `complex` können für a und b Instanzen beliebiger numerischer Datentypen übergeben werden:

```
>>> divmod(2.5, 1.3)
(1.0, 1.2)
>>> divmod(11, 4)
(2, 3)
```

enumerate(iterable)

Die Funktion `enumerate` erzeugt ein iterierbares Objekt, das nicht allein über die Elemente von *iterable* iteriert, sondern über Tupel der folgenden Form: $(i, \text{iterable}[i])$. Dabei ist i ein Schleifenzähler, der bei 0 beginnt. Die Schleife wird beendet, wenn i den Wert $\text{len}(\text{iterable})-1$ hat. Diese Tupelstrukturen werden deutlich, wenn man das Ergebnis eines `enumerate`-Aufrufs in eine Liste konvertiert:

```
>>> list(enumerate(["a", "b", "c", "d"]))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Damit eignet sich `enumerate` besonders für `for`-Schleifen, in denen ein numerischer Schleifenzähler mitgeführt werden soll. Innerhalb einer `for`-Schleife kann `enumerate` folgendermaßen verwendet werden:

```
for i, wert in enumerate(iterable):
    print("Der Wert von iterable an", i, "ter Stelle ist:", wert)
```

Angenommen, der obige Code würde für eine Liste `iterable = [1,2,3,4,5]` ausgeführt, so käme folgende Ausgabe zustande:

```
Der Wert von iterable an 0 ter Stelle ist: 1
Der Wert von iterable an 1 ter Stelle ist: 2
Der Wert von iterable an 2 ter Stelle ist: 3
Der Wert von iterable an 3 ter Stelle ist: 4
Der Wert von iterable an 4 ter Stelle ist: 5
```

filter(function, list)

Die Funktion `filter` erwartet ein Funktionsobjekt als ersten und eine Liste als zweiten Parameter. Der Parameter *function* muss eine Funktion oder Lambda-Form sein, die einen Parameter erwartet und einen booleschen Wert zurückgibt.

Die Funktion `filter` ruft für jedes Element der Liste *list* die Funktion *function* auf und erzeugt ein iterierbares Objekt, das alle Elemente von *list* durchläuft, für die *function* `True` zurückgegeben hat. Dies soll an folgendem Beispiel erklärt werden, in dem `filter` dazu verwendet wird, um aus einer Liste von ganzen Zahlen die ungeraden Zahlen herauszufiltern:

```
def fun(prm):
    return (prm%2 == 0)
fobj = filter(fun, [1,2,3,4,5,6,7,8,9,10])
print(list(fobj))
```

Das zurückgegebene iterierbare Objekt kann beispielsweise in einer `for`-Schleife durchlaufen oder, wie in diesem Beispiel, mittels `list` in eine Liste überführt und ausgegeben werden. Die Ausgabe des Beispiels lautet:

```
[2, 4, 6, 8, 10]
```

float([x])

Erzeugt eine Instanz des Datentyps `float`. Wenn der Parameter *x* nicht angegeben wurde, wird der Wert der Instanz mit `0.0`, andernfalls mit dem übergebenen Wert initialisiert. Mit Ausnahme von `complex` können Instanzen alle numerischen Datentypen für *x* übergeben werden.

```
>>> float()
0.0
>>> float(5)
5.0
```

Außerdem ist es möglich, für *x* einen String zu übergeben, der eine Gleitkommazahl enthält:

```
>>> float("1e30")
1e+30
>>> float("0.5")
0.5
```

format(value[, format_spec])

Gibt den Wert *value* gemäß der Formatangabe *format_spec* aus. Beispielsweise lässt sich ein Geldbetrag bei der Ausgabe folgendermaßen auf zwei Nachkommastellen runden:

```
>>> format(1.23456, ".2f") + "€"
'1.23€'
```

Ausführliche Informationen zu Formatangaben finden Sie in Abschnitt 8.5.3 über Stringformatierungen.

frozenset(iterable)

Erzeugt eine Instanz des Datentyps `frozenset` zum Speichern einer unveränderlichen Menge. Wenn der Parameter *iterable* angegeben wurde, so werden die Elemente der erzeugten Menge diesem iterierbaren Objekt entnommen. Wenn der Parameter *iterable* nicht angegeben wurde, erzeugt `frozenset` eine leere Menge.

Beachten Sie zum einen, dass ein `frozenset` keine veränderlichen Elemente enthalten darf, und zum anderen, dass jedes Element nur einmal in einer Menge vorkommen kann.

```
>>> frozenset()
frozenset()
>>> frozenset({1,2,3,4,5})
frozenset({1, 2, 3, 4, 5})
>>> frozenset("PyYYYYYYthon")
frozenset({'h', 'o', 'n', 'P', 't', 'y'})
```

globals()

Die Built-in Function `globals` gibt ein Dictionary mit allen globalen Referenzen des aktuellen Namensraums zurück. Die Schlüssel entsprechen den Referenznamen als Strings und die Werte den jeweiligen Instanzen.

```
>>> a = 1
>>> b = {}
>>> c = [1,2,3]
>>> globals()
{'a': 1, 'c': [1, 2, 3], 'b': {}, '__builtins__': <module 'builtins' (built-in)>, '__package__': None, '__name__': '__main__', '__doc__': None}
```

Das zurückgegebene Dictionary enthält neben den vorher angelegten noch weitere Instanzen, die im globalen Namensraum existieren. Diese vordefinierten Referenzen haben wir bisher noch nicht besprochen, lassen Sie sich davon also nicht stören.

hash(object)

Berechnet den Hash-Wert der Instanz *object* und gibt ihn zurück. Bei einem Hash-Wert handelt es sich um eine ganze Zahl, die aus Typ und Wert der Instanz er-

zeugt wird. Ein solcher Wert wird verwendet, um effektiv zwei komplexere Instanzen auf Gleichheit prüfen zu können. So werden beispielsweise die Schlüssel eines Dictionarys intern durch ihre Hash-Werte verwaltet.

```
>>> hash(12345)
12345
>>> hash("Hallo Welt")
-962533610
>>> hash((1,2,3,4))
89902565
```

Beachten Sie den Unterschied zwischen veränderlichen (mutable) und unveränderlichen (immutable) Instanzen. Aus Letzteren kann zwar formal auch ein Hash-Wert errechnet werden, dieser wäre aber nur so lange gültig, wie die Instanz nicht verändert wurde. Aus diesem Grund ist es nicht sinnvoll, Hash-Werte von veränderlichen Instanzen zu berechnen; veränderliche Instanzen sind »unhashable«:

```
>>> hash([1,2,3,4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

help([object])

Die Funktion `help` startet die interaktive Hilfe von Python. Wenn der Parameter *object* ein String ist, wird dieser im Hilfesystem nachgeschlagen. Sollte es sich um eine andere Instanz handeln, wird eine dynamische Hilfeseite zu dieser generiert.

hex(x)

Erzeugt einen String, der die als Parameter *x* übergebene ganze Zahl in Hexadezimalschreibweise enthält. Die Zahl entspricht, wie sie im String erscheint, dem Python-Literal für Hexadezimalzahlen.

```
>>> hex(12)
'0xc'
>>> hex(0xFF)
'0xff'
>>> hex(-33)
'-0x21'
```

id(object)

Die Funktion `id` gibt die Identität einer beliebigen Instanz zurück. Bei der Identität einer Instanz handelt es sich um eine ganze Zahl, die die Instanz eindeutig identifiziert.

```
>>> id(1)
134537016
>>> id(2)
134537004
```

input([prompt])

Liest eine Eingabe vom Benutzer ein und gibt sie in Form eines Strings zurück. Der Parameter *prompt* ist optional. Hier kann ein String angegeben werden, der vor der Eingabeaufforderung ausgegeben werden soll.

```
>>> s = input("Geben Sie einen Text ein: ")
Geben Sie einen Text ein: Python ist gut
>>> s
'Python ist gut'
```

Hinweis

Das Verhalten der Built-in Funktion `input` wurde mit Python 3.0 verändert. In früheren Versionen wurde die Eingabe des Benutzers als Python-Code vom Interpreter ausgeführt und das Ergebnis dieser Ausführung in Form eines Strings zurückgegeben. Die »alte« `input`-Funktion entsprach also folgendem Code:

```
>>> eval(input("Prompt: "))
Prompt: 2+2
4
```

Die `input`-Funktion, wie sie in aktuellen Versionen von Python existiert, hieß in früheren Versionen `raw_input`.

int([x[, radix]])

Erzeugt eine Instanz des Datentyps `int`. Die Instanz kann durch Angabe von *x* mit einem Wert initialisiert werden. Wenn kein Parameter angegeben wird, erhält die erzeugte Instanz den Wert 0.

Wenn der Parameter *x* als String übergeben wird, so erwartet die Funktion `int`, dass dieser String den gewünschten Wert der Instanz enthält. Durch den optionalen Parameter *radix* kann die Basis des Zahlensystems angegeben werden, in dem die Zahl geschrieben wurde.

```
>>> int(5)
5
>>> int("FF", 16)
255
>>> int(hex(12), 16)
12
```

len(s)

Gibt die Länge bzw. die Anzahl der Elemente von *s* zurück.

```
>>> len("Hallo Welt")
10
>>> len([1,2,3,4,5])
5
```

list([sequence])

Erzeugt eine Instanz des Datentyps `list` aus den Elementen von *sequence*. Der Parameter *sequence* muss ein iterierbares Objekt sein. Wenn er weggelassen wird, wird eine leere Liste erzeugt.

```
>>> list()
[]
>>> list((1,2,3,4))
[1, 2, 3, 4]
>>> list({"a": 1, "b": 2})
['a', 'b']
```

Die Funktion `list` kann, wie bereits mehrfach demonstriert, dazu verwendet werden, ein beliebiges iterierbares Objekt in eine Liste zu überführen:

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

locals()

Die Built-in Function `locals` gibt ein Dictionary mit allen lokalen Referenzen des aktuellen Namensraums zurück. Die Schlüssel entsprechen den Referenznamen als Strings und die Werte den jeweiligen Instanzen. Dies soll an folgendem Beispiel deutlich werden:

```
def f(a, b, c):
    d = a + b + c
    print(locals())
f(1, 2, 3)
```

Dieses Beispiel erzeugt folgende Ausgabe:

```
{'a': 1, 'c': 3, 'b': 2, 'd': 6}
```

Beachten Sie, dass der Aufruf von `locals` im Namensraum des Hauptprogramms äquivalent ist zum Aufruf von `globals`.

map(function, list, ...)

Diese Funktion erwartet ein Funktionsobjekt als ersten und eine Liste als zweiten Parameter. Optional können weitere Listen übergeben werden, die aber die gleiche Länge wie die erste haben müssen. Die Funktion *function* muss genauso viele Parameter erwarten, wie Listen übergeben wurden, und aus den Parametern einen Rückgabewert erzeugen.

Die Funktion `map` ruft *function* für jedes Element der übergebenen Liste auf und gibt ein iterierbares Objekt zurück, das die jeweiligen Rückgabewerte von *function* durchläuft. Sollten mehrere Listen übergeben werden, so werden *function* die jeweils n-ten Elemente aller Listen übergeben. Beachten Sie, dass *function* aus diesem Grund unbedingt genau so viele Parameter erwarten muss, wie Listen übergeben werden, und dass alle übergebenen Listen gleich viele Elemente enthalten müssen.

Im folgenden Beispiel wird das Funktionsobjekt durch eine Lambda-Form erstellt. Es ist auch möglich, eine echte Funktion zu definieren und ihren Namen zu übergeben.

```
>>> f = lambda x: x**2
>>> ergebnis = map(f, [1,2,3,4])
>>> list(ergebnis)
[1, 4, 9, 16]
```

Hier wird `map` dazu verwendet, eine Liste mit den Quadraten der Elemente einer zweiten Liste zu erzeugen.

```
>>> f = lambda x, y: x+y
>>> ergebnis = map(f, [1,2,3,4], [1,2,3,4])
>>> list(ergebnis)
[2, 4, 6, 8]
```

Hier wird `map` dazu verwendet, aus zwei Listen eine zu erzeugen, die die Summen der jeweiligen Elemente beider Quelllisten enthält.

In beiden Beispielen wurden Listen verwendet, die ausschließlich numerische Elemente enthielten. Das muss nicht unbedingt sein. Welche Elemente eine Liste enthalten darf, hängt davon ab, welche Instanzen für *function* als Parameter verwendet werden dürfen.

Das letzte Beispiel wird durch Abbildung 10.2 veranschaulicht.

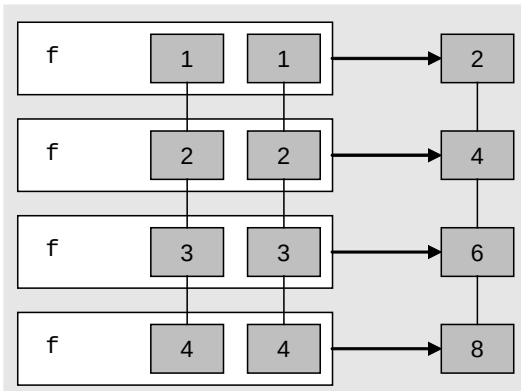


Abbildung 10.2 Arbeitsweise der Built-in Function map

Die eingehenden und ausgehenden Listen sind jeweils senkrecht dargestellt.

max(s[, args...][key])

Wenn keine zusätzlichen Parameter übergeben werden, erwartet `max` eine Sequenz und gibt ihr größtes Element zurück. Die übergebene Instanz eines sequentiellen Datentyps muss Elemente enthalten:

```
>>> max([2,4,1,9,5])
9
>>> max("Hallo Welt")
't'
```

Wenn mehrere Parameter übergeben werden, so verhält sich `max` so, dass der größte übergebene Parameter zurückgegeben wird:

```
>>> max(3, 5, 1, 99, 123, 45)
123
>>> max("Hallo", "Welt", "!")
'Welt'
```

Für beide Verwendungsarten von `max` kann eine optionale Funktion als Schlüsselwortparameter übergeben werden, die für jedes Element der übergebenen Sequenz bzw. jeden Parameter aufgerufen wird, bevor das größte Element festgestellt wird. So ist es mit `key` möglich, aus den übergebenen Datensätzen eine für die Ordnungsrelation relevante Information zu extrahieren.

In folgendem Beispiel soll `key` dazu verwendet werden, die Funktion `max` für Strings *case insensitive* zu machen. Dazu zeigen wir zunächst den normalen Aufruf ohne `key`:

```
>>> max("a", "P", "q", "X")
'q'
```

Ohne eigene *key*-Funktion wird der größte Parameter unter Berücksichtigung von Groß- und Kleinbuchstaben ermittelt. Folgende *key*-Funktion konvertiert zuvor alle Buchstaben in Kleinbuchstaben:

```
>>> f = lambda x: x.lower()
>>> max("a", "P", "q", "X", key=f)
'X'
```

Durch die *key*-Funktion wird der größte Parameter anhand der durch *f* modifizierten Werte ermittelt, jedoch unmodifiziert zurückgegeben.

min(s[, args...][key])

Die Funktion *min* verhält sich wie *max*, ermittelt jedoch das kleinste Element einer Sequenz bzw. den kleinsten übergebenen Parameter.

oct(x)

Die Funktion *oct* erzeugt einen String, der die übergebene ganze Zahl *x* in Oktalschreibweise enthält.

```
>>> oct(123)
'0o173'
>>> oct(0o777)
'0o777'
```

open(filename[, mode[, bufsize]])

Öffnet eine Datei im gewünschten Modus und gibt das erzeugte Dateiojekt zurück. Eine vollständige Beschreibung der Funktion finden Sie in Abschnitt 9.4, »Verwendung des Dateiobjekts«.

ord(c)

Die Funktion *ord* erwartet einen String der Länge 1 und gibt den Unicode-Code des enthaltenen Zeichens zurück. Wenn es sich um einen Unicode-String handelt, wird der Unicode-Code des Zeichens zurückgegeben.

```
>>> ord("P")
80
>>> ord("€")
8364
```

pow(x, y[, z])

Berechnet $x ** y$ oder, wenn z angegeben wurde, $x ** y \% z$. Beachten Sie, dass diese Berechnung unter Verwendung des Parameters z performanter ist als die Ausdrücke $\text{pow}(x, y) \% z$ bzw. $x ** y \% z$.

```
>>> 7 ** 5 % 4
3
>>> pow(7, 5, 4)
3
```

print([object, ...][, sep=''][, end='\n'][, file=sys.stdout])

Die Funktion `print` schreibt die Textentsprechungen der für *object*, ... übergebenen Instanzen in den Datenstrom *file*. Bislang haben wir `print` nur dazu verwendet, auf den Bildschirm bzw. in die Standardausgabe zu schreiben. Hier sehen wir, dass `print` es über den Schlüsselwortparameter *file* ermöglicht, in ein beliebiges zum Schreiben geöffnetes Dateiobjekt zu schreiben:

```
>>> f = open("datei.txt", "w")
>>> print("Hallo Welt", file=f)
>>> f.close()
```

Über den Schlüsselwortparameter *sep*, der mit einem Leerzeichen vorbelegt ist, wird das Trennzeichen angegeben, das zwischen zwei auszugebenden Werten stehen soll:

```
>>> print("Hallo", "Welt")
Hallo Welt
>>> print("Hallo", "Welt", sep=" du schöne ")
Hallo du schöne Welt
>>> print("Hallo", "du", "schöne", "Welt", sep="-")
Hallo-du-schöne-Welt
```

Über den zweiten Schlüsselwortparameter *end* wird bestimmt, welches Zeichen `print` als Letztes, also nach erfolgter Ausgabe aller übergebenen Instanzen, ausgeben soll. Vorbelegt ist dieser Parameter mit einem Newline-Zeichen.

```
>>> print("Hallo", end=" Welt\n")
Hallo Welt
>>> print("Hallo", "Welt", end="AAAA")
Hallo WeltAAAA>>>
```

Im letzten Beispiel befindet sich der Eingabeprompt des Interpreters direkt hinter der von `print` erzeugten Ausgabe, weil im Gegensatz zum Standardverhalten von `print` am Ende kein Newline-Zeichen ausgegeben wurde.

range([start,]stop[, step])

Die Funktion `range` erzeugt ein iterierbares Objekt über fortlaufende, numerische Werte. Dabei wird mit *start* begonnen, vor *stop* aufgehört und in jedem Schritt der vorherige Wert um *step* erhöht. Sowohl *start* als auch *step* sind optional und mit 0 bzw. 1 vorbelegt.

Beachten Sie, dass *stop* eine Grenze angibt, die nicht erreicht wird. Die Nummerierung beginnt also bei 0 und endet einen Schritt, bevor *stop* erreicht würde.

Bei dem von `range` zurückgegebenen iterierbaren Objekt handelt es sich um ein sogenanntes `range`-Objekt. Dies wird bei der Ausgabe im interaktiven Modus folgendermaßen angezeigt:

```
>>> range(10)
range(0, 10)
```

Um zu veranschaulichen, über welche Zahlen das `range`-Objekt iteriert, wurde es in den folgenden Beispielen mit `list` in eine Liste überführt:

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

Es ist möglich, eine negative Schrittweite anzugeben:

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

Beachten Sie, falls Sie mit älteren Versionen von Python arbeiten, dass `range` erst seit Python 3.0 ein `range`-Objekt zurückgibt. Zuvor wurde eine Liste mit den gewünschten Elementen erzeugt und zurückgegeben.

repr(object)

Gibt einen String zurück, der eine druckbare Repräsentation der Instanz *object* enthält. Für viele Instanzen versucht `repr`, den Python-Code in den String zu schreiben, der die entsprechende Instanz erzeugen würde. Für manche Instanzen ist dies jedoch nicht möglich bzw. nicht praktikabel. In einem solchen Fall gibt `repr` zumindest den Typ der Instanz aus.

```
>>> repr([1,2,3,4])
'[1, 2, 3, 4]'
```

```
>>> repr(0x34)
'52'
>>> repr(set([1,2,3,4]))
'set([1, 2, 3, 4])'
>>> repr(open("datei.txt", "w"))
"<open file 'datei.txt', mode 'w' at 0xb7bea0f8>"
```

reversed(seq)

Mit `reversed` kann eine Sequenz *seq* sehr effizient rückwärts durchlaufen werden:³

```
>>> for i in reversed([1, 2, 3, 4, 5, 6]):
...     print(i)
6
5
4
3
2
1
```

round(x[, n])

Rundet die Gleitkommazahl *x* auf *n* Nachkommastellen. Der Parameter *n* ist optional und mit 0 vorbelegt.

```
>>> round(0.5, 4)
0.5
>>> round(-0.5)
-1.0
>>> round(0.5234234234234, 5)
0.52342
```

set([iterable])

Erzeugt eine Instanz des Datentyps `set`. Wenn angegeben, werden alle Elemente des iterierbaren Objekts *iterable* in das Set übernommen. Beachten Sie, dass ein Set keine Dubletten enthalten darf, jedes in *iterable* mehrfach vorkommende Element also nur einmal eingetragen wird.

```
>>> set()
set()
>>> set("Hallo Welt")
set({'a', ' ', 'e', 'H', 'l', 'o', 't', 'W'})
```

3 Die Built-in Function `reversed` ist nicht auf Sequenzen beschränkt, sondern funktioniert für jedes beliebige iterierbare Objekt.

```
>>> set({1,2,3,4})
set({1, 2, 3, 4})
```

sorted(iterable[, key[, reverse]])

Die Funktion `sorted` erzeugt aus den Elementen von *iterable* eine sortierte Liste:

```
>>> sorted([3,1,6,2,9,1,8])
[1, 1, 2, 3, 6, 8, 9]
>>> sorted("Hallo Welt")
[' ', 'H', 'W', 'a', 'e', 'l', 'l', 'l', 'o', 't']
```

Die Funktion akzeptiert zwei weitere Schlüsselwortparameter, um das Sortieren der Elemente zu beeinflussen:

- Durch den Schlüsselwortparameter *key* kann eine Funktion übergeben werden, die die für den Vergleich wichtige Information aus den Elementen extrahiert. Die Funktion muss einen Parameter akzeptieren und einen Rückgabewert zurückgeben. Die hier übergebene Funktion hat die gleiche Schnittstelle und die gleiche Bedeutung wie die, die bei der Built-in Function `max` über den Schlüsselwortparameter *key* übergeben werden kann.

```
>>> f = lambda x: x.lower()
>>> sorted("Hallo Welt", key=f)
[' ', 'a', 'e', 'H', 'l', 'l', 'l', 'o', 't', 'W']
```

- Der Schlüsselwortparameter *reverse* muss ein boolescher Wert sein und ist mit `False` vorbelegt. Wird er auf `True` gesetzt, so veranlasst dies `sorted`, die Sortierreihenfolge umzukehren.

```
>>> sorted([3,1,6,2,9,1,8], reverse=True)
[9, 8, 6, 3, 2, 1, 1]
```

Die obigen Schlüsselwortparameter können selbstverständlich nicht nur isoliert, sondern auch gemeinsam übergeben werden.

Die Built-in Function `sort` verhält sich im Wesentlichen wie die `sort`-Methode eines Strings. Weitere Beispiele für die Verwendung von `sort` finden Sie daher in Abschnitt 8.5.3, »Strings – str, bytes«.

str([object[, encoding[, errors]])

Erzeugt einen String, der eine lesbare Beschreibung der Instanz *object* enthält. Wenn *object* nicht übergeben wird, erzeugt `str` einen leeren String.

```
>>> str(None)
'None'
>>> str()
''
```

```
>>> str(12345)
'12345'
>>> str(str)
"<class 'str'"
```

Die Funktion `str` kann dazu verwendet werden, einen `bytes`-String oder eine `bytearray`-Instanz in einen String zu überführen. Dieser Prozess wird Dekodieren genannt, und es muss dazu mindestens einer der Parameter *encoding* und *errors* angegeben worden sein:

```
>>> b = bytearray([1,2,3])
>>> str(b, "utf-8")
'\x01\x02\x03'

>>> b = bytes("Hallö Wölt", "utf-8", "strict")
>>> str(b)
"b'Hall\\xc3\\xb6 W\\xc3\\xb6lt'"
>>> str(b, "utf-8")
'Hallö Wölt'
```

Dabei muss für den Parameter *encoding* ein String übergeben werden, der das Encoding enthält, mit dem der `bytes`-String kodiert wurde, in diesem Fall `utf-8`. Der Parameter *errors* wurde in obigem Beispiel nicht angegeben und bestimmt, wie mit Dekodierungsfehlern zu verfahren ist. Die folgende Tabelle listet die möglichen Werte für *errors* und ihre Bedeutung auf:

errors	Beschreibung
"strict"	Bei einem Dekodierungsfehler wird eine <code>ValueError</code> -Exception geworfen.
"ignore"	Fehler bei der Dekodierung werden ignoriert.
"replace"	Ein Zeichen, das nicht dekodiert werden konnte, wird durch das Unicode-Zeichen U+FFFD, auch Replacement Character genannt, ersetzt.

Tabelle 10.1 Mögliche Werte des Parameters `errors`

Hinweis

Beachten Sie, dass der Datentyp `str` mit Python 3.0 einer Überarbeitung unterzogen wurde. Im Gegensatz zu dem Datentyp `str` aus Python 2.x ist er in Python 3 dazu gedacht, Unicode-Text aufzunehmen. Er ist somit vergleichbar mit dem Datentyp `unicode` aus Python 2. Der dortige Datentyp `str` lässt sich vergleichen mit dem `bytes`-String aus Python 3.

Weitere Informationen über die Datentypen `str` und `bytes` sowie über Unicode finden Sie in Abschnitt 8.5.3, »Strings – `str`, `bytes`«.

sum(sequence[, start])

Die Funktion `sum` berechnet die Summe aller Elemente von *sequence* und gibt das Ergebnis zurück. Wenn der optionale Parameter *start* angegeben wurde, so fließt dieser als Startwert der Berechnung ebenfalls in die Summe mit ein.

```
>>> sum([1,2,3,4])
10
>>> sum([1,2,3,4], 2)
12
>>> sum([4,3,2,1], 2)
12
```

tuple([sequence])

Erzeugt eine Instanz des Datentyps `tuple` und überträgt dabei, wenn angegeben, alle Elemente von *sequence* in diese neue Instanz.

```
>>> tuple()
()
>>> tuple([1,2,3,4])
(1, 2, 3, 4)
```

type(object)

Die Funktion `type` gibt den Datentyp der übergebenen Instanz *object* zurück.

```
>>> type(1)
<type 'int'>
>>> type("Hallo Welt") == str
True
>>> type(sum)
<class 'builtin_function_or_method'>
```

zip(iterable, ...)

Die Funktion `zip` nimmt beliebig viele, gleich lange iterierbare Objekte als Parameter. Sollten nicht alle die gleiche Länge haben, werden die längeren auf die Länge des kürzesten dieser Objekte beschnitten.

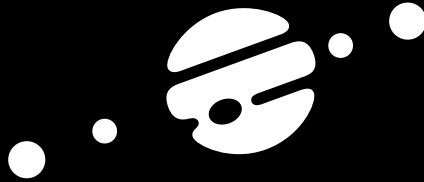
Als Rückgabewert wird ein iterierbares Objekt erzeugt, das über Tupel iteriert, die im *i*-ten Iterationsschritt die jeweils *i*-ten Elemente der übergebenen Sequenzen enthalten.

```
>>> ergebnis = zip([1,2,3,4], [5,6,7,8], [9,10,11,12])
>>> list(ergebnis)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
>>> ergebnis = zip("Hallo Welt", "HaWe")
```



```
>>> list(ergebnis)
[('H', 'H'), ('a', 'a'), ('l', 'W'), ('l', 'e')]
```

Dies waren noch nicht alle Built-in Functions, da einige für Themen gedacht sind, die bisher noch nicht behandelt wurden. Im Anhang finden Sie eine tabellarische Übersicht über alle Built-in Functions, inklusive eines Verweises, wo die jeweilige Funktion detailliert besprochen wird.



Teil II
Fortgeschrittene Programmiertechniken

»Divide et impera!«
– Julius Caesar

11 Modularisierung

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

- Zum einen kann jedes Python-Programm sogenannte *Bibliotheken* (engl. *libraries*) einbinden. Eine Bibliothek dient häufig einem ganz bestimmten Zweck, wie etwa der Arbeit mit Dateien eines bestimmten Dateiformats, und stellt üblicherweise Datentypen oder Funktionen bereit, die nach dem Einbinden verwendet werden können. Es ist möglich, eigene Bibliotheken zu schreiben oder eine Bibliothek eines Drittanbieters zu installieren. Ein gutes Argument für Python ist die umfangreiche *Standardbibliothek*, die im Lieferumfang enthalten ist. Sie bietet eine hohe Grundfunktionalität, die in jeder Python-Umgebung verfügbar ist.
- Die zweite Möglichkeit zur Modularisierung sind *lokale Module*. Darunter versteht man die Kapselung einzelner Programmteile – auch hier üblicherweise Datentypen oder Funktionen – in eigene Programmdateien. Diese Dateien können wie Bibliotheken eingebunden werden, sind aber in keinem anderen Python-Programm verfügbar. Diese Form der Modularisierung hilft bei der Programmierung ungemein, da sie dem Programmierer die Möglichkeit gibt, sehr langen Programmcode überschaubar auf verschiedene Programmdateien aufzuteilen.

In Python besteht der einzige Unterschied zwischen Bibliotheken und lokalen Modulen darin, wo sie gespeichert sind. Während sich lokale Module in der Regel im Verzeichnis des Hauptprogramms bzw. in einem Unterverzeichnis desselben befinden, sind Bibliotheken in einigen festgelegten Verzeichnissen der Python-Installation gespeichert.¹

¹ Selbstgeschriebene Bibliotheken können Sie in das Unterverzeichnis *site-packages* der Python-Installation speichern. Dort werden üblicherweise auch Bibliotheken von Drittanbietern installiert.

11.1 Einbinden externer Programmbibliotheken

Eine Bibliothek, sei es ein Teil der Standardbibliothek oder eine selbstgeschriebene, kann mithilfe der `import`-Anweisung eingebunden werden. Wir werden in den Beispielen hauptsächlich das Modul `math` der Standardbibliothek verwenden. Das ist ein Modul, das mathematische Funktionen wie `sin` oder `cos` sowie mathematische Konstanten wie `pi` bereitstellt. Um sich diese Funktionalität in einem Programm zunutze machen zu können, ist folgende `import`-Anweisung nötig:

```
import math
```

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

```
import math, random
```

Dies ist äquivalent zu:

```
import math
import random
```

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

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

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

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

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

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

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

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

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

Hinweis

Der Sinn von Namensräumen ist es, thematisch abgegrenzte Bereiche, also zum Beispiel eine Bibliothek, zu kapseln und über einen gemeinsamen Namen anzusprechen. Wenn Sie den kompletten Inhalt einer Bibliothek in den globalen Namensraum eines Programms einbinden, kann es vorkommen, dass die Bibliothek mit eventuell vorhandenen Referenzen interferiert. In einem solchen Fall werden die bereits bestehenden Referenzen kommentarlos überschrieben, wie das folgende Beispiel zeigt:

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

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

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

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

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

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

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

Hinweis

Die Aufzählung der mit einer `from/import`-Anweisung zu importierenden Objekte kann unter Umständen recht lang werden. In solchen Fällen darf sie in runde Klammern gefasst werden. Der Vorteil dieser Schreibweise ist, dass eingeklammerte Ausdrücke beliebig formatiert, unter anderem auch auf mehrere Zeilen umbrochen werden dürfen:

```
from math import (sin, cos, tan,
                  sinh, cosh, tanh)
```

Beachten Sie, dass diese Schreibweise bei einer normalen `import`-Anweisung nicht möglich ist.

11.2 Eigene Module

Nachdem Sie in die unendlichen Weiten der `import`-Anweisung eingeführt wurden, möchten wir uns damit beschäftigen, wie Module selbst erstellt und eingebunden werden können. Beachten Sie, dass es sich hier nicht um eine Bibliothek handelt, die in jedem Python-Programm zur Verfügung steht, sondern um ein Modul, das nur lokal in Ihrem Python-Programm genutzt werden kann. Von der Verwendung her unterscheiden sich Module und Bibliotheken kaum. In diesem Abschnitt soll ein Programm erstellt werden, das eine ganze Zahl einliest, deren Fakultät und Kehrwert berechnet und die Ergebnisse ausgibt. Die mathematischen Berechnungen sollen dabei nicht nur in Funktionen, sondern auch in einem eigenen Modul gekapselt werden. Dazu schreiben wir diese zunächst in eine Datei namens *mathematik.py*:

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

def kehr(n):
    return 1.0 / n
```

Die Funktionen sollten selbsterklärend sein. Beachten Sie, dass die Datei *mathematik.py* selbst keinerlei Code ausführt, sondern nur Funktionen bereitstellt, die aus anderen Modulen heraus aufgerufen werden können.

Jetzt erstellen wir eine Programmdatei namens *programm.py*, in der das Hauptprogramm stehen soll. Beide Dateien müssen sich im selben Verzeichnis befinden.

den. Im Hauptprogramm importieren wir zunächst das lokale Modul `mathematik`. Der Modulname eines lokalen Moduls entspricht dem Dateinamen der zugehörigen Programmdatei ohne Dateierweiterung. Beachten Sie, dass der Modulname den Regeln der Namensgebung eines Bezeichners folgen muss. Das bedeutet insbesondere, dass, abgesehen von dem Punkt vor der Dateierweiterung, kein Punkt im Dateinamen erlaubt ist.

```
import mathematik

while True:
    zahl = int(input("Geben Sie eine ganze Zahl ein: "))
    print("Fakultaet: ", mathematik.fak(zahl))
    print("Kehrwert: ", mathematik.kehr(zahl))
```

Sie sehen, dass Sie das lokale Modul im Hauptprogramm wie eine Bibliothek importieren und verwenden können.

Durch das Erstellen eigener Module kann es leicht zu Namenskonflikten mit der Standardbibliothek kommen. Beispielsweise hätten wir unsere obige Programmdatei auch `math.py` und das Modul demzufolge `math` nennen können. Dieses Modul stünde im Konflikt mit der Bibliothek `math`. Für solche Fälle ist dem Interpreter eine Reihenfolge vorgegeben, nach der er zu verfahren hat, wenn ein Modul oder eine Bibliothek importiert werden soll:

- ▶ Zunächst wird der lokale Programmordner nach einer Datei mit dem entsprechenden Namen durchsucht. In dem oben geschilderten Konfliktfall stünde bereits im ersten Schritt fest, dass ein lokales Modul namens `math` existiert. Wenn ein solches lokales Modul existiert, wird dieses eingebunden und keine weitere Suche durchgeführt.
- ▶ Wenn kein lokales Modul des angegebenen Namens gefunden wurde, wird die Suche auf Bibliotheken ausgeweitet.
- ▶ Wenn auch keine Bibliothek mit dem angegebenen Namen gefunden wurde, wird ein `ImportError` erzeugt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named bla
```

11.2.1 Modulinterne Referenzen

In jedem Modul existieren globale Variablen, die Informationen über das Modul selbst enthalten. An dieser Stelle soll ein Überblick über diese recht überschaubare Anzahl von Referenzen gegeben werden. Beachten Sie, dass es sich jeweils um zwei Unterstriche vor und hinter dem Namen der Referenz handelt.

Referenz	Beschreibung
<code>__builtins__</code>	Referenziert ein Dictionary, das die Namen aller eingebauten Typen und Funktionen als Schlüssel und die mit den Namen verknüpften Instanzen als Werte enthält.
<code>__file__</code>	Referenziert einen String, der den Namen der Programmdatei des Moduls inklusive Pfad enthält. Nicht bei Modulen der Standardbibliothek verfügbar.
<code>__name__</code>	Referenziert einen String, der den Namen des Moduls enthält.

Tabelle 11.1 Globale Variablen in einem Modul

11.3 Pakete

Python ermöglicht es Ihnen, mehrere Module in einem sogenannten *Paket* zu kapseln. Das ist vorteilhaft, wenn diese Module thematisch zusammengehören. Ein Paket kann, im Gegensatz zu einem einzelnen Modul, beliebig viele weitere Pakete enthalten, die ihrerseits wieder Module bzw. Pakete enthalten können.

Um ein Paket zu erstellen, muss im Wesentlichen ein Unterordner im Programmverzeichnis erzeugt werden. Der Name des Ordners entspricht dem Namen des Pakets. Zusätzlich muss in diesem Ordner eine Programmdatei namens `__init__.py` existieren. (Beachten Sie, dass es sich um jeweils zwei Unterstriche vor und hinter »init« handelt.) Diese Datei darf leer, muss aber vorhanden sein und enthält Initialisierungscode, der beim Einbinden des Paketes einmalig ausgeführt wird. Ein Programm mit mehreren Paketen und Unterpaketen hat also eine solche oder ähnliche Verzeichnisstruktur wie in Abbildung 11.1.

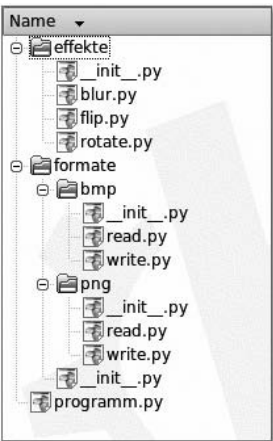


Abbildung 11.1 Paketstruktur eines Beispielprogramms

Es handelt sich um die Verzeichnisstruktur eines fiktiven Bildbearbeitungsprogramms. Das Hauptprogramm befindet sich in der Datei *programm.py*. Neben dem Hauptprogramm existieren im Programmverzeichnis zwei Pakete:

- ▶ Das Paket `effekte` soll bestimmte Effekte auf ein bereits geladenes Bild anwenden. Dazu enthält das Paket neben der Datei `__init__.py` drei Module, die jeweils einen grundlegenden Effekt durchführen. Es handelt sich um die Module `blur` (zum Verwischen des Bildes), `flip` (zum Spiegeln des Bildes) und `rotate` (zum Drehen des Bildes).
- ▶ Das Paket `formate` soll dazu in der Lage sein, bestimmte Grafikformate zu lesen und schreiben. Dazu definiert es in seiner `__init__.py` zwei Funktionen namens `leseBild` und `schreibeBild`. Wir möchten nicht näher auf Funktionschnittstellen oder Ähnliches eingehen, sondern relativ abstrakt bleiben. Damit das Lesen und Schreiben von Grafiken diverser Formate möglich ist, enthält das Paket `formate` zwei Unterpakete namens `bmp` und `png`, die je zwei Module zum Lesen bzw. Schreiben des entsprechenden Formats enthalten.

Im Hauptprogramm sollen zunächst die Pakete `effekte` und `formate` eingebunden und verwendet werden. Dies ermöglicht die `import`-Anweisung:

```
import effekte, formate
```

bzw.:

```
import effekte
import formate
```

Beachten Sie, dass es zu einem Namenskonflikt kommt, wenn beispielsweise neben dem Paket `effekte` ein Modul gleichen Namens, also eine Programmdatei namens *effekte.py*, existiert. Es ist grundsätzlich so, dass bei Namensgleichheit ein Paket Vorrang vor einem Modul hat, es also keine Möglichkeit mehr gibt, das Modul zu importieren.

Durch die `import`-Anweisung wird die Programmdatei `__init__.py` des einzubindenden Paketes ausgeführt und der Inhalt dieser Datei als Modul in einem eigenen Namensraum verfügbar gemacht. So könnten Sie nach den obigen `import`-Anweisungen folgendermaßen auf die Funktionen `leseBild` und `schreibeBild` zugreifen:

```
formate.leseBild()
formate.schreibeBild()
```

Um das nun geladene Bild zu modifizieren, soll diesmal ein Modul des Paketes `effekte` geladen werden. Auch dies ist mit der `import`-Anweisung möglich. Der

Paketname wird durch einen Punkt vom Modulnamen getrennt. Auf diese Weise kann ein Modul aus einer beliebigen Paketstruktur importiert werden:

```
import effekte.blur
```

In diesem Fall wurde das Paket `effekte` vorher eingebunden. Wenn dies nicht der Fall gewesen wäre, so würde das Importieren von `effekte.blur` dafür sorgen, dass zunächst das Paket `effekte` eingebunden und die dazugehörige `__init__.py` ausgeführt würde. Danach wird das Untermodul `blur` eingebunden. Das Modul kann fortan wie jedes andere verwendet werden:

```
effekte.blur.verschwemmeBild()
```

Beachten Sie, dass sich das Verhalten der hier besprochenen Version der `import`-Anweisung verändert, wenn Sie sich in einer Paketstruktur befinden. Dies soll das Thema des nächsten Abschnitts sein.

11.3.1 Absolute und relative Import-Anweisungen

Große Bibliotheken bestehen häufig nicht nur aus einem Modul oder Paket, sondern enthalten diverse Unterpakete, definieren also eine beliebig komplexe *Paketstruktur*. In einer solchen Paketstruktur ist eine Variante der `import`-Anweisung denkbar, die ein Unterpaket anhand einer relativen Pfadangabe einbindet, beispielsweise das Paket mit dem Namen `xyz` zwei Ebenen über dem einbindenden Paket. Eine solche spezielle `import`-Anweisung existiert seit Python 2.5 und wird *relative import-Anweisung* genannt.

Seit Python 3.0 führt die normale `import`-Anweisung innerhalb einer Paketstruktur einen sogenannten *absoluten Import* durch. Das bedeutet, dass über die bisher besprochene Syntax

```
import xyz
```

kein Modul oder Unterpaket `xyz` im lokalen Paketverzeichnis eingebunden wird, sondern stets ein Modul oder Paket aus einem globalen Bibliotheksverzeichnis.² Wenn das Modul `xyz` im globalen Namensraum nicht existiert, wird nicht auf das lokale Paketverzeichnis zurückgegriffen, sondern eine `ImportError`-Exception geworfen.

2 Das globale Bibliotheksverzeichnis können Sie über die im Modul `sys` enthaltene Liste `path` in Erfahrung bringen. Näheres dazu finden Sie in Abschnitt 17.3, »Zugriff auf die Laufzeitumgebung – `sys`«.

Um ein Paket in der lokalen Paketstruktur einzubinden, müssen wir uns der relativen `import`-Anweisung bedienen, die folgendermaßen geschrieben wird:

```
from . import xyz
```

Diese Anweisung bindet das Paket (oder das Modul) `xyz` aus dem Verzeichnis ein, das zwischen `from` und `import` angegeben wird. Ein Punkt steht dabei für das aktuelle Verzeichnis. Jeder weitere Punkt symbolisiert das ein Level höher gelegene Verzeichnis. Die Anweisung

```
from ...math import pi
```

importiert beispielsweise das Objekt `pi` aus dem Modul `math`, das sich zwei Ebenen über dem aktuellen Paketverzeichnis befindet.

Wenn eine relative `import`-Anweisung außerhalb einer Paketstruktur ausgeführt wird, beispielsweise im interaktiven Modus, wird eine `ValueError`-Exception geworfen:

```
>>> from . import bla
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Attempted relative import in non-package
```

Beachten Sie, dass die eingangs besprochenen Möglichkeiten zur Umbenennung eines eingebundenen Pakets oder Moduls auch bei relativen `import`-Anweisungen wie erwartet funktionieren:

```
from . import xyz as bla
```

Diese Anweisung bindet das Modul oder das Paket `xyz` aus dem lokalen Paketverzeichnis unter dem Namen `bla` ein.

11.3.2 Importieren aller Module eines Pakets

Bisher konnte mit

```
from abc import *
```

der gesamte Inhalt eines Moduls in den aktuellen Namensraum importiert werden. Dies funktioniert für Pakete nicht. Der Grund dafür ist, dass einige Betriebssysteme, darunter vor allem Windows, bei Datei- und Ordnernamen nicht zwischen Groß- und Kleinschreibung unterscheiden – Python aber sehr wohl. Angenommen, die obige Anweisung würde wie gehabt funktionieren und `abc` wäre ein Paket, so wäre es beispielsweise unter Windows völlig unklar, ob ein

Untermodul namens `modul` als `Modul`, `MODUL` oder `modul` eingebunden werden soll.

Aus diesem Grund importiert die obige Anweisung nicht alle im Paket enthaltenen Module in den aktuellen Namensraum, sondern importiert nur das Paket an sich und führt den Initialisierungscode in `__init__.py` aus. Sowohl alle in dieser Datei angelegten Elemente als auch alle Elemente von eventuell vorher importierten Modulen dieses Pakets werden in den aktuellen Namensraum eingeführt.

Es gibt zwei Möglichkeiten, das gewünschte Verhalten der obigen Anweisung zu erreichen. Beide müssen vom Autor des Pakets implementiert werden.

- ▶ Zum einen können alle Module des Pakets innerhalb der `__init__.py` per `import`-Anweisung importiert werden. Dies hätte zur Folge, dass sie beim Einbinden des Pakets und damit nach dem Ausführen des Codes der `__init__.py`-Datei eingebunden wären.
- ▶ Zum anderen kann dies durch Anlegen einer Referenz namens `__all__` geschehen. Diese muss eine Liste von Strings mit den zu importierenden Modulnamen referenzieren:

```
__all__ = ["blur", "flip", "rotate"]
```

Es liegt im Ermessen des Programmierers, welches Verhalten `from abc import *` bei seinen Paketen zeigen soll. Beachten Sie aber, dass das Importieren des kompletten Modul- bzw. Paketinhalts in den aktuellen Namensraum zu unerwünschten Namenskonflikten führen kann. Aus diesem Grund sollten Sie importierte Module stets in einem eigenen Namensraum führen.

11.3.3 Die Built-in Function `__import__`

Es existiert eine Built-in Functions, die sich auf Modularisierung, also auf das Einbinden von Modulen und Paketen, bezieht. Diese Funktion wurde in Abschnitt 10.7, »Vordefinierte Funktionen«, nicht erläutert, da das Konzept der Modularisierung Ihnen zu diesem Zeitpunkt noch nicht bekannt war. Aus diesem Grund soll die Beschreibung der Built-in Function `__import__` an dieser Stelle nachgeholt werden. Beachten Sie, dass diese Funktion nur in wenigen Fällen benötigt und daher an dieser Stelle nur oberflächlich erläutert wird. Ausführliche Informationen über die Funktionen finden Sie in der Python-Dokumentation.

`__import__(name[, globals[, locals[, fromlist[, level]]])`

Die Built-in Function `__import__` wird von der `import`-Anweisung verwendet, um ein Modul oder Paket einzubinden. Die Funktion existiert hauptsächlich, damit sie vom Programmierer überschrieben werden kann, um das Verhalten der

`import`-Anweisung zu verändern. Zum Überschreiben der Funktion muss eine neue Funktion mit gleicher Schnittstelle erstellt und dem Namen `__import__` zugewiesen werden.

Die Funktion `__import__` bindet das Modul oder Paket *name* ein und gibt den erzeugten Namensraum zurück. Dabei kann für *globals* und *locals* jeweils ein Dictionary übergeben werden, das alle Referenzen des globalen bzw. lokalen Namensraums enthält. Ein solches Dictionary wird von den Built-in Functions `globals` und `locals` erstellt. Für den vierten Parameter, *fromlist*, kann eine Liste mit Namen übergeben werden, die aus dem Modul *name* eingebunden werden sollen. Der fünfte Parameter, *level*, gibt an, ob absolutes oder relatives Importverhalten verwendet werden soll (vgl. Abschnitt 11.3.1). Der voreingestellte Wert von `-1` weist die Funktion `__import__` dazu an, sowohl absolutes als auch relatives Importverhalten zu zeigen. Ein Wert von `0` schreibt absolutes Importverhalten vor, während ein positiver Wert größer null die Anzahl der übergeordneten Verzeichnisse festlegt, die beim relativen Importverhalten einbezogen werden sollen.

Die beiden `import`-Anweisungen

```
import bla
from blubb import hallo, welt
```

resultieren intern in den folgenden Aufrufen von `__import__`:

```
__import__("bla")
__import__("blubb", globals(), locals(), ["hallo", "welt"], -1)
```

So viel zum Thema Modularisierung. Wenden wir uns nun einem weiteren interessanten Themengebiet zu: der objektorientierten Programmierung in Python.

»Abstraction is selective ignorance.«
– Andrew Koenig

12 Objektorientierung

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

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

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

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

Wir gehen modellhaft davon aus, dass jedes Konto einen "Inhaber" hat, der durch einen String mit seinem Namen identifiziert wird. Das Konto hat eine ganzzahlige "Kontonummer", um es von allen anderen Konten zu unterscheiden. Mit der Gleitkommazahl, die mit dem Schlüssel "Kontostand" verknüpft ist, wird das

aktuelle Guthaben in Euro gespeichert. Die Schlüssel "MaxTagesumsatz" und "UmsatzHeute" dienen dazu, den Tagesumsatz eines jeden Kunden zu seinem eigenen Schutz auf ein bestimmtes Limit zu begrenzen. "MaxTagesumsatz" gibt dabei an, wie viel Geld pro Tag maximal von dem bzw. auf das Konto bewegt werden darf. Mit "UmsatzHeute" »merkt« sich das System, wie viel am heutigen Tag schon umgesetzt worden ist. Zu Beginn eines neuen Tages wird dieser Wert wieder auf null gesetzt. Die von uns betrachteten Konten sollen prinzipiell nicht überzogen werden können, der Kontostand bleibt also immer positiv.

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

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

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

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

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

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

```

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

```

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

Die letzten Operationen für unsere Modellkonten sind das Ein- beziehungsweise Auszahlen am Geldautomaten oder Bankschalter. Beide Funktionen benötigen als Parameter das betreffende Konto und den jeweiligen Geldbetrag. Da die Funktionen sehr einfach sind, möchten wir uns nicht weiter mit Erklärungen aufhalten, sondern direkt den Quellcode präsentieren:

```

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

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

```

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

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

```

def zeige_konto(konto):
    print("Konto von {}".format(konto["Inhaber"]))
    print("Aktueller Kontostand: {:.2f} Euro".format(
        konto["Kontostand"]))
    print("(Heute schon {:.2f} von {} umgesetzt)".format(
        konto["UmsatzHeute"], konto["MaxTagesumsatz"]))

```

Mit diesen Definitionen könnten wir beispielsweise folgende Bankoperationen simulieren:

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

Zuerst eröffnet Heinz Meier ein neues Konto `k1` mit der Kontonummer 567123 mit dem Startguthaben von 12.350 Euro. Erwin Schmidt zahlt 15.000 Euro auf sein neues Konto `k2` mit der Kontonummer 396754 ein. Beide haben den standardmäßigen maximalen Tagesumsatz von 1.500 Euro gewählt. Nun treten die beiden in geschäftlichen Kontakt miteinander, wobei Herr Schmid einen DVD-Recorder von Herrn Meier für 160 Euro kauft und ihn per Überweisung bezahlt. Am selben Tag erwirbt Herr Meier Herrn Schmidts gebrauchten Spitzenlaptop, der für 1.000 Euro den Besitzer wechselt. Als Herr Meier in den Abendstunden stark an der Heimkinoanlage von Herrn Schmid interessiert ist und ihm dafür 500 Euro überweisen möchte, wird er enttäuscht, denn die Überweisung schlägt fehl. Völlig verdattert zieht Herr Schmidt den voreiligen Schluss, er habe zu wenig Geld auf seinem Konto. Deshalb möchte er den Betrag auf sein Konto einzahlen und anschließend erneut überweisen. Als aber auch die Einzahlung abgelehnt wird, wendet er sich an einen Bankangestellten. Dieser lässt sich die Informationen der beteiligten Konten anzeigen. Dabei sieht er, dass die gewünschte Überweisung das Tageslimit von Herrn Schmidts Konto überschreitet und deshalb nicht ausgeführt werden kann.

Wie Sie sehen, arbeitet unsere Banksimulation wie erwartet und ermöglicht uns eine relativ einfache Handhabung von Kontodaten. Sie weist aber einige unschöne Eigenheiten auf, wir im Folgenden besprechen werden.

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

```
>>> k1 = neues_konto("Heinz Meier", 567123, 12350.0)
>>> k2 = neues_konto("Erwin Schmidt", 396754, 15000.0)
>>> k1["Kontostand"] = [3, "Hehe, das gibt einen tollen Fehler"]
>>> geldtransfer(k1, k2, 160)
Traceback (most recent call last):
  [...]
TypeError: unorderable types: list() < int()
```

Wir wünschen uns also eine Möglichkeit, die eigentlichen Daten, also im Beispiel das Konto, mit den Verarbeitungsfunktionen zu einer Einheit zu koppeln und diese Verbindung vor direkten Zugriffen auf die enthaltenen Daten zu schützen, um ihre Konsistenz zu sichern. Genau diese Wünsche befriedigt die Objektorientierung, indem sie Daten und Verarbeitungsfunktionen zu sogenannten *Objekten* zusammenfasst. Dabei werden die Daten eines solchen Objekts *Attribute* und die Verarbeitungsfunktionen *Methoden* genannt. Attribute und Methoden werden unter dem Begriff *Member* einer Klasse zusammengefasst. Schematisch ließe sich das Objekt eines Kontos also folgendermaßen darstellen:

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

Tabelle 12.1 Schema eines Konto-Objekts

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

Angenommen, `k1` und `k2` seien Konto-Objekte, wie sie das obige Schema zeigt, mit den Daten von Herrn Meier und Herrn Schmidt; dann könnten wir das letzte Beispiel folgendermaßen formulieren (der Code ist so natürlich noch nicht lauffähig, da die Definition für die Konto-Objekte fehlt):

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

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

12.1 Klassen

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

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

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

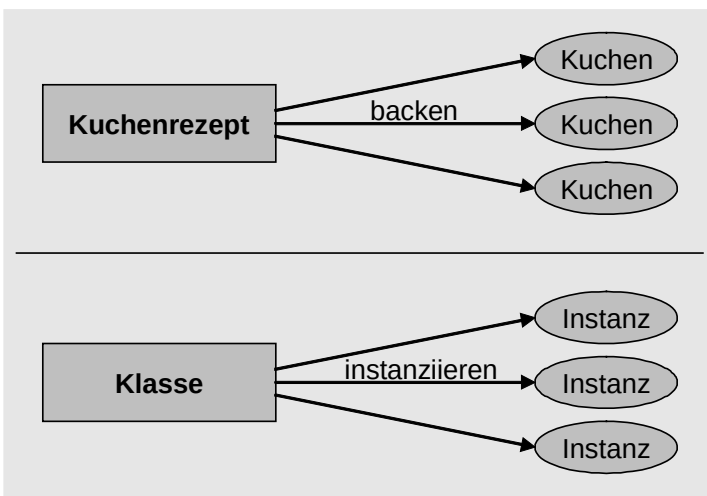


Abbildung 12.1 Analogie von Rezept/Kuchen und Klasse/Objekt

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

```
class Konto:
    pass
```

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

```
>>> Konto()
<__main__.Konto object at 0xb787776c>
```

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

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

12.1.1 Definieren von Methoden

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

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

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

    def einzahlen(self, betrag):
        pass

    def auszahlen(self, betrag):
        pass

    def zeige_konto(self):
        pass
```

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

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

Hier wird an die Methode `einzahlen` eine Referenz auf das Konto `k` übergeben, auf das dann innerhalb von `einzahlen` über den Parameter `self` zugegriffen werden kann.

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

12.1.2 Konstruktor, Destruktor und die Erzeugung von Attributen

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

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

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

```
>>> Beispielklasse()
Hier spricht der Konstruktor
<__main__.Beispielklasse object at 0xb7877a4c>
```

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

Dem Konstruktor steht der sogenannte *Destruktor* (engl. *destruct* = »zerstören«) gegenüber, der immer dann aufgerufen wird, wenn eine Instanz von der Garbage

Collection aus dem Speicher entfernt wird. Ein Destruktor ist eine bis auf `self` parameterlose Methode, die auf den Namen `__del__` hört:

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

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

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

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

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

```
>>> v1 = Beispielklasse()
Hier spricht der Konstruktor
>>> v2 = v1
>>> del v1
>>> del v2
Und hier kommt der Destruktor
```

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

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

Neue Attribute anlegen

Da es die Hauptaufgabe eines Konstruktors ist, einen konsistenten Initialzustand einer Instanz herzustellen und sie damit in einen benutzbaren Zustand zu versetzen, sollten alle Attribute einer Klasse auch dort definiert werden.¹ Die Definition neuer Attribute erfolgt durch eine einfache Wertezuweisung, wie Sie sie von normalen Variablen kennen. Damit können wir die Funktion `neues_konto` durch den Konstruktor der Klasse `Konto` ersetzen, der dann wie folgt implementiert werden kann; für den Parameter `self` wird dabei beim Aufruf automatisch eine Referenz auf die neu erzeugte `Konto`-Instanz übergeben:

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

`# hier kommen die restlichen Methoden hin`

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

```
class Konto:
    # hier kommt der Konstruktor hin

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

¹ Es gibt sehr wenige Sonderfälle, in denen diese Regel eine unpraktische Einschränkung ist. Deshalb müssen Sie nicht zwingend alle Attribute in der `__init__`-Methode definieren. Sie sollten aber im Regelfall, soweit es möglich ist, alle Attribute Ihrer Klassen im Konstruktor anlegen.

```

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

# hier wären die restlichen Methoden

```

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

```

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

```

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

12.1.3 Private Member

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

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

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

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

Im Gegensatz zu vielen anderen Programmiersprachen, die dieses Konzept mit eigenen Schlüsselwörtern implementieren, legt in Python der Name eines Members fest, ob es von außen explizit verwendet werden soll oder nicht. Dabei gibt es drei Kategorien:

Namensschema	Bezeichnung	Bedeutung
name	<i>public</i> (öffentlich)	Normale Member ohne führende Unterstriche sind sowohl innerhalb einer Klasse also auch von außen les- und schreibbar.
<u>name</u>	<i>protected</i> (geschützt)	Auf Members, deren Name mit einem Unterstrich beginnt, kann zwar sowohl von innen als auch von außen lesend und schreibend zugegriffen werden, aber der Entwickler einer Klasse teilt den anderen Programmierern dadurch mit, dass dieses Member nicht direkt benutzt werden sollte. ²
<u><u>name</u></u>	<i>private</i> (privat)	Namen mit zwei führenden Unterstrichen sind für wirklich private Member gedacht, die von außen nicht sichtbar sind und deshalb nur über Methoden der Klasse verändert und ausgelesen werden können. ³

Tabelle 12.2 Namensschemata für öffentliche, geschützte und private Member

Protected Members sind weiterhin nach außen sichtbar und voll veränderbar. Sie sind nur nach einer Konvention geschützt, die es allen Programmierern empfiehlt, solche Attribute von außen nicht zu benutzen. Es handelt sich hierbei um eine Schnittstellendefinition, die nicht durch eine technische Lese- bzw. Schreibsperrre erreicht wird, sondern auf einer Konvention zwischen allen Python-Programmierern beruht: Member, die mit einem Unterstrich beginnen, sollen von außen nicht benutzt werden. Wer es trotzdem tut, sollte sich darüber im Klaren sein, dass dies zu nicht beabsichtigtem Verhalten führen kann. Der Vorteil einer

² Insbesondere sollten Sie sich nicht darauf verlassen, dass als *protected* gekennzeichnete Member in neuen Versionen einer Programmbibliothek erhalten bleiben.

³ Wenn man es ganz genau nimmt, sind auch diese Member nicht wirklich gegen Zugriffe von außen geschützt: Sie werden intern von Python durch Namen des Schemas `_Klassenname_Attributname` ersetzt, und deshalb führen Versuche, von außen auf die ursprünglichen Namen zuzugreifen, zu Fehlern. Über den geänderten Namen kann aber weiterhin von überall aus auf die Attribute zugegriffen werden.

solchen Privatisierung durch eine Abmachung besteht gegenüber der technischen Sperre darin, dass immer noch auf die Member zugegriffen werden kann, wenn dies unbedingt erforderlich sein sollte. Dies erleichtert beispielsweise das Entwickeln von Debuggern zur Fehlersuche in Programmen oder Analysetools enorm.

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

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

    # hier wären die restlichen Methoden
```

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

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

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

```
class Konto:
    # hier wäre der Konstruktor
```

```
def kontostand(self):
    return self.__Kontostand
```

```
# hier wären die restlichen Methoden
```

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

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

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

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

```
class Konto:
    # hier wäre der Konstruktor

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

    # Setter-Methode für das Tageslimit
    def setMaxTagesumsatz(self, neues_limit):
        if (type(neues_limit) in (float, int) and
```

```

    neues_limit > 0):
        self.__MaxTagesumsatz = neues_limit
        return True
    else:
        return False

```

hier wären die restlichen Methoden

Das Methoden-Paar `maxTagesumsatz` und `setMaxTagesumsatz` ermöglicht nun den komfortablen und trotzdem sicheren Zugriff auf den maximalen Tagesumsatz, indem sichergestellt wird, dass nur gültige Werte gespeichert werden. Die Setter-Methode prüft, ob der Datentyp von `neues_limit` entweder `float` oder `int` ist und ob sein Wert im gültigen Bereich liegt, und setzt abhängig vom Ausgang dieser Prüfung das Attribut `__MaxTagesumsatz` auf den neuen Wert oder eben nicht. Anhand des Rückgabewertes der Funktion kann der Bankangestellte dann sehen, ob er einen Fehler bei der Übergabe gemacht hat.⁴

12.1.4 Versteckte Setter und Getter

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

```

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

```

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

```

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

```

4 Eine elegantere Methode, die aufrufende Ebene auf solche Fehler hinzuweisen, lernen Sie in Abschnitt 13.1, »Exception Handling«, kennen. Sie könnten dann beispielsweise bei ungültigen Werten einen `ValueError` produzieren.

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

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

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

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

```
class Konto:
    # hier wäre der Konstruktor

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

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

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

    # hier wären die restlichen Methoden
```


Die `print`-Anweisungen dienen nur dazu, dass wir in unserem Beispiel gleich sehen können, dass die Methoden auch wirklich aufgerufen werden. Außerdem wurden die Rückgabewerte von `setMaxTagesumsatz` entfernt, da diese die aufrufende Ebene nicht mehr erreichen können und somit sinnlos geworden sind.⁵

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

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

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

12.1.5 Statische Member

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

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

5 Um Fehler zu signalisieren, sollte der Setter Exceptions werfen. Wie das geht, lernen Sie in Abschnitt 13.1, »Exception Handling«.

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

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

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

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

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

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

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

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

# hier wären die restlichen Methoden
```

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

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

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

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

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

```

class Konto:
    Anzahl = 0 # Zu Beginn ist die Instanzanzahl 0

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

    zeigeAnzahl = staticmethod(zeigeAnzahl)

# Die restlichen Member wären hier

```

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

```

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

```

12.2 Vererbung

Bisher haben wir nur objektorientierte Techniken behandelt, die durch Kapselung von Daten und Definition von Schnittstellen die Konsistenz der Objekte sichern. Eines der zu Anfang des Kapitels angesprochenen Ziele der Objektorientierung war es aber auch, dass unsere Programme auch leicht veränderlich sind, so dass sie auf Probleme angewandt werden können, die dem ursprünglichen Problem ähnlich sind. Dieses Ziel erreichen wir aber mit den bis jetzt eingeführten Techniken noch nicht.

Wir haben im letzten Abschnitt unsere Klasse `Konto` so erweitert, dass sie über ein statisches Attribut die Anzahl ihrer Instanzen nachhalten konnte. Wenn wir nun eine neue Klasse definieren wollten – nehmen wir beispielhaft eine Klasse, die Angestellte der Bank beschreibt – und diese ebenfalls die Anzahl ihrer eigenen Instanzen – in dem Fall also die Zahl der Angestellten – ermitteln soll, so müssten wir den Quellcode für das Instanzzählen ein weiteres Mal in die Klasse `Angestellter` schreiben. Es wäre wünschenswert, einmal festzulegen, wie eine Klasse ihre eigenen Instanzen zählt, und diese Fähigkeit ohne erneutes Aufschreiben des Codes auf neue Klassen übertragen zu können.

Dieses Konzept, Fähigkeiten einer Klasse auf eine andere zu übertragen, nennt man *Vererbung*, wobei alle Member, also sowohl Attribute als auch Methoden, von der Mutter- auf die Tochterklasse übertragen werden. In unserem Beispiel hätten wir also eine Mutterklasse `Zaehler`, die die Instanzzählung implementiert und von der die Klassen `Konto` und `Angestellter` diese Fähigkeit erben:

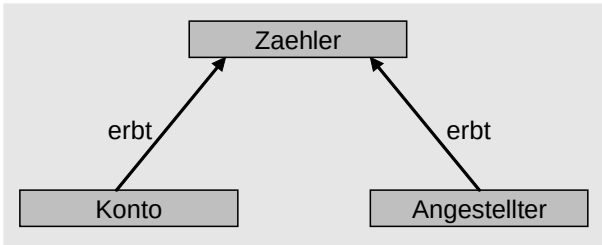


Abbildung 12.2 »Konto« und »Angestellter« erben von »Zaehler«.

Man spricht auch davon, dass die *Basisklasse* `Zaehler` ihre Member an die beiden *Subklassen*, `Konto` und `Angestellter`, vererbt.

Wir wollen nun das angegebene Beispiel in Python implementieren, wobei wir uns zuerst der `Zaehler`-Klasse zuwenden:

```

class Zaehler:
    Anzahl = 0

    def __init__(self):
        type(self).Anzahl += 1

    def __del__(self):
        type(self).Anzahl -= 1
  
```

Die Definition enthält bis auf den Zugriff auf das Attribut `Anzahl` mittels `type(self)` nichts Neues. Wir können deshalb nicht mehr direkt über den Klassennamen `per Zaehler.Anzahl` auf das Attribut zugreifen, weil wir von der Klasse erben wollen und die Subklassen jeweils ihr eigenes statisches Attribut `Anzahl` haben sollen. Würden wir mit `Zaehler.Anzahl` arbeiten, könnten wir damit die Gesamtanzahl der `Konto`- und `Angestellter`-Instanzen berechnen. Mithilfe von `type` lässt sich der Datentyp einer Instanz ermitteln, und das nutzen wir, um den Zähler abhängig davon, welchen Typ `self` hat, für die richtige Klasse zu ändern.

Um nun unsere Klasse `Konto` von `Zaehler` erben zu lassen, müssen wir hinter den Klassennamen `Konto` die gewünschte Basisklasse `Zaehler` in Klammern schreiben:

```

class Konto(Zaehler):
    def __init__(self, inhaber, kontonummer, kontostand,
                  max_tagesumsatz=1500):

        Zaehler.__init__(self) # Wichtige Zeile - siehe unten

        self.__Inhaber = inhaber
        self.__Kontonummer = kontonummer
  
```

```

self.__Kontostand = kontostand
self.__MaxTagesumsatz = max_tagesumsatz
self.__UmsatzHeute = 0

```

```

# hier wären die restlichen Methoden

```

Im Wesentlichen haben sich bei der neuen Definition von `Konto` nur das schon angesprochene Einfügen des geklammerten Basisklassennamens `Zaehler` hinter `Konto` vorgenommen und die erste Zeile des Konstruktors geändert. Den Konstruktor der Basisklasse rufen wir mit `Zaehler.__init__(self)` auf, um unser `Konto` auch als Zähler benutzen zu können. Dies ist deshalb notwendig, weil eine Klasse nur eine Methode `__init__` haben kann. Bei der Vererbung tritt nun oft der Fall ein, dass die erbende Klasse Methoden definiert, die auch schon in der Basisklasse vorhanden waren – in unserem Beispiel eben der Konstruktor `__init__`. In einem solchen Fall werden die Methoden der Basisklasse mit denen, die die Subklasse selbst definiert, überschrieben, so dass im Beispiel `self.__init__` eine Referenz auf den Konstruktor von `Konto` und nicht auf den von `Zaehler` enthält. Um trotzdem auf solche überschriebenen Methoden zugreifen zu können, ersetzt man beim Aufruf das `self` vor dem Punkt durch den Namen der entsprechenden Basisklasse und übergibt `self` explizit als Parameter. Würde `Zaehler.__init__` noch weitere Parameter erwarten, so würden diese wie üblich durch Kommata getrennt dahinter geschrieben.

Sie sollten sich außerdem als wichtige Regel merken, dass Sie im Konstruktor einer abgeleiteten Klasse immer den Konstruktor der Basisklasse aufrufen müssen, weil Ihre Instanzen sonst aufgrund der fehlenden Initialisierung in einen nicht definierten Zustand übergehen und sich damit unerwartet verhalten können.

In unserem Fall würde die Instanzzählung ohne den Aufruf des Konstruktors der Basisklasse nicht funktionieren, da der Zähler nicht mit 0 initialisiert würde.

Natürlich können Sie von einer erbenden Klasse weitere Klassen erben lassen, so dass ganze »Stammbäume« entstehen. Wenn Sie beispielsweise bei der Speicherung der Bankangestellten eigene Klassen für jeden Tätigkeitsbereich definieren möchten, so könnten diese von der Klasse `Angestellter` erben, die wiederum `Zaehler` als Basisklasse hat:

```

class Angestellter(Zaehler):
    def __init__(self, name, stundenlohn, stunden_pro_woche):
        Zaehler.__init__(self)
        self.Name = name
        self.Stundenlohn = stundenlohn
        self.StundenProWoche = stunden_pro_woche

```

```

def befoerdere(self, neue_position):
    # hier würde der Code für eine Beförderung stehen
    pass

```

Unsere Angestellten haben der Einfachheit halber nur ihren Namen, ihren Stundenlohn und ihre durchschnittliche Arbeitszeit pro Woche in Stunden als Attribute. Nun könnten wir die beiden speziellen Angestellten, Sekretarin und Bankdirektor, definieren, die jeweils von der Klasse Angestellter erben:

```

class Sekretarin(Angestellter):
    def __init__(self, name):
        Angestellter.__init__(self, name, 15, 30)

class Bankdirektor(Angestellter):
    def __init__(self, name, dienstwagen):
        Angestellter.__init__(self, name, 150, 50)
        self.Dienstwagen = dienstwagen

```

Da es in unserer Bank Standardarbeitszeiten und einheitliche Gehälter für jede Position gibt, brauchen wir diese Informationen nicht mehr an den Konstruktor der abgeleiteten Klassen zu übergeben, sondern sie werden bei dem Aufruf des Konstruktors der Basisklasse intern weitergegeben. Die Sekretarin hat in unserem einfachen Beispiel neben den von Angestellter geerbten Mitgliedern keine weiteren Attribute oder Methoden, und der Bankdirektor bekommt neben dem »Erbgut« nur noch ein neues Attribut für seinen Dienstwagen dazu.

Mithilfe des Konzepts der Vererbung wird Ihr Programmtext in hohem Maße wiederverwendbar, vorausgesetzt, Sie machen sich bei der Strukturierung Ihrer Programme entsprechende Gedanken und zerlegen sie in sinnvoll aufgeteilte Klassen.

12.2.1 Mehrfachvererbung

Bisher haben wir eine Subklasse immer von genau einer Basisklasse erben lassen. Es gibt aber Situationen, in denen eine Klasse die Fähigkeiten von zwei oder noch mehr Basisklassen erben soll, um das gewünschte Ergebnis zu erzielen. Dieses Konzept, bei dem eine Klasse von mehreren Basisklassen erbt, wird *Mehrfachvererbung* genannt. Möchten Sie eine Klasse von mehreren Basisklassen erben lassen, müssen Sie die Basisklassen durch Kommata getrennt in die Klammern hinter den Klassennamen schreiben:

```

class NeueKlasse(Basisklasse1, Basisklasse2, Basisklasse3, ...):
    # Definition von Methoden und Attributen
    pass

```

Wir werden die Mehrfachvererbung an einem einfachen Beispiel verdeutlichen: Angenommen, wir möchten eine Klasse für die Beschreibung von Hausbooten entwickeln, so könnten wir einfach jeweils eine Klasse für die Beschreibung eines Hauses und eine für die eines Bootes definieren, so dass wir durch Vererbung Spezialformen wie das Ferienhaus oder das Rennboot von jeweils einer der Klassen erben lassen könnten.⁶

Unsere Hausbootklasse soll die Eigenschaften von beiden Klassen, `Haus` und `Boot`, erben. Die beiden Klassen für das Haus und das Boot könnten in stark vereinfachter Form folgendermaßen aussehen:

```
class Haus:
    def __init__(self, anzahl_stockwerke, anzahl_zimmer,
                  flaeche, hausnummer):
        self.AnzahlStockwerke = anzahl_stockwerke
        self.AnzahlZimmer = anzahl_zimmer
        self.Flache = flaeche
        self.Hausnummer = hausnummer

        self.HaustuerOffen = False

    def oeffneHaustuer(self):
        self.HaustuerOffen = True

    def schliesseHaustuer(self):
        self.HaustuerOffen = False

class Boot:
    def __init__(self, laenge, tiefgang, motorleistung):
        self.Laenge = laenge
        self.Tiefgang = tiefgang
        self.Motorleistung = motorleistung

        self.MotorIstEingeschaltet = False
        self.AnkerGeworfen = True

    def starteMotor(self):
        self.MotorIstEingeschaltet = True

    def stoppeMotor(self):
        self.MotorIstEingeschaltet = False
```

⁶ Dieses Beispiel ist zugegebenermaßen relativ praxisfern, eignet sich aber trotzdem gut, um das Konzept der Mehrfachvererbung zu veranschaulichen.


```

def werfeAnker(self):
    self.AnkerGeworfen = True

def ankerLichten(self):
    self.AnkerGeworfen = False

```

Die Klasse `Haus` kann sich einige grundlegende Eigenschaften eines Hauses merken und außerdem speichern, ob die Haustür gerade offen oder geschlossen ist. Außerdem bietet sie zum Öffnen und Schließen der Tür entsprechende Methoden an.

Mit der Klasse `Boot` kann man die Länge, den Tiefgang und die Motorleistung in PS speichern. Sie verfügt zusätzlich über Eigenschaften für den Status des Motors und des Ankers, die auch jeweils über Methoden gesetzt werden können.

Nun lassen wir unsere neue Klasse namens `Hausboot` von den Klassen `Haus` und `Boot` erben, wodurch sie alle Fähigkeiten von ihnen übernimmt. Da wir keine zusätzliche Funktionalität hinzufügen wollen, definieren wir nur einen Konstruktor für die Klasse `Hausboot`, der die Parameter an die Konstruktoren von `Haus` und `Boot` weitergibt:

```

class Hausboot(Haus, Boot):
    def __init__(self, anzahl_stockwerke, anzahl_zimmer,
                  flaeche, hausnummer,
                  laenge, tiefgang, motorleistung):

        Haus.__init__(self, anzahl_stockwerke, anzahl_zimmer,
                      flaeche, hausnummer)
        Boot.__init__(self, laenge, tiefgang, motorleistung)

```

Nun können wir eine Instanz der Klasse `Hausboot` erzeugen und zur Demonstration den Anker werfen und den Motor starten:

```

>>> mein_hausboot = Hausboot(2, 10, 200, 5, 20, 1.5, 1000)
>>> mein_hausboot.AnzahlStockwerke
2
>>> mein_hausboot.starteMotor()
>>> mein_hausboot.MotorIstEingeschaltet
True
>>> mein_hausboot.AnkerGeworfen
False
>>> mein_hausboot.werfeAnker()
>>> mein_hausboot.AnkerGeworfen
True

```

Wie das Beispiel zeigt, können wir die Instanz `mein_hausboot` problemlos wie ein `Haus` und wie ein `Boot` verwenden.

Mehrfachvererbung wird erst dann knifflig, wenn einer Klasse gleichnamige Attribute oder Methoden von verschiedenen Basisklassen vererbt werden.

Was wäre beispielsweise passiert, wenn die Klasse `Hausboot` keinen eigenen Konstruktor definiert hätte, der die Konstruktoren beider Basisklassen aufruft? Wäre der Konstruktor der Basisklasse `Haus` oder der der Klasse `Boot`, oder wären vielleicht beide aufgerufen worden?

Wenn in Python eine Klasse von mehreren Basisklassen gleichnamige Member erbt, wird nach der Reihenfolge entschieden, in der die Basisklassen angegeben werden: Es werden immer zuerst die Eigenschaften der weiter links stehenden Basisklasse vererbt.

Wenn wir also eine Klasse `Hausboot2` definieren, die ebenfalls von `Haus` und `Boot` erbt und deren Klassenkörper ausschließlich aus einer `pass`-Anweisung besteht, würde `Hausboot2` die `__init__`-Methode von `Haus` erben:

```
class Hausboot2(Haus, Boot):
    pass
```

```
>>> mein_hausboot2 = Hausboot2()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    mein_hausboot2 = Hausboot2()
TypeError: __init__() takes exactly 5 positional arguments (1 given)
```

Die Fehlermeldung teilt uns mit, dass der Konstruktor von `Hausboot2` genau fünf Parameter erwartet, was genau der Parameteranzahl des Konstruktors von `Haus` entspricht. Da die `__init__`-Methode von `Boot` nur vier Parameter benötigt, handelt es sich beim Konstruktor von `Hausboot2` also um den der `Haus`-Klasse.

Mögliche Probleme der Mehrfachvererbung

Es ist kein Zufall, dass nur wenige Sprachen das Konzept der Mehrfachvererbung unterstützen, da Programme, die es verwenden, anfällig für schwer auffindbare Fehler sind, weil gleichnamige Member auch dann überschrieben werden, wenn sie semantisch nichts miteinander zu tun haben.

Besonders kritisch wird es dann, wenn eine Klasse über Umwege mehrmals von derselben Basisklasse erbt. Betrachten wir einmal folgende vereinfachte Klassenhierarchie:

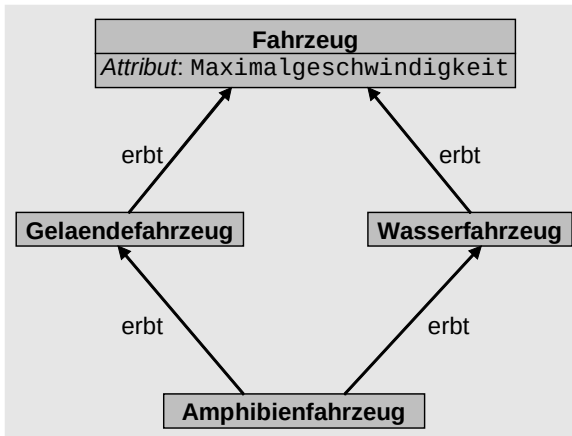


Abbildung 12.3 »Amphibienfahrzeug« erbt auf zwei Wegen von »Fahrzeug«.

Die Klasse `Amphibienfahrzeug` hat exakt ein Attribut `Maximalgeschwindigkeit`, das sie entweder von `Geländefahrzeug` oder von `Wasserfahrzeug` erbt, je nachdem, in welcher Reihenfolge die beiden Basisklassen bei der Definition von `Amphibienfahrzeug` angegeben wurden. Dies ist aber nicht sinnvoll, da sich die jeweilige `Maximalgeschwindigkeit` zu Lande und zu Wasser in der Regel unterscheidet. Eine brauchbare Klasse zur Beschreibung von Amphibienfahrzeugen lässt sich also nicht durch die gezeigte Mehrfachvererbung definieren, wie es die Intuition raten würde.

Sie sollten in Ihren eigenen Programmen sehr genau darauf achten, dass Sie nur dann Mehrfachvererbungen einsetzen, wenn dadurch keine Konflikte entstehen können, die den Sinn der resultierenden Klasse entstellen – und nach Möglichkeit ganz auf Mehrfachvererbungen verzichten.

12.3 Magic Members

Es gibt in Python eine Reihe spezieller Methoden und Attribute, um Klassen besondere Fähigkeiten zu geben. Die Namen dieser Member beginnen und enden jeweils mit zwei Unterstrichen `__`. Im Laufe der letzten Abschnitte haben Sie bereits zwei dieser sogenannten *Magic Members* kennengelernt: den Konstruktor namens `__init__` und den Destruktor namens `__del__`.

Der Umgang mit den Methoden und Attributen ist insofern »magisch«, als dass sie in der Regel nicht direkt mit ihrem Namen benutzt, sondern bei Bedarf implizit im Hintergrund verwendet werden. Der Konstruktor `__init__` wird z. B. immer dann aufgerufen, wenn ein neues Objekt einer Klasse erzeugt wird, auch

wenn kein expliziter Aufruf mit zum Beispiel `Klassenname.__init__()` an der entsprechenden Stelle steht.

Mit vielen Magic Members lässt sich das Verhalten von Built-in Functions und Operatoren für die eigenen Klassen anpassen, so dass die Instanzen Ihrer Klassen beispielsweise sinnvoll mit den Vergleichsoperatoren `<` und `>` verglichen werden können.

Wir werden Ihnen im Folgenden eine Liste präsentieren, die häufig genutzte Magic Members mit ihrer Bedeutung auflistet. Wegen der großen Anzahl verzichten wir dabei bei vielen der besprochenen Methoden und Attribute auf Beispiele. Wir bitten Sie, für genauere Informationen Pythons Online-Dokumentation zu konsultieren.

12.3.1 Allgemeine Magic Members

`__init__(self, ...)`

Der Destruktor einer Klasse. Wird beim Erzeugen einer neuen Instanz aufgerufen. Näheres können Sie in Abschnitt 12.1.2, »Konstruktor, Destruktor und die Erzeugung von Attributen«, nachlesen.

`__del__(self)`

Der Destruktor einer Klasse. Wird beim Zerstören einer neuen Instanz aufgerufen. Weitere Informationen finden Sie in Abschnitt 12.1.2, »Konstruktor, Destruktor und die Erzeugung von Attributen«.

`__repr__(self)`

Der Rückgabewert von `obj.__repr__` gibt an, was `repr(obj)` zurückgeben soll. Dies sollte nach Möglichkeit gültiger Python-Code sein, der beim Ausführen die Instanz `obj` erzeugt.

`__str__(self)`

Der Rückgabewert von `obj.__str__` gibt an, was `str(obj)` zurückgeben soll. Dies sollte nach Möglichkeit eine für den Menschen lesbare Repräsentation von `obj` sein.

Zugriff auf Attribute anpassen

Die Methoden in diesem Abschnitt dienen dazu, festzulegen, wie Python vorgehen soll, wenn die Attribute einer Instanz gelesen oder geschrieben werden. Da die Standardmechanismen in den meisten Fällen das gewünschte Resultat bewirken, werden Sie diese Methoden nur selten überschreiben.

`__dict__`

Jede Instanz besitzt ein Attribut namens `__dict__`, das die Member der Instanz in einem Dictionary speichert.

Die beiden folgenden Codezeilen produzieren also das gleiche Ergebnis, vorausgesetzt, `obj` ist eine Instanz einer Klasse, die ein Attribut `A` definiert:

```
>>> obj.A
"Der Wert des Attributs A"
>>> obj.__dict__["A"]
"Der Wert des Attributs A"
```

`__getattr__(self, name)`

Wird dann aufgerufen, wenn das Attribut mit dem Namen *name* gelesen wird, aber nicht existiert.

Die Methode `__getattr__` sollte entweder einen Wert zurückgeben, der für das Attribut gelten soll, oder einen `AttributeError` erzeugen.

`__getattribute__(self, name)`

Wird immer aufgerufen, wenn der Wert des Attributs mit dem Namen *name* gelesen wird, auch wenn das Attribut bereits existiert.

Implementiert eine Klasse sowohl `__getattr__` als auch `__getattribute__`, wird nur letztere Funktion beim Lesen von Attributen aufgerufen, es sei denn, `__getattribute__` ruft selbst `__getattr__` auf.

Wichtig

Greifen Sie innerhalb von `__getattribute__` niemals mit `self.attribut` auf die Attribute der Instanz zu, weil dies eine endlose Rekursion zur Folge hätte.

Benutzen Sie stattdessen immer `__getattribute__` der Basisklasse, zum Beispiel `object.__getattribute__(self, "attribut")`.

`__setattr__(self, name, value)`

Die Methode `__setattr__` wird immer dann aufgerufen, wenn der Wert eines Attributs per Zuweisung geändert oder ein neues Attribut erzeugt wird. Der Parameter *name* gibt dabei einen String an, der den Namen des zu verändernden Attributs enthält. Mit *value* wird der neue Wert übergeben.

Mit `__setattr__` lässt sich zum Beispiel festlegen, welche Attribute eine Instanz überhaupt haben darf, indem alle anderen Werte einfach ignoriert oder mit Fehlerausgaben quittiert werden.

Wichtig

Verwenden Sie niemals eine Zuweisung der Form `self.attribut = wert` innerhalb von `__setattr__`, um die Attribute auf bestimmte Werte zu setzen, da dies eine endlose Rekursion bewirken würde: Bei jeder Zuweisung würde `__setattr__` erneut aufgerufen.

Um Attributwerte mit `__setattr__` zu verändern, können Sie auf das Attribut `__dict__` zurückgreifen: `self.__dict__["attribut"] = wert`.

`__delattr__(self, name)`

Wird aufgerufen, wenn das Attribut mit dem Namen *name* per `del` gelöscht wird.

`__slots__`

Mit dem `__slots__`-Attribut können die Member einer Instanz in der Klasse genau definiert werden. Normalerweise ist es problemlos möglich, auch nach der Instanziierung neue Attribute und Methoden für eine Instanz zu erstellen bzw. Member zu löschen, wie das folgende Beispiel zeigt:

```
>>> class Test:
    def __init__(self):
        self.A = 1
        self.B = 2
>>> t = Test()
>>> t.A
1
>>> t.C = 1337
>>> t.C
1337
>>> del t.A
>>> t.A
Traceback (most recent call last):
  File "<pyshe11#12>", line 1, in <module>
    t.A
AttributeError: 'Test' object has no attribute 'A'
```

Dieses Verhalten ist oft aus mehreren Gründen nicht erwünscht:

Das dynamische Erstellen und Löschen von Membern kann zu schwer lokalisierbaren Fehlern führen und das Kapselungsprinzip verletzen. Außerdem muss der Interpreter Aufwand treiben, um die Dynamik der Member zu gewährleisten. Gerade bei Klassen, die sehr oft instanziiert werden sollen, kann dies zu Speicher- und Geschwindigkeitsproblemen führen.

Deshalb kann mit `__slots__` angegeben werden, welche Member eine Instanz einer Klasse haben darf. Erzeugen Sie zu diesem Zweck ein statisches Attribut

namens `__slots__`, dem Sie eine Sequenz der Namen zuweisen, die die Attribute und Methoden der Instanzen haben dürfen. Alle Versuche, auf andere Member als die mit `__slots__` definierten zuzugreifen, führen dann zu Fehlern. Außerdem benutzt Python für solche Instanzen eine effizientere Technik, um die Attribute und Methoden zu speichern, als bei »normalen« Klassen.

Im folgenden Beispiel darf die Klasse `Test` nur die Attribute namens `A` und `B` haben:

```
>>> class Test:
    __slots__ = ("A", "B")
    def __init__(self):
        self.A = 1
        self.B = 2

>>> t = Test()
>>> t.A
1
>>> t.C = 1337
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    t.C = 1337
AttributeError: 'Test' object has no attribute 'C'
>>> del t.A
>>> t.A
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    t.A
AttributeError: 'Test' object has no attribute 'A'
```

Wie Sie sehen, schlägt das Erstellen des neuen Attributs `C` mit einem `AttributeError` fehl. Es ist allerdings immer noch möglich, bereits vorhandene Attribute per `del` zu löschen. Diese können allerdings auch wieder erzeugt werden, sofern sie in der `__slots__`-Liste stehen.

Wichtig

Eine `__slots__`-Definition lässt sich nicht auf Subklassen vererben.

Vergleichsoperatoren

Die folgenden *Magic Methods* dienen dazu, das Verhalten der Vergleichsoperatoren für die Klasse anzupassen. Man nennt diese Anpassung auch *Überladen* des Operators.

Um beispielsweise zwei Kontoklassen zu vergleichen, kann die Kontonummer herangezogen werden. Damit gibt es eine sinnvolle Interpretation für den Vergleich mit `==` bei Konten. Die Magic Method für Vergleiche mit `==` heißt `__eq__` (von engl. *equal* = »gleich«) und erwartet als Parameter eine Instanz, mit der das Objekt verglichen werden soll, für das `__eq__` aufgerufen wurde.

Der folgende Beispielcode erweitert unsere `Konto`-Klasse aus der Einführung zur Objektorientierung um die Fähigkeit, sinnvoll mit `==` verglichen zu werden:

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

    def __eq__(self, k2):
        return self.Kontonummer == k2.Kontonummer
```

Nun erzeugen wir drei Konten, wobei zwei die gleiche Kontonummer haben, und vergleichen sie mit dem `==`-Operator. Das Szenario wird natürlich immer ein Wunschtraum für Donald Duck bleiben:

```
>>> konto1 = Konto("Dagobert Duck", 1337, 9999999999999999)
>>> konto2 = Konto("Donald Duck", 1337, 1.5)
>>> konto3 = Konto("Gustav Gans", 2674, "50000")
>>> konto1 == konto2
True
>>> konto1 == konto3
False
```

Die Anweisung `konto1 == konto2` wird intern von Python beim Ausführen durch `konto1.__eq__(konto2)` ersetzt.

Neben der `__eq__`-Methode gibt es eine Reihe weiterer Vergleichsmethoden, die jeweils einem Vergleichsoperator entsprechen. Alle diese Methoden erwarten neben `self` einen weiteren Parameter, der die Instanz referenzieren muss, mit der `self` verglichen werden soll.

Die folgende Tabelle zeigt alle Vergleichsmethoden mit ihren Entsprechungen. Die Herkunftstabelle kann Ihnen unter Umständen helfen, sich die Methodennamen und ihre Bedeutung besser zu merken:

Methode	Operator	Herkunft
<code>__lt__(self, other)</code>	<code><</code>	<i>lower than</i> (dt. »kleiner als«)
<code>__le__(self, other)</code>	<code><=</code>	<i>less or equal</i> (dt. »kleiner oder gleich«)
<code>__eq__(self, other)</code>	<code>==</code>	<i>equal</i> (dt. »gleich«)
<code>__ne__(self, other)</code>	<code>!=</code>	<i>not equal</i> (dt. »ungleich«)
<code>__gt__(self, other)</code>	<code>></code>	<i>greater than</i> (dt. »größer als«)
<code>__ge__(self, other)</code>	<code>>=</code>	<i>greater or equal</i> (dt. »größer oder gleich«)

Tabelle 12.3 Die Magic Methods für Vergleiche

Wichtig
Wenn eine Klasse keine der Methoden <code>__eq__</code> oder <code>__ne__</code> implementiert, werden Instanzen der Klasse mittels <code>==</code> und <code>!=</code> anhand ihrer Identität miteinander verglichen.

`__hash__(self)`

Die `__hash__`-Methode einer Instanz bestimmt, welchen Wert die Built-in Funktion `hash` für die Instanz zurückgeben soll. Die Hash-Werte müssen Ganzzahlen sein und sind insbesondere für die Verwendung von Instanzen als Schlüssel für Dictionaries von Bedeutung.

Die einzige Bedingung für gültige Hash-Werte ist, dass Objekte, die bei Vergleichen mit `==` als gleich angesehen werden, auch den gleichen Hash-Wert besitzen.

`__bool__(self)`

Die `__bool__`-Methode sollte einen Wahrheitswert (`True` oder `False`) zurückgeben, der angibt, wie das Objekt in eine `bool`-Instanz umzuwandeln ist.

Ist `__bool__` nicht implementiert, wird stattdessen der Rückgabewert von `__len__` verwendet. Sind beide Methoden nicht vorhanden, werden alle Instanzen der betreffenden Klasse als `True` behandelt.

Hinweis
In Python-Versionen vor 3.0 hieß die Methode <code>__nonzero__</code> anstelle von <code>__bool__</code> .

`__call__(self[, args...])`

Mit der `__call__`-Methode werden die Instanzen einer Klasse wie Funktionen aufrufbar.

Das folgende Beispiel implementiert eine Klasse `Potenz`, die dazu dient, Potenzen zu berechnen. Welcher Exponent dabei verwendet werden soll, wird dem Kon-

struktur als Parameter übergeben. Durch die `__call__`-Methode können die Instanzen von `Potenz` wie Funktionen aufgerufen werden, um Potenzen zu berechnen:

```
class Potenz:
    def __init__(self, exponent):
        self.Exponent = exponent

    def __call__(self, basis):
        return basis ** self.Exponent
```

Nun können wir bequem mit Potenzen arbeiten:

```
>>> dreier_potenz = Potenz(3)
>>> dreier_potenz(2)
8
>>> dreier_potenz(5)
125
```

12.3.2 Datentypen emulieren

In Python entscheiden die Methoden, die ein Datentyp implementiert, zu welcher Kategorie von Datentypen er gehört. Deshalb ist es möglich, Ihre eigenen Datentypen beispielsweise wie numerische oder sequentielle Datentypen »aussehen« zu lassen, indem sie die entsprechende Schnittstelle implementieren.

Sie werden im Folgenden die Methoden kennenlernen, die ein Datentyp implementieren muss, um ein numerischer Datentyp zu sein. Außerdem werden die Schnittstellen von Sequenzen und Mappings behandelt.

Numerische Datentypen emulieren

Ein numerischer Datentyp muss vor allem eine Reihe von Operatoren definieren.

Binäre Operatoren

Als Erstes gibt es die sogenannten binären Operatoren, die zwei Operanden erwarten. Hierzu zählen unter anderem `+`, `-`, `*` und `/`.

Alle Methoden zum Überladen von binären Operatoren erwarten einen Parameter, der den zweiten Operanden referenziert. Ihr Rückgabewert muss eine neue Instanz sein, die das Ergebnis der Rechnung enthält.

Wenn Python einen Ausdruck auswertet, der binäre Operatoren enthält, werden intern automatisch die entsprechenden Methoden aufgerufen. Die folgenden bei-

den Befehle sind vollkommen gleichwertig, wobei die Klammern um die 1 aus syntaktischen Gründen notwendig sind:

```
>>> 1 + 2
3
>>> (1).__add__(2)
3
```

Als Beispiel werden wir eine kleine Klasse zum Verwalten von Längenangaben mit Einheiten implementieren, die die Operatoren für Addition und Subtraktion unterstützt.

Die Klasse wird intern alle Maße für die Berechnungen in Meter umwandeln. Ihre Definition sieht dann folgendermaßen aus:

```
class Laenge:
    Umrechnung = {"m" : 1, "cm" : 0.01, "mm" : 0.001,
                  "dm" : 10, "km" : 1000,
                  "ft" : 0.3048,  # Fuß
                  "in" : 0.0254,  # Zoll
                  "mi" : 1609344  # Meilen
                  }

    def __init__(self, zahlenwert, einheit):
        self.Zahlenwert = zahlenwert
        self.Einheit = einheit

    def __str__(self):
        return "{0:f}{1}".format(self.Zahlenwert, self.Einheit)

    def __add__(self, other):
        z = self.Zahlenwert * Laenge.Umrechnung[self.Einheit]
        z += other.Zahlenwert * Laenge.Umrechnung[other.Einheit]
        z /= Laenge.Umrechnung[self.Einheit]
        return Laenge(z, self.Einheit)

    def __sub__(self, other):
        z = self.Zahlenwert * Laenge.Umrechnung[self.Einheit]
        z -= other.Zahlenwert * Laenge.Umrechnung[other.Einheit]
        z /= Laenge.Umrechnung[self.Einheit]
        return Laenge(z, self.Einheit)
```

Das Dictionary `Laenge.Umrechnung` enthält Faktoren, mit denen geläufige Längenmaße in Meter umgerechnet werden. Die Methoden `__add__` und `__sub__` überladen jeweils den Operator für Addition + bzw. den für Subtraktion -, indem

sie zuerst die Zahlenwerte beider Operanden gemäß ihrer Einheiten in Meter umwandeln, verrechnen und schließlich wieder in die Einheit des weiter links stehenden Operanden konvertieren.

In der nachstehenden Tabelle sind alle binären Operatoren und die entsprechenden Magic Methods aufgelistet:

Operator	Magic Method
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
**	<code>__pow__(self, other[, modulo])</code>
%	<code>__mod__(self, other)</code>
>>	<code>__lshift__(self, other)</code>
<<	<code>__rshift__(self, other)</code>
&	<code>__and__(self, other)</code>
	<code>__or__(self, other)</code>
^	<code>__xor__(self, other)</code>

Tabelle 12.4 Magic Methods für binäre Operatoren

Die Methoden sollten bei erfolgreicher Rechnung das Ergebnis zurückgeben. Ist es nicht möglich, den Operanden `other` zu verarbeiten, sollte `NotImplemented` zurückgegeben werden.

Binäre Operatoren mit umgekehrter Operandenreihenfolge

Wenn Python einen Ausdruck der Form `Operand1 Operator Operand2` wie beispielsweise `2 * "abc"` auswerten soll, wird zuerst versucht, eine passende Methode vom ersten Operanden zu benutzen. Existiert diese nicht oder gibt sie `NotImplemented` zurück, wird versucht, beim zweiten Operanden eine entsprechende Methode zu finden. Dies geschieht aber nur dann, wenn die beiden Operanden nicht vom gleichen Datentyp sind. Allerdings muss der zweite Operand eine spezielle Methode für vertauschte Operanden implementieren.

Die folgende Tabelle listet alle dafür verfügbaren Methodennamen und die entsprechenden Operatoren auf:

Operator	Magic Method
+	<code>__radd__(self, other)</code>
-	<code>__rsub__(self, other)</code>
*	<code>__rmul__(self, other)</code>
/	<code>__rtruediv__(self, other)</code>
//	<code>__rfloordiv__(self, other)</code>
**	<code>__rpow__(self, other[, modulo])</code>
%	<code>__rmod__(self, other)</code>
>>	<code>__rlshift__(self, other)</code>
<<	<code>__rrshift__(self, other)</code>
&	<code>__rand__(self, other)</code>
	<code>__ror__(self, other)</code>
^	<code>__rxor__(self, other)</code>

Tabelle 12.5 Magic Methods für binäre Operatoren des rechten Operanden

Für nicht unterstützte Werte von `other` sollte auch hier `NotImplemented` zurückgegeben werden.

Erweiterte Zuweisungen

Neben den Operatoren selbst können auch die erweiterten Zuweisungen überladen werden. Bei einer erweiterten Zuweisung wird ein Gleichheitszeichen benutzt, dem der jeweilige Operator vorangestellt wird:

```
>>> a = 10
>>> a += 5
>>> a
15
```

Standardmäßig verwendet Python für solche Zuweisungen den Operator selbst, so dass `a += 5` intern wie `a = a + 5` ausgeführt wird. Diese Vorgehensweise hat für komplexe Datentypen wie beispielsweise Listen den Nachteil, dass immer eine komplett neue Liste erzeugt werden muss. Deshalb können Sie gezielt die erweiterten Zuweisungen anpassen, um die Effizienz des Programms zu verbessern.

In der folgenden Tabelle stehen alle Operatoren für erweiterte Zuweisungen und die entsprechenden Methoden:

Operator	Magic Method
+=	<code>__iadd__(self, other)</code>
-=	<code>__isub__(self, other)</code>
*=	<code>__imul__(self, other)</code>
/=	<code>__itruediv__(self, other)</code>
//=	<code>__ifloordiv__(self, other)</code>
**=	<code>__ipow__(self, other[, modulo])</code>
%=	<code>__imod__(self, other)</code>
>>=	<code>__ilshift__(self, other)</code>
<<=	<code>__irshift__(self, other)</code>
&=	<code>__iand__(self, other)</code>
=	<code>__ior__(self, other)</code>
^=	<code>__ixor__(self, other)</code>

Tabelle 12.6 Methoden für die erweiterte Zuweisung

Unäre Operatoren

Mit den folgenden Methoden werden die unären Operatoren überladen. Unäre Operatoren erwarten im Gegensatz zu den binären Operatoren nur einen Parameter. Zu den unären Operatoren zählen die Vorzeichen `+` und `-`, die Built-in Function `abs` zur Bestimmung des absoluten Werts und die Tilde `~`, um das Komplement eines Wertes zu berechnen:

Operator	Magic Method
<code>+</code>	<code>__pos__(self)</code>
<code>-</code>	<code>__neg__(self)</code>
<code>abs</code>	<code>__abs__(self)</code>
<code>~</code>	<code>__invert__(self)</code>

Tabelle 12.7 Magic Methods für die unären Operatoren

Built-in Functions und Umwandlung in andere Typen

Zu guter Letzt gibt es einen Satz von Magic Methods, mit dem bestimmt werden kann, welche Werte bestimmte Built-in Functions für Instanzen der Klasse zurückgeben sollen. Für diesen Zweck existieren die folgenden Methoden:

Built-in	Magic Method
complex	<code>__complex__(self)</code>
int	<code>__int__(self)</code>
long	<code>__long__(self)</code>
float	<code>__float__(self)</code>
oct	<code>__oct__(self)</code>
hex	<code>__hex__(self)</code>

Tabelle 12.8 Methoden zum Festlegen der Built-in-Rückgabewerte

`__index__(self)`

Wenn ein Datentyp außerdem als Index benutzt werden soll, wie er z. B. für das Slicing benötigt wird, muss er die parameterlose Methode `__index__(self)` überschreiben. Der Rückgabewert von `__index__` muss eine Ganzzahl (`int` oder `long`) sein.

`__round__(self, [n])`

Diese Methode soll den Wert der Instanz auf `n` Nachkommastellen genau runden und als `float` zurückgeben. Sie wird von der `round`-Built-in verwendet.

Container emulieren

Mithilfe der folgenden Methoden ist es möglich, eigene sequentielle oder Mapping-Datentypen zu implementieren, die sich genauso wie Listen oder Dictionaries benutzen lassen.

Wenn Sie einen sequentiellen Datentyp entwickeln möchten, müssen alle Indizes `i` der Sequenz Ganzzahlen sein und die Gleichung $0 \leq i < N$ erfüllen. `N` ist dabei als Länge der Sequenz definiert.

Alle Mapping-Datentypen sollten zusätzlich zu den nachfolgend besprochenen Magic Methods weitere Methoden implementieren, die in der nachstehenden Tabelle aufgelistet sind:⁷

⁷ Wenn Ihnen die hier angegebenen Beschreibungen nicht ausführlich genug sind, können Sie sich noch einmal den Abschnitt 8.6.1, »Dictionary – dict – dict«, ansehen.

Methode	Bedeutung
<code>m.keys()</code>	Gibt eine Liste der Schlüssel von <code>m</code> zurück.
<code>m.values()</code>	Gibt eine Liste der Werte von <code>m</code> zurück.
<code>m.items()</code>	Gibt eine Liste der Schlüssel-Wert-Paare von <code>m</code> zurück.
<code>m.has_key(k)</code>	Prüft, ob der Schlüssel <code>k</code> in <code>m</code> existiert.
<code>m.get(k[, d])</code>	Wenn der Schlüssel <code>k</code> in <code>m</code> existiert, wird <code>m[k]</code> zurückgegeben, ansonsten <code>d</code> .
<code>m.clear()</code>	Entfernt alle Elemente aus <code>m</code> .
<code>m.setdefault(k[, x])</code>	Wenn der Schlüssel <code>k</code> in <code>m</code> existiert, wird <code>m[k]</code> zurückgegeben. Gibt es den Schlüssel <code>k</code> nicht in <code>m</code> , wird <code>m[k]</code> auf den Wert <code>x</code> gesetzt und <code>x</code> zurückgegeben.
<code>m.pop(k[, d])</code>	Wenn der Schlüssel <code>k</code> in <code>m</code> existiert, wird <code>m[k]</code> zurückgegeben und danach mit <code>del</code> gelöscht. Gibt es den Schlüssel <code>k</code> nicht in <code>m</code> , so wird <code>d</code> zurückgegeben.
<code>m.popitem()</code>	Gibt ein willkürlich ausgewähltes Schlüssel-Wert-Paar von <code>m</code> zurück und entfernt es anschließend aus <code>m</code> .
<code>m.copy()</code>	Gibt eine Kopie von <code>m</code> zurück.
<code>m.update(b)</code>	Übernimmt alle Schlüssel-Wert-Paare von <code>b</code> in <code>m</code> . Vorhandene Einträge werden dabei überschrieben.

Tabelle 12.9 Nötige Methoden für Mapping-Typen

Nun folgt die Besprechung der Magic Methods für sequentielle und Mapping-Datentypen:

`__len__(self)`

Liefert eine Ganzzahl zurück, die die Länge der Sequenz oder des Mapping-Objekts angibt.

`__getitem__(self, key)`

Wird aufgerufen, wenn mit dem `[]`-Operator ein Element des Containers gelesen wird. Der Parameter `key` gibt dabei den Index des gesuchten Elements an.

Ein sehr einfaches Beispiel gibt immer den Index selbst als Element des Containers zurück:

```
>>> class MeinContainer:
    def __getitem__(self, key):
        return key
>>> obj = MeinContainer()
```



```
>>> obj[10]
10
>>> obj[1337]
1337
```

Anstelle eines Zahlenwertes als Index können Sie ab Python 3.0 auch ein sogenanntes `slice`-Objekt übergeben. Diese `slice`-Objekte dienen dazu, Slicing zu realisieren. Folgende Aufrufe sind dabei gleichwertig:

```
>>> a = "Hallo Welt"
>>> a[1:5]
'allo'
>>> a[slice(1,5)]
'allo'
```

Es handelt sich bei `slice` um einen einfachen Datentyp, dessen Konstruktor die folgende Schnittstelle besitzt:

```
slice(start, stop, [step=None])
```

Dabei beschreiben `start` und `stop` die Grenzen des Bereichs, und der optionale Parameter `step` gibt die Schrittweite an. Objekte vom Typ `slice` haben entsprechende Attribute `start`, `stop`, `step`, mit denen Sie auf die Werte zugreifen. Ausführliches zum Thema Slicing finden Sie in Abschnitt 8.5.3, »Strings – str, bytes«.

Wenn der übergebene Index `key` ungültig ist, sollte `__getitem__` einen `IndexError` produzieren.

`__setitem__(self, key, value)`

Muss das Element mit dem Index `key` auf den Wert `value` setzen.

Diese Methode sollte nur dann implementiert werden, wenn der Datentyp das Verändern und Hinzufügen von Elementen unterstützen soll.

Bei ungültigen `key`-Werten sollte ein `IndexError` erzeugt werden.

`__delitem__(self, key)`

Muss das Element mit dem Index `key` aus dem Container entfernen.

Bei ungültigen `key`-Werten sollte ein `IndexError` erzeugt werden.

`__iter__(self)`

Muss einen Iterator über die Werte des sequentiellen Datentyps bzw. über die Schlüssel des Mapping-Typs zurückgeben. Genaues zu Iteratoren können Sie in Abschnitt 13.5, »Iteratoren«, nachlesen.

`__reversed__(self)`

Muss einen Iterator über die Werte des sequentiellen Datentyps zurückgeben, der die Elemente in umgekehrter Reihenfolge durchläuft. Die Methode `__reversed__` wird von der Built-in Function `reversed` benutzt.

Implementiert eine Klasse die Methode `__reversed__` nicht, benutzt die Built-in `reversed` die Methode `__getitem__` zum umgekehrten Durchlaufen der Sequenz, was in der Regel ineffizienter ist.

`__contains__(self, item)`

Muss einen Wahrheitswert zurückgeben, der angibt, ob der sequentielle Datentyp ein Element mit dem Wert von *item* enthält. Handelt es sich um einen Mapping-Typ, wird geprüft, ob es einen Schlüssel mit dem Wert von *item* gibt.

Diese Methode wird von den Operatoren `in` und `not in` benutzt.

Allerdings ist es nicht notwendig, `__contains__` zu implementieren, wenn bereits `__iter__` für den Typ definiert worden ist. Mit `__contains__` kann der Datentyp nur eine unter Umständen effizientere Prüfung anbieten, da nicht wie bei `__iter__` erst die Elemente der Sequenz durchlaufen werden müssen.

12.4 Objektphilosophie

Seitdem in Python 2.3 Datentypen und Klassen vereinigt wurden, ist Python von Grund auf objektorientiert. Das bedeutet, dass im Prinzip alles, mit dem Sie bei der Arbeit mit Python in Berührung kommen, eine Instanz irgendeiner Klasse ist. Von der einfachen Zahl bis zu den Klassen⁸ selbst hat dabei jedes Objekt seine eigenen Attribute und Methoden.

Insbesondere ist es möglich, von eingebauten Datentypen wie `list` oder `dict` zu erben.

Das folgende Beispiel zeigt eine Subklasse von `list`, die den Durchschnittswert ihrer Elemente berechnen kann. Sollte ein Element einen anderen Datentyp als `int` oder `float` haben, wird es einfach ignoriert:

```
class ListeMitDurchschnitt(list):
    def durchschnitt(self):
        summe, i = 0, 0
```

⁸ Der Datentyp von Klassen-Instanzen sind sogenannte Metaklassen, deren Verwendung in diesem Buch nicht behandelt wird.

```

    for e in self:
        if type(e) in (int, float):
            summe += e
            i += 1
    return summe / i

```

Der Datentyp `ListeMitDurchschnitt` kann nun genau wie der Datentyp `list` verwendet werden:

```

>>> l = ListeMitDurchschnitt((2, 3, 4))
>>> l
[2, 3, 4]
>>> l.append(5)
>>> l.durchschnitt()
3.5

```

Durch die konsequente Objektorientierung werden Python-Programme noch leichter zu entwickeln und wiederzuverwenden.

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

13 Weitere Spracheigenschaften

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

13.1 Exception Handling

Stellen Sie sich einmal ein Programm vor, das über eine vergleichsweise tiefe Aufrufhierarchie verfügt, das heißt, dass Funktionen weitere Unterfunktionen aufrufen, die ihrerseits wieder Funktionen aufrufen. Es ist häufig so, dass die übergeordneten Funktionen nicht korrekt weiterarbeiten können, wenn in einer ihrer Unterfunktionen ein Fehler aufgetreten ist. Es ist also notwendig, die Information, dass ein Fehler aufgetreten ist, durch die Aufrufhierarchie nach oben zu schleusen, damit jede übergeordnete Funktion auf den Fehler reagieren und sich daran anpassen kann.

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

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

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

Bisher haben wir bei einer solchen Ausgabe

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

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

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

13.1.1 Eingebaute Exceptions

In Python existieren eine Reihe von eingebauten Exceptions, zum Beispiel die bereits bekannten Exceptions `SyntaxError`, `NameError` oder `TypeError`. Solche Exceptions werden von Funktionen der Standardbibliothek oder vom Interpreter selbst geworfen. Diese Exceptions sind eingebaut, das bedeutet, dass sie zu jeder Zeit im Quelltext verwendet werden können:

```
>>> NameError
<class 'NameError'>
```

```
>>> SyntaxError
<class 'SyntaxError'>
```

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

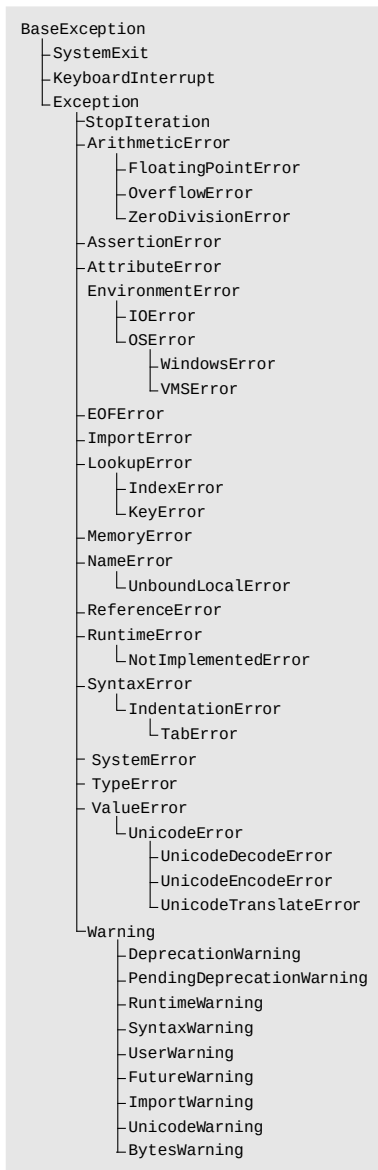


Abbildung 13.1 Vererbungshierarchie der eingebauten Exceptions

BaseException

Die Klasse `BaseException` ist die Basisklasse aller Exceptions und stellt damit eine gewisse Grundfunktionalität bereit, die folglich für alle Exception-Typen vorhanden ist. Aus diesem Grund soll sie hier besprochen werden.

Die Grundfunktionalität, die `BaseException` bereitstellt, besteht lediglich aus einem wesentlichen Attribut namens `args`. Dabei handelt es sich um ein Tupel, in dem alle Parameter abgelegt werden, die der Exception bei ihrer Instantiierung übergeben wurden. Über diese Parameter ist es dann später beim Fangen der Exception möglich, detaillierte Informationen über den aufgetretenen Fehler zu erhalten. Die Verwendung des Attributs `args` demonstriert nun das folgende Beispiel:

```
>>> e = BaseException("Hallo Welt")
>>> e.args
('Hallo Welt',)
>>> e = BaseException("Hallo Welt",1,2,3,4,5)
>>> e.args
('Hallo Welt', 1, 2, 3, 4, 5)
```

Soweit zunächst zur direkten Verwendung der Exception-Klassen. Eine Erklärung aller eingebauten Exception-Klassen finden Sie im Anhang.

13.1.2 Werfen einer Exception

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

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

Dazu wird das Schlüsselwort `raise`, gefolgt von einer Instanz, geschrieben. Diese darf nur Instanz einer selbst erstellten Klasse oder eines vordefinierten Exception-Typs sein. Das Werfen von Instanzen anderer Datentypen, insbesondere von Strings, ist nicht möglich:

```
>>> raise "Hallo Welt"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exceptions must derive from BaseException
```

Im folgenden Abschnitt möchten wir besprechen, wie Exceptions im Programm abgefangen werden können, so dass sie nicht in einem Traceback enden, sondern

zur Ausnahmebehandlung eingesetzt werden können. Beachten Sie, dass wir sowohl in diesem als auch im nächsten Abschnitt bei den eingebauten Exceptions bleiben. Selbstdefinierte Exceptions werden das Thema von Abschnitt 13.1.4, »Eigene Exceptions«, sein.

13.1.3 Abfangen einer Exception

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

Nach Ihrem bisherigen Kenntnisstand würde die Funktion folgendermaßen aussehen:

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

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

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

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

- ▶ Der `try`-Block wird durch das Schlüsselwort `try` eingeleitet, gefolgt von einem Doppelpunkt und, um eine Ebene weiter eingerückt, einem beliebigen Codeblock. Dieser Codeblock wird zunächst ausgeführt. Wenn in diesem Codeblock eine Exception auftritt, wird seine Ausführung sofort beendet und der `except`-Zweig der Anweisung ausgeführt.
- ▶ Der `except`-Zweig wird durch das Schlüsselwort `except` eingeleitet, gefolgt von einer optionalen Liste von Exception-Typen, für die dieser `except`-Zweig ausgeführt werden soll. Beachten Sie, dass mehrere Exception-Typen in Form

eines Tupels angegeben werden müssen. Dazu werden Sie später noch ein Beispiel sehen. Hinter der Liste der Exception-Typen kann, ebenfalls optional, das Schlüsselwort `as` gefolgt von einem frei wählbaren Bezeichner stehen. Hier legen Sie fest, unter welchem Namen Sie die gefangene Exception-Instanz im `except`-Zweig referenzieren können. Auf diesem Weg können Sie beispielsweise auf die in dem `args`-Attribut der Exception-Instanz abgelegten Informationen zugreifen. Auch dazu werden wir im Verlauf dieses Kapitels noch Beispiele sehen.

Die Angabe von `as e` im obigen Code ist für dieses Beispiel nicht relevant und wurde nur der Vollständigkeit halber hinzugefügt.

Danach folgt ein Doppelpunkt und, um eine Ebene weiter eingerückt, ein beliebiger Codeblock. Dieser Codeblock wird nur dann ausgeführt, wenn innerhalb des `try`-Blocks eine der aufgelisteten Exceptions geworfen wurde.

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

```
try:
    Anweisung
    :
    Anweisung
except Exceptiontyp as Bezeichner:
    Anweisung
    :
    Anweisung
```

Abbildung 13.2 Struktur einer `try/except`-Anweisung

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

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

Die Funktion soll nun dahingehend erweitert werden, dass auch ein `TypeError` abgefangen und dann ebenfalls `None` zurückgegeben wird. Dazu haben wir im Wesentlichen drei Möglichkeiten. Die erste wäre es, die Liste der abzufangenden Exception-Typen im vorhandenen `except`-Zweig um den `TypeError` zu erwei-

tern. Beachten Sie dabei, dass zwei oder mehr Exception-Typen im Kopf eines `except`-Zweiges als Tupel angegeben werden müssen.

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

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

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

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

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

Hinweis

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

Eine Exception ist nichts anderes als eine Instanz einer bestimmten Klasse. Von Darum stellt sich die Frage, ob und wie man innerhalb eines `except`-Zweiges Zugriff auf die geworfene Instanz erlangt. Das ist durch Angabe des bereits angesprochenen `as` Bezeichner-Teils im Kopf des `except`-Zweigs möglich. Unter dem dort angegebenen Namen können wir nun innerhalb des Codeblocks auf die geworfene Exception-Instanz zugreifen. Dies könnte folgendermaßen aussehen:

```
try:
    print([1,2,3][10])
```

```
except (IndexError, TypeError) as e:
    print("Fehlermeldung:", e.args[0])
```

Die Ausgabe des obigen Beispiels lautet:

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

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

Beachten Sie, dass sowohl der `else`- als auch der `finally`-Zweig ans Ende der `try/except`-Anweisung geschrieben werden müssen. Wenn beide Zweige vorkommen, muss der `else`-Zweig vor dem `finally`-Zweig stehen. Abbildung 13.3 zeigt eine vollständige `try/except`-Anweisung.

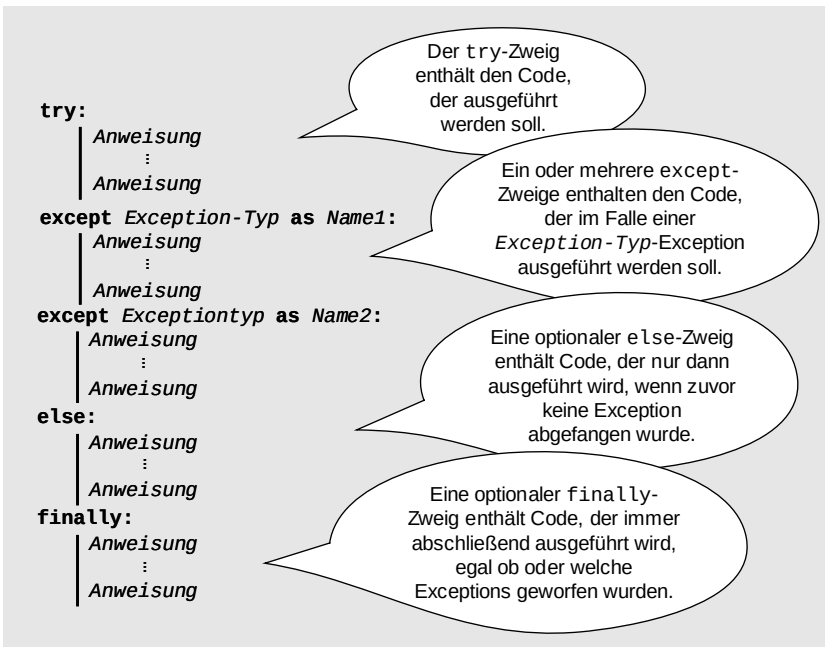


Abbildung 13.3 Eine vollständige `try/except`-Anweisung

Abschließend noch einige Bemerkungen dazu, wie eine `try/except`-Anweisung ausgeführt wird. Zunächst wird der dem `try`-Zweig zugehörige Code ausgeführt. Sollte innerhalb dieses Codes eine Exception geworfen werden, so wird der dem entsprechenden `except`-Zweig zugehörige Code ausgeführt. Ist kein passender `except`-Zweig vorhanden, so wird die Exception nicht abgefangen und endet, wenn sie auch anderswo nicht abgefangen wird, als `Traceback` auf dem Bildschirm. Sollte im `try`-Zweig keine Exception geworfen werden, so wird keiner der `except`-Zweige ausgeführt, sondern zunächst der `else`- und dann der `finally`-Zweig, wobei beide Zweige optional sind. Beachten Sie, dass der `finally`-Zweig in jedem Fall, also auch wenn Exceptions aufgetreten sind, zum Schluss ausgeführt wird.

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

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

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

13.1.4 Eigene Exceptions

Beim Werfen und Abfangen von Exceptions sind Sie nicht auf den eingebauten Satz von Exception-Typen beschränkt, vielmehr können Sie selbst beliebige neue Typen erstellen. Dazu brauchen Sie lediglich eine eigene Klasse zu erstellen, die von der Exception-Basisklasse `Exception` erbt, und dann ganz nach Anforderung weitere Attribute und Methoden zum Umgang mit Ihrer persönlichen Exception hinzufügen. Die folgende Beispielfunktion soll zwei ganze Zahlen dividieren und im Falle einer Division durch null keinen `ZeroDivisionError`, sondern einen

eigenen Exception-Typ mit weiteren Informationen werfen. Dazu definieren wir zunächst eine von `Exception` abgeleitete Klasse und fügen ein Attribut für den Zähler der Division hinzu:

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

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

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

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

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in division
__main__.DivisionByZeroError
```

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

```
try:
    division(12, 0)
except DivisionByZeroError, e:
    print("Nulldivision: {0} / 0".format(e.zaehler))
```

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

```
Nulldivision: 12 / 0
```

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

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

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

Ein Traceback, der durch diese Exception verursacht wird, sähe folgendermaßen aus:

```
>>> division(12, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in division
__main__.DivisionByZeroError: Division durch null
```

13.1.5 Erneutes Werfen einer Exception

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

```
def funktion3():
    raise TypeError

def funktion2():
    funktion3()

def funktion1():
    funktion2()

funktion1()
```

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

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

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

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

```
def funktion3():
    raise TypeError

def funktion2():
    funktion3()

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

funktion1()
```

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

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

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

```
def funktion3():
    raise
```

```
def funktion2():
    funktion3()

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

funktion1()
```

Der in diesem Beispiel ausgegebene Traceback sieht folgendermaßen aus:

```
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    funktion1()
  File "test.py", line 11, in funktion1
    funktion2()
  File "test.py", line 7, in funktion2
    funktion3()
  File "test.py", line 4, in funktion3
    raise TypeError
TypeError
```

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

13.1.6 Exception Chaining

Gelegentlich kommt es vor, dass man innerhalb eines `except`-Zweiges in die Verlegenheit kommt, eine weitere Exception zu werfen. Das Problem dabei ist, dass die ursprünglich in diesem `except`-Zweig gefangene Exception verlorengeht. Das ist problematisch, da möglicherweise genau das Auftreten dieser Exception dazu beigetragen hat, dass die zweite Exception geworfen werden musste. Diese verwirrende Situation soll anhand eines Beispiels geklärt werden:

```
try:
    [1,2,3][128]
except IndexError as e:
    raise RuntimeError("Schlimmer Fehler") from e
```

Im `try`-Zweig wird versucht, auf das 128-te Element einer 3-elementigen Liste zuzugreifen, was eine `IndexError`-Exception provoziert. Diese wird im `except`-

Zweig gefangen und zusätzlich eine `RuntimeError`-Exception mit einer ausdrucksvollen Fehlermeldung geworfen. Dieser `RuntimeError`-Exception wird dabei die zuvor gefangene `IndexError`-Exception angehängt, was sich auch am entstehenden Traceback ablesen lässt:

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    [1,2,3][128]
IndexError: list index out of range
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    raise RuntimeError("Schlimmer Fehler") from e
RuntimeError: Schlimmer Fehler
```

Beachten Sie, dass es sich bei der endgültigen Exception um eine `RuntimeError`-Exception handelt. Sie kann nicht von einem `except`-Zweig gefangen werden, der `IndexError`-Exceptions behandelt.

Die Verwendung der `raise ... from`-Syntax war in obigem Beispiel eigentlich nicht notwendig, da Python in solchen Fällen die vorangegangene Exception bereits implizit an die neu geworfene Exception anhängt. Dennoch zeigt sich hier eine flexible und interessante Möglichkeit, um eine beliebige zweite Exception an die zu werfende, eigentliche Exception anzuhängen.

Abschließend sei gesagt, dass die hier vorgestellten Techniken zum Exception Handling ungemein beim Schreiben von strukturiertem und lesbarem Code helfen, so dass Sie sie verinnerlichen sollten. Wir werden auch im Laufe dieses Buches immer wieder Exceptions verwenden.

13.2 Comprehensions

In diesem Abschnitt möchten wir uns auf ein interessantes Feature von Python stürzen, die sogenannten *Comprehensions*. Das sind spezielle Anweisungen, mit denen Sie eine neue Liste bzw. ein neues Dictionary oder Set mit generischem Inhalt erzeugen. Das bedeutet, Sie geben eine Erzeugungsvorschrift an, nach der die jeweilige Instanz mit Werten gefüllt wird.

Während List Comprehensions bereits seit längerem in Python existieren, sind Dict Comprehensions und Set Comprehensions ein Novum von Python 3.0.

13.2.1 List Comprehensions

Es ist ein häufig auftretendes Problem, dass man aus den Elementen einer bestehenden Liste eine neue Liste erstellen möchte, deren Elemente aus denen der alten Liste berechnet wurden. Bislang würden Sie dies entweder sehr umständlich in einer `for`-Schleife erledigen oder die Built-in Functions `map` und `filter` einsetzen. Letzteres ist zwar relativ kurz, bedarf jedoch einer Funktion, die auf jedes Element der Liste angewandt wird. Das ist umständlich und ineffizient.

Python unterstützt eine sehr viel flexiblere Syntax, die für gerade diesen Zweck geschaffen wurde: die sogenannten *List Comprehensions*. Die folgende List Comprehension erzeugt aus einer Liste mit ganzen Zahlen eine neue Liste, die die Quadrate dieser Zahlen enthält:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> [x**2 for x in lst]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Eine List Comprehension wird in eckige Klammern gefasst und besteht zunächst aus einem Ausdruck, gefolgt von beliebig vielen `for/in`-Bereichen. Ein `for/in`-Bereich lehnt sich an die Syntax der `for`-Schleife an und gibt an, mit welchem Bezeichner über welche Liste iteriert wird – in diesem Fall mit dem Bezeichner `x` über die Liste `lst`. Der angegebene Bezeichner kann im Ausdruck zu Beginn der List Comprehension verwendet werden. Das Ergebnis einer List Comprehension ist eine neue Liste, die als Elemente die Ergebnisse des Ausdrucks in jedem Iterationsschritt enthält. Die Funktionsweise der obigen List Comprehension lässt sich folgendermaßen zusammenfassen:

Für jedes Element `x` der Liste `lst` bilde das Quadrat von `x`, und füge das Ergebnis in die Ergebnisliste ein.

Dies ist die einfachste Form der List Comprehension. Der `for/in`-Bereich lässt sich um eine Fallunterscheidung erweitern, so dass nur bestimmte Elemente in die neue Liste übernommen werden. So könnten wir die obige List Comprehension beispielsweise dahingehend erweitern, dass nur die Quadrate gerader Zahlen gebildet werden:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> [x**2 for x in lst if x%2 == 0]
[4, 16, 36, 64]
```

Dazu wird der `for/in`-Bereich um das Schlüsselwort `if` erweitert, auf das eine Bedingung folgt. Nur wenn diese Bedingung `True` ergibt, wird das berechnete Element in die Ergebnisliste aufgenommen.

Diese Form der List Comprehension lässt sich also folgendermaßen beschreiben:

Für jedes Element x der Liste `lst` – sofern es sich bei x um eine gerade Zahl handelt – bilde das Quadrat von x , und füge das Ergebnis in die Ergebnisliste ein.

Als nächstes Beispiel soll eine List Comprehension dazu verwendet werden, zwei als Listen dargestellte Vektoren zu addieren. Die Addition zweier Vektoren erfolgt koordinatenweise, also in unserem Fall Element für Element:

```
>>> v1 = [1, 7, -5]
>>> v2 = [-9, 3, 12]
>>> [v1[i] + v2[i] for i in range(3)]
[-8, 10, 7]
```

Dazu wird eine von `range` erzeugte Liste von Indizes in der List Comprehension durchlaufen. In jedem Durchlauf werden die jeweiligen Koordinaten addiert und an die Ergebnisliste angehängt.

Es wurde bereits gesagt, dass eine List Comprehension beliebig viele `for/in`-Bereiche haben kann. Diese können wie verschachtelte `for`-Schleifen betrachtet werden. Im Folgenden möchten wir ein Beispiel besprechen, in dem diese Eigenschaft von Nutzen ist. Zunächst definieren wir zwei Listen:

```
>>> lst1 = ["A", "B", "C"]
>>> lst2 = ["D", "E", "F"]
```

Eine List Comprehension soll nun eine Liste erstellen, die alle möglichen Buchstabenkombinationen enthält, die gebildet werden können, indem man zunächst einen Buchstaben aus `lst1` und dann einen aus `lst2` wählt. Die Kombinationen sollen jeweils als Tupel in der Liste stehen:

```
>>> [(a,b) for a in lst1 for b in lst2]
[('A', 'D'), ('A', 'E'), ('A', 'F'), ('B', 'D'), ('B', 'E'),
 ('B', 'F'), ('C', 'D'), ('C', 'E'), ('C', 'F')]
```

Diese List Comprehension kann folgendermaßen beschrieben werden:

Für jedes Element a der Liste `lst1` gehe über alle Elemente b von `lst2`, und füge jeweils das Tupel (a, b) in die Ergebnisliste ein.

List Comprehensions bieten einen interessanten und eleganten Weg, sehr komplexe Operationen platzsparend zu schreiben. Besonders möchten wir noch einmal auf die Effizienz von List Comprehensions hinweisen. So kann eine List Comprehension stets schneller ausgeführt werden als beispielsweise eine äquivalente `for`-Schleife.

Viele Probleme, bei denen List Comprehensions zum Einsatz kommen, könnten auch durch die Built-in Functions `map`, `filter` oder durch eine Kombination der beiden gelöst werden, jedoch sind List Comprehensions zumeist besser lesbar und führen zu einem übersichtlicheren Quellcode.

13.2.2 Dict Comprehensions

Seit Version 3.0 bietet Python einen zu den List Comprehensions analogen Weg an, um ein Dictionary zu erzeugen. Dies nennt sich dann eine *Dictionary Comprehension* bzw. kurz *Dict Comprehension*.

Der Aufbau einer Dict Comprehension ist ähnlich wie der einer List Comprehension, weswegen wir direkt mit einem Beispiel einsteigen:

```
>>> lst = ["Donald", "Dagobert", "Daisy"]
>>> {k:len(k) for k in lst}
{'Donald': 6, 'Dagobert': 8, 'Daisy': 5}
```

Hier wurde mithilfe einer Dict Comprehension ein Dictionary erzeugt, das eine vorgegebene Liste von Strings als Schlüssel und die Längen des jeweiligen Schlüsselstrings als Wert enthält.

Beim Betrachten des Beispiels fallen sofort zwei Unterschiede zu den List Comprehensions auf:

- ▶ Im Gegensatz zu einer List Comprehension wird eine Dict Comprehension in geschweifte Klammern gefasst.
- ▶ Bei einer Dict Comprehension muss in jedem Durchlauf der Schleife ein Schlüssel-Wert-Paar zum Dictionary hinzugefügt werden. Dieses steht am Anfang der Comprehension, wobei Schlüssel und Wert durch einen Doppelpunkt voneinander getrennt sind.

Sonst können Sie eine Dict Comprehension verwenden, wie Sie es bereits von List Comprehensions her kennen. Beide Typen lassen sich sogar gemeinsam nutzen. Dazu noch ein Beispiel:

```
>>> lst1 = ["A", "B", "C"]
>>> lst2 = [2, 4, 6]
>>> {k:[k*i for i in lst2] for k in lst1}
{'A': ['AA', 'AAAA', 'AAAAAA'],
 'C': ['CC', 'CCCC', 'CCCCCC'],
 'B': ['BB', 'BBBB', 'BBBBBB']}
```

Dieser Code erzeugt ein Dictionary, das zu jedem Schlüssel mithilfe einer List Comprehension eine Liste als Wert erzeugt, die jeweils das Zwei-, Vier- und Sechsfache des Schlüssels enthält.

13.2.3 Set Comprehensions

Der dritte wichtige Datentyp, für den ebenfalls eine Comprehension-Syntax existiert, ist das Set. Eine *Set Comprehension* wird, wie eine Dict Comprehension, in geschweifte Klammern eingefasst. Im Gegensatz zur Dict Comprehension fehlen allerdings der Doppelpunkt und der dahinter angegebene Wert:

```
>>> lst = [1,2,3,4,5,6,7,8,9]
>>> {i**2 for i in lst}
{64, 1, 36, 81, 9, 16, 49, 25, 4}
```

Eine Set Comprehension funktioniert also, abgesehen von den geschweiften Klammern, völlig analog zur List Comprehension. Es bedarf also keiner weiteren Beispiele, um sie erfolgreich einzusetzen.

13.3 Docstrings

In Abschnitt 5.3, »Kommentare«, wurde der sogenannte Blockkommentar eingeführt. Ein Blockkommentar wird folgendermaßen geschrieben:

```
"""
Dies ist ein Blockkommentar.
Er kann mehrere Zeilen umfassen.
"""
```

Der Name Blockkommentar wird den Möglichkeiten, die diese Notation bietet, jedoch nicht ganz gerecht. In der Python-Terminologie wird ein in drei doppelte oder einfache Hochkommata eingefasster Text *Docstring* genannt, kurz für »Documentation String«.

Docstrings sind dazu gedacht, Funktionen, Module oder Klassen zu beschreiben. Diese Beschreibungen können durch externe Tools oder beispielsweise die Built-in Function `help` gelesen und wiedergegeben werden. Auf diese Weise lassen sich sehr einfach Dokumentationen aus den – eigentlich programminternen – Kommentaren erzeugen.

Die folgenden beiden Beispiele zeigen eine Klasse und eine Funktion jeweils mit einem Docstring dokumentiert. Beachten Sie, dass ein Docstring immer am Anfang des Funktions- bzw. Klassenkörpers stehen muss, um als Docstring erkannt zu werden. Ein Docstring kann durchaus auch an anderen Stellen stehen, kann dann jedoch keiner Klasse oder Funktion zugeordnet werden und fungiert somit nur als Blockkommentar.

```

class MeineKlasse:
    """Beispiel fuer Docstrings.

    Diese Klasse zeigt, wie Docstrings verwendet
    werden.
    """
    pass

def MeineFunktion():
    """Diese Funktion macht nichts.

    Im Ernst, diese Funktion macht wirklich nichts.
    """
    pass

```

Um den Docstring programmintern verwenden zu können, besitzt jede Instanz ein Attribut namens `__doc__`, das ihren Docstring enthält. Beachten Sie, dass auch Funktionsobjekte und eingebundene Module Instanzen sind:

```

>>> print(MeineKlasse.__doc__)
Beispiel fuer Docstrings.
    Diese Klasse zeigt, wie Docstrings verwendet
    werden.

>>> print(MeineFunktion.__doc__)
Diese Funktion macht nichts.
    Im Ernst, diese Funktion macht wirklich nichts.

```

Auch ein Modul kann durch einen Docstring kommentiert werden. Der Docstring eines Moduls muss zu Beginn der entsprechenden Programmdatei stehen und ist ebenfalls über das Attribut `__doc__` erreichbar. Beispielsweise kann folgendermaßen der Docstring des Moduls `math` der Standardbibliothek ausgelesen werden:

```

>>> import math
>>> math.__doc__
'This module is always available. It provides access to the\
nmathematical functions defined by the C standard.'

```

Sobald Sie damit anfangen, größere Programme in Python zu realisieren, sollten Sie Funktionen, Methoden, Klassen und Module mit Docstrings versehen. Das hilft nicht nur beim Programmieren selbst, sondern auch beim späteren Erstellen einer Programmdokumentation.

13.4 Generatoren

In diesem Abschnitt werden wir uns mit dem Konzept der Generatoren beschäftigen, die eine komfortable Möglichkeit anbieten, Reihen von Werten zu verarbeiten. Weil sich das noch sehr abstrakt anhört, wollen wir direkt mit einem Beispiel beginnen. Sie erinnern sich sicherlich noch an die Built-in Function `range`, die im Zusammenhang mit `for`-Schleifen eine wichtige Rolle spielt:

```
>>> for i in range(10):
        print(i, end=" ")
0 1 2 3 4 5 6 7 8 9
```

Wie wir bereits wissen, gibt `range(10)` ein iterierbares Objekt zurück, mit dem sich die Zahlen 0 bis 9 in der Schleife durchlaufen lassen. Sie haben bereits gelernt, dass `range` dafür keine Liste mit diesen Zahlen erzeugt, sondern sie erst bei Bedarf generiert. Es kommt sehr häufig vor, dass man eine Liste von Objekten mit einer Schleife verarbeiten möchte, ohne dass dabei die gesamte Liste als solche im Speicher liegen muss. Für das obige Beispiel bedeutet dies, dass wir zwar die Zahlen von 0 bis 9 verarbeiten, die Liste `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` aber zu keiner Zeit benötigen.

Dieses Prinzip möchte man nun verallgemeinern, um beliebige Sequenzen von Objekten, die nicht alle zusammen im Speicher stehen müssen, mithilfe von Schleifen durchlaufen zu können. Beispielsweise möchten wir gerne über die ersten n Quadratzahlen iterieren.

An dieser Stelle kommen die sogenannten *Generatoren* ins Spiel. Ein Generator ist eine Funktion, die bei jedem Aufruf das nächste Element einer virtuellen Sequenz zurückgibt. Für unser Beispiel bräuchten wir also einen Generator, der nacheinander die ersten n Quadratzahlen zurückgibt. Die Definition dieser auch *Generatorfunktionen* genannten Konstrukte ist der von normalen Funktionen sehr ähnlich. Der von uns benötigte Generator, wir nennen ihn `square_generator`, lässt sich folgendermaßen implementieren (wundern Sie sich bitte nicht über das `yield`, es wird im Anschluss erklärt):

```
def square_generator(n):
    i = 1
    while i <= n:
        yield i*i
        i += 1
```

Mit diesem Generator können wir nun auf sehr elegante Weise die ersten 10 Quadratzahlen auf dem Bildschirm ausgeben:

```
>>> for i in square_generator(10):
    print(i, end=" ")
1 4 9 16 25 36 49 64 81 100
```

Der Funktionsaufruf `square_generator(10)` gibt ein iterierbares Objekt (die generator-Instanz) zurück, das mit einer `for`-Schleife durchlaufen werden kann.

Der Knackpunkt bei Generatoren liegt in dem `yield`-Statement, mit dem wir die einzelnen Werte der virtuellen Sequenz zurückgeben. Die Syntax von `yield` unterscheidet sich dabei nicht von der des `return`-Statements und muss deshalb nicht weiter erläutert werden. Entscheidend ist, wie `yield` sich im Vergleich zu `return` auf die Verarbeitung des Programms auswirkt.

Wird in einer normalen Funktion während eines Programmlaufs ein `return` erreicht, wird der Kontrollfluss an die nächsthöhere Ebene zurückgegeben und der Funktionslauf beendet. Außerdem werden alle lokalen Variablen der Funktion wieder freigegeben. Bei einem erneuten Aufruf der Funktion würde Python wieder ganz am Anfang der Funktion beginnen und die komplette Funktion erneut ausführen.

Im Gegensatz dazu werden beim Erreichen einer `yield`-Anweisung die aktuelle Position innerhalb der Generatorfunktion und ihre lokalen Variablen gespeichert, und es erfolgt ein Rücksprung in das aufrufende Programm mit dem hinter `yield` angegebenen Wert. Beim nächsten Iterationsschritt macht Python dann hinter dem zuletzt ausgeführten `yield` weiter und kann wieder auf die alten lokalen Variablen, in dem Fall `i` und `n`, zugreifen. Erst wenn das Ende der Funktion erreicht wird, beginnen die endgültigen Aufräumarbeiten.

Generatoren sind sehr flexibel und können durchaus mehrere `yield`-Anweisungen enthalten:

```
def generator_mit_mehreren_yields():
    a = 10
    yield a
    yield a*2
    b = 5
    yield a+b
```

Auch dieser Generator kann mit einer `for`-Schleife durchlaufen werden:

```
>>> for i in generator_mit_mehreren_yields():
    print(i, end=" ")
10 20 15
```

Im ersten Iterationsschritt wird die lokale Variable `a` in der Generatorfunktion angelegt und ihr Wert dann mit `yield a` an die Schleife übergeben. Beim nächsten

Schleifendurchlauf wird dann bei `yield a*2` weitergemacht, wobei die zurückgegebene 20 zeigt, dass der Wert von `a` tatsächlich zwischen den Aufrufen erhalten geblieben ist. Während des letzten Iterationsschritts erzeugen wir zusätzlich die lokale Variable `b` mit dem Wert 5 und geben die Summe von `a` und `b` an die Schleife weiter, wodurch die 15 ausgegeben wird. Da nun das Ende der Generatorfunktion erreicht ist, bricht die Schleife nach drei Durchläufen ab.

Es ist auch möglich, eine Generatorfunktion frühzeitig zu verlassen, wenn dies erforderlich sein sollte. Um dies zu erreichen, benutzt man das `return`-Statement ohne Rückgabewert. Der folgende Generator erzeugt abhängig vom Wert des optionalen Parameters `auch_jungen` eine Folge aus zwei Mädchennamen oder zwei Mädchen- und Jungennamen:

```
def namen(auch_jungen=True):
    yield "Meggi"
    yield "Katharina"
    if not auch_jungen:
        return
    yield "Florian"
    yield "Ramin"
```

Mithilfe der Built-in Function `list` können wir aus den Werten des Generators eine Liste erstellen, die entweder nur "Sonja" und "Lisa" oder zusätzlich "Florian" und "Jan" enthält:

```
>>> list(namen())
['Meggi', 'Katharina', 'Florian', 'Ramin']
>>> list(namen(False))
['Meggi', 'Katharina']
```

Generator Expressions

Sie erinnern sich sicherlich noch an die sogenannten List Comprehensions, mit denen Sie auf einfache Weise Listen erzeugen konnten. Mit solchen List Comprehensions konnten Sie beispielsweise eine Liste mit den ersten zehn Quadratzahlen generieren:

```
>>> [i*i for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Wenn wir nun die Summe dieser ersten zehn Quadratzahlen bestimmen wollten, könnten wir das mithilfe der Built-in Function `sum` erreichen, indem wir schreiben:

```
>>> sum([i*i for i in range(1, 11)])
385
```

So weit, so gut. Allerdings wurde hier eine nicht benötigte `list`-Instanz erzeugt, die Speicherplatz vergeudet.

Um auch in solchen Fällen nicht auf den Komfort von List Comprehensions verzichten zu müssen, wurden sogenannte *Generator Expressions* eingeführt. Generator Expressions sehen genauso aus wie die entsprechenden List Comprehensions, mit der Ausnahme, dass statt der eckigen Klammern `[]` die runden Klammern `()` als Begrenzung verwendet werden. Damit können wir das obige Beispiel speicherschonend mit einer Generator Expression formulieren:

```
>>> sum((i*i for i in range(1, 11)))
385
```

Die umschließenden runden Klammern können entfallen, wenn der Ausdruck sowieso schon geklammert ist. In unserem `sum`-Beispiel können wir also ein Klammerpaar entfernen:

```
>>> sum(i*i for i in range(1, 11))
385
```

Generatoren können Ihnen helfen, Ihre Programme sowohl in der Lesbarkeit als auch hinsichtlich der Ausführungsgeschwindigkeit zu verbessern. Immer dann, wenn Sie es mit einer komplizierten und dadurch schlecht lesbaren `while`-Schleife zu tun haben, sollten Sie prüfen, ob ein Generator die Aufgabe nicht eleganter übernehmen kann.

Wir haben uns in diesem Abschnitt auf die Definition von Generatoren und ihre Anwendung in der `for`-Schleife oder mit `list` beschränkt. Im folgenden Abschnitt werden Sie die Hintergründe und die technische Umsetzung kennenlernen, denn hinter den Generatoren und der `for`-Schleife steht das Konzept der Iteratoren.

13.5 Iteratoren

Sie sind bei der Lektüre dieses Buchs schon oft mit dem Begriff »iterierbares Objekt« konfrontiert worden, wobei Ihnen bisher nur gesagt wurde, dass Sie solche Instanzen beispielsweise mit einer `for`-Schleife durchlaufen oder bestimmten Funktionen, wie `list`, als Parameter übergeben konnten. In diesem Abschnitt werden wir uns nun endlich mit den Hintergründen und Funktionsweisen dieser Objekte befassen.

Ein sogenannter *Iterator* ist eine Abstraktionsschicht, die es ermöglicht, auf die Elemente einer Sequenz über eine standardisierte Schnittstelle zuzugreifen.

Bisher mussten Sie für den Zugriff auf die Elemente einer Sequenz oder eines Dictionaries immer eine Referenz auf den Container, also die `list`- oder `dict`-Instanz, sowie den Index des jeweiligen Elements benutzen. Dies hatte den Nachteil, dass Sie dafür immer die Art der Indizes kennen mussten, die die Datenstruktur anbot, weshalb Sie den Code für jeden Datentyp anpassen mussten. Nun ist aber insbesondere das Durchlaufen aller Elemente einer Sequenz oder eines anderen Objekts, das mehrere Elemente speichert, eine Operation, die unabhängig von dem jeweiligen Datentyp immer auf das Gleiche hinausläuft. Um beispielsweise alle Elemente einer Sequenz auszugeben, benötigen Sie nacheinander Zugriff auf die Elemente, wobei egal ist, ob dieser nun über numerische Indizes oder irgendeine andere Art von Schlüsseln bereitgestellt wird.

Deshalb wurden Iteratoren eingeführt, mit denen der jeweilige Datentyp sich selbst um die Bereitstellung der Elemente kümmert und die konkrete Implementation hinter einer einheitlichen Schnittstelle versteckt. Die dazu festgelegte Schnittstelle heißt *Iterator-Protokoll* und ist folgendermaßen definiert:

Jede iterierbare Instanz muss eine parameterlose `__iter__`-Methode implementieren, die ein *Iterator-Objekt* zurückgibt. Das Iterator-Objekt muss ebenfalls eine `__iter__`-Methode besitzen, die einfach eine Referenz auf das Objekt selbst zurückgibt. Außerdem muss es eine `__next__`-Methode aufweisen, die bei jedem Aufruf das nächste Element des zu durchlaufenden Containers zurückgibt. Ist das Ende der Iteration erreicht, muss die `__next__`-Methode die `StopIteration`-Exception mittels `raise` werfen.

Um die Iteration starten zu können, muss über die Built-in Function `iter` eine Referenz auf den Iterator ermittelt werden. Die Anweisung `iter(Objekt)` ruft dabei die `__iter__`-Methode der Instanz `Objekt` auf und reicht das Ergebnis als Rückgabewert an die aufrufende Ebene weiter. Von der zurückgegebenen Iterator-Instanz kann dann so lange die `__next__`-Methode aufgerufen werden, bis diese die `StopIteration`-Exception wirft.

Um mehr Licht in diese abstrakte Beschreibung zu bringen, werden wir eine Klasse entwickeln, die uns über die Fibonacci-Folge iterieren lässt. Die Fibonacci-Folge ist eine Folge aus ganzen Zahlen, wobei jedes Element $f(n)$ durch die Summe seiner beiden Vorgänger $f(n-2) + f(n-1)$ berechnet werden kann. Die beiden ersten Elemente werden per Definition auf $f(1) = f(2) = 1$ gesetzt. Der Anfang der unendlichen Folge ist in der nachstehenden Tabelle gezeigt:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
f(n)	1	1	2	3	5	8	13	21	34	55	89	144	233	377

Tabelle 13.1 Die ersten 14 Elemente der Fibonacci-Folge

Die Folge kann unter anderem dazu verwendet werden, die idealisierte Entwicklung von Kaninchenpopulationen zu berechnen. Außerdem konvergiert der Quotient von aufeinanderfolgenden Elementen für große n gegen den Goldenen Schnitt ($\Phi = 1,618\dots$), einem Verhältnis, das sich sehr oft in der Natur findet.

```
class Fibonacci:
    def __init__(self, max_n):
        self.MaxN = max_n
        self.N = 0
        self.A = 0
        self.B = 0

    def __iter__(self):
        self.N = 0
        self.A = 0
        self.B = 1
        return self

    def __next__(self):
        if self.N < self.MaxN:
            self.N += 1
            self.A, self.B = self.B, self.A + self.B
            return self.A
        else:
            raise StopIteration
```

Unsere Klasse `Fibonacci` erwartet als Parameter für ihren Konstruktor die Nummer des Elements, nach dem die Iteration stoppen soll. Diese Nummer speichern wir in dem Attribut `MaxN` und zählen dann mit dem Attribut `N`, wie viele Elemente bereits zurückgegeben wurden. Um uns zwischen den `__next__`-Aufrufen die aktuelle Position in der Folge zu merken und um das nächste Element berechnen zu können, speichern wir das zuletzt zurückgegebene Element und seinen Nachfolger in den Attributen `A` und `B` der `Fibonacci`-Klasse. Wir werden keine separate Iterator-Klasse definieren und lassen deshalb die `__iter__`-Methode eine Referenz auf die `Fibonacci`-Instanz selbst, also `self`, zurückgeben. Außerdem müssen beim Beginn des Durchlaufens die Speicher für das letzte nächste Element mit ihren Anfangswerten 0 bzw. 1 belegt und muss der `N`-Zähler auf 0 gesetzt werden. Die `__next__`-Methode kümmert sich um die Berechnung des aktuellen Elements der Folge und aktualisiert die Zwischenspeicher und den Zähler. Ist das Ende der gewünschten Teilfolge erreicht, wird `StopIteration` geworfen.

Die Klasse lässt sich nun mit allen Konstrukten verarbeiten, die das Iterator-Protokoll unterstützen, wie beispielsweise die `for`-Schleife und die Built-in Functions `list` oder `sum`:

```
>>> for f in Fibonacci(14):
    print(f, end=" ")
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> list(Fibonacci(16))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
>>> sum(Fibonacci(60))
4052739537880
```

Mit einer kleinen Subklasse von `Fibonacci` können wir auch einen Iterator erzeugen, der uns die Verhältnisse zweier aufeinanderfolgender Fibonacci-Zahlen durchlaufen lässt. Dabei sieht man sehr schnell, dass sich die Quotienten dem Goldenen Schnitt nähern. Die Subklasse muss nur die `__next__`-Methode der `Fibonacci`-Klasse überschreiben und dann statt der Folgeelemente die Quotienten zurückgeben. Dabei kommt es uns zugute, dass wir in dem Attribut `B` bereits den Wert des nächsten Elements im Voraus berechnen. Die Implementation sieht dann folgendermaßen aus:

```
class GoldenerSchnitt(Fibonacci):
    def __next__(self):
        Fibonacci.__next__(self)
        return self.B / self.A
```

In Python-Versionen vor 3.0 musste man an dieser Stelle `self.B` vor der Division in eine `float`-Instanz konvertieren, da sonst eine Integerdivision, also eine Division ohne Nachkommastellen, ausgeführt worden wäre. Seit Python 3.0 können wir uns die Konvertierung sparen, da mit dem Divisionsoperator `/` immer mit Nachkommaanteil gerechnet wird. Schon die ersten vierzehn Elemente dieser Folge lassen die Konvergenz erkennen. (Der Goldene Schnitt, bis auf sechs Nachkommastellen gerundet, lautet 1,618034.)

```
>>> for g in GoldenerSchnitt(14):
    print("{0:.6f}".format(g), end=" ")
1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619
048 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037
```

Es ist durchaus üblich, die `__iter__`-Methode eines iterierbaren Objekts als Generator zu implementieren. Im Falle unserer Fibonacci-Folge läuft diese Technik auf wesentlich eleganteren Code hinaus, weil wir uns nun nicht mehr den Status des Iterators zwischen den `__next__`-Aufrufen merken müssen und auch die explizite Definition von `__next__` entfällt:

```
class Fibonacci2:
    def __init__(self, max_n):
        self.MaxN = max_n
```

```
def __iter__(self):
    n = 0
    a, b = 0, 1
    for n in range(self.MaxN):
        a, b = b, a + b
    yield a
```

Instanzen der Klasse `Fibonacci2` verhalten sich bei der Iteration genau wie die Lösung ohne Generator-Ansatz:

```
>>> list(Fibonacci2(10))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Allerdings ließe sich die Klasse `GoldenerSchnitt` nicht mehr so einfach als Subklasse von `Fibonacci2` implementieren, da die Zwischenspeicherung der Werte und auch die `__next__`-Methode nun in dem Generator gekapselt sind.

Benutzung von Iteratoren

Nun haben Sie gelernt, wie Sie eine gültige Iterator-Schnittstelle in ihren eigenen Klassen implementieren können. Wir werden diese Thematik jetzt von der anderen Seite betrachten und uns damit beschäftigen, wie die Benutzung dieser Iterator-Schnittstelle aussieht, damit Sie auch Funktionen schreiben können, die nicht Listen oder andere Sequenzen, sondern beliebige iterierbare Instanzen verarbeiten können.

Wir betrachten zu diesem Zweck eine einfache `for`-Schleife und werden dann hinter die Kulissen schauen, indem wir eine äquivalente Schleife ohne `for` programmieren werden, die explizit das Iterator-Protokoll benutzt:

```
>>> for i in range(10):
        print(i, end=" ")
0 1 2 3 4 5 6 7 8 9
```

Wie Sie bereits wissen, benötigen wir zum Durchlaufen einer Sequenz das dazugehörige Iterator-Objekt. Dieses liefert uns die Built-in Funktion `iter`, die, wie schon vorigen Abschnitt erklärt, die `__iter__`-Methode des übergebenen Objekts aufruft:

```
>>> iter(range(10))
<range_iterator object at 0xb7b1eb18>
```

Über die `__next__`-Methode des Iterator-Objekts ermitteln wir nun der Reihe nach alle Elemente:

```
>>> i = iter(range(3))
>>> i.__next__()
0
```

```
>>> i.__next__()
1
>>> i.__next__()
2
>>> i.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Wird `i.__next__` nach dem Zurückgeben des letzten Elements erneut aufgerufen, wirft die Methode erwartungsgemäß die `StopIteration`-Exception. Wenn wir diese Exception mit einer `try/except`-Anweisung abfangen, können wir die `for`-Schleife folgendermaßen nachbauen:

```
>>> i = iter(range(10))
>>> while True:
    try:
        print(i.__next__(), end=" ")
    except StopIteration:
        break
0 1 2 3 4 5 6 7 8 9
```

Natürlich soll dieses Beispiel keine Aufforderung sein, in Zukunft keine `for`-Schleifen mehr zu benutzen. Das Ziel unserer Bemühungen war es, Ihnen ein besseres Verständnis für die Benutzung von Iteratoren zu vermitteln. Die `for`-Schleife in Python ist natürlich nicht wie in dem Beispiel implementiert, sondern in eine optimierte Routine des Python-Interpreters ausgelagert. Dadurch erlaubt der Iterator-Ansatz auch eine Geschwindigkeitssteigerung, weil die Iteration durch eine maschinennahe C-Schleife übernommen werden kann.

Die `for`-Schleife kann im Übrigen auch über einen Iterator selbst iterieren und muss diesen nicht selbst erzeugen. Die folgenden beiden Schleifen sind also äquivalent:

```
>>> for i in range(3):
    print(i, end=" ")
0 1 2
>>> for i in iter(range(3)):
    print(i, end=" ")
0 1 2
```

Dass `for` dabei, wie in der alternativen `while`-Schleife verdeutlicht, noch einmal selbst `iter` aufruft, ist insofern kein Problem, als die `__iter__`-Methode eines Iterator-Objekts eine Referenz auf das Objekt selbst zurückgeben muss. Ist `a` ein Iterator-Objekt, so gilt immer `a is iter(a)`, wie das folgende Beispiel noch einmal verdeutlicht:

```
>>> a = iter(range(10)) # einen range-Iterator erzeugen
>>> a is iter(a)
True
```

Im Gegensatz dazu muss die `__iter__`-Methode eines iterierbaren Objekts weder eine Referenz auf sich selbst noch immer dieselbe Iterator-Instanz zurückgeben:

```
>>> a = list((1, 2, 3)) # ein iterierbares Objekt erzeugen
>>> iter(a) is iter(a)
False
```

Im Umkehrschluss bedeutet dies, dass die Built-in Function `iter` bei Aufrufen für dasselbe iterierbare Objekt verschiedene Iteratoren zurückgeben kann.

Dieses Verhalten kann zu relativ schwer auffindbaren Fehlern führen. Stellen Sie sich einmal vor, Sie lesen eine Textdatei ein, die eine bestimmte Schlüsselzeile enthält. Alles, was vor dieser Schlüsselzeile steht, ist für Ihr Programm vollkommen uninteressant, denn Sie interessieren sich nur für den dahinterstehenden Teil. Da Sie bereits wissen, dass man über die Zeilen einer Datei mittels einer eleganten `for`-Schleife iterieren kann, könnten Sie auf folgende Scheinlösung kommen:

```
datei = open("textdatei.txt", "r")
for zeile in datei:
    if zeile.strip() == "Schlüsselzeile":
        break

for zeile in datei:
    print(zeile)
```

Der Grund, warum mit diesem Miniprogramm nicht nur die interessanten Zeilen hinter der "Schlüsselzeile", sondern alle Zeilen der Datei ausgegeben werden, liegt darin, dass beide Schleifen jeweils ihren eigenen Iterator für die Datei erzeugt haben. Deshalb wurde zu Beginn der zweiten `for`-Schleife die Leseposition innerhalb der Datei wieder an den Anfang gesetzt und somit die gesamte Datei ausgegeben. Mit ein paar kleinen Änderungen können wir die Schleifen aber dazu zwingen, sich einen Iterator zu teilen, und erreichen damit das gewünschte Verhalten:

```
datei = open("textdatei.txt", "r")
datei_iterator = iter(datei)
for zeile in datei_iterator:
    if zeile.strip() == "Schlüsselzeile":
        break

for zeile in datei_iterator:
    print(zeile)
```


Da die impliziten `iter`-Aufrufe am Anfang der beiden `for`-Schleifen nun Referenzen auf denselben Iterator zurückgeben, erscheinen nur die interessanten Informationen auf dem Bildschirm. Es ist also in manchen Fällen durchaus sinnvoll, explizit Iteratoren zu erzeugen und mit diesen zu arbeiten.

Nachteile von Iteratoren gegenüber dem direkten Zugriff über Indizes

Neben den schon angesprochenen Vorteilen, dass einmal geschriebener Code für alle Datentypen, die das Iterator-Interface implementieren, gilt und dass durch die maschinennahe Implementation der Schnittstelle die Ausführung der Programme beschleunigt werden kann, haben Iteratoren auch Nachteile.

Iteratoren eignen sich hervorragend, um alle Elemente einer Sequenz zu durchlaufen und dies einheitlich für alle Container-Datentypen umzusetzen. Mit Indizes ist aber auch möglich, in beliebiger Reihenfolge auf die Elemente zuzugreifen und ihre Werte zu verändern, was mit dem Iterator-Ansatz nicht möglich ist.

Insofern lassen sich die Indizes nicht vollständig durch Iteratoren ersetzen, sondern werden für Spezialfälle durch sie ergänzt.

Alternative Definition für iterierbare Objekte

Neben der oben beschriebenen Definition für iterierbare Objekte gibt es eine weitere Möglichkeit, eine Klasse iterierbar zu machen. Da es bei sehr vielen Folgen und Containern möglich ist, die Elemente einfach durchzunummerieren und über ganzzahlige Indizes anzusprechen, haben sich die Python-Entwickler dazu entschlossen, dass ein Objekt schon dann iterierbar ist, wenn man seine Elemente über die `__getitem__`-Methode, also den `[]`-Operator, ansprechen kann. Ruft man die Built-in Funktion `iter` mit einer solchen Instanz als Parameter auf, kümmert Python sich um die Erzeugung des Iterators. Bei jedem Aufruf der `__next__`-Methode des erzeugten Iterators wird die `__getitem__`-Methode der iterierbaren Instanz aufgerufen, wobei immer eine Ganzzahl als Parameter übergeben wird. Die Zählung der übergebenen Indizes beginnt bei 0 und endet erst, wenn die `__getitem__`-Methode einen `IndexError` produziert, sobald ein ungültiger Index übergeben wurde.

Beispielsweise könnte eine Klasse zum Iterieren über die ersten `max_n` Quadratzahlen folgendermaßen aussehen, wenn sie zudem noch das Bestimmen ihrer Länge mittels `len` unterstützt:

```
class Quadrature:
    def __init__(self, max_n):
        self.MaxN = max_n
```

```

def __getitem__(self, index):
    index += 1 # 0*0 ist nicht sehr interessant...
    if index > len(self) or index < 1:
        raise IndexError
    return index*index

def __len__(self):
    return self.MaxN

```

Zur Demonstration dieses versteckten Iterators lassen wir uns eine Liste mit den ersten zwanzig Quadratzahlen ausgeben:

```

>>> list(Quadrade(20))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
 289, 324, 361, 400]

```

Diese Art von Iterator-Definition sollte nur in seltenen Fällen benutzt werden, da sie einerseits wenig elegant und andererseits meistens langsamer als eine explizite Implementation des Iterator-Protokolls ist.

Funktionsiteratoren

Die letzte Möglichkeit, in Python auf Iteratoren zurückzugreifen, stellen sogenannte *Funktionsiteratoren* dar. Funktionsiteratoren sind Objekte, die eine bestimmte Funktion so lange aufrufen, bis diese einen bestimmten Wert, den *Terminator* der Folge, zurückgibt. Einen Funktionsiterator erzeugen Sie mit der Built-in Function `iter`, wobei Sie als ersten Parameter eine Referenz auf die Funktion, über die Sie iterieren möchten, und als zweiten Parameter der Wert des Terminators übergeben.

`iter(funktion, terminator)`

Ein gutes Beispiel ist die Methode `readline` des `file`-Objekts, die so lange den Wert der nächsten Zeile zurückgibt, bis das Ende der Datei erreicht wurde. Wenn sich keine weiteren Daten mehr hinter der aktuellen Leseposition der `file`-Instanz befinden, gibt `readline` einen leeren String zurück. Läge im aktuellen Arbeitsverzeichnis eine Datei namens *freunde.txt*, die die vier Namen "Lucas", "Florian", "Lars" und "John" in je einer separaten Zeile enthält, so könnten wir folgendermaßen über sie iterieren:

```

>>> datei = open("freunde.txt")
>>> for zeile in iter(datei.readline, ""):
    print(zeile.strip(), end=" ")
Lucas Florian Lars John

```

Anmerkung

Dieses Beispiel dient nur der Veranschaulichung von Funktionsiteratoren. Über die Zeilen einer Datei können Sie natürlich auch weiterhin direkt mit

```
>>> for zeile in datei:
        print(zeile.strip(), end=" ")
iterieren.
```

13.6 Interpreter im Interpreter

In bestimmten Fällen ist es nützlich, vom Benutzer eingegebenen oder anderweitig zur Laufzeit geladenen Python-Code aus einem Python-Programm heraus auszuführen. Stellen Sie sich einmal vor, Sie wollten ein Programm schreiben, das Wertetabellen für beliebige Funktionen mit einem ganzzahligen Parameter darstellt. Für ein solches Programm muss der Benutzer die Funktion festlegen können. Anstatt dafür eine eigene Sprache zu definieren und einen eigenen Parser und Compiler zu schreiben, bietet es sich an, Funktionsdefinitionen in Python-Syntax zu erlauben.

Mithilfe der `exec`-Built-in können wir genau dies erreichen. Pythons `exec`-Anweisung erwartet einen String als Parameter, der den auszuführenden Code enthält. Alternativ kann auch ein geöffnetes Datei-Objekt an `exec` übergeben werden.

Um beispielsweise eine vom Benutzer eingegebene Funktion für die Ausgabe einer kleinen Wertetabelle zu benutzen, dient der folgende Code-Schnipsel:

```
print("Definieren Sie eine Funktion f mit einem Parameter:")
definition = input()
exec(definition)
for i in range(5):
    print("f({0}) = {1:f}".format(i, f(i)))
```

Ein Programmlauf könnte dann wie folgt aussehen:

```
Definieren Sie eine Funktion f mit einem Parameter:
def f(x): return x*x
f(0) = 0.000000
f(1) = 1.000000
f(2) = 4.000000
f(3) = 9.000000
f(4) = 16.000000
```

Wie Sie sehen, ist die Funktion `f`, die von dem Benutzer definiert wurde, nach dem Ausführen von `exec` im lokalen Namensraum unseres Programms verfügbar,

denn wir können sie ganz normal aufrufen. Ebenso kann der Benutzer neue Variablen anlegen oder den Wert bereits bestehender Variablen auslesen, was allerdings ein Sicherheitsrisiko darstellt.

Um die Sicherheit zu erhöhen, können Sie den mit `exec` ausgeführten Code in einem eigenen Namensraum »einsperren«. Alle neuen Variablen, Klassen und Funktionen werden in diesem gesonderten Namensraum abgelegt. Außerdem sind dem `exec`-Code nur noch die Variablen zugänglich, die in seinem Namensraum vorhanden sind. Ein Namensraum ist ein einfaches Dictionary, das den Referenznamen ihre Werte zuordnet. Um einem `exec`-Statement einen eigenen Namensraum zu geben, stellt man das Dictionary als zweiten Parameter hintenan:

```
>>> kontext = {"pi" : 3.1459}
>>> exec("print(pi)", kontext)
3.1459
```

Die vollständige Schnittstelle von `exec` hat zwei Parameter für den Kontext, einen für die globalen und einen für die lokalen Variablen:

`exec(object[, globals[, locals]])`

Wir haben in unserem Beispiel also nur einen globalen Kontext festgelegt.

Alle Referenzen, die innerhalb des `exec`-Codes definiert wurden, sind anschließend auch in dem übergebenen Kontext definiert. Damit sichern wir unser Einstiegsbeispiel gegen ungewollte Seiteneffekte ab. Den Wert der Kreiszahl π wollen wir dem Benutzer auch für seine Funktionen zugänglich machen:

```
print("Definieren Sie eine Funktion f mit einem Parameter:")
definition = input()
kontext = {"pi" : 3.1459}
exec(definition, kontext)
for i in range(5):
    print("f({0}) = {1}".format(i, kontext['f'](i)))
```

Ein Beispiellauf, in dem der Benutzer eine Funktion für die Berechnung der Kreisfläche anhand des Kreisradius eingibt, sähe dann so aus:

```
Definieren Sie eine Funktion f mit einem Parameter:
def f(r): return pi * r**2
f(0) = 0.000000
f(1) = 3.145900
f(2) = 12.583600
f(3) = 28.313100
f(4) = 50.334400
```

Ausdrücke auswerten mit eval

Während mit `exec` beliebiger Python-Code ausgeführt werden kann, dient die Built-in Function `eval` dazu, Python-Ausdrücke auszuwerten und das Ergebnis zurückzugeben:

```
>>> eval("5 * 4")
20
```

Auch der von `eval` ausgewertete Ausdruck hat standardmäßig Zugriff auf alle Variablen des aktuellen Kontexts. Genau wie bei `exec` kann durch die beiden zusätzlichen Parameter `globals` und `locals` ein benutzerdefinierter Kontext festgelegt werden.

```
>>> x = 10
>>> eval("5 * x")
50
>>> eval("5 * x", {})
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    eval("5 * x", {})
  File "<string>", line 1, in <module>
NameError: name 'x' is not defined
```

Beim ersten Aufruf von `eval` konnten wir auf die globale Variable `x` zugreifen, weil der Kontext einfach kopiert wurde. Dem zweiten Aufruf hingegen übergaben wir ein leeres Dictionary als Kontext, weshalb der versuchte Zugriff auf `x` mit einer Exception quittiert wurde.

Die vollständige Schnittstelle von `eval` sieht folgendermaßen aus:

```
eval(source [, globals[, locals]])
```

13.7 Geplante Sprachelemente

Die Sprache Python befindet sich in ständiger Entwicklung, und jede neue Version bringt neue Sprachelemente mit sich, die alten Python-Code unter Umständen inkompatibel mit der neusten Version des Interpreters machen. Zwar geben sich die Entwickler Mühe, größtmögliche Kompatibilität zu wahren, doch ist durch das bloße Hinzufügen eines Schlüsselwortes schon derjenige Code inkompatibel geworden, der das neue Schlüsselwort als normalen Bezeichner verwendet.

Der Interpreter besitzt eine Art Modus, mit dem sich einige ausgewählte Sprachelemente der kommenden Python-Version bereits mit der aktuellen Version testen lassen. Dies soll den Wechsel von einer Version zur nächsten vereinfachen,

da bereits gegen einige neue Features der nächsten Version getestet werden kann, bevor diese herausgegeben wird.

Zum Einbinden eines geplanten Features wird eine `import`-Anweisung verwendet:

```
from __future__ import sprachelement
```

Die Sprachelemente können verwendet werden, als wären sie in einem Modul namens `__future__` gekapselt. Beachten Sie aber, dass Sie mit dem Modul `__future__` nicht ganz so frei umgehen können, wie Sie das von anderen Modulen her gewohnt sind. Sie dürfen es beispielsweise nur am Anfang einer Programmdatei einbinden. Vor einer solchen `import`-Anweisung dürfen nur Kommentare, leere Zeilen oder andere *Future Imports* stehen.

Wir möchten hier nicht näher auf die einzelnen Features und ihre Verwendung eingehen, da sie mitunter allzu speziell sind und meist aus älteren Python-Versionen stammen. Es ist jedoch immer interessant, ein wenig mit den geplanten Features herumzuspielen und sich selbst ein Bild davon zu machen.

13.8 Die with-Anweisung

Es gibt Operationen, die in einem bestimmten Kontext ausgeführt werden müssen und bei denen sichergestellt werden muss, dass der Kontext jederzeit korrekt deinitialisiert wird, beispielsweise auch, wenn eine Exception auftritt. Als Beispiel für einen solchen Kontext dient das Dateiojekt. Es muss sichergestellt sein, dass die `close`-Methode des Dateiojekts gerufen wird, selbst wenn zwischen dem Aufruf von `open` und dem der `close`-Methode des Dateiojekts eine Exception geworfen wurde. Dazu ist mit den herkömmlichen Sprachelementen Pythons folgende `try/finally`-Anweisung nötig:

```
f = open("datei.txt", "r")
try:
    print(f.read())
finally:
    f.close()
```

Zunächst wird eine Datei namens *datei.txt* zum Lesen geöffnet. Die darauffolgende `try/finally`-Anweisung stellt sicher, dass `f.close` in jedem Fall aufgerufen wird. Dieses Beispiel lässt sich mit der `with`-Anweisung folgendermaßen formulieren:

```
with open("programm.py", "r") as f:
    print(f.read())
```

Die `with`-Anweisung besteht aus dem Schlüsselwort `with`, gefolgt von einer Instanz. Optional können auf die Instanz das Schlüsselwort `as` und ein Bezeichner folgen. Dieser Bezeichner wird *Target* genannt, und seine Bedeutung hängt von der verwendeten Instanz ab. Im obigen Beispiel referenziert `f` das geöffnete Dateiojekt. Um zu verstehen, was bei einer `with`-Anweisung genau passiert, definieren wir im nächsten Beispiel eine eigene Klasse, die sich mit der `with`-Anweisung verwenden lässt. Eine solche Klasse wird *Kontextmanager* genannt.

Die Klasse `MeinLogfile` ist dafür gedacht, eine rudimentäre Logdatei zu führen. Dazu implementiert sie die Funktion `eintrag`, die eine neue Zeile in die Logdatei schreibt. Die Klassendefinition sieht folgendermaßen aus:

```
class MeinLogfile:
    def __init__(self, logfile):
        self.logfile = logfile
        self.f = None

    def eintrag(self, text):
        self.f.write("==>{0}\n".format(text))

    def __enter__(self):
        self.f = open(self.logfile, "w")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.f.close()
```

Zu den beiden ersten Methoden der Klasse ist nicht viel zu sagen. Dem Konstruktor `__init__` wird der Dateiname der Logdatei übergeben, der intern im Attribut `self.logfile` gespeichert wird. Zusätzlich wird das Attribut `self.f` angelegt, das später das geöffnete Dateiojekt referenzieren soll.

Die Methode `eintrag` hat die Aufgabe, den übergebenen Text in die Logdatei zu schreiben. Dazu ruft sie einfach die Methode `write` des Dateiojekts auf. Beachten Sie, dass die Methode `eintrag` nur innerhalb einer `with`-Anweisung aufgerufen werden kann, da das Dateiojekt erst in den folgenden Magic Functions geöffnet und geschlossen wird.

Die angesprochenen Magic Functions `__enter__` und `__exit__` sind das Herzstück der Klasse und müssen implementiert werden, wenn die Klasse im Zusammenhang mit `with` verwendet werden soll. Die Methode `__enter__` wird aufgerufen, wenn der Kontext aufgebaut, also bevor der Körper der `with`-Anweisung ausgeführt wird. Die Methode bekommt keine Parameter, gibt aber einen Wert zurück. Der Rückgabewert von `__enter__` wird später vom Target-Bezeichner referenziert, sofern einer angegeben wurde. Im Falle unserer Beispielklasse wird

die Datei `self.logfile` zum Schreiben geöffnet und mit `return self` eine Referenz auf die eigene Instanz zurückgegeben.

Die zweite Magic Function `__exit__` wird aufgerufen, wenn der Kontext verlassen wird, also nachdem der Körper der `with`-Anweisung entweder vollständig durchlaufen oder durch eine Exception vorzeitig abgebrochen wurde. Im Falle der Beispielklasse wird das geöffnete Dateiobjekt `self.f` geschlossen. Näheres zu den drei Parametern der Methode `__exit__` folgt weiter unten. Die soeben erstellte Klasse `MeinLogfile` lässt sich folgendermaßen mit `with` verwenden:

```
inst = MeinLogfile("logfile.txt")
with inst as log:
    log.eintrag("Hallo Welt")
    log.eintrag("Na, wie gehts?")
```

Zur Erklärung: Zunächst wird eine Instanz der Klasse `MeinLogfile` erstellt und dabei der Dateiname `logfile.txt` übergeben. Die `with`-Anweisung bewirkt als Erstes, dass die Methode `__enter__` der Instanz `inst` ausgeführt und ihr Rückgabewert durch `log` referenziert wird. Dann wird der Körper der `with`-Anweisung ausgeführt, in dem insgesamt zweimal die Methode `eintrag` aufgerufen und damit Text in die Logdatei geschrieben wird. Nachdem der Anweisungskörper ausgeführt worden ist, wird einmalig die Methode `__exit__` der Instanz `inst` aufgerufen.

Im Folgenden sollen die Magic Functions `__enter__` und `__exit__` vollständig erläutert werden.

`__enter__(self)`

Diese Magic Function wird einmalig zum Öffnen des Kontexts aufgerufen, bevor der Körper der `with`-Anweisung ausgeführt wird. Der Rückgabewert dieser Methode wird im Körper der `with`-Anweisung vom Target-Bezeichner referenziert.

`__exit__(self, exc_type, exc_value, traceback)`

Die Magic Function `__exit__` wird einmalig zum Schließen des Kontexts aufgerufen, nachdem der Körper der `with`-Anweisung ausgeführt worden ist. Die drei Parameter `exc_type`, `exc_value` und `traceback` spezifizieren Typ, Wert und Traceback-Objekt einer eventuell innerhalb des `with`-Anweisungskörpers geworfenen Exception. Wenn keine Exception geworfen wurde, referenzieren alle drei Parameter `None`. Wie mit einer geworfenen Exception weiter verfahren wird, steuern Sie mit dem Rückgabewert der Methode `__exit__`: Gibt die Methode `True` zurück, wird die Exception unterdrückt. Bei einem Rückgabewert von `False` wird die Exception erneut geworfen.

13.9 Function Annotations

Seit Python 3.0 gibt es eine Syntax, mit der Sie die Parameter und den Rückgabewert einer Funktion mit einer sogenannten Annotation, einer Anmerkung, versehen können. Bevor wir uns in einem Beispiel von der Nützlichkeit dieser Annotations überzeugen, besprechen wir zunächst, wo solche Annotations bei der Funktionsdefinition syntaktisch untergebracht werden:

```
def funktion(p1: Annotation1, p2: Annotation2) -> Annotation3:
    Funktionskörper
```

Bei der Definition einer Funktion kann hinter jeden Parameter ein Doppelpunkt, gefolgt von einer Annotation, geschrieben werden. Eine Annotation darf dabei ein beliebiger Python-Ausdruck sein. Die Angabe einer Annotation ist völlig optional, und die Funktionsschnittstelle darf auch durchaus nur teilweise mit Annotations versehen werden. Hinter der Parameterliste kann eine ebenfalls optionale Annotation für den Rückgabewert der Funktion geschrieben werden. Diese wird durch einen Pfeil (->) eingeleitet. Erst hinter dieser Annotation folgt der Doppelpunkt, der den Funktionskörper einleitet.

Eine Funktion, die wie die obige mit Annotations versehen wurde, verhält sich nicht anders als eine äquivalente Funktion ohne Annotations. Man könnte sagen: Dem Python-Interpreter sind Annotations egal. Das Interessante an Function Annotations ist, dass man sie über das Attribut `__annotations__` des Funktionsobjektes auslesen kann. Da Annotations beliebige Ausdrücke sein dürfen, kann der Programmierer hier also eine Information pro Parameter und Rückgabewert »speichern«, auf die er zu einem späteren Zeitpunkt – beispielsweise, wenn die Funktion mit konkreten Parameterwerten aufgerufen wird – zurückkommt.

Dabei werden die Annotations über das Attribut `__annotations__` in Form eines Dictionarys zugänglich gemacht. Dieses Dictionary enthält die Parameternamen bzw. "return" für die Annotation des Rückgabewertes als Schlüssel und die jeweiligen Annotation-Ausdrücke als Werte. Für die obige schematische Funktionsdefinition sähe dieses Dictionary also folgendermaßen aus:

```
funktion.__annotations__ =
{
    "p1" : Annotation1,
    "p2" : Annotation2,

    "return" : Annotation3
}
```

Mit Function Annotations könnten Sie also beispielsweise eine Typüberprüfung an der Funktionsschnittstelle durchführen. Dies soll unser Beispiel sein. Dazu definieren wir zunächst eine Funktion samt Annotations:

```
def strmult(s: str, n: int) -> str:
    return a*b
```

Die Funktion `strmult` hat die Aufgabe, einen String `s` `n`-mal hintereinandergeschrieben zurückzugeben. Das geschieht durch Multiplikation von `s` und `n`. Es wäre natürlich kein Gewinn, wenn jede Funktion ihre eigenen Parameter auf Richtigkeit überprüfen müsste, das würde auch ohne Function Annotations funktionieren. Wir schreiben jetzt eine Funktion `call`, die dazu in der Lage ist, eine beliebige Funktion, deren Schnittstelle vollständig durch Annotations beschrieben ist, aufzurufen bzw. eine Exception zu werfen, wenn einer der übergebenen Parameter einen falschen Typ hat:

```
def call(f, **kwargs):
    for arg in kwargs:
        if arg not in f.__annotations__:
            raise TypeError("Parameter '{0}'"
                           " unbekannt".format(arg))

        if type(kwargs[arg]) != f.__annotations__[arg]:
            raise TypeError("Parameter '{0}'"
                           " hat ungültigen Typ".format(arg))

    ret = f(**kwargs)
    if type(ret) != f.__annotations__["return"]:
        raise TypeError("Ungültiger Rückgabewert")
    return ret
```

Die Funktion `call` bekommt ein Funktionsobjekt und beliebig viele Schlüsselwortparameter übergeben. Dann greift sie für jeden übergebenen Schlüsselwortparameter auf das Annotation-Dictionary des Funktionsobjektes `f` zu und prüft, ob ein Parameter dieses Namens überhaupt in der Funktionsdefinition von `f` vorkommt, und wenn ja, ob die für diesen Parameter übergebene Instanz den richtigen Typ hat. Ist eines von beidem nicht der Fall, wird eine entsprechende Exception geworfen.

Wenn alle Parameter korrekt übergeben wurden, wird das Funktionsobjekt `f` aufgerufen und der Rückgabewert gespeichert. Dessen Typ wird dann mit dem Datentyp verglichen, der in der Annotation für den Rückgabewert angegeben wurde; wenn er abweicht, wird eine Exception geworfen. Ist alles gutgegangen, wird der Rückgabewert der Funktion `f` von `call` durchgereicht:

```
>>> call(strmult, s="Hallo", n=3)
'HalloHalloHallo'

>>> call(strmult, s="Hallo", n="Welt")
Traceback (most recent call last):
  [...]
TypeError: Parameter 'n' hat ungültigen Typ

>>> call(strmult, s=13, n=37)
Traceback (most recent call last):
  [...]
TypeError: Parameter 's' hat ungltigen Typ
```

Um die Überprüfung auf den Rückgabewert testen zu können, muss natürlich die Definition der Funktion `strmult` verändert werden.

13.10 Function Decorator

Aus Kapitel 12, »Objektorientierung«, kennen Sie sicherlich noch die Built-in Function `staticmethod`, die folgendermaßen verwendet wurde:

```
class MeineKlasse:
    def methode():
        pass

    methode2 = staticmethod(methode)
```

Durch diese Schreibweise wird zunächst eine Methode angelegt und später durch die Built-in Function `staticmethod` modifiziert. Die angelegte Methode wird dann mit dem modifizierten Funktionsobjekt überschrieben.

Diese Art, `staticmethod` anzuwenden, ist zwar richtig und funktioniert, ist aber gleichzeitig auch unidiomatisch und nicht gerade gut lesbar. Aus diesem Grund unterstützt Python eine eigene Notation, um den obigen Code lesbarer zu gestalten. Das folgende Beispiel ist zu dem vorherigen äquivalent:

```
class MeineKlasse:
    @staticmethod
    def methode():
        pass
```

Die Funktion, die die angelegte Methode modifizieren soll, wird nach einem `@`-Zeichen vor die Methodendefinition geschrieben. Eine solche Notation wird *Function Decorator* genannt. Allerdings sind Function Decorators nicht auf den Einsatz mit `staticmethod` beschränkt, vielmehr können Sie beliebige Decorators

erstellen. Auf diese Weise können Sie eine Funktion durch bloßes Hinzufügen eines Decorators um eine gewisse Funktionalität erweitern.

Function Decorators können nicht nur auf Methoden angewendet werden, sondern genauso auf Funktionen. Zudem können sie ineinander verschachtelt werden, wie folgendes Beispiel zeigt:

```
@dec1
@dec2
def funktion():
    pass
```

Diese Funktionsdefinition ist äquivalent zu folgendem Code:

```
def funktion():
    pass
funktion = dec1(dec2(funktion))
```

Es erübrigt sich zu sagen, dass sowohl `dec1` als auch `dec2` implementiert werden müssen, bevor die Beispiele lauffähig sind.

Das jetzt folgende Beispiel soll einen interessanten Ansatz zum Cachen (dt. »Zwischenspeichern«) von Funktionsaufrufen zeigen, bei dem die Ergebnisse von komplexen Berechnungen automatisch gespeichert werden. Diese können dann beim nächsten Funktionsaufruf mit den gleichen Parametern wiedergegeben werden, ohne die Berechnungen erneut durchführen zu müssen. Das Caching einer Funktion soll allein durch Angabe eines Function Decorators erfolgen, also ohne in die Funktion selbst einzugreifen, und zudem mit beliebigen Funktionschnittstellen, also beliebigen Funktionen, arbeiten können. Dazu sehen wir uns zunächst die Definition der Berechnungsfunktion an, die in diesem Fall die Fakultät einer ganzen Zahl berechnet, inklusive Function Decorator:

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

Die Berechnung einer Fakultät sollte Ihnen inzwischen geläufig sein. Interessant ist hier allerdings der Function Decorator, denn es handelt sich hierbei nicht um eine Funktion, sondern um eine Klasse namens `CacheDecorator`, die im Decorator instantiiert wird. Sie erinnern sich sicherlich, dass eine Klasse durch Implementieren der Magic Function `__call__` aufrufbar gemacht werden kann und sich damit wie ein Funktionsobjekt verhält. Wir müssen diesen Umweg gehen, da wir die Ergebnisse der Berechnungen so speichern müssen, dass sie auch in

späteren Aufrufen des Decorators noch verfügbar sind. Das ist mit einer Funktion nicht möglich, wohl aber mit einer Klasse. Die Definition der Decorator-Klasse sieht folgendermaßen aus:

```
class CacheDecorator:

    def __init__(self):
        self.cache = {}
        self.func = None

    def cachedFunc(self, *args):
        if args not in self.cache:
            self.cache[args] = self.func(*args)
        return self.cache[args]

    def __call__(self, func):
        self.func = func
        return self.cachedFunc
```

Im Konstruktor der Klasse `CacheDecorator` wird ein leeres Dictionary für die zwischengespeicherten Werte angelegt. Neben dem Konstruktor ist unter anderem die Methode `__call__` implementiert. Durch diese Methode werden Instanzen der Klasse aufrufbar, können also wie ein Funktionsobjekt verwendet werden. Um als Function Decorator verwendet werden zu können, muss die Methode `__call__` ein Funktionsobjekt als Parameter akzeptieren und ein Funktionsobjekt zurückgeben, das fortan als veränderte Version der ursprünglich angelegten Funktion mit dieser assoziiert wird. In diesem Fall gibt `__call__` das Funktionsobjekt der Methode `cachedFunc` zurück.

Die Methode `cachedFunc` soll also fortan anstelle der ursprünglich angelegten Funktion aufgerufen werden. Damit sie ihre Aufgabe erledigen kann, hat sie Zugriff auf das Funktionsobjekt der eigentlichen Funktion, das von dem Attribut `self.func` referenziert wird. Die Methode `cachedFunc` akzeptiert beliebig viele Positional Arguments, da sie später für beliebige Funktionen und damit beliebige Funktionsschnittstellen arbeiten muss. Diese Argumente sind innerhalb der Methode als Tupel verfügbar.

Jetzt wird geprüft, ob das Tupel mit den übergebenen Argumenten bereits als Schlüssel im Dictionary `self.cache` existiert. Wenn ja, wurde die Funktion bereits mit exakt den gleichen Argumenten aufgerufen, und der im Cache gespeicherte Rückgabewert kann direkt zurückgegeben werden. Ist der Schlüssel nicht vorhanden, wird die Berechnungsfunktion `self.func` mit den übergebenen Argumenten aufgerufen und das Ergebnis im Cache gespeichert. Anschließend wird es zurückgegeben.

Um zu testen, ob das Speichern der Werte funktioniert, wird das Beispiel um zwei Ausgaben erweitert, je nachdem, ob ein Ergebnis neu berechnet oder aus dem Cache geladen wurde. Und tatsächlich, es funktioniert:

```
>>> fak(10)
Ergebnis berechnet
3628800
>>> fak(20)
Ergebnis berechnet
2432902008176640000
>>> fak(20)
Ergebnis geladen
2432902008176640000
>>> fak(10)
Ergebnis geladen
3628800
```

Wie Sie sehen, wurden die ersten beiden Ergebnisse berechnet, während die letzten beiden aus dem internen Cache geladen wurden. Diese Form des Cachings bietet je nach Anwendungsbereich und Komplexität der Berechnung erhebliche Geschwindigkeitsvorteile.

13.11 assert

Mithilfe des Schlüsselworts `assert` lassen sich Konsistenzabfragen in ein Python-Programm integrieren. Durch das Schreiben einer `assert`-Anweisung legt der Programmierer eine Bedingung fest, die für die Ausführung des Programms essenziell ist und die bei Erreichen der `assert`-Anweisung zu jeder Zeit `True` ergeben muss. Wenn die Bedingung einer `assert`-Anweisung `False` ergibt, wird eine `AssertionError`-Exception geworfen. In der folgenden Sitzung im interaktiven Modus wurden mehrere `assert`-Anweisungen eingegeben:

```
>>> import math
>>> assert math.log(1) == 0
>>> assert math.sqrt(4) == 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert math.sqrt(9) == 3
>>>
```

Die `assert`-Anweisung ist damit ein wichtiges Hilfsmittel zum Aufspüren von Fehlern und ermöglicht es, den Programmlauf zu beenden, wenn bestimmte Vo-

raussetzungen nicht gegeben sind. Häufig prüft man an Schlüsselstellen im Programm mit `assert`, ob alle Referenzen die erwarteten Werte referenzieren, um eventuelle Fehlberechnungen rechtzeitig und umfassend erkennen zu können.

Beachten Sie, dass `assert`-Anweisungen üblicherweise nur während der Entwicklung eines Programms benötigt werden und in einem fertigen Programm eher stören würden. Deswegen werden `assert`-Anweisungen nur dann ausgeführt, wenn die globale Konstante `__debug__` `True` referenziert. Diese Konstante referenziert nur dann `False`, wenn der Interpreter mit der Kommandozeilenoption `-O` gestartet wurde. Wenn die Konstante `__debug__` `False` referenziert, werden `assert`-Anweisungen ignoriert und haben damit keinen Einfluss mehr auf die Laufzeit Ihres Programms.

Beachten Sie, dass Sie den Wert von `__debug__` im Programm selbst nicht verändern dürfen, sondern nur über die Kommandozeilenoption `-O` bestimmen können, ob `assert`-Anweisungen ausgeführt oder ignoriert werden sollen.

13.12 Weitere Aspekte der Syntax

Das Thema dieses Abschnitts sollen kleinere Aspekte der Python-Syntax sein, die bisher vernachlässigt wurden. Allgemein gilt, dass die hier besprochenen Notationen keineswegs notwendig oder unumgänglich sind. Entscheiden Sie ganz nach Ihren Vorlieben, ob und in welchem Umfang Sie sie einsetzen möchten.

13.12.1 Umbrechen langer Zeilen

Sicherlich haben Sie bereits einige eigene Python-Programme geschrieben, und dabei ist die ein oder andere recht lange Quellcodezeile entstanden. Viele Programmierer beschränken die Länge ihrer Quellcodezeilen, damit beispielsweise mehrere Quellcodezeilen nebeneinander auf den Bildschirm passen oder der Code auch auf Geräten mit einer festen Zeichenbreite angenehm zu lesen ist. Eine geläufige maximale Zeilenlänge ist 80 Zeichen. Doch welche Möglichkeiten bietet Python, überlange Zeilen umzubrechen, so dass eine maximale Zeilenlänge eingehalten werden kann?

Sie wissen bereits, dass Sie Ihren Quellcode innerhalb von Klammern beliebig umbrechen dürfen, doch an vielen anderen Stellen sind Sie an die strengen syntaktischen Regeln von Python gebunden. Durch Einsatz der Backslash-Notation ist es möglich, Quellcode an nahezu beliebigen Stellen in eine neue Zeile umzubrechen:

```
>>> var \
... = \
... 10
>>> var
10
```

Grundsätzlich kann ein Backslash überall da stehen, wo auch ein Leerzeichen hätte stehen können. Somit ist auch ein Backslash innerhalb eines Strings möglich:

```
>>> "Hallo \
... Welt"
'Hallo Welt'
```

Beachten Sie dabei aber, dass eine Einrückung des umbrochenen Teils des Strings Leerzeichen in den String schreibt. Aus diesem Grund sollten Sie folgende Variante, einen String in mehrere Zeilen zu schreiben, vorziehen:

```
>>> "Hallo " \
... "Welt"
'Hallo Welt'
```

Allgemein kann die Backslash-Notation die Lesbarkeit des Quellcodes sowohl vermindern als auch, beispielsweise bei sehr langen Strings, erhöhen. Grundsätzlich sollten Sie nach Möglichkeit versuchen, lesbaren Code zu erzeugen, was unter anderem bedeutet, den Backslash nicht im Übermaß zu verwenden.

13.12.2 Zusammenfügen mehrerer Zeilen

Genau so, wie Sie eine einzeilige Anweisung mithilfe des Backslashes auf mehrere Zeilen umbrechen, können Sie mehrere einzeilige Anweisungen in eine Zeile zusammenfassen. Dazu werden die Anweisungen durch ein Semikolon voneinander getrennt:

```
>>> print("Hallo"); print("Welt")
Hallo
Welt
```

Anweisungen, die aus einem Anweisungskopf und einem Anweisungskörper bestehen, können auch ohne Einsatz eines Semikolons in eine Zeile gefasst werden, sofern der Anweisungskörper selbst aus nicht mehr als einer Zeile besteht:

```
>>> x = True
>>> if x: print("Hallo Welt")
...
Hallo Welt
```


Sollte der Anweisungskörper mehrere Zeilen lang sein, so können diese selbstverständlich durch ein Semikolon zusammengefasst werden:

```
>>> x = True
>>> if x: print("Hallo"); print("Welt")
...
Hallo
Welt
```

Alle durch ein Semikolon zusammengeführten Anweisungen werden so behandelt, als wären sie gleich weit eingerückt. Allein ein Doppelpunkt vermag die Eindrückungstiefe zu vergrößern. Aus diesem Grund gibt es im obigen Beispiel keine Möglichkeit, in derselben Zeile eine Anweisung zu schreiben, die nicht mehr im Körper der `if`-Anweisung steht.

Beachten Sie, dass beim Einsatz des Backslashes und vor allem des Semikolons schnell unleserlicher Code geschrieben wird. Verwenden Sie beide Notationen daher nur, wenn Sie meinen, dass es der Lesbarkeit und Übersichtlichkeit dienlich ist.



Teil III
Die Standardbibliothek

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

14 Mathematik

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

14.1 Mathematische Funktionen – `math`, `cmath`

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

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

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

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

```
>>> import math
```

14.1.1 Mathematische Konstanten

math.pi

Die Kreiszahl Pi (π).

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

Die Konstante ist auch in `cmath` vorhanden.

math.e

Die Eulersche Zahl e .

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

Die Konstante ist auch in `cmath` vorhanden.

14.1.2 Zahlentheoretische Funktionen

math.ceil(x)

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

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

math.fabs(x)

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

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

math.floor(x)

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

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

math.fmod(x, y)

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

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

math.frexp(x)

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

$$x = m \cdot 2^e$$

zu sehen.

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

math.ldexp(m, e)

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

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

math.modf(x)

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

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

14.1.3 Exponential- und Logarithmusfunktionen

math.exp(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.log(x[, base])

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.log10(x)

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

Die Funktion ist auch in `cmath` vorhanden.

math.pow(x, y)

Berechnet x^y . Es können für x , insbesondere aber auch für y , negative Zahlen oder Gleitkommazahlen übergeben werden. Beachten Sie, dass `math.pow` stets eine reelle Zahl zurückgibt und im Falle eines komplexen Ergebnisses eine `ValueError`-Exception wirft.

Diese Funktion ist äquivalent zur Built-in Function `pow`.

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

math.sqrt(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

14.1.4 Trigonometrische Funktionen**math.acos(x)**

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.asin(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.atan(x)

Berechnet den Arkustangens von x . Der Arkustangens ist die Umkehrfunktion des Tangens. Der Rückgabewert von `atan` ist eine Gleitkommazahl, wird im Bogenmaß angegeben und liegt im Bereich von $-\pi/2$ bis $+\pi/2$.

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

Die Funktion ist auch in `cmath` vorhanden.

math.atan2(y, x)

Berechnet $\text{atan}(y / x)$. Im Gegensatz zur `atan`-Funktion beachtet `atan2` die Vorzeichen der Parameter x und y und kann somit Winkel für alle Quadranten berechnen. Mithilfe der Funktion `atan2` lassen sich beispielsweise sehr elegant kartesische Koordinaten in Polarkoordinaten umrechnen.

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

math.cos(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.hypot(x, y)

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

$$\text{hypot}(x, y) = \sqrt{x^2 + y^2}$$

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

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

math.sin(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.tan(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

14.1.5 Winkelfunktionen

math.degrees(x)

Rechnet den Winkel x vom Bogenmaß in Grad um. Das Ergebnis ist immer eine Gleitkommazahl und wird nach der Formel $360x / 2\pi$ berechnet.

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

math.radians(x)

Rechnet den Winkel x von Grad ins Bogenmaß um. Das Ergebnis ist immer eine Gleitkommazahl und wird nach der Formel $2\pi \cdot x / 360$ berechnet.

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

14.1.6 Hyperbolische Funktionen

math.cosh(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.sinh(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

math.tanh(x)

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

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

Die Funktion ist auch in `cmath` vorhanden.

14.1.7 Funktionen aus `cmath`

In diesem Abschnitt werden die Funktionen aus `cmath` vorgestellt, die keine Entsprechung im Modul `math` haben.

phase(x)

Gibt die Phase (häufig auch Argument oder Winkel genannt) der komplexen Zahl x zurück.

polar(x)

Konvertiert die komplexe Zahl x in ihre Polardarstellung. Das Ergebnis ist ein Tupel, das den Radius r und den Winkel φ von x enthält.

rect(r, phi)

Das Gegenstück zu `polar`. Die Funktion `rect` konvertiert eine in Polardarstellung durch den Radius r und den Winkel ϕ gegebene komplexe Zahl in ihre kartesische Darstellung. Das Ergebnis wird als `complex`-Instanz zurückgegeben.

14.2 Zufallszahlengenerator – `random`

Das Modul `random` der Standardbibliothek erzeugt Pseudozufallszahlen und bietet zudem einige zusätzliche Funktionen, um zufallsgesteuerte Operationen auf Basisdatentypen anzuwenden.

Beachten Sie, dass das Modul `random` keine echten Zufallszahlen erzeugen kann, sondern sogenannte Pseudozufallszahlen. Echte Zufallszahlen sind für einen Computer nicht berechenbar. Ein Generator für Pseudozufallszahlen wird mit einer ganzen Zahl initialisiert und erzeugt aufgrund dieser Basis eine deterministische, aber scheinbar zufällige Abfolge von Pseudozufallszahlen. Diese Zahlenfolge wiederholt sich dabei nach einer gewissen Anzahl von erzeugten Zufallszahlen. Im Falle des in Python standardmäßig verwendeten Algorithmus beträgt diese Periode $2^{19937} - 1$ Zahlen.

Bevor Sie die Beispiele dieses Abschnitts ausprobieren können, müssen Sie selbstverständlich das Modul `random` einbinden:

```
>>> import random
```

Steuerungsfunktionen

random.seed([x])

Initialisiert den Zufallszahlengenerator mit der Instanz *x*. Wenn es sich bei *x* um eine ganze Zahl handelt, wird der Zufallszahlengenerator direkt mit dieser Zahl, ansonsten mit dem Hash-Wert der übergebenen Instanz initialisiert.

Wenn kein Parameter übergeben wird, wird der Zufallszahlengenerator mit der aktuellen Systemzeit initialisiert. Auf diese Weise können die erzeugten Zahlen als quasi-zufällig angesehen werden.

Wird der Zufallszahlengenerator zu unterschiedlichen Zeiten mit demselben Wert initialisiert, erzeugt er jeweils dieselbe Zahlenfolge.

random.getstate()

Die Funktion `getstate` gibt ein Tupel zurück, das den aktuellen Status des Zufallszahlengenerators beschreibt. Mithilfe der Funktion `setstate` lässt sich damit der Status des Generators speichern und zu einem späteren Zeitpunkt, beispielsweise nach zwischenzeitlicher Neuinitialisierung, wiederherstellen.

random.setstate()

Die Funktion `setstate` akzeptiert ein von `getstate` erzeugtes Tupel und überführt den Zufallszahlengenerator in den durch dieses Tupel beschriebenen Status.

```
>>> state = random.getstate()
>>> random.setstate(state)
```

random.getrandbits(k)

Erzeugt eine ganze Zahl, deren Bitfolge aus *k* zufälligen Bits besteht. Das Ergebnis ist, unabhängig von der verwendeten Bitzahl, immer eine Instanz des Datentyps `long`.

```
>>> random.getrandbits(8)
149L
>>> random.getrandbits(8)
187L
```

Funktionen für ganze Zahlen

random.randrange([start,]stop[, step])

Gibt ein zufällig gewähltes Element der Liste zurück, die ein Aufruf der Built-in Function `range` mit gleichen Parametern erzeugen würde. Das heißt, es wird eine Zufallszahl *n* zwischen *start* und *stop* erzeugt, für die gilt: $start + n \cdot step$.

```
>>> random.randrange(0, 50, 2)
40
```

random.randint(a, b)

Erzeugt eine zufällige, ganze Zahl n , so dass gilt: $a \leq n \leq b$.

```
>>> random.randint(0, 10)
2
>>> random.randint(0, 10)
7
```

Funktionen für Sequenzen**random.choice(seq)**

Gibt ein zufällig gewähltes Element der Sequenz *seq* zurück. Die übergebene Sequenz darf nicht leer sein.

```
>>> random.choice([1,2,3,4,5])
5
>>> random.choice([1,2,3,4,5])
2
```

Im Beispiel wurde der Einfachheit halber eine Liste mit ausschließlich numerischen Elementen verwendet. Dies muss nicht unbedingt sein, es darf ein beliebiger sequentieller Datentyp mit beliebigen Elementen übergeben werden.

random.shuffle(x[, random])

Die Funktion `shuffle` bringt die Elemente der Sequenz *x* in eine zufällige Reihenfolge. Beachten Sie, dass diese Funktion nicht seiteneffektfrei ist, sondern die übergebene Sequenz an sich bearbeitet wird. Aus diesem Grund dürfen für *x* auch nur Instanzen veränderlicher sequentieller Datentypen übergeben werden.

Als optionaler Parameter *random* kann ein Funktionsobjekt übergeben werden, das über die gleiche Schnittstelle verfügt wie die Funktion `random.random`, die später beschrieben wird. Durch Implementieren einer solchen Funktion ist es möglich, `shuffle` einen eigenen Zufallszahlengenerator vorzugeben.

```
>>> l = [1,2,3,4]
>>> random.shuffle(l)
>>> l
[1, 4, 3, 2]
```

random.sample(population, k)

Die Funktion `sample` bekommt eine Sequenz *population* und eine ganze Zahl *k* als Parameter übergeben. Das Ergebnis ist eine neue Liste mit *k* zufällig gewählten Elementen aus *population*. Auf diese Weise könnte beispielsweise eine gewisse Anzahl von Gewinnern aus einer Liste von Lotterieteilnehmern gezogen werden. Beachten Sie, dass auch die Reihenfolge der erzeugten Liste zufällig ist und die

Ziehungen bei mehrmaligem Funktionsaufruf mit Wiederholungen durchgeführt werden.

```
>>> pop = [1,2,3,4,5,6,7,8,9,10]
>>> random.sample(pop, 3)
[7, 8, 5]
>>> random.sample(pop, 3)
[5, 9, 7]
```

Die Funktion `sample` kann insbesondere auch in Kombination mit der Built-in Function `range` verwendet werden:

```
>>> random.sample(range(10000000), 3)
[4571575, 2648561, 2009814]
```

Spezielle Verteilungen

random.random()

Gibt die nächste Zufallszahl zurück. Der Rückgabewert ist eine Gleitkommazahl zwischen 0.0 und 1.0.

```
>>> random.random()
0.067300272273646655
>>> random.random()
0.52544342703734148
```

random.uniform(a, b)

Erzeugt eine gleichverteilte zufällige Gleitkommazahl n , so dass gilt: $a \leq n \leq b$.

```
>>> random.uniform(0.5, 0.6)
0.5618673220051662
```

random.betavariate(alpha, beta)

Erzeugt Zufallszahlen, die statistisch der Betaverteilung entsprechen. Die Parameter *alpha* und *beta* müssen numerische Werte größer als -1 sein, und der Rückgabewert liegt zwischen 0 und 1.

```
>>> random.betavariate(2.5, 1.0)
0.76494009914551264
```

random.expovariate(lambd)

Erzeugt Zufallszahlen, die statistisch der Exponentialverteilung entsprechen. Der Parameter `lambd` ist 1.0 geteilt durch das gewünschte arithmetische Mittel. Der Rückgabewert liegt zwischen 0 und positiv unendlich.

```
>>> random.expovariate(0.5)
0.85259287178065613
```

random.gammavariate(alpha, beta)

Erzeugt Zufallszahlen, die statistisch der Gammaverteilung entsprechen. Die Parameter *alpha* und *beta* müssen numerische Werte größer als 0 sein.

```
>>> random.gammavariate(1.3, 0.5)
1.1608977325106138
```

random.gauss(mu, sigma)

Erzeugt Zufallszahlen, die statistisch der Gauß-Verteilung entsprechen. Der Parameter *mu* entspricht dem arithmetischen Mittel und *sigma* der Standardabweichung.

```
>>> random.gauss(0.5, 1.9)
1.0084579933596225
```

random.lognormvariate(mu, sigma)

Erzeugt Zufallszahlen, die statistisch der logarithmischen Normalverteilung entsprechen. Der Parameter *mu* entspricht dem arithmetischen Mittel und *sigma* der Standardabweichung.

```
>>> random.lognormvariate(0.5, 1.9)
0.25625006871810202
```

random.normalvariate(mu, sigma)

Erzeugt Zufallszahlen, die statistisch der Normal- oder Gauß-Verteilung entsprechen. Der Parameter *mu* entspricht dem arithmetischen Mittel und *sigma* der Standardabweichung. Die Funktion ist damit äquivalent zu `gauss`.

```
>>> random.normalvariate(0.5, 1.9)
1.9176550196262139
```

random.vonmisesvariate(mu, kappa)

Erzeugt Zufallszahlen, die statistisch der Von-Mises-Verteilung entsprechen. Der Parameter *mu* entspricht dem mittleren Winkel in Radiant und *kappa* dem Konzentrationsparameter, der größer oder gleich 0 sein muss.

```
>>> random.vonmisesvariate(0.5, 1.9)
2.0502913847498458
```

random.paretovariate(alpha)

Erzeugt Zufallszahlen, die statistisch der Pareto-Verteilung entsprechen.

```
>>> random.paretovariate(0.5)
43.528372368738189
```

random.weibullvariate(alpha, beta)

Erzeugt Zufallszahlen, die statistisch der Weibull-Verteilung entsprechen.

```
>>> random.weibullvariate(0.5, 1.9)
0.23610339261628124
```

Alternative Generatoren**random.SystemRandom([seed])**

Das Modul `random` enthält zusätzlich zu den oben erläuterten Funktionen eine Klasse namens `SystemRandom`, die es ermöglicht, den Zufallszahlengenerator des Betriebssystems zu verwenden statt des Python-eigenen. Beachten Sie, dass diese Klasse nicht auf allen, aber auf den gängigsten Betriebssystemen existiert.

Beim Instantiieren der Klasse kann eine Zahl oder Instanz zur Initialisierung des Zufallszahlengenerators übergeben werden. Danach lässt sich die Klasse `SystemRandom` wie das Modul `random` verwenden, da sie die meisten im Modul enthaltenen Funktionen als Methode implementiert.

Beachten Sie jedoch, dass nicht die komplette Funktionalität von `random` in `SystemRandom` zur Verfügung steht. So wird ein Aufruf der Methode `seed` ignoriert, während Aufrufe der Methoden `getstate` und `setstate` eine `NotImplementedError`-Exception werfen.

```
>>> sr = random.SystemRandom()
>>> sr.randint(1, 10)
9
```

14.3 Präzise Dezimalzahlen – decimal

Sicherlich erinnern Sie sich noch an folgendes Beispiel, das zeigen sollte, dass der eingebaute Datentyp `float` nicht unendlich präzise ist:

```
>>> 0.9
0.90000000000000002
```

Das liegt daran, dass nicht jede Dezimalzahl durch das interne Speichermodell von `float` dargestellt werden kann, sondern nur mit einer gewissen Genauigkeit angenähert wird. Diese Einschränkung wird jedoch aus Gründen der Effizienz in Kauf genommen. Als wir über Gleitkommazahlen gesprochen haben, wurde Abhilfe durch ein Modul versprochen, und dieses Modul heißt `decimal`. Es muss aber noch einmal deutlich darauf hingewiesen werden, dass diese Abhilfe auf Kosten der Performance geht.

Das Modul `decimal` enthält im Wesentlichen den Datentyp `Decimal`, der Dezimalzahlen mit einer beliebigen Präzision speichern und verarbeiten kann. In diesem Abschnitt möchten wir Sie in die Verwendung des Datentyps einführen, die sich an die Verwendung der vorhandenen numerischen Datentypen anlehnt. Um die Beispiele auszuführen, müssen Sie den Datentyp zuerst einbinden:

```
>>> from decimal import Decimal
```

Hinweis

Das hier besprochene Modul `decimal` folgt in seiner Funktionsweise der *General Decimal Arithmetic Specification* von IBM. Aus diesem Grund ist es möglich, dass Ihnen ein ähnliches Modul bereits von einer anderen Programmiersprache her bekannt ist.

Es existieren beispielsweise Bibliotheken, die das `decimal`-Modul in gleicher oder abgewandelter Form für C, C++, Java oder Perl implementieren.

14.3.1 Verwendung des Datentyps

Es existiert kein Literal, mit dem Sie Instanzen des Datentyps `Decimal` direkt erzeugen könnten, wie es beispielsweise bei `float` der Fall ist. Um eine `Decimal`-Instanz mit einem bestimmten Wert zu erzeugen, müssen Sie den Datentyp explizit instantiieren. Den Wert können Sie dem Konstruktor in Form eines Strings übergeben:

```
>>> Decimal("0.9")
Decimal("0.9")
>>> Decimal("1.33e7")
Decimal("1.33E+7")
```

Dies ist die geläufigste Art, `Decimal` zu instantiieren. Es ist außerdem möglich, dem Konstruktor eine ganze Zahl oder ein Tupel zu übergeben:

```
>>> Decimal(123)
Decimal("123")
>>> Decimal((0, (3, 1, 4, 1), -3))
Decimal("3.141")
```

Im zweiten Fall bestimmt das erste Element des Tupels das Vorzeichen, wobei 0 für eine positive und 1 für eine negative Zahl steht. Das zweite Element muss ein weiteres Tupel sein, das alle Ziffern der Zahl enthält. Das dritte Element des Tupels entspricht dem Exponenten der zuvor angegebenen Zahl.

Beachten Sie, dass es ausdrücklich nicht möglich ist, bei der Instantiierung eine Gleitkommazahl direkt zu übergeben, da sich sonst die Ungenauigkeiten von `float` auf den Datentyp `Decimal` übertragen würden.

Sobald eine `Decimal`-Instanz erzeugt wurde, kann sie wie eine Instanz eines bekannten numerischen Datentyps verwendet werden. Das bedeutet insbesondere, dass alle von diesen Datentypen her bekannten Operatoren auch für `Decimal` definiert sind. Es ist zudem möglich, `Decimal` in Operationen mit anderen numerischen Datentypen zu verwenden. Kurzum: `Decimal` passt sich nahezu perfekt in die bestehende Welt der numerischen Datentypen ein.

```
>>> Decimal("0.9") * 5
Decimal("4.5")
>>> Decimal("0.9") / 10
Decimal("0.09")
>>> Decimal("0.9") % Decimal("1.0")
Decimal("0.9")
```

Eine Besonderheit des Datentyps ist es, abschließende Nullen beim Nachkommaanteil einer Dezimalzahl beizubehalten, obwohl diese eigentlich überflüssig sind. Das ist beispielsweise beim Rechnen mit Geldbeträgen von Nutzen:

```
>>> Decimal("2.50") + Decimal("4.20")
Decimal("6.70")
```

Ein `Decimal`-Wert lässt sich in einen Wert eines beliebigen anderen numerischen Datentyps überführen. Beachten Sie, dass solche Konvertierungen im Falle von `Decimal` in der Regel verlustbehaftet sind, der Wert also an Genauigkeit verliert.

```
>>> float(Decimal("1.337"))
1.337
>>> float(Decimal("0.9"))
0.90000000000000002
>>> int(Decimal("1.337"))
1
```

Diese Eigenschaft ermöglicht es, `Decimal`-Instanzen ganz selbstverständlich als Parameter von beispielsweise Built-in Functions oder Funktionen der Bibliothek `math` zu übergeben:

```
>>> import math
>>> math.sqrt(Decimal("2"))
1.4142135623730951
```

Beachten Sie dabei, dass von diesen Funktionen auch in einem solchen Fall niemals eine `Decimal`-Instanz zurückgegeben wird. Unter Verwendung des Moduls `math` laufen Sie also Gefahr, durch den `float`-Rückgabewert an Genauigkeit zu verlieren.

Diese Beschränkung lässt sich bei vielen mathematischen Operationen durch Verwendung der entsprechenden Operatoren umgehen, da diese das Ergebnis in

jedem Fall als `Decimal`-Instanz zurückgeben. Für einige mathematische Funktionen stellt eine `Decimal`-Instanz spezielle Methoden bereit. Jede dieser Methoden erlaubt es, neben ihren spezifischen Parametern ein sogenanntes `Context`-Objekt zu übergeben. Ein solches `Context`-Objekt beschreibt den Kontext, in dem die Berechnungen durchgeführt werden sollen, beispielsweise also auf wie viele Nachkommastellen genau gerundet werden soll. Näheres zum `Context`-Objekt erfahren Sie weiter hinten in diesem Abschnitt.

Die wichtigsten Methoden einer `Decimal`-Instanz `d` lauten:

Methode	Bedeutung
<code>d.exp([context])</code>	e^d
<code>d.fma(other, third[, context])</code>	$d \cdot other + third^1$
<code>d.ln([context])</code>	$\log_e(d)$
<code>d.log10([context])</code>	$\log_{10}(d)$
<code>d.sqrt([context])</code>	$\sqrt{d}$

Tabelle 14.1 Mathematische Methoden des Datentyps `Decimal`

Die Verwendung dieser Methoden demonstriert das folgende Beispiel:

```
>>> d = Decimal("9")
>>> d.sqrt()
Decimal('3')
>>> d.ln()
Decimal('2.197224577336219382790490474')
>>> d.fma(2, -7)
Decimal('11')
```

Tipp

Das Programmieren mit dem Datentyp `Decimal` ist mit viel Schreibarbeit verbunden, da kein Literal für diesen Datentyp existiert. Viele Python-Programmierer behelfen sich damit, dem Datentyp einen kürzeren Namen zu verpassen:

```
>>> from decimal import Decimal as D
>>> D("1.5e-7")
Decimal("1.5E-7")
```

1 Der Vorteil dieser Methode ist, dass sie die Berechnung »in einem Guss« durchführt, dass also nicht mit einem gerundeten Zwischenergebnis der Multiplikation weitergerechnet wird.

14.3.2 Nichtnumerische Werte

Aus Abschnitt 8.3.2, »Gleitkommazahlen – float«, kennen Sie bereits die Werte `nan` und `inf` des Datentyps `float`, die immer dann auftraten, wenn eine Berechnung nicht möglich war bzw. eine Zahl den Zahlenraum von `float` sprengte. Selbst konnten Sie diese Werte allerdings nicht vergeben.

Der Datentyp `Decimal` baut auf diesem Ansatz auf und ermöglicht es Ihnen zudem, `Decimal`-Instanzen mit einem solchen Zustand zu initialisieren. Folgende Werte sind möglich:

Wert	Bedeutung
<code>Infinity</code> , <code>Inf</code>	positiv unendlich
<code>-Infinity</code> , <code>-Inf</code>	negativ unendlich
<code>NaN</code>	ungültiger Wert (»Not a Number«)
<code>sNaN</code>	ungültiger Wert (»signaling Not a Number«) Der Unterschied zu <code>NaN</code> besteht darin, dass eine Exception geworfen wird, sobald versucht wird, mit <code>sNaN</code> weiterzurechnen. Rechenoperationen mit <code>NaN</code> funktionieren anstandslos, ergeben allerdings immer wieder <code>NaN</code> .

Tabelle 14.2 Nichtnumerische Werte des Datentyps `Decimal`

Diese nichtnumerischen Werte können wie Zahlen verwendet werden:

```
>>> Decimal("NaN") + Decimal("42.42")
Decimal("NaN")
>>> Decimal("Infinity") + Decimal("Infinity")
Decimal("Infinity")
>>> Decimal("sNaN") + Decimal("42.42")
Traceback (most recent call last):
[...]
decimal.InvalidOperation: sNaN
>>> Decimal("Inf") - Decimal("Inf")
Traceback (most recent call last):
[...]
decimal.InvalidOperation: -INF + INF
```

14.3.3 Das Context-Objekt

Eingangs wurde erwähnt, dass es der Datentyp `Decimal` erlaubt, Dezimalzahlen mit beliebiger Genauigkeit zu speichern. Die Genauigkeit, das heißt die Anzahl der Nachkommastellen, ist eine von mehreren globalen Einstellungen, die innerhalb eines sogenannten `Context`-Objekts gekapselt werden.

Um auf den aktuellen Kontext der arithmetischen Operationen zugreifen zu können, existieren innerhalb des Moduls `decimal` die Funktionen `getcontext` und `setcontext`.

An dieser Stelle möchten wir nur auf drei Attribute des `Context`-Objekts eingehen, die die Berechnungen beeinflussen können:

prec

Das Attribut `prec` (für »precision«) ermöglicht es, die Genauigkeit der `Decimal`-Instanzen des aktuellen Kontextes zu bestimmen. Der Wert versteht sich als Anzahl der zu berechnenden Nachkommastellen und ist eine ganze Zahl.

```
>>> c = decimal.getcontext()
>>> c.prec = 3
>>> Decimal("1.23456789") * Decimal("2.3456789")
Decimal("2.90")
```

Emin, Emax

Die Attribute `Emin` und `Emax` ermöglichen es, die maximale bzw. minimale Größe des Exponenten festzulegen. Beide müssen eine ganze Zahl referenzieren. Wenn das Ergebnis einer Berechnung dieses Limit überschreitet, wird eine Exception geworfen.

```
>>> c = decimal.getcontext()
>>> c.Emax = 9
>>> Decimal("1e100") * Decimal("1e100")
Traceback (most recent call last):
[...]
decimal.Overflow: above Emax
```

Dieser Abschnitt kann allenfalls als grundlegende Einführung in das Modul `decimal` verstanden werden, denn dieses Modul bietet noch viele weitere Möglichkeiten, Berechnungen anzustellen oder Ergebnisse dieser Berechnungen genau an die eigenen Bedürfnisse anzupassen. Sollte also Ihr Interesse an diesem Modul geweckt worden sein, fühlen Sie sich dazu ermutigt, insbesondere in der Python-Dokumentation nach weiteren Verwendungswegen zu forschen.

Beachten Sie aber, dass üblicherweise kein Bedarf an solch präzisen Berechnungen besteht, wie sie der Datentyp `Decimal` ermöglicht. Der Geschwindigkeitsvorteil von `float` wiegt in der Regel schwerer als der Genauigkeitsgewinn von `Decimal`.

14.4 Spezielle Generatoren – itertools

An dieser Stelle möchten wir Ihnen das Modul `itertools` der Standardbibliothek vorstellen, das eine Reihe von Generatorfunktionen enthält, die man im Programmieralltag immer wieder benötigt und sich sonst selbst schreiben müsste. So ist es mit `itertools` beispielsweise möglich, über alle Kombinationen oder Permutationen aus Elementen einer gegebenen Liste zu iterieren. Dies rechtfertigt auch die Einordnung von `itertools` in der Kategorie »Mathematik«.

Im Folgenden sollen die wichtigsten der in `itertools` enthaltenen Generatoren vorgestellt werden. Um die Beispiele nachvollziehen zu können, müssen Sie zuvor natürlich das Modul `itertools` importiert haben.

Beachten Sie, dass die von den Generatorfunktionen zurückgegebenen Iteratoren in den folgenden Beispielen zur Verdeutlichung des Prinzips mittels `list` in eine Liste überführt und ausgegeben werden. In der Praxis durchläuft man die von den `itertools`-Generatoren erzeugten Iteratoren üblicherweise mit einer `for`-Schleife.

chain(*iterables)

Die Funktion `chain` erzeugt einen Iterator, der der Reihe nach alle Elemente der übergebenen iterierbaren Objekte durchläuft:

```
>>> list(itertools.chain("ABC", "DEF"))
['A', 'B', 'C', 'D', 'E', 'F']
```

Sie sehen, dass zuerst die Elemente des ersten und dann die Elemente des zweiten übergebenen Strings durchlaufen werden.

In einigen Fällen ist es ungünstig, die iterierbaren Objekte einzeln als Parameter zu übergeben. Dafür gibt es die Funktion `chain.from_iterable`, die eine Sequenz von iterierbaren Objekten als einzigen Parameter erwartet:

```
>>> list(itertools.chain.from_iterable(["ABC", "DEF", "GHI"]))
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
```

Abgesehen von der Parameterfrage, sind die beiden Funktionen äquivalent.

combinations(iterable, r)

Durchläuft alle r -elementigen Kombinationen aus *iterable*. Bei einer Kombination wird nicht auf die Reihenfolge der zusammengestellten Elemente geachtet. Das Vertauschen von Elementen einer Kombination führt also nicht zu einer neuen Kombination. Im folgenden Beispiel sollen alle 4-stelligen Kombinationen aus den Zahlen von 0 bis 4 durchlaufen werden:

```
>>> list(itertools.combinations(range(5), 4))
[(0, 1, 2, 3), (0, 1, 2, 4), (0, 1, 3, 4),
 (0, 2, 3, 4), (1, 2, 3, 4)]
```

Sie sehen, dass die Anordnung (4, 1, 0, 2) nicht aufgeführt ist, da sie sich nur durch Vertauschung der Elemente aus der Kombination (0, 1, 2, 4) ergibt.

Anhand des nächsten Beispiels sehen Sie, dass die bestimmten Kombinationen von der Reihenfolge der Elemente in *iterable* abhängen:

```
>>> list(itertools.combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations("CBA", 2))
[('C', 'B'), ('C', 'A'), ('B', 'A')]
```

Wenn Sie an einem Generator interessiert sind, der auf die Reihenfolge der Elemente achtet, möchten Sie alle Permutationen durchlaufen. In diesem Fall ist die Funktion `permutations` die beste Wahl.

count([n])

Erzeugt einen Iterator, der beginnend mit n alle ganzen Zahlen der Reihe nach durchläuft. Der Parameter n ist mit 0 vorbelegt. Beachten Sie, dass dieser Iterator von selbst nicht aufhört zu zählen und Sie Gefahr laufen, Endlosschleifen zu produzieren, wenn Sie `count` unbedacht verwenden.

```
>>> for i in itertools.count(-5):
...     print(i)
...     if i >= 0:
...         break
...
-5
-4
-3
-2
-1
0
```

Interessant ist `count` auch in Verbindung mit der Built-in Function `map`. Dies soll anhand des folgenden Beispiels demonstriert werden, das die Quadratzahlen zwischen 0 und 30 ausgibt:

```
m = map(lambda x: x**2, itertools.count())
for i in m:
    if i > 30:
        break
    print(i)
```

cycle(iterable)

Durchläuft alle Elemente des iterierbaren Objekts *iterable* und fängt danach wieder von vorn an.

Beachten Sie, dass sich die Funktion `cycle` intern eine Kopie jedes Elements von *iterable* anlegt und diese beim erneuten Durchlaufen verwendet. Das hat je nach Größe von *iterable* einen signifikanten Speicherverbrauch zur Folge.

dropwhile(predicate, iterable)

Die Funktion `dropwhile` bekommt ein iterierbares Objekt *iterable* und eine Funktion *predicate* übergeben. Sie ruft zunächst für alle Elemente von *iterable* die Funktion *predicate* auf und übergeht jedes Element, für das *predicate* `True` zurückgegeben hat. Nachdem *predicate* zum ersten Mal `False` zurückgegeben hat, wird jedes nachfolgende Element von *iterable* durchlaufen, unabhängig davon, was *predicate* für dieses Element zurückgibt. Dies soll an einem Beispiel erläutert werden:

```
>>> p = lambda x: x.islower()
>>> list(itertools.dropwhile(p, "abcdefghijklmnopqrStuvWXYZ"))
['H', 'I', 'J', 'K', 'L', 'M', 'n', 'o', 'p', 'q', 'r', 's', 't',
 'u', 'v', 'w', 'x', 'y', 'z']
```

Im Beispiel sollen alle Buchstaben nach den Kleinbuchstaben am Anfang in die Ergebnisliste aufgenommen werden. Sie sehen, dass auch Kleinbuchstaben im Ergebnis enthalten sind, nachdem die Prädikatfunktion `p` zum ersten Mal `True` zurückgegeben hat.

filterfalse(predicate, iterable)

Durchläuft alle Elemente von *iterable*, für die die Funktion *predicate* `False` zurückgibt. Ein Aufruf von `filterfalse` ist damit äquivalent zur folgenden Generator Expression:

```
(x for x in iterable if not predicate(x))
```

Im folgenden Beispiel sollen nur die Großbuchstaben eines Strings durchlaufen werden:

```
>>> p = lambda x: x.islower()
>>> list(itertools.filterfalse(p, "abcDEFghiJkLmnoP"))
['D', 'E', 'F', 'J', 'K', 'L', 'P']

>>> list((x for x in "abcDEFghiJkLmnoP" if not p(x)))
['D', 'E', 'F', 'J', 'K', 'L', 'P']
```


islice(iterable[, start], stop[, step])

Die Funktion `islice` bildet das Slicing, das Sie von den sequentiellen Datentypen her kennen, auf beliebige iterierbare Objekte ab. Die Funktion erzeugt dabei einen Iterator, der bei dem Element mit der laufenden Nummer *start* beginnt, vor dem Element mit der Nummer *stop* aufhört und in jedem Schritt um *step* Elemente weiterspringt:

```
>>> list(itertools.islice("ABCDEFGHIJKL", 2, 8, 2))
['C', 'E', 'G']

>>> "ABCDEFGHIJKL"[2:8:2]
'CEG'
```

permutations(iterable[, r])

Erzeugt einen Iterator über alle *r*-stelligen Permutationen aus Elementen des iterierbaren Objekts *iterable*.

```
>>> list(itertools.permutations(range(3), 2))
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

Wie Sie sehen, sind die Anordnungen (0,1) und (1,0) beide in der Ergebnisliste enthalten. Bei Permutationen kommt es im Gegensatz zu den Kombinationen auf die Reihenfolge der Anordnung an.

```
>>> list(itertools.permutations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'),
 ('B', 'C'), ('C', 'A'), ('C', 'B')]

>>> list(itertools.permutations("CBA", 2))
[('C', 'B'), ('C', 'A'), ('B', 'C'),
 ('B', 'A'), ('A', 'C'), ('A', 'B')]
```

Dieses Beispiel zeigt, dass auch hier die Reihenfolge der Permutationen in der Ergebnisliste von der Reihenfolge der zu permutierenden Elemente in *iterable* abhängt.

product(*iterables[, repeat])

Erzeugt einen Iterator, der das sogenannte kartesische Produkt der übergebenen iterierbaren Objekte durchläuft. Das Bilden des kartesischen Produkts kommt dem Bilden aller Tupel aus je einem Element eines jeden übergebenen iterierbaren Objektes gleich. Dabei steht ein Element in dem Tupel genau an der Stelle, an der auch das iterierbare Objekt in der Parameterliste steht, aus dem es stammt. Dies soll an folgendem Beispiel veranschaulicht werden:

```
>>> list(itertools.product("ABC", [1,2]))
[('A', 1), ('A', 2), ('B', 1), ('B', 2), ('C', 1), ('C', 2)]
```

Hier wurde jedes Zeichen aus dem String "ABC" einmal mit allen Elementen der Liste [1,2] in Verbindung gebracht.

Über den optionalen Schlüsselwortparameter *repeat* kann ein iterierbares Objekt beispielsweise mehrmals mit sich selbst »multipliziert« werden, ohne dass Sie es der Funktion mehrfach übergeben müssten:

```
>>> list(itertools.product("AB", "AB", "AB"))
[('A', 'A', 'A'), ('A', 'A', 'B'), ('A', 'B', 'A'),
 ('A', 'B', 'B'), ('B', 'A', 'A'), ('B', 'A', 'B'),
 ('B', 'B', 'A'), ('B', 'B', 'B')]

>>> list(itertools.product("AB", repeat=3))
[('A', 'A', 'A'), ('A', 'A', 'B'), ('A', 'B', 'A'),
 ('A', 'B', 'B'), ('B', 'A', 'A'), ('B', 'A', 'B'),
 ('B', 'B', 'A'), ('B', 'B', 'B')]
```

repeat(object[, times])

Erzeugt einen Iterator, der nur das Objekt *object* zurückgibt, dies aber fortwährend. Optional können Sie über den Parameter *times* festlegen, wie viele Iterationsschritte durchgeführt werden sollen:

```
>>> list(itertools.repeat("A", 10))
['A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A', 'A']
```

takewhile(predicate, iterable)

Die Funktion *takewhile* ist das Gegenstück zu *dropwhile*. Sie erzeugt einen Iterator, der so lange die Elemente von *iterable* durchläuft, wie die Funktion *predicate* für die Elemente *True* zurückgibt. Sobald ein *predicate*-Aufruf *False* ergeben hat, bricht der Iterator ab.

```
>>> p = lambda x: x.islower()
>>> list(itertools.takewhile(p, "abcdefGHIjkLMNOp"))
['a', 'b', 'c', 'd', 'e', 'f']
```

In diesem Fall wurde *takewhile* verwendet, um nur die Kleinbuchstaben am Anfang des übergebenen Strings zu durchlaufen.

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

15 Strings

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

15.1 Reguläre Ausdrücke – re

Das Modul `re` der Standardbibliothek bietet umfangreiche Möglichkeiten zum Arbeiten mit sogenannten *regulären Ausdrücken* (engl. *regular expressions*). In einem solchen regulären Ausdruck wird durch eine spezielle Syntax ein Textmuster beschrieben, das dann auf verschiedene Texte oder Textfragmente angewendet werden kann. Grundsätzlich gibt es zwei große Anwendungsbereiche von regulären Ausdrücken.

Im ersten Bereich, beim sogenannten *Matching*, wird geprüft, ob ein Textabschnitt auf das Muster des regulären Ausdrucks passt oder nicht. Ein häufiges Beispiel für Matching ist ein Test, ob eine eingegebene E-Mail-Adresse syntaktisch gültig ist.

Die zweite Einsatzmöglichkeit von regulären Ausdrücken ist das sogenannte *Searching*, bei dem innerhalb eines größeren Textes nach Textfragmenten gesucht wird, die auf einen regulären Ausdruck passen. Es handelt sich dabei um eine eigene Disziplin, da dieses Verhalten vom Programmierer selbst nicht effizient durch Einsatz des Matchings implementiert werden kann. Ein Anwendungsbeispiel ist der Syntax Highlighter Ihrer Python-Umgebung, der durch Searching nach speziellen Codeabschnitten wie Schlüsselwörtern oder Strings sucht, um diese grafisch hervorzuheben.

Ein regulärer Ausdruck ist in Python ein String, der die entsprechenden Regeln enthält. Im Gegensatz zu manch anderen Programmiersprachen existiert hier kein eigenes Literal zu diesem Zweck. Sollten Sie sich mit regulären Ausdrücken

bereits auskennen, sind Sie vielleicht gerade auf ein Problem aufmerksam geworden, denn der Backslash ist ein sehr wichtiges Zeichen zur Beschreibung regulärer Ausdrücke, und ausgerechnet dieses Zeichen trägt innerhalb eines Strings bereits eine Bedeutung: Normalerweise leitet ein Backslash eine Escape-Sequenz ein. Sie können nun entweder immer die Escape-Sequenz für einen Backslash ("`\\`") verwenden oder, was empfehlenswerter ist, auf Pythons Raw-Strings zurückgreifen, in denen keine Escape-Sequenzen möglich sind. Zur Erinnerung: Raw-Strings werden in Python durch ein vorangestelltes `r` gekennzeichnet:

```
r"Hallo Welt"
```

Im Folgenden möchten wir Sie in die komplexe Syntax regulärer Ausdrücke einweihen. Allein zu diesem Thema sind bereits ganze Bücher erschienen, weswegen die Beschreibung hier vergleichsweise knapp, aber grundlegend ausfallen soll. Es gibt verschiedene Notationen zur Beschreibung regulärer Ausdrücke. Python hält sich an die Syntax, die in der Programmiersprache Perl verwendet wird.

15.1.1 Syntax regulärer Ausdrücke

Grundsätzlich ist der String

```
r"python"
```

bereits ein regulärer Ausdruck. Dieser würde exakt auf den String `"python"` passen. Diese direkt angegebenen einzelnen Buchstaben werden *Zeichenliterale* genannt. Beachten Sie unbedingt, dass Zeichenliterale innerhalb regulärer Ausdrücke *case sensitive* sind, das heißt, dass der obige Ausdruck nicht auf den String `"Python"` passen würde.

In regulären Ausdrücken können eine ganze Reihe von Steuerungszeichen verwendet werden, die den Ausdruck flexibler und mächtiger machen. Diese sollen im Folgenden besprochen werden.

Beliebige Zeichen

Die einfachste Verallgemeinerung, die innerhalb eines regulären Ausdrucks verwendet werden kann, ist die Kennzeichnung eines beliebigen Zeichens durch einen Punkt. So passt der Ausdruck

```
r".ython"
```

sowohl auf `"python"` und `"Python"` als auch auf `"Jython"`, nicht jedoch auf `"Blython"`, da es sich nur um ein einzelnes beliebiges Zeichen handelt. Ein durch einen Punkt gekennzeichnetes beliebiges Zeichen darf nicht weggelassen werden. Der obige Ausdruck würde demzufolge nicht auf `"ython"` passen.

Zeichenklassen

Abgesehen davon, ein Zeichen ausdrücklich als beliebig zu kennzeichnen, ist es auch möglich, eine Klasse von Zeichen vorzugeben, die an dieser Stelle vorkommen dürfen. Dazu werden die gültigen Zeichen in eckige Klammern an die entsprechende Position geschrieben:

```
r"[jp]ython"
```

Dieser reguläre Ausdruck arbeitet ähnlich wie der des letzten Abschnitts, lässt jedoch nur die Buchstaben `j` und `p` als erstes Zeichen des Wortes zu. Damit passt der Ausdruck sowohl auf `"jython"` als auch auf `"python"`, jedoch nicht auf `"Python"`, `"jpython"` oder `"ython"`. Um auch die jeweiligen Großbuchstaben im Wort zu erlauben, können Sie den Ausdruck folgendermaßen erweitern:

```
r"[jJpP]ython"
```

Innerhalb einer Zeichenklasse ist es ebenfalls möglich, ganze Bereiche von Zeichen zuzulassen. Dadurch wird folgende Syntax verwendet:

```
r"[A-Z]ython"
```

Dieser reguläre Ausdruck lässt jeden Großbuchstaben als Anfangsbuchstaben des Wortes durch, beispielsweise aber keinen Kleinbuchstaben und keine Zahl. Um mehrere Bereiche zuzulassen, schreiben Sie diese ganz einfach hintereinander:

```
r"[A-Ra-r]ython"
```

Dieser reguläre Ausdruck passt beispielsweise sowohl auf `"Qython"` als auch auf `"qython"`, nicht aber auf `"Sython"` oder `"3ython"`.

Auch Ziffernbereiche können als Zeichenklasse verwendet werden:

```
r"[0-9]ython"
```

Als letzte Möglichkeit, die eine Zeichengruppe bietet, können Zeichen oder Zeichenbereiche ausgeschlossen werden. Dazu wird zu Beginn der Zeichengruppe ein Zirkumflex (^) geschrieben. So erlaubt der reguläre Ausdruck

```
r"[^p]ython"
```

jedes Zeichen, abgesehen von einem großen oder kleinen »P«. Demzufolge würden sowohl `"Sython"` als auch `"wython"` passen, während `"Python"` und `"python"` außen vor bleiben.

Beachten Sie, dass es innerhalb einer Zeichenklasse, abgesehen vom Bindestrich und dem Zirkumflex, keine Zeichen mit spezieller Bedeutung gibt. Das heißt insbesondere, dass ein Punkt in einer Zeichenklasse tatsächlich das Zeichen `.` bedeutet und nicht etwa ein beliebiges Zeichen.

Quantoren

Bisher können wir in einem regulären Ausdruck bestimmte Regeln für einzelne Zeichen aufstellen. Wir stünden allerdings vor einem Problem, wenn wir an einer bestimmten Stelle des Wortes eine gewisse Anzahl oder gar beliebig viele dieser Zeichen erlauben wollten. Für diesen Zweck werden sogenannte *Quantoren* eingesetzt. Das sind spezielle Zeichen, die hinter ein einzelnes Zeichenliteral oder eine Zeichenklasse geschrieben werden und kennzeichnen, wie oft diese auftreten dürfen. Die folgende Tabelle listet alle Quantoren auf und erläutert kurz ihre Bedeutung. Danach werden wir Beispiele für die Verwendung von Quantoren bringen.

Quantor	Bedeutung
?	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse darf entweder keinmal oder einmal vorkommen.
*	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse darf beliebig oft hintereinander vorkommen, das heißt unter anderem, dass sie auch weggelassen werden kann.
+	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse darf beliebig oft hintereinander vorkommen, mindestens aber einmal. Sie darf also nicht weggelassen werden.

Tabelle 15.1 Quantoren in regulären Ausdrücken

Die folgenden drei Beispiele zeigen einen regulären Ausdruck mit je einem Quantor. Nachfolgend soll besprochen werden, wie sich Quantoren auf die Bedeutung des Ausdrucks auswirken.

- ▶ `r"P[Yy]?thon"`
Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes ein höchstens einmaliges Auftreten des großen oder kleinen »Y«. Damit passt der Ausdruck auf die Wörter "Python" und "Pthon", beispielsweise jedoch nicht auf "Pyython".
- ▶ `r"P[Yy]*thon"`
Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes ein beliebig häufiges Auftreten des großen oder kleinen »Y«. Damit passt der Ausdruck auf die Wörter "Python", "Pthon" und "PyYYYYyython", beispielsweise jedoch nicht auf "Pzthon".
- ▶ `r"P[Yy]+thon"`
Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes ein mindestens einmaliges Auftreten des großen oder kleinen »Y«. Damit passt der

Ausdruck auf die Wörter "Python", "PYthon" und "PyYYYYython", beispielsweise jedoch nicht auf "Pthon".

Neben diesen allgemeinen Quantoren gibt es eine Syntax, die es ermöglicht, exakt anzugeben, wie viele Wiederholungen einer Zeichengruppe erlaubt sind. Dabei werden die Unter- und Obergrenzen für Wiederholungen in geschweifte Klammern hinter das entsprechende Zeichen bzw. die entsprechende Zeichengruppe geschrieben. Die folgende Tabelle listet die Möglichkeiten der Notation auf:

Quantor	Bedeutung
{anz}	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse muss exakt <code>anz</code> -mal vorkommen.
{min,}	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse muss mindestens <code>min</code> -mal vorkommen.
{,max}	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse darf maximal <code>max</code> -mal vorkommen.
{min,max}	Das vorangegangene Zeichen bzw. die vorangegangene Zeichenklasse muss mindestens <code>min</code> -mal und darf maximal <code>max</code> -mal vorkommen.

Tabelle 15.2 Quantoren in regulären Ausdrücken

Auch für diese Quantoren möchten wir das bisherige Beispiel abändern und untersuchen, was sie für Auswirkungen haben.

- ▶ `r"P[Yy]{2}thon"`
Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes exakt zwei jeweils große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter "Pypython" oder "PYython", beispielsweise jedoch nicht auf "Pyyython".
- ▶ `r"P[Yy]{2,}thon"`
Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes mindestens zwei jeweils große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter "Pypython", "PYython" und "PyYYYYython", beispielsweise jedoch nicht auf "Python".
- ▶ `r"P[Yy]{,2}thon"`
Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes maximal zwei jeweils große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter "Python", "Pthon" und "PYython", beispielsweise jedoch nicht auf "Pyyython".

► `r"P[Yy]{1,2}thon"`

Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes mindestens ein und maximal zwei große oder kleine »Y«. Damit passt der Ausdruck auf die Wörter "Python" oder "PYython", beispielsweise jedoch nicht auf "Pthon" oder "PYYYYthon".

Vordefinierte Zeichenklassen

Damit Sie nicht bei jedem regulären Ausdruck das Rad neu erfunden müssen, existiert eine Reihe von vordefinierten Zeichenklassen, die beispielsweise alle Ziffern oder alle alphanumerischen Zeichen umfassen. Diese Zeichenklassen werden bei der Arbeit mit regulären Ausdrücken sehr häufig benötigt und können deswegen durch einen speziellen Code abgekürzt werden. Jeder dieser Codes beginnt mit einem Backslash. Die folgende Tabelle listet alle vordefinierten Zeichenklassen mit ihren Bedeutungen auf.

Zeichenklasse	Bedeutung
<code>\d</code>	Passt auf alle Zeichen, die Ziffern des Dezimalsystems sind. Äquivalent zu <code>[0-9]</code> .
<code>\D</code>	Passt auf alle Zeichen, die nicht Ziffern des Dezimalsystems sind. Äquivalent zu <code>[^0-9]</code> .
<code>\s</code>	Passt auf alle Whitespace-Zeichen. Äquivalent zu <code>[\t\n\r\f\v]</code> .
<code>\S</code>	Passt auf alle Zeichen, die kein Whitespace sind. Äquivalent zu <code>[^\t\n\r\f\v]</code> .
<code>\w</code>	Passt auf alle alphanumerischen Zeichen und den Unterstrich. Äquivalent zu <code>[a-zA-z0-9_]</code> .
<code>\W</code>	Passt auf alle Zeichen, die nicht alphanumerisch und kein Unterstrich sind. Äquivalent zu <code>[^a-zA-Z0-9_]</code> .

Tabelle 15.3 Vordefinierte Zeichenklassen in regulären Ausdrücken

Diese vordefinierten Zeichenklassen können wie ein normales Zeichen im regulären Ausdruck verwendet werden. So passt der Ausdruck

`r"P\w*th\dn"`

auf die Wörter "Pyth0n" oder "P_th1n", beispielsweise jedoch nicht auf "Python".

Beachten Sie, dass die üblichen Escape-Sequenzen, die innerhalb eines Strings verwendet werden können, auch innerhalb eines regulären Ausdrucks – selbst wenn er in einem Raw-String geschrieben wird – ihre Bedeutung behalten und nicht mit den hier vorgestellten Zeichenklassen interferieren. Gebräuchlich sind hier vor allem `\n`, `\t`, `\r` oder `\\`, insbesondere aber auch `\x`.

Zudem ist es mit dem Backslash möglich, einem Sonderzeichen die spezielle Bedeutung zu nehmen, die es innerhalb eines regulären Ausdrucks trägt. Auf diese Weise können Sie zum Beispiel mit den Zeichen `*` oder `+` arbeiten, ohne dass diese als Quantoren angesehen werden. So passt der folgende reguläre Ausdruck

```
r"\*Py\\.\\.\\.on\*"
```

allein auf den String `"*Py...on*"`.

Weitere Sonderzeichen

Für gewisse Einsatzgebiete wird es unbedingt verlangt, Regeln aufstellen zu können, die über die bloße Zeichenebene hinausgehen. So wäre es beispielsweise interessant, einen regulären Ausdruck zu erschaffen, der nur passt, wenn sich das Wort am Ende oder Anfang einer Textzeile befindet. Für solche und ähnliche Fälle gibt es einen bestimmten Satz an zusätzlichen Sonderzeichen, die genau so angewendet werden wie die vordefinierten Zeichenklassen.

Die folgende Tabelle listet alle zusätzlichen Sonderzeichen auf und gibt zu jedem eine kurze Erklärung. In der Tabelle finden Sie einige Anmerkungen zu sogenannten Flags. Das sind Einstellungen, die entweder aktiviert oder deaktiviert werden können und die Auswertung eines regulären Ausdrucks beeinflussen. Näheres dazu, wie Sie diese Einstellungen setzen können, erfahren Sie im Laufe dieses Abschnitts.

Sonderzeichen	Bedeutung
\A	Passt nur am Anfang eines Strings.
\b	Passt nur am Anfang oder Ende eines Wortes. Ein Wort kann aus allen Zeichen der Klasse \w bestehen und wird durch ein Zeichen der Klasse \s begrenzt.
\B	Passt nur, wenn es sich nicht um den Anfang oder das Ende eines Wortes handelt.
\Z	Passt nur am Ende eines Strings.
^	Passt nur am Anfang eines Strings. Beachten Sie, dass das Zeichen ^ zwei Bedeutungen hat und innerhalb einer Zeichenklasse die aufgelisteten Zeichen ausschließt. Wenn das MULTILINE-Flag gesetzt wurde, passt ^ auch direkt nach jedem Newline-Zeichen innerhalb des Strings.
\$	Passt nur am Ende eines Strings. Wenn das MULTILINE-Flag gesetzt wurde, passt \$ auch direkt vor jedem Newline-Zeichen innerhalb des Strings.

Tabelle 15.4 Vordefinierte Zeichenklassen in regulären Ausdrücken

Im konkreten Beispiel passt also der reguläre Ausdruck

```
r"\APython\Z"
```

nur bei dem String "Python", nicht jedoch bei den Strings "abcPythonabc" oder "Pythonabc".

Die hier besprochenen Beispiele beziehen sich hauptsächlich auf das Matching von regulären Ausdrücken, weswegen Ihnen die Bedeutung dieser Sonderzeichen möglicherweise noch nicht ersichtlich ist. Diese Sonderzeichen sind aber gerade beim Searching von unerlässlicher Wichtigkeit. Stellen Sie sich einmal vor, Sie würden in einem Text nach allen Vorkommen einer bestimmten Zeichenkette am Zeilenanfang suchen wollen. Dies wäre nur durch Einsatz des Sonderzeichens `^` möglich.

Genügsame Quantoren

Wir haben bereits die Quantoren `?`, `*` und `+` besprochen. Diese werden in der Terminologie regulärer Ausdrücke als »gefräßig« (engl. *greedy*) bezeichnet. Diese Klassifizierung ist nur beim Searching von Bedeutung. Betrachten Sie dazu einmal folgenden regulären Ausdruck:

```
r"Py.*on"
```

Dieser Ausdruck passt auf jeden Teilstring, der mit `Py` beginnt und mit `on` endet. Dazwischen können beliebig viele nicht näher spezifizierte Zeichen stehen. Behalten Sie im Hinterkopf, dass wir uns beim Searching befinden, der Ausdruck also dazu verwendet werden soll, aus einem längeren String verschiedene Teilstrings zu isolieren, die auf den regulären Ausdruck passen.

Nun möchten wir den regulären Ausdruck gedanklich auf den folgenden String anwenden:

```
"Python Python Python"
```

Sie meinen, dass drei Ergebnisse gefunden werden? Irrtum, es handelt sich um exakt ein Ergebnis, nämlich den Teilstring "Python Python Python". Zur Erklärung: Es wurde der »gefräßige« Quantor `*` eingesetzt. Ein solcher gefräßiger Quantor hat die Ambition, die maximal mögliche Anzahl Zeichen zu »verschlingen«. Beim Searching wird also, solange die »gefräßigen« Quantoren eingesetzt werden, stets der größtmögliche passende String gefunden.

Dieses Verhalten lässt sich umkehren, so dass immer der kleinstmögliche passende String gefunden wird. Dazu können Sie an jeden Quantor ein Fragezeichen anfügen. Dadurch wird der Quantor »genügsam« (engl. *non-greedy*). Angenommen, das Searching auf dem obigen String wäre mit dem regulären Ausdruck

```
r"Py.*?on"
```

durchgeführt worden, so wäre als Ergebnis tatsächlich dreimal der Teilstring "Python" gefunden worden. Dies funktioniert für die Quantoren `?`, `*`, `+` und `{}`.

Gruppen

Ein Teil eines regulären Ausdrucks kann durch runde Klammern zu einer sogenannten *Gruppe* zusammengefasst werden. Eine solche Gruppierung hat im Wesentlichen drei Vorteile:

- Eine Gruppe kann als Einheit betrachtet und als solche natürlich auch mit einem Quantor versehen werden. Auf diese Weise lässt sich beispielsweise das mehrmalige Auftreten einer bestimmten Zeichenkette erlauben:

```
r"( ?Python)+ ist gut"
```

In diesem Ausdruck existiert eine Gruppe um den Teilausdruck `r" ?Python"`. Dieser Teilausdruck passt auf den String "Python" mit einem optionalen Leerzeichen zu Beginn. Die gesamte Gruppe kann nun beliebig oft vorkommen, womit der obige reguläre Ausdruck sowohl auf "Python ist gut" als auch auf "Python Python Python ist gut" passt.

Beachten Sie das Leerzeichen zu Beginn der Gruppe, um die Funktionsweise des Ausdrucks zu verstehen.

- Der zweite Vorteil einer Gruppe ist, dass Sie auf sie zugreifen können, nachdem das Searching bzw. Matching durchgeführt wurde. Das heißt, Sie könnten beispielsweise überprüfen, ob eine eingegebene URL gültig ist, und gleichzeitig Subdomain, Domain und TLD herausfiltern.

Näheres dazu, wie der Zugriff auf Gruppen funktioniert, erfahren Sie in Abschnitt 15.1.2, »Verwendung des Moduls«.

- Es gibt Gruppen, die in einem regulären Ausdruck häufiger gebraucht werden. Um diese nicht jedes Mal erneut schreiben zu müssen, werden Gruppen mit 1 beginnend durchnummeriert und können dann anhand ihres Index referenziert werden. Eine solche Referenz besteht aus einem Backslash, gefolgt von dem Index der jeweiligen Gruppe, und passt auf den gleichen Teilstring, auf den die Gruppe gepasst hat. So passt der reguläre Ausdruck `r"(Python) \1"` auf "Python Python".

Alternativen

Eine weitere Möglichkeit, die die Syntax regulärer Ausdrücke vorsieht, sind sogenannte *Alternativen*. Im Prinzip handelt es sich dabei um nichts anderes als um eine ODER-Verknüpfung zweier Zeichen oder Zeichengruppen, wie Sie sie be-

reits von dem Operator `or` her kennen. Diese Verknüpfung wird durch den senkrechten Strich `|`, auch *Pipe* genannt, durchgeführt.

```
r"P(ython|eter)"
```

Dieser reguläre Ausdruck passt sowohl auf den String `"Python"` als auch auf `"Peter"`. Durch die Gruppe kann später ausgelesen werden, welche der beiden Alternativen aufgetreten ist.

Extensions

Damit wäre die Syntax regulärer Ausdrücke beschrieben. Zusätzlich zu dieser mehr oder weniger standardisierten Syntax erlaubt Python die Verwendung sogenannter *Extensions*. Eine Extension ist folgendermaßen aufgebaut:

```
(?...)
```

Die drei Punkte werden durch eine Kennung der gewünschten Extension und weitere extensionspezifische Angaben ersetzt. Diese Syntax wurde gewählt, da eine öffnende Klammer, gefolgt von einem Fragezeichen, keine syntaktisch sinnvolle Bedeutung hat und demzufolge »frei« war. Beachten Sie aber, dass eine Extension in der Regel keine neue Gruppe erzeugt, auch wenn die runden Klammern dies nahelegen. Nachfolgend möchten wir näher auf die Extensions eingehen, die in Pythons regulären Ausdrücken verwendet werden können.

(?aiLmsux)

Diese Extension erlaubt es, ein oder mehrere Flags für den gesamten regulären Ausdruck zu setzen. Der Begriff *Flag* ist bereits verwendet worden und beschreibt eine bestimmte Einstellung, die entweder aktiviert oder deaktiviert werden kann. Ein Flag kann entweder im regulären Ausdruck selbst, eben durch diese Extension, oder durch einen Parameter der Funktion `re.compile` gesetzt werden. Im Zusammenhang mit dieser Funktion werden wir näher darauf eingehen, welche Flags wofür stehen. Das Flag `i` macht den regulären Ausdruck beispielsweise *case insensitive*:

```
r"(?i)P"
```

Dieser Ausdruck passt sowohl auf `"P"` als auch auf `"p"`.

(?:...)

Diese Extension wird wie normale runde Klammern verwendet, erzeugt dabei aber keine Gruppe. Das heißt, auf einen durch diese Extension eingeklammerten Teilausdruck können Sie später nicht zugreifen. Ansonsten ist diese Syntax äquivalent zu runden Klammern:

```
r"(?:abc|def)"
```

(?P<name>...)

Diese Extension erzeugt eine Gruppe mit dem angegebenen Namen. Das Besondere an einer solchen benannten Gruppe ist, dass sie nicht allein über ihren Index, sondern auch über ihren Namen referenziert werden kann. Der Name muss ein gültiger Bezeichner sein:

```
r"(?P<hallowelt>abc|def)"
```

(?P=name)

Passt auf all das, auf das die bereits definierte Gruppe mit dem Namen *name* gepasst hat. Diese Extension erlaubt es also, eine benannte Gruppe zu referenzieren.

```
r"(?P<py>[Pp]ython) ist, wie (?P=py) sein sollte"
```

Dieser reguläre Ausdruck passt auf den String "Python ist, wie Python sein sollte".

(?#...)

Diese Extension stellt einen Kommentar dar. Der Inhalt der Klammern wird schlicht ignoriert:

```
r"Py(?#lalala)thon"
```

(?=...)

Passt nur dann, wenn der reguläre Ausdruck ... als Nächstes passt. Diese Extension greift also vor, ohne in der Auswertung des Ausdrucks tatsächlich voranzuschreiten.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?!...)

Passt nur dann, wenn der reguläre Ausdruck ... als Nächstes nicht passt. Diese Extension ist das Gegenstück zu der vorherigen.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?<=...)

Passt nur, wenn der reguläre Ausdruck ... zuvor gepasst hat. Diese Extension greift also auf bereits ausgewertete Teile des Strings zurück, ohne die Auswertung selbst zurückzuwerfen.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?<!...)

Passt nur, wenn der reguläre Ausdruck ... zuvor nicht gepasst hat. Diese Extension ist damit das Gegenstück zu der vorherigen.

Diese Extension ist vor allem beim Searching von Bedeutung.

(?(id/name)yes-pattern|no-pattern)

Diese recht kompliziert anmutende Extension kann in einem regulären Ausdruck als eine Art Fallunterscheidung verwendet werden. Abhängig davon, ob eine Gruppe mit dem angegebenen Index bzw. dem angegebenen Namen auf einen Teilstring gepasst hat, wird entweder (im positiven Fall) auf das *yes-pattern* oder (im negativen Fall) auf das *no-pattern* getestet. Das *no-pattern* wird durch einen senkrechten Strich vom *yes-pattern* getrennt, kann aber auch weggelassen werden. Vielleicht ist der Sinn dieser Extension noch nicht ganz klar geworden, deshalb folgendes Beispiel:

```
r"(?P<klammer>\()?Python(?:klammer\))"
```

In diesem Ausdruck wird zunächst eine Gruppe namens `klammer` erstellt, die maximal einmal vorkommen darf und aus einer öffnenden, runden Klammer besteht. Danach folgt die Zeichenkette `Python`, und schlussendlich wird durch die Extension eine schließende Klammer gefordert, sofern zuvor eine öffnende aufgetreten ist, also sofern die Gruppe `klammer` zuvor gepasst hat. Damit passt der reguläre Ausdruck auf die Strings `"Python"` und `"(Python)"`, beispielsweise aber nicht auf `"(Python"`.

Damit wäre den syntaktischen Regeln für reguläre Ausdrücke Genüge getan. Auch wenn dieser Abschnitt möglicherweise etwas trocken und theoretisch war, ist es durchaus wichtig, sich mit regulären Ausdrücken auseinanderzusetzen, denn in vielen Fällen ist der Einsatz regulärer Ausdrücke besonders elegant.

In den folgenden Abschnitten möchten wir über die praktische Anwendung regulärer Ausdrücke in Python reden. Dazu gehört zunächst einmal die Verwendung des Moduls `re`. Danach werden wir jeweils ein kleines Beispielprojekt zum Matching bzw. Searching zeigen.

15.1.2 Verwendung des Moduls

Nachdem Sie in die unendlichen Weiten der regulären Ausdrücke eingeführt wurden, werden wir uns hier um ihre konkrete Verwendung in Python kümmern. Die Beispiele dieses Abschnitts werden im interaktiven Modus durchgeführt und setzen voraus, dass das Modul `re` eingebunden wurde:

```
>>> import re
```

Flags

Im vorherigen Abschnitt wurden mehrfach die sogenannten *Flags* angesprochen. Das sind bestimmte Einstellungen, die die Auswertung eines regulären Ausdrucks beeinflussen. Flags können Sie entweder im Ausdruck selbst durch eine Extension oder als Parameter einer der im Modul `re` verfügbaren Funktionen angeben. Sie beeinflussen nur den Ausdruck, der aktuell verarbeitet wird, und verbleiben nicht nachhaltig im System. Jedes Flag ist als Konstante im Modul `re` enthalten und kann über eine Lang- oder eine Kurzversion seines Namens angesprochen werden. Die folgende Tabelle listet alle Flags auf und erläutert ihre Bedeutung.

Alias	Name	Bedeutung
re.A	re.ASCII	Beschränkt die Zeichenklassen <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\s</code> und <code>\S</code> auf den ASCII-Zeichensatz.
re.I	re.IGNORECASE	Macht die Auswertung des regulären Ausdrucks <i>case insensitive</i> , das heißt, dass die Zeichengruppe <code>[A-Z]</code> sowohl auf Groß- als auch auf Kleinbuchstaben passen würde.
re.L	re.LOCALE	Gibt an, dass bestimmte vordefinierte Zeichenklassen von der aktuellen Lokalisierung abhängig gemacht werden sollen. Das betrifft die Gruppen <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\s</code> und <code>\S</code> .
re.M	re.MULTILINE	Wenn dieses Flag gesetzt wurde, passt <code>^</code> sowohl zu Beginn des Strings als auch nach jedem Newline-Zeichen und <code>\$</code> vor jedem Newline-Zeichen. Normalerweise passen <code>^</code> und <code>\$</code> nur am Anfang bzw. am Ende des Strings.
re.S	re.DOTALL	Wenn dieses Flag gesetzt wurde, passt das Sonderzeichen <code>.</code> tatsächlich auf jedes Zeichen. Normalerweise passt der Punkt auf jedes Zeichen außer auf das Newline-Zeichen <code>\n</code> .
re.U	re.UNICODE	Wenn dieses Flag gesetzt wurde, passen sich die vordefinierten Zeichenklassen dem Unicode-Standard an. Das heißt, dass dann auch Nicht-ASCII-Zeichen als Buchstabe oder Ziffer eingestuft werden. Dieses Flag ist seit Python 3.0 standardmäßig gesetzt.
re.X	re.VERBOSE	Das Setzen dieses Flags erlaubt es Ihnen, einen regulären Ausdruck zu formatieren. Wenn es gesetzt wurde, werden Whitespace-Zeichen wie Leerzeichen, Tabulatoren oder Newline-Zeichen ignoriert, solange sie nicht durch einen Backslash eingeleitet werden. Zudem leitet ein <code>#</code> -Zeichen einen Kommentar ein. Das heißt, alles hinter diesem Zeichen bis zu einem Newline-Zeichen wird ignoriert.

Tabelle 15.5 Flags

Funktionen

Neben den Flags enthält das Modul `re` noch einige Funktionen, die im Folgenden besprochen werden sollen.

`re.compile(pattern[, flags])`

Kompiliert den regulären Ausdruck *pattern* zu einem *Regular-Expression-Objekt*, im Folgenden *RE-Objekt* genannt. Bei mehreren Operationen auf demselben regulären Ausdruck lohnt es sich, diesen zu kompilieren, da diese Operationen dann wesentlich schneller durchgeführt werden können. Zum Durchführen der Operationen bietet das RE-Objekt im Wesentlichen die gleiche Funktionalität wie das Modul `re`.

Um die Auswertung des Ausdrucks zu beeinflussen, können Sie ein oder mehrere Flags angeben. Wenn es sich um mehrere handelt, müssen Sie sie durch das bitweise ODER `|` trennen.

```
>>> c1 = re.compile(r"P[yY]thon")
>>> c2 = re.compile(r"P[yY]thon", re.I)
>>> c3 = re.compile(r"P[yY]thon", re.I | re.S)
```

Die Angabe von Flags ist bei den meisten Funktionen des Moduls `re` über den Parameter *flags* möglich. Wir werden darauf in Zukunft nicht mehr eingehen.

Näheres zum RE-Objekt folgt im nächsten Abschnitt.

`re.search(pattern, string[, flags])`

Durchsucht den String *string* nach einem Teilstring, auf den der reguläre Ausdruck *pattern* passt. Der erste gefundene Teilstring wird in Form eines sogenannten *Match-Objekts* zurückgegeben. Näheres zur Verwendung des Match-Objekts erfahren Sie im entsprechenden Abschnitt weiter unten.

Wenn kein Ergebnis gefunden wurde, gibt die Funktion `None` zurück.

```
>>> re.search(r"P[Yy]thon", "Nimm doch Python")
<sre.SRE_Match object at 0xb7bd7f00>
```

`re.match(pattern, string[, flags])`

Wenn null oder mehr Zeichen am Anfang des Strings *string* auf den regulären Ausdruck *pattern* passen, wird diese Übereinstimmung in Form eines Match-Objekts zurückgegeben. Wenn keine Übereinstimmung gefunden wurde, wird `None` zurückgegeben.

```
>>> print(re.match(r"P[Yy]thon", "PYthon"))
None
```

```
>>> re.match(r"P[Yy]thon", "PYthon")
<_sre.SRE_Match object at 0xb7bd7f00>
```

re.split(pattern, string[, maxsplit])

Der String *string* wird nach Übereinstimmungen mit dem regulären Ausdruck *pattern* durchsucht. Alle passenden Teilstings werden als Trennzeichen angesehen, und die dazwischenliegenden Teile werden als Liste von Strings zurückgegeben.

```
>>> re.split(r"\s", "Python Python Python")
['Python', 'Python', 'Python']
```

Eventuell vorkommende Gruppen innerhalb des regulären Ausdrucks werden ebenfalls als Elemente dieser Liste zurückgegeben:

```
>>> re.split(r"\s(?:.*?)\s", "Python oder Python und Python")
['Python', 'oder', 'Python', 'und', 'Python']
```

In diesem regulären Ausdruck werden alle von zwei Whitespaces umgebenen Wörter als Trennzeichen behandelt.

Wenn der Parameter *maxsplit* angegeben wurde und ungleich 0 ist, wird der String maximal *maxsplit*-mal unterteilt. Der Reststring wird als letztes Element der Liste zurückgegeben.

re.findall(pattern, string[, flags])

Sucht im String *string* nach Übereinstimmungen mit dem regulären Ausdruck *pattern*. Alle gefundenen, nicht überlappenden Übereinstimmungen werden in Form einer Liste von Strings zurückgegeben:

```
>>> re.findall(r"P[Yy]thon", "Python oder PYthon und Python")
['Python', 'Python', 'Python']
```

Wenn *pattern* ein oder mehrere Gruppen enthält, werden diese anstelle der übereinstimmenden Teilstings in die Ergebnisliste geschrieben.

```
>>> re.findall(r"P([Yy])thon", "Python oder PYthon und Python")
['y', 'Y', 'y']
>>> re.findall(r"P([Yy])th(.*?)n", "Python oder PYthon und Python")
[('y', 'o'), ('Y', 'o'), ('y', 'o')]
```

Bei mehreren Gruppen handelt es sich um eine Liste von Tupeln.

re.finditer(pattern, string[, flags])

Sucht im String *string* nach Übereinstimmungen mit dem regulären Ausdruck *pattern*. Das Ergebnis ist ein Iterator, der über alle gefundenen, nicht überlappenden Übereinstimmungen jeweils als Match-Objekt iteriert.

re.sub(pattern, repl, string[, count])

Die Funktion *sub* sucht im String *string* nach nicht überlappenden Übereinstimmungen mit dem regulären Ausdruck *pattern*. Es wird eine Kopie des Strings *string* zurückgegeben, in dem alle passenden Teilstrings durch den String *repl* ersetzt wurden:

```
>>> re.sub(r"[Jj]a[Vv]a", "Python", "Java oder java und jaVa")
'Python oder Python und Python'
```

Statt eines Strings kann für *repl* auch ein Funktionsobjekt übergeben werden. Dieses wird für jede gefundene Übereinstimmung aufgerufen und bekommt das jeweilige Match-Objekt als einzigen Parameter. Der übereinstimmende Teilstring wird durch den Rückgabewert der Funktion ersetzt.

Es ist möglich, durch die Schreibweisen `\g<name>` oder `\g<index>` Gruppen des regulären Ausdrucks zu referenzieren:

```
>>> re.sub(r"([Jj]ava)", "Python statt \g<1>", "Nimm doch Java")
'Nimm doch Python statt Java'
```

Durch den optionalen Parameter *count* kann die maximale Anzahl an Ersetzungen festgelegt werden, die vorgenommen werden dürfen.

re.subn(pattern, repl, string[, count])

Funktioniert ähnlich wie *sub*, mit dem Unterschied, dass ein Tupel zurückgegeben wird, in dem zum einen der neue String und zum anderen die Anzahl der vorgenommenen Ersetzungen stehen:

```
>>> re.subn(r"([Jj]ava)", "Python statt \g<1>", "Nimm doch Java")
('Nimm doch Python statt Java', 1)
```

re.escape(string)

Wandelt alle nicht-alphanumerischen Zeichen von *string* in ihre entsprechende Escape-Sequenz um und gibt das Ergebnis als String zurück. Diese Funktion ist besonders dann sinnvoll, wenn Sie einen String in einen regulären Ausdruck einbetten möchten, aber nicht sicher sein können, ob Sonderzeichen, beispielsweise ein Punkt, enthalten sind.

```
>>> re.escape("Funktioniert das wirklich? ... (ja!)")
'Funktioniert\\ das\\ wirklich\\?\\ \\\\.\\.\\.\\.\\.\\. \\(ja\\!\\.\\.\\)'
```

Beachten Sie, dass die Escape-Sequenzen im Stringliteral jeweils durch einen doppelten Backslash eingeleitet werden. Das liegt daran, dass das Ergebnis als String und nicht als Raw-String zurückgegeben wird.

Das Regular-Expression-Objekt

Ein Regular-Expression-Objekt, im Folgenden RE-Objekt genannt, wird erzeugt, wenn ein regulärer Ausdruck kompiliert wurde. Das Kompilieren eines regulären Ausdrucks ist sinnvoll, wenn mehrere Operationen mit ihm durchgeführt werden sollen. Diese werden dann zusammengenommen wesentlich schneller durchgeführt, als wenn Sie die Funktionen `match` oder `search` direkt aufrufen.

Damit Searching- und Matching-Operationen mit einem kompilierten regulären Ausdruck durchgeführt werden können, besitzt das RE-Objekt eine Funktionalität, die deckungsgleich ist mit der des `re`-Moduls. Das bedeutet, dass für das RE-Objekt größtenteils die Funktionen des `re`-Moduls als Methoden implementiert sind, selbstverständlich mit gewissen Änderungen der Schnittstelle.

Wir werden hier nicht genau auf die Funktionsweise der Methoden eingehen, sondern nur einen Vergleich zu den Funktionen des `re`-Moduls ziehen. Dennoch ist es aufgrund der Änderungen bei den Schnittstellen wichtig, alle Methoden zu behandeln. Die Beispiele verstehen sich in folgendem Kontext:

```
>>> import re
>>> c = re.compile(r"P[Yy]th.n")
```

Das bedeutet: Es existiert ein RE-Objekt namens `c`, dem der reguläre Ausdruck `r"P[Yy]th.n"` zugrunde liegt.

c.match(string[, pos[, endpos]])

Äquivalent zur Funktion `re.match`. Die optionalen Parameter *pos* und *endpos* geben, wenn sie ungleich 0 sind, zwei Indizes an, zwischen denen das Matching durchgeführt werden soll. Wenn sie nicht angegeben wurden, wird das Matching auf dem gesamten String durchgeführt.

```
>>> print(c.match("Pythoon"))
None
>>> c.match("Python")
<_sre.SRE_Match object at 0xb7c49e58>
```

c.search(string[, pos[, endpos]])

Äquivalent zur Funktion `re.search`. Die optionalen Parameter *pos* und *endpos* haben dieselbe Bedeutung wie bei der Methode `match`.

```
>>> c.search("Dies ist Python")
<_sre.SRE_Match object at 0xb7c49e58>
```

c.split(string[, maxsplit])

Äquivalent zur Funktion `re.split`.

```
>>> c.split("halloweltPythonhallowelt")
['hallowelt', 'hallowelt']
```

c.findall(string[, pos[, endpos]])

Äquivalent zur Funktion `re.findall`. Die optionalen Parameter *pos* und *endpos* haben dieselbe Bedeutung wie bei der Methode `match`.

```
>>> c.findall("Python Python Python")
['Python', 'Python', 'Python']
```

c.finditer(string[, pos[, endpos]])

Äquivalent zur Funktion `re.finditer`. Die optionalen Parameter *pos* und *endpos* haben dieselbe Bedeutung wie bei der Methode `match`.

c.sub(repl, string[, count])

Äquivalent zur Funktion `re.sub`.

c.subn(repl, string[, count])

Äquivalent zur Funktion `re.subn`.

Neben diesen Methoden enthält das RE-Objekt drei Attribute, die das Arbeiten mit dem Objekt erleichtern.

c.flags

Das Attribut *flags* ist eine ganze Zahl und enthält alle gesetzten Flags. Beachten Sie, dass Flags selbst auch ganze Zahlen sind und eine Kombination von Flags durch ihr bitweises ODER repräsentiert wird. Die zu setzenden Flags werden beim Erzeugen des RE-Objekts der Funktion `re.compile` übergeben. Wenn kein Flag übergeben wird, ist der Wert des Attributs 32, bedingt durch das seit Python 3.0 standardmäßig gesetzte Flag `re.UNICODE`.

```
>>> c.flags
32
```

Um zu testen, ob ein bestimmtes Flag gesetzt ist, kann das bitweise UND verwendet werden:

```
>>> c1 = re.compile(r"P[Yy]th.n", re.I)
>>> c1.flags
34
>>> c1.flags & re.I
2
>>> c1.flags & re.M
0
```

Das bitweise UND zwischen dem Attribut *flags* und einem nicht gesetzten Flag ergibt immer 0.

c.groupindex

Das Attribut *groupindex* ist ein Dictionary, das alle Namen benannter Gruppen als Schlüssel enthält und die Indizes dieser Gruppen als Werte. Eine benannte Gruppe wird durch die Extension (*?P<name>...*) erzeugt.

```
>>> c2 = re.compile(r"(?P<gruppe1>P[Yy])(?P<gruppe2>th.n)")
>>> c2.groupindex
{'gruppe1': 1, 'gruppe2': 2}
```

c.pattern

Das Attribut *pattern* ist ein String und enthält den regulären Ausdruck, der dem RE-Objekt zugrunde liegt.

```
>>> c.pattern
'P[Yy]th.n'
```

Das Match-Objekt

Nachdem wir das RE-Objekt besprochen haben, wenden wir uns einem wesentlich interessanteren Objekt zu, dem Match-Objekt. Eine solche Instanz wird zurückgegeben, wenn eine Match- oder Search-Operation Übereinstimmungen gefunden hat. Das Match-Objekt enthält nähere Details zu diesen gefundenen Übereinstimmungen.

Die Beispiele in diesem Abschnitt verstehen sich in folgendem Kontext:

```
>>> import re
>>> c = re.compile(r"(P[Yy])(th.n)")
```

Das Match-Objekt verfügt über folgende Methoden:

m.expand(template)

Die Methode `expand` erlaubt es, den String *template* mit Informationen zu füllen, die aus der Matching- bzw. Searching-Operation stammen. So können über `\g<index>` und `\g<name>` die Teilstrings eingefügt werden, die auf die jeweiligen Gruppen gepasst haben. Beachten Sie unbedingt, dass Sie *template* wegen der Backslashes als Raw-String angeben sollten.

```
>>> m = c.match("Python")
>>> m.expand(r"Hallo \g<1> Welt \g<2>")
'Hallo Py Welt thon'
```

m.group([group1, ...])

Die Methode `group` erlaubt einen komfortablen Zugriff auf die Teilstrings, die auf die verschiedenen Gruppen des regulären Ausdrucks gepasst haben. Wenn nur ein Argument übergeben wurde, ist der Rückgabewert ein String, ansonsten ein Tupel von Strings. Wenn eine Gruppe auf keinen Teilstring gepasst hat, wird für diese `None` zurückgegeben. Ein Index von 0 gibt alle Gruppen zurück.

```
>>> m = c.match("Python")
>>> m.group(0)
'Python'
>>> m.group(1)
'Py'
>>> m.group(1, 2)
('Py', 'thon')
```

m.groups([default])

Gibt ein Tupel zurück, das alle Teilstrings enthält, die auf eine der im regulären Ausdruck enthaltenen Gruppen gepasst haben. Der optionale Parameter *default* erlaubt es, den Wert festzulegen, der in das Tupel geschrieben wird, wenn auf eine Gruppe kein Teilstring gepasst hat. Der Parameter ist mit `None` vorbelegt.

```
>>> m = c.match("Python")
>>> m.groups()
('Py', 'thon')
```

m.groupdict([default])

Gibt ein Dictionary zurück, das die Namen aller benannten Gruppen als Schlüssel und die jeweils passenden Teilstrings als Werte enthält. Der Parameter *default* hat die gleiche Bedeutung wie bei der Methode `groups`.

```
>>> c2 = re.compile(r"(?P<gruppe>P[Yy])(th.n)")
>>> m2 = c2.match("Python")
```

```
>>> m2.groupdict()
{'gruppe': 'Py'}
```

m.start([group]), end([group])

Gibt den Start- bzw. Endindex des Teilstrings zurück, der auf die Gruppe *group* gepasst hat. Der optionale Parameter *group* ist mit 0 vorbelegt.

```
m = c.match("Python")
>>> m.start(2)
2
>>> m.end(2)
6
```

m.span([group])

Gibt das Tupel (m.start(group), m.end(group)) zurück.

```
>>> m = c.match("Python")
>>> m.span(2)
(2, 6)
```

Neben den soeben beschriebenen Methoden besitzt das Match-Objekt sechs Attribute, die im Folgenden beschrieben werden sollen.

m.pos, m.endpos

Die Methoden *match* und *search* des RE-Objekts besitzen zwei Parameter namens *pos* und *endpos*. Die Attribute *pos* und *endpos* des Match-Objekts erlauben den Zugriff auf die dort zuletzt übergebenen Werte.

m.lastindex

Der Index der Gruppe, die bei der Auswertung als Letzte auf einen Teilstring gepasst hat, oder *None*, wenn keine Gruppe gepasst hat.

m.lastgroup

Der Name der symbolischen Gruppe, die bei der Auswertung als Letzte auf einen Teilstring gepasst hat, oder *None*, wenn keine Gruppe gepasst hat.

m.re

Der ursprüngliche reguläre Ausdruck als String.

m.string

Der String, der der *match*- bzw. *search*-Methode des RE-Objekts zuletzt übergeben wurde.

15.1.3 Ein einfaches Beispielprogramm – Searching

Bisher wurde sowohl die Syntax regulärer Ausdrücke als auch deren Verwendung durch das Modul `re` der Standardbibliothek besprochen. Eigentlich ist die Thematik damit erschöpfend behandelt, doch wir möchten, um auch einer praxisorientierten Einführung gerecht zu werden, an dieser Stelle zwei kleine Beispielprojekte vorstellen, die stark auf reguläre Ausdrücke setzen. Zunächst erklären wir in diesem relativ einfach gehaltenen Programm das Searching und im nächsten, etwas komplexeren Beispiel das Matching.

Mithilfe des Searchings werden Muster innerhalb eines längeren Textes gefunden und herausgefiltert. In unserem Beispielprogramm soll das Searching dazu dienen, alle Links aus einer beliebigen HTML-Datei mitsamt Beschreibung herauszulesen. Dazu müssen wir uns zunächst den Aufbau eines HTML-Links vergegenwärtigen:

```
<a href="URL">Beschreibung</a>
```

Dazu ist zu sagen, dass HTML nicht zwischen Groß- und Kleinschreibung unterscheidet, wir den regulären Ausdruck also mit dem `IGNORECASE`-Flag verwenden sollten. Des Weiteren handelt es sich bei dem obigen Beispiel um die einfachste Form eines HTML-Links, denn neben der URL und der Beschreibung können weitere Angaben gemacht werden.

Der folgende reguläre Ausdruck passt sowohl auf den oben beschriebenen als auch auf weitere, komplexere HTML-Links:

```
r"<a.*href=[\"'\"](.?)*[\"'\"].*>(.*?)</a>"
```

Wichtig ist, dass der reguläre Ausdruck zwei Gruppen enthält, jeweils für die URL und die Beschreibung, so dass diese beiden Angaben später bequem ausgelesen werden können. Außerdem sollten Sie unbedingt beachten, dass innerhalb dieser Gruppen »genügsame« Quantoren eingesetzt wurden, da sonst mehrere Links fälschlicherweise zu einem zusammengefasst werden könnten.

Doch nun zum Beispielprogramm:

```
import re

f = open("test.html", "r")
html = f.read()
f.close()

it = re.finditer(r"<[a].*href=[\"'\"](.?)*[\"'\"].*>(.*?)</[a]>",
                 html, re.I)

for m in it:
    print("Name: {0}, Link: {1}".format(m.group(2), m.group(1)))
```

Zunächst wird eine beliebige HTML-Datei, in diesem Fall *test.html*, geöffnet und mithilfe der Methode `read` des Dateiobjekts ausgelesen. Danach wird die Funktion `finditer` des Moduls `re` aufgerufen, um alle Übereinstimmungen mit dem vorhin besprochenen regulären Ausdruck im HTML-Code zu finden. Das Ergebnis wird als Iterator zurückgegeben und von `it` referenziert.

Schlussendlich wird über `it` iteriert. In jedem Iterationsschritt ist die aktuelle Übereinstimmung als `Match`-Objekt `m` verfügbar. Jetzt werden nur noch die Teilstrings ausgegeben, die auf die beiden Gruppen des regulären Ausdrucks gepasst haben.

Sie können das Programm mit beliebigen HTML-Seiten testen. Besuchen Sie dazu im Internet eine möglichst komplexe Website, beispielsweise die eines Nachrichtenmagazins, und speichern Sie diese als HTML-Datei ab. Sie werden sehen, dass das Beispielprogramm auch hier die enthaltenen Links findet.

Das hier vorgestellte Programm schreit geradezu danach, erweitert zu werden. Beispielsweise könnte es neben Links noch andere Teile des HTML-Codes, wie enthaltene Bilder oder Überschriften, auslesen.

15.1.4 Ein komplexeres Beispielprogramm – Matching

Es ist allgemein – und besonders im Web – ein häufiges Problem, eingegebene Formulardaten zu validieren und die wichtigen Informationen aus den Eingaben herauszufiltern. Dies ist selbstverständlich auch mit normalen String-Operationen möglich, doch mutiert der Code bei solchen Versuchen schnell zu einem unförmigen Batzen von Irgendwas. Das angesprochene Problem lässt sich durch reguläre Ausdrücke sehr elegant und mit nur geringem Quellcodeaufwand lösen. Unser Beispielprogramm soll aus einer Art elektronischer Visitenkarte alle relevanten Informationen auslesen und maschinenlesbar aufbereiten. Die Visitenkarte ist in einer Textdatei in folgendem Format gespeichert:

```
Name: Max Mustermann
Addr: Musterstr 123
      12345 Musterhausen
Tel:  +49 1234 56789
```

Das Programm soll nun diese Textdatei einlesen, die enthaltenen Informationen extrahieren und zu einem solchen Dictionary aufbereiten:

```
{
  'Tel': ('+49', '1234', '56789'),
  'Name': ('Max', 'Mustermann'),
  'Addr': ('Musterstr', '123', '12345', 'Musterhausen')
}
```

In der Textdatei soll dabei immer nur ein Datensatz stehen.

Zunächst einmal möchten wir etwas detaillierter auf die Funktionsweise des Beispielprogramms eingehen. Die Visitenkarte besteht aus verschiedenen Informationen, denen immer eine Überschrift bzw. Kategorie gegeben wurde (»Name«, »Addr« und »Tel«). Die Kategorie von der Information zu trennen, ist keine komplizierte Angelegenheit, da der Doppelpunkt innerhalb der Informationen nicht vorkommt und somit in jeder Zeile einzigartig ist. Ein Problem ist die dritte Zeile, da hier keine explizite Überschrift gegeben ist. In einem solchen Fall wird die Zeile an die Information der vorherigen Überschrift angehängt. Auf diese Weise lässt sich ein Dictionary erzeugen, das die Überschriften auf die jeweiligen Informationen mappt.

Die Informationen werden allerdings zeilenweise aus der Datei ausgelesen. Das ist nicht optimal, da wir die Daten ausdrücklich maschinenlesbar einlesen wollten, das heißt insbesondere nach Einzelinformationen getrennt. Für diese Arbeit bieten sich reguläre Ausdrücke förmlich an.

Kommen wir zur konkreten Implementierung. Dazu schreiben wir zunächst eine Funktion, die die Daten zeilenweise einliest und zu einem Dictionary aufbereitet:

```
def leseDatei(datei):
    d = {}
    f = open(datei)
    for zeile in f:
        if ":" in zeile:
            key, d[key] = (s.strip() for s in zeile.split(":"))
        elif "key" in locals():
            d[key] += "\n{0}".format(zeile.strip())
    f.close()
    return d
```

Die Funktion `leseDatei` bekommt den String `datei` mit einer Pfadangabe übergeben. Innerhalb der Funktion wird die Datei zeilenweise eingelesen. Jede Zeile wird anhand des Doppelpunktes in die beiden Teile »Überschrift« und »Information« aufgeteilt und, durch Einsatz der Methode `strip`, von überflüssigen Leerzeichen befreit. Danach werden Überschrift und Information in das Dictionary `d` geschrieben, und die jeweils aktuelle Überschrift wird zusätzlich durch `key` referenziert.

Wenn in einer Zeile kein Doppelpunkt vorkommt, wurde die Information auf mehrere Zeilen umbrochen. Das bedeutet für uns, dass wir zunächst auch die Methode `strip` auf den kompletten Zeileninhalt anwenden und sie dann unter der Überschrift `key` an den bereits bestehenden Wert im Dictionary anhängen. Zu

diesem Zweck muss die Referenz `key` selbstverständlich existieren. Da diese erst innerhalb der `if`-Anweisung angelegt wird, wird vorausgesetzt, dass eine Zeile mit Doppelpunkt vor einer Zeile ohne Doppelpunkt kommen muss. Obwohl es keine syntaktisch sinnvolle Datei gibt, in der das nicht gilt, überprüfen wir im `elif`-Zweig explizit, ob die Referenz `key` existiert.

Das Resultat dieser Funktion ist ein Dictionary mit den Überschriften als Schlüssel und den dazugehörigen Informationen (in Form von Strings) als Werte. Die zweite Funktion des Beispiels soll die Daten mithilfe regulärer Ausdrücke analysieren und dann als Tupel im Dictionary ablegen. Dazu erzeugen wir zunächst ein Dictionary namens `regexp`, das für jede Überschrift einen regulären Ausdruck bereitstellt, der verwendet werden kann, um die Information zu validieren:

```
regexp = {
    "Name" : r"([A-Za-z]+)\s([A-Za-z]+)",
    "Addr" : r"([A-Za-z]+)\s(\d+)\s*(\d{5})\s([A-Za-z]+)",
    "Tel"  : r"(\+\d{2})\s(\d{4})\s(\d{3,})"
```

Diese regulären Ausdrücke verfügen über mehrere Gruppen, um das Aufteilen der Information in die verschiedenen Einzelinformationen zu erleichtern.

Die Funktion, mit der die Daten analysiert werden, sieht folgendermaßen aus:

```
def analysiereDaten(daten, regexp):
    for key in daten:
        m = re.match(regexp[key], daten[key])
        if not m:
            return False
        daten[key] = m.groups()
    return True
```

Die Funktion `analysiereDaten` bekommt zwei Dictionaries als Parameter übergeben: zum einen das soeben erstellte Dictionary `regexp` und zum anderen das Dictionary, das von der vorherigen Funktion erstellt wurde und die eingelesenen Daten enthält.

Die Funktion iteriert in einer `for`-Schleife über das Dictionary `daten` und wendet, jeweils passend zur aktuellen Überschrift, mithilfe der Funktion `re.match` den regulären Ausdruck auf den eingelesenen String an. Das zurückgegebene Match-Objekt wird durch `m` referenziert.

Nachfolgend wird getestet, ob `re.match` `None` zurückgegeben hat. Ist das der Fall, gibt die Funktion `analysiereDaten` ihrerseits `False` zurück. Andernfalls wird der aktuelle Wert des Dictionarys `daten` mit den Teilstrings überschrieben, die auf die einzelnen Gruppen der regulären Ausdrücke gepasst haben. Die Methode

`group` des `Match`-Objekts gibt ein Tupel von Strings zurück. Nach dem Durchlaufen der Funktion `analysiereDaten` enthält das Dictionary die gewünschten Daten in aufbereiteter Form.

Zu guter Letzt fehlt noch der Code, der den Anstoß zum Einlesen und Aufbereiten der Daten gibt:

```
daten = leseDatei("id.txt")
if analysiereDaten(daten, regexp):
    print(daten)
else:
    print("Die Angaben sind fehlerhaft")
```

Je nachdem, welchen Wahrheitswert die Funktion `analysiereDaten` zurückgegeben hat, werden die aufbereiteten Daten oder eine Fehlermeldung ausgegeben.

Hoffentlich haben Ihnen die beiden Beispiele geholfen, einen praxisbezogenen Einstieg in die Welt der regulären Ausdrücke zu finden. Bleibt noch zu sagen, dass das dargebotene Programm zwar funktioniert, aber nicht perfekt ist. Fühlen Sie sich dazu ermutigt, es zu erweitern oder anzupassen. So erlauben die regulären Ausdrücke beispielsweise noch keine Umlaute oder Interpunktionszeichen im Straßennamen. Sie könnten beispielsweise auch Visitenkarte und Programm um die Angabe einer E-Mail-Adresse erweitern.

15.2 Lokalisierung von Programmen – gettext

Das Modul `gettext` der Standardbibliothek ist bei der *Internationalisierung* und *Lokalisierung* von Python-Programmen von Nutzen. Mit Internationalisierung, auch als »I18N« abgekürzt, wird der Vorgang bezeichnet, die Benutzerschnittstelle eines Programms so zu abstrahieren, dass sie sehr leicht an andere sprachliche oder kulturelle Umgebungen angepasst werden kann. Als Lokalisierung, abgekürzt auch »L10N«, wird dann das Anpassen des Programms an die Gegebenheiten eines bestimmten Landes oder einer Region bezeichnet. Beachten Sie, dass sich das Modul `gettext` dabei auf die Übersetzung von Strings beschränkt. Andere Unterschiede, wie beispielsweise Datumsformate oder Währungssymbole, werden nicht berücksichtigt.

Das Modul `gettext` lehnt sich an die *GNU gettext API* an, die als Teil des GNU-Projekts weitverbreitet ist und die rein sprachliche Anpassung eines Programms gewährleistet. Das Modul erlaubt es, eine möglichst genaue Nachbildung der GNU `gettext API` zu verwenden. Zudem ist eine gegenüber der GNU `gettext API` etwas abstraktere, objektorientierte Schnittstelle vorhanden, auf die wir uns in diesem Abschnitt beziehen werden.

Zunächst ein paar Worte dazu, wie die Lokalisierung eines Programms vonstattengeht. Der Programmierer schreibt sein Programm, in dem die Benutzerführung vorzugsweise in englischer Sprache stattfindet. Zur Internationalisierung des Programms wird jeder String, der ausgegeben werden soll, durch eine sogenannte *Wrapper-Funktion* geschickt. Das ist eine Funktion, die den nicht-lokalisierten englischen String als Parameter übergeben bekommt und die passende Übersetzung zurückgibt.

Intern verwendet `gettext` zur Übersetzung verschiedene Sprachkompile. Das sind Binärdateien, die die Übersetzung des Programms in jeweils eine bestimmte Sprache enthalten. Diese Binärdateien werden aufgrund ihrer Dateiendung *.mo-Dateien* genannt. Wie diese Dateien erzeugt werden, wird unter anderem Inhalt des nächsten Abschnitts sein.

15.2.1 Beispiel für die Verwendung von gettext

Das Modul `gettext` spielt immer dann eine Rolle, wenn ein Programm veröffentlicht und somit einer großen Gruppe von Anwendern zugänglich gemacht wird. Dabei ist es besonders bei Open-Source-Projekten üblich, dass das Programm ursprünglich nur in einer oder zwei Sprachen veröffentlicht wird und weitere Übersetzungen später von den Anwendern erstellt und an den Autor geschickt werden. Damit dies funktioniert, sollte der Programmierer jedoch zumindest die Übersetzbarkeit seines Programms gewährleisten.

An dieser Stelle zeigen wir die Verwendung von `gettext` an einem kleinen Beispielprogramm. Dabei kann selbstverständlich nicht die vollständige Funktionalität von `gettext` zum Tragen kommen. Der Quellcode des Beispielprogramms sieht folgendermaßen aus:

```
import gettext
import random

trans = gettext.translation("meinprogramm", "locale", ["de"])
trans.install()

werte = []
while True:
    w = input(_("Please enter a value: "))
    if not w:
        break
    werte.append(w)
print(_("The random choice is {0}").format(random.choice(werte)))
```

Das Programm selbst ist unspektakulär – es liest so lange Strings vom Benutzer ein, bis einer dieser Strings leer ist, der Benutzer also  gedrückt hat, ohne eine

Eingabe zu tätigen. Dann wählt das Programm zufällig einen dieser Strings und gibt ihn aus. Mit diesem Programm könnte also beispielsweise eine zufällig gewählte Person einer Gruppe für den nächsten Samstagabend zum Fahrer ernannt werden. Beachten Sie aber besonders, dass die Interaktion mit dem Benutzer ausschließlich auf Englisch geschieht, jeder String, der ausgegeben wird, aber vorher durch eine Funktion namens `_` geschickt wird. Beachten Sie außerdem bei der `print`-Ausgabe, dass die Funktion `_` um den mit einem Platzhalter behafteten String geschrieben wurde, also bevor der Platzhalter durch dynamischen Inhalt ersetzt wurde. Das ist wichtig, da sonst keine Übersetzung erfolgen kann.

Der eigentlich interessante Teil des Programms sind die beiden Zeilen nach den `import`-Anweisungen:

```
trans = gettext.translation("meinprogramm", "locale", ["de"])
trans.install()
```

Hier wird ein sogenanntes Translation-Objekt erstellt. Das ist eine Instanz, die die Übersetzung aller Strings in eine bestimmte Sprache gewährleistet. Um ein solches Objekt zu erstellen, wird die Funktion `gettext.translation` aufgerufen. Diese bekommt einen frei wählbaren Namen, die sogenannte *Domain*, als ersten Parameter. Der zweite Parameter ist das Unterverzeichnis, in dem sich die Übersetzungen befinden, und der dritte Parameter ist schließlich eine Liste von Sprachen. Das Translation-Objekt übersetzt nun in die erste Sprache aus der Liste, für die ein Sprachkompilat gefunden werden kann.

Durch Aufruf der Methode `install` des Translation-Objekts installiert dieses seine interne Übersetzungsmethode als Funktion `_` im lokalen Namensraum. Damit werden alle Strings, mit denen die Funktion `_` aufgerufen wird, in die Sprache übersetzt, für die das Translation-Objekt steht, sofern denn eine Übersetzung verfügbar ist.

Erstellen des Sprachkompilats

Zum Erstellen des Sprachkompilats müssen Sie zunächst eine Liste aller zu übersetzenden Strings erstellen. Das sind all jene, die vor der Ausgabe durch die Funktion `_` geschickt werden. Da es eine unzumutbare Arbeit wäre, diese Liste von Hand anzufertigen, ist in Python ein Programm namens *pygettext.py*¹ im Liefer-

1 Unter Windows finden Sie das Programm *pygettext.py* im Unterordner *Tools/i18n* Ihrer Python-Installation. Unter Unix-ähnlichen Betriebssystemen wie beispielsweise Linux sollte sich die Programmdatei im Systempfad befinden und direkt ausführbar sein.

Möglicherweise installiert Ihre Linux-Distribution das Programm *pygettext.py* von Python 3 unter einem veränderten Namen, beispielsweise *pygettext3.py*, um die parallele Installation von Python 2 und 3 zu ermöglichen.

umfang enthalten, das genau dies für Sie erledigt. Das Programm erstellt eine sogenannte *.po-Datei*. Das ist eine für Menschen lesbare Variante des *.mo*-Dateiformats. Diese *.po*-Datei wird dann von den Übersetzern in verschiedene Sprachen übersetzt. Dies kann von Hand geschehen oder durch Einsatz diverser Tools, die für diesen Zweck existieren. Die für unser Beispielprogramm erzeugte *.po*-Datei sieht folgendermaßen aus:

```
[...]

#: main.py:9
msgid "Please enter a value: "
msgstr "Bitte geben Sie einen Wert ein: "

#: main.py:13
msgid "The random choice is {0}"
msgstr "Die Zufallswahl ist {0}"
```

Anstelle der Auslassungszeichen enthält die Datei Informationen wie etwa den Autor, die verwendete Software oder das Encoding der Datei.

Eine übersetzte *.po*-Datei wird durch das Programm *msgfmt.py*², das ebenfalls zum Lieferumfang von Python gehört, in das binäre *.mo*-Format kompiliert. Ein fertiges Sprachkompilat muss sich in folgendem Ordner befinden, damit es von *gettext* als solches gefunden wird:

Programmverzeichnis/Unterordner/Sprache/LC_MESSAGES/Domain.mo

Der Name des Verzeichnisses *Unterordner* wird beim Aufruf der Funktion *gettext.translate* angegeben und war in unserem Beispiel *locale*. Dieses Verzeichnis muss für jede vorhandene Sprache ein weiteres Verzeichnis enthalten, das seinerseits über ein Unterverzeichnis *LC_MESSAGES* verfügen muss. Das Sprachkompilat selbst muss die im Programm angegebene Domain als Namen haben.

In unserem Beispielprogramm muss das Sprachkompilat also in folgendem Verzeichnis liegen:

Programmverzeichnis/locale/de/LC_MESSAGES/meinprogramm.mo

2 Unter Windows finden Sie das Programm *msgfmt.py* im Unterordner *Tools/i18n* Ihrer Python-Installation. Unter Unix-ähnlichen Betriebssystemen wie beispielsweise Linux sollte sich die Programmdatei im Systempfad befinden und direkt ausführbar sein. Hier gilt derselbe Hinweis wie bei *pygettext.py*.

Wenn das Sprachkompilat nicht vorhanden ist, wird beim Aufruf der Funktion `gettext.translate` eine entsprechende Exception geworfen:

```
Traceback (most recent call last):
  [...]
IOError: [Errno 2] No translation file found for domain: 'meinprogramm'
```

Wenn das Sprachkompilat an seinem Platz ist, werden Sie beim Ausführen des Programms feststellen, dass alle Strings ins Deutsche übersetzt wurden:

```
Bitte geben Sie einen Wert ein: Donald Duck
Bitte geben Sie einen Wert ein: Daisy Duck
Bitte geben Sie einen Wert ein: Onkel Dagobert
Bitte geben Sie einen Wert ein:
Die Zufallswahl ist Donald Duck
```

Mit dem Modul `gettext` ist es also möglich, ein Programm ohne großen Aufwand in mehrere Sprachen zu übersetzen und darüber hinaus zu einem späteren Zeitpunkt weitere Sprachen hinzuzufügen, ohne große Änderungen am Programm selbst durchführen zu müssen.

15.3 Hash-Funktionen – hashlib

Das Modul `hashlib` der Standardbibliothek implementiert die gängigsten sogenannten *Hash-Funktionen*. Ganz allgemein sind das sehr komplexe Algorithmen, die aus einem Parameter, zumeist einem String, einen sogenannten *Hash-Wert* berechnen. Wozu kann ein solcher Hash-Wert verwendet werden?

Stellen Sie sich einmal vor, Sie würden eine Foren-Software entwickeln, die später für eine Community im Internet eingesetzt werden soll. Bevor ein Benutzer Beiträge im Forum verfassen darf, muss er sich mit seinem Benutzernamen und dem dazu passenden Passwort anmelden. Natürlich ist es im Sinne des Forenbetreibers und vor allem des Benutzers selbst, dass das Passwort nicht in falsche Hände gerät. Es stellt sich also die Frage, wie die Anmeldeprozedur möglichst sicher gestaltet werden kann.

Die intuitivste Möglichkeit wäre es, Benutzernamen und Passwort an die Foren-Software zu übermitteln. Dort werden diese beiden Informationen mit den Anmeldedaten aller Benutzer verglichen, und bei einem Treffer wird der Zugang zum Forum ermöglicht.

Würde eine solche Software die Anmeldeprozedur tatsächlich so durchführen, müssten Benutzername und Passwort im Klartext in der internen Datenbank des

Forums gespeichert werden. Das ist beim Benutzernamen kein größeres Problem, da es sich dabei im Allgemeinen um eine öffentliche Information handelt. Doch das Passwort im Klartext in einer solchen Datenbank zu speichern, wäre grob fahrlässig. Stellen Sie sich einmal vor, ein Angreifer würde über eine Sicherheitslücke in einem anderen Teil der Software Zugriff auf die Datenbank erlangen. Er wäre sofort im Besitz aller Passwörter der angemeldeten Benutzer. Das wird besonders dann brisant, wenn man bedenkt, dass viele Leute das gleiche Passwort für mehrere Benutzerkonten verwenden.

Wünschenswert wäre es also, die Korrektheit eines Passworts mit an Sicherheit grenzender Wahrscheinlichkeit zu ermitteln, ohne Referenzpasswörter im Klartext speichern zu müssen. Und genau hier kommen Hash-Funktionen ins Spiel. Eine Hash-Funktion bekommt einen Parameter übergeben und errechnet daraus eine Art Prüfsumme, den sogenannten *Hash-Wert*. Wenn sich jetzt also ein neuer Benutzer bei der Foren-Software anmeldet und sein Passwort wählt, wird dieses nicht im Klartext in die Datenbank eingetragen, sondern es wird der Hash-Wert des Passworts gespeichert.

Beim Einloggen schickt der Benutzer sein Passwort an den Server. Dieser errechnet dann den Hash-Wert des übertragenen Passworts und vergleicht ihn mit den gespeicherten Hash-Werten.

Damit eine solche Anmeldeprozedur funktioniert und ein potentieller Angreifer auch mit Zugriff auf die Datenbank keine Passwörter errechnen kann, müssen Hash-Funktionen einige Bedingungen erfüllen:

- ▶ Eine Hash-Funktion stellt eine *Einwegkodierung* dar. Das heißt, dass die Berechnung des Hash-Wertes nicht umkehrbar ist, man also aus einem Hash-Wert nicht auf den ursprünglichen Parameter schließen kann.
- ▶ Bei Hash-Funktionen treten grundsätzlich sogenannte *Kollisionen* auf, das sind zwei verschiedene Parameter, die denselben Hash-Wert ergeben. Ein wesentlicher Schritt zum Knacken einer Hash-Funktion ist es, solche Kollisionen berechnen zu können. Eine Hash-Funktion sollte also die Berechnung von Kollisionen so stark erschweren, dass sie nur unter extrem hohem Zeitaufwand zu bestimmen wären.
- ▶ Eine Hash-Funktion sollte möglichst willkürlich sein, so dass man nicht aufgrund eines ähnlichen Hash-Wertes darauf schließen kann, dass man in der Nähe des gesuchten Passworts ist. Sobald der Parameter der Hash-Funktion minimal verändert wird, sollte ein völlig verschiedener Hash-Wert berechnet werden.

- Zu guter Letzt sollte eine Hash-Funktion selbstverständlich schnell zu berechnen sein. Außerdem müssen sich die entstehenden Hash-Werte untereinander effizient vergleichen lassen.

Das Anwendungsfeld von Hash-Funktionen ist weit gefächert. So werden sie, abgesehen von dem obigen Passwortbeispiel, unter anderem auch zum Vergleichen großer Dateien verwendet. Anstatt diese Dateien untereinander Byte für Byte zu vergleichen, werden ihre Hash-Werte berechnet und verglichen. Mit den Hash-Werten lässt sich sagen, ob die Dateien mit Sicherheit verschieden oder mit großer Wahrscheinlichkeit identisch sind. Das ist besonders dann interessant, wenn es aufgrund eingeschränkter Bandbreite gar nicht möglich ist, die Dateien direkt zu vergleichen. So ist der Vergleich der Hash-Werte beispielsweise die effizienteste Methode, die Authentizität einer aus dem Internet heruntergeladenen Datei zu überprüfen.

Beachten Sie, dass die Wahrscheinlichkeit einer Kollision bei den im Modul `hashlib` implementierten Verfahren sehr gering, aber theoretisch immer noch vorhanden ist.

15.3.1 Verwendung des Moduls

Zunächst enthält das Modul `hashlib` eine Reihe von Klassen, die jeweils einen Hash-Algorithmus implementieren:

Klasse	Algorithmus	Beschreibung
<code>hashlib.md5</code>	MD5	<i>Message-Digest Algorithm 5</i> Erzeugt aus einem beliebigen String einen 128-Bit-Hash-Wert.
<code>hashlib.sha1</code>	SHA-1	<i>Secure Hash Algorithm 1</i> Erzeugt aus einem beliebigen String einen 160-Bit-Hash-Wert. Beachten Sie, dass der SHA-1-Algorithmus bereits ansatzweise geknackt wurde, also nicht mehr zum Speichern von Passwörtern verwendet werden sollte.
<code>hashlib.sha224</code>	SHA-224	<i>Secure Hash Algorithm 224</i> Erzeugt aus einem beliebigen String einen 224-Bit-Hash-Wert.
<code>hashlib.sha256</code>	SHA-256	<i>Secure Hash Algorithm 256</i> Erzeugt aus einem beliebigen String einen 256-Bit-Hash-Wert.

Tabelle 15.6 Unterstützte Hash-Funktionen

Klasse	Algorithmus	Beschreibung
<code>hashlib.sha384</code>	SHA-384	<i>Secure Hash Algorithm 384</i> Erzeugt aus einem beliebigen String einen 384-Bit-Hash-Wert.
<code>hashlib.sha512</code>	SHA-512	<i>Secure Hash Algorithm 512</i> Erzeugt aus einem beliebigen String einen 512-Bit-Hash-Wert.

Tabelle 15.6 Unterstützte Hash-Funktionen (Forts.)

Die Verwendung dieser Klassen ist identisch. Deshalb wird sie hier exemplarisch an der Klasse `md5` gezeigt.

Beim Instantiieren der Klasse `md5` wird eine `bytes`-Instanz übergeben, deren Hash-Wert berechnet werden soll. Das Berechnen eines Hash-Wertes aus einem String ist seit Python 3.0 nicht mehr möglich.

```
>>> import hashlib
>>> m = hashlib.md5(b"Hallo Welt")
```

Durch Aufruf der Methode `digest` wird der berechnete Hash-Wert als Bytefolge zurückgegeben. Beachten Sie, dass die zurückgegebene `bytes`-Instanz durchaus nicht-druckbare Zeichen enthalten kann.

```
>>> m.digest()
b'\x72\x09\xaet\x8aL\x04\x0e\xba\xdcQ\xa8'
```

Durch Aufruf der Methode `hexdigest` wird der berechnete Hash-Wert als String zurückgegeben, der eine Folge von zweistelligen Hexadezimalzahlen enthält. Diese Hexadezimalzahlen repräsentieren jeweils ein Byte des Hash-Wertes. Der zurückgegebene String enthält ausschließlich druckbare Zeichen.

```
>>> m.hexdigest()
'5c372a32c9ae748a4c040ebadc51a829'
```

15.3.2 Beispiel

Das folgende kleine Beispielprogramm verwendet das Modul `hashlib`, um einen Passwortschutz zu realisieren. Das Passwort soll dabei nicht als Klartext im Quelltext gespeichert werden, sondern als Hash-Wert. Dadurch ist gewährleistet, dass die Passwörter nicht einsehbar sind, selbst wenn jemand in den Besitz der Hash-Werte kommen sollte. Auch anmeldepflichtige Internetportale wie beispielsweise Foren speichern die Passwörter der Benutzer als Hash-Wert.

```
import hashlib

pwhash = "578127b714de227824ab105689da0ed2"

m = hashlib.md5(bytes(input("Ihr Passwort bitte: "), "utf-8"))
if pwhash == m.hexdigest():
    print("Zugriff erlaubt")
else:
    print("Zugriff verweigert")
```

Das Programm liest ein Passwort vom Benutzer ein, errechnet den MD5-Hash-Wert dieses Passworts und vergleicht ihn mit dem gespeicherten Hash-Wert. Der vorher berechnete Hash-Wert `pwhash` ist in diesem Fall im Programm vorgegeben. Unter normalen Umständen stünde er mit anderen Hash-Werten in einer Datenbank oder wäre in einer Datei gespeichert. Wenn beide Werte übereinstimmen, wird symbolisch »Zugriff erlaubt« ausgegeben. Das Passwort für dieses Programm lautet »Mein Passwort«.

16 Datum und Zeit

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

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

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

Wir werden im Folgenden beide Module und ihre Funktionen beleuchten.

16.1 Elementare Zeitfunktionen – `time`

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

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

Ein Unix-Timestamp mit dem Wert `1190132696.0` markiert beispielsweise den 18.09.2007 um 18:24 Uhr und 56 Sekunden, da seit dem Beginn der Unix-Epoche bis zu diesem Zeitpunkt genau 1190132696,0 Sekunden vergangen sind.

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

Die Lokalzeit ist abhängig von dem Standort der jeweiligen Uhr und bezieht sich darauf, was die Uhren an diesem Standort anzeigen müssen, um richtig zu gehen.

Als koordinierte Weltzeit wird die Lokalzeit auf dem Null-Meridian verstanden, der unter anderem durch Großbritannien verläuft. Die koordinierte Weltzeit wird mit *UTC* für *Coordinated Universal Time* abgekürzt.¹ Alle Lokalzeiten lassen sich relativ zur UTC angeben, indem man die Abweichung in Stunden nennt. Beispielsweise hat Mitteleuropa die Lokalzeit UTC+1, was bedeutet, dass unsere Uhren im Vergleich zu denen in Großbritannien um eine Stunde vorgehen.

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

Neben der schon angesprochenen Zeitdarstellung durch Unix-Timestamps gibt es ein weiteres Format, das durch einen eigenen Datentyp namens `struct_time` implementiert wird. Instanzen des Typs `struct_time` haben neun Attribute, die wahlweise über einen Index oder ihren Namen angesprochen werden können. Die folgende Tabelle zeigt den genauen Aufbau des Datentyps:

Index	Attributname	Bedeutung und Wertebereich
0	<code>tm_year</code>	Die Jahreszahl des Zeitstempels Werte ² : 1970–2038
1	<code>tm_mon</code>	Nummer des Monats Werte: 1–12
2	<code>tm_mday</code>	Nummer des Tags im Monat Werte: 1–31
3	<code>tm_hour</code>	Stunde der Uhrzeit des Zeitstempels Werte: 0–23
4	<code>tm_min</code>	Minute der Uhrzeit des Zeitstempels Werte: 0–59

Tabelle 16.1 Aufbau des Datentyps `struct_time`

1 Nein, die Abkürzung UTC für *Coordinated Universal Time* ist nicht fehlerhaft, sondern rührt daher, dass man einen Kompromiss zwischen der englischen Variante »Coordinated Universal Time« und der französischen Bezeichnung »Temps Universel Coordonné« finden wollte.

2 Diese Begrenzung kommt durch den Wertebereich für die Unix-Timestamps zustande. Und ja, alle Programme, die auf Unix-Zeitstempel setzen, werden im Jahr 2038 ein Problem bekommen ...

Index	Attributname	Bedeutung und Wertebereich
5	tm_sec	Sekunde der Uhrzeit des aktuellen Zeitstempels Werte ³ : 0–61
6	tm_wday	Nummer des Wochentages Werte: 0–6 (0 entspricht Montag)
7	tm_yday	Nummer des Tages im Jahr Werte: 0–366
8	tm_isdst	Gibt an, ob der Zeitstempel durch die Sommerzeit angepasst wurde. Werte: 0 für »Nein«, 1 für »Ja« und -1 für »Unbekannt«

Tabelle 16.1 Aufbau des Datentyps struct_time (Forts.)

Allen Funktionen, die struct_time-Instanzen als Parameter erwarten, können Sie alternativ auch ein Tupel mit neun Elementen übergeben, das für die entsprechenden Indizes die gewünschten Werte enthält.

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

Attribute

time.accept2dyear

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

time.altzone

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

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

time.daylight

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

- Es ist tatsächlich der Bereich von 0 bis 61, um sogenannte Schaltsekunden zu kompensieren. Schaltsekunden dienen dazu, die Ungenauigkeiten der Erdrotation bei Zeitangaben auszugleichen. Sie werden sich in der Regel nicht darum kümmern müssen.

time.struct_time

Referenz auf den eingangs besprochenen Datentyp `struct_time`.

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

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

time.timezone

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

time.tzname

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

```
>>> time.tzname
('CET', 'CEST')
```

Funktionen**time.asctime([t])**

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

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

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

time.clock()

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

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

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

```
>>> start = time.clock()
>>> rechenintensive_funktion()
>>> ende = time.clock()
>>> print("Die Funktion lief "
...       "{0:1.2f} Sekunden".format(ende - start))
Die Funktion lief 7.46 Sekunden
```

time.ctime([secs])

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

time.gmtime([secs])

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

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

```
>>> time.gmtime()
time.struct_time(tm_year=2009, tm_mon=1, tm_mday=18, tm_hour=16,
tm_min=11, tm_sec=45, tm_wday=6, tm_yday=18, tm_isdst=0)
```

Das obige Beispiel wurde also nach UTC am 18.01.2009 um 16:11 Uhr ausgeführt.

time.localtime([secs])

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

time.mktime(t)

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

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

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

time.sleep(secs)

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

time.strftime(format[, t])

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

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

Platzhalter	Bedeutung
%a	lokale Abkürzung für den Namen des Wochentags
%A	der komplette Name des Wochentags in der lokalen Sprache
%b	lokale Abkürzung für den Namen des Monats
%B	der vollständige Name des Monats in der lokalen Sprache
%c	das Format für eine angemessene Datums- und Zeitdarstellung auf der lokalen Plattform
%d	Nummer des Tages im aktuellen Monat. Ergibt einen String der Länge 2 im Bereich [01,31].
%H	Stunde im 24-Stunden-Format. Das Ergebnis hat immer zwei Ziffern und liegt im Bereich [00,23].
%I	Stunde im 12-Stunden-Format. Das Ergebnis hat immer zwei Ziffern und liegt im Bereich [01,12].
%j	Nummer des Tages im Jahr. Das Ergebnis hat immer drei Ziffern und liegt im Bereich [001, 366].
%m	Nummer des Monats bestehend aus zwei Ziffern im Bereich [01,12]
%M	Minute als Zahl mit zwei Ziffern. Liegt immer im Bereich [00,59].
%p	Die lokale Entsprechung für AM bzw. PM ⁴
%S	Sekunde als Zahl mit zwei Ziffern. Liegt immer im Bereich [00,61].

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

4 Von lat. »*Ante Meridiem*« (dt. »vor dem Mittag«) bzw. lat. »*Post Meridiem*« (»nach dem Mittag«)

Platzhalter	Bedeutung
%U	Nummer der aktuellen Woche im Jahr, wobei der Sonntag als erster Tag der Woche betrachtet wird. Das Ergebnis hat immer zwei Ziffern und liegt im Bereich [01,53]. Der Zeitraum am Anfang eines Jahres vor dem ersten Sonntag wird als 0. Woche gewertet.
%w	Nummer des aktuellen Tages in der Woche. Sonntag wird als 0. Tag betrachtet. Das Ergebnis liegt im Bereich [0,6].
%W	Wie %U, nur dass statt des Sonntags der Montag als erster Tag der Woche betrachtet wird.
%x	Datumsformat der lokalen Plattform
%X	Zeitformat der lokalen Plattform
%y	Jahr ohne Jahrhundertangabe. Das Ergebnis besteht immer aus zwei Ziffern und liegt im Bereich [00,99].
%Y	komplette Jahreszahl mit Jahrhundertangabe
%Z	Name der lokalen Zeitzone oder ein leerer String, wenn keine lokale Zeitzone festgelegt wurde
%%	Ergibt ein Prozentzeichen % im Resultatstring.

Tabelle 16.2 Übersicht über alle Platzhalter der `time.strftime`-Funktion (Forts.)

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

```
>>> time.strftime("%d.%m.%Y um %H:%M:%S Uhr")
'20.01.2009 um 12:50:41 Uhr'
```

time.strptime(string[, format])

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

```
>>> zeit_string = '19.09.2007 um 00:21:17 Uhr'
>>> time.strptime(zeit_string, "%d.%m.%Y um %H:%M:%S Uhr")
time.struct_time(tm_year=2007, tm_mon=9, tm_mday=19, tm_hour=0,
tm_min=21, tm_sec=17, tm_wday=2, tm_yday=262, tm_isdst=-1)
```

Geben Sie den optionalen Parameter *format* nicht an, wird der Standardwert `"%a %b %d %H:%M:%S %Y"` verwendet. Dies entspricht dem Ausgabeformat von `time.ctime`.

time.time()

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

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

16.2 Komfortable Datumsfunktionen – `datetime`

Das Modul `datetime` ist im Vergleich zum `time`-Modul wesentlich abstrakter und durch seine eigenen Zeit- und Datumsstypen auch wesentlich angenehmer zu benutzen.

Das Modul unterscheidet zwei Arten von Datums- und Zeitobjekten: die sogenannten *naiven* und die *bewussten* Objekte.

Ein *naives* Objekt kümmert sich nicht darum, auf welche Zeitzone sich sein Wert bezieht, und enthält auch keine Informationen darüber, wohingegen ein *bewusstes* Objekt mit Informationen zu seiner Zeitzone verknüpft ist. Ihre Programme können selbst entscheiden, ob die von ihnen benutzten Objekte *naiv* oder *bewusst* sind.

Wie das genau funktioniert, wird hier nicht näher thematisiert. Weitere Informationen darüber finden Sie in der Python-Dokumentation.

Konstanten des Moduls `datetime`

Es gibt zwei Konstanten, die das `datetime`-Modul definiert, um den Wertebereich für die Jahreszahlen zu definieren:

`datetime.MINYEAR`

Der minimal mögliche Wert für eine Jahreszahl. Der Wert ist in der Regel 1.

`datetime.MAXYEAR`

Der maximal mögliche Wert für eine Jahreszahl. Der Wert ist in der Regel 9999.

Die fünf Datentypen von `datetime`

Das Modul `datetime` definiert fünf eigene Datentypen für den Umgang mit Datum und Zeit. Alle diese Datentypen sind *immutable*.

`datetime.date`

Ein Datentyp zum Speichern von Datumsangaben. Alle Instanzen dieses Datentyps sind prinzipiell *naiv*, kümmern sich also nicht um die Gegebenheiten der lokalen Zeitzone.

datetime.time

Mit `datetime.time` werden Zeitpunkte an einem Tag gespeichert. Dabei wird idealisiert angenommen, dass jeder Tag $24 * 60 * 60$ Sekunden umfasst und dass es keine Schaltsekunden gibt.

datetime.datetime

Die Kombination aus `datetime.date` und `datetime.time` zum Speichern von ganzen Zeitpunkten, die sowohl ein Datum als auch eine Uhrzeit umfassen. Der Datentyp `datetime.datetime` ist der wichtigste des Moduls.

datetime.timedelta

Es ist möglich, Differenzen zwischen `datetime.date`- und auch `datetime.datetime`-Instanzen zu bilden. Die Ergebnisse solcher Subtraktionen sind `datetime.timedelta`-Objekte.

datetime.tzinfo

Dieser Typ wird benötigt, um mit Zeitzonen umzugehen. Dafür muss das Programm eine Subklasse von `datetime.tzinfo` erzeugen und bestimmte Methoden überschreiben.

Aus Platzgründen werden wir diesen Datentyp nicht behandeln. Allerdings finden Sie in der Python-Dokumentation ein gutes Beispiel für die Implementation einer `datetime.tzinfo`-Klasse.

16.2.1 datetime.date

Hier werden wir die Attribute und Methoden des Datentyps `datetime.date` behandeln.

Konstruktoren der Klasse datetime.date

Es gibt drei Konstruktoren für `datetime.date`-Instanzen:

datetime.date(year, month, day)

Erzeugt eine neue Instanz des Datentyps `datetime.date`, die den durch die Parameter festgelegten Tag repräsentiert. Dabei müssen die Parameter folgenden Bedingungen genügen:

- ▶ `datetime.MINYEAR <= year <= datetime.MAXYEAR`
- ▶ `1 <= month <= 12`
- ▶ `1 <= day <= (Anzahl der Tage des übergebenen Monats)`

```
>>> geburtstag = datetime.date(1987, 11, 3)
>>> geburtstag
datetime.date(1987, 11, 3)
```

datetime.date.today()

Erzeugt eine neue `datetime.date`-Instanz, die den aktuellen Tag repräsentiert:

```
>>> datetime.date.today()
datetime.date(2007, 9, 19)
```

datetime.date.fromtimestamp(timestamp)

Erzeugt ein neues `datetime.date`-Objekt, das das Datum des übergebenen Unix-Timestamps speichert.

Klassen-Member von datetime.date

datetime.date.min

Ein Klassenattribut, das den frühesten Tag enthält, der durch den `datetime.date`-Typ abgebildet werden kann. Wie das folgende Beispiel zeigt, ist dies der 1. Januar im Jahr 1:

```
>>> datetime.date.min
datetime.date(1, 1, 1)
```

datetime.date.max

Das Klassenattribut `datetime.date.max` speichert eine `datetime.date`-Instanz, die den spätesten Tag repräsentiert, der von `datetime.date` verwaltet werden kann: den 31.12. im Jahr 9999.

```
>>> datetime.date.max
datetime.date(9999, 12, 31)
```

Operatoren für datetime.date-Instanzen

Sie können Differenzen zwischen zwei `datetime.date`-Instanzen bilden. Das Ergebnis einer solchen Subtraktion ist ein `datetime.timedelta`-Objekt:

```
>>> datetime.date(1987, 11, 3) - datetime.date(1987, 7, 26)
datetime.timedelta(100)
```

In dem Beispiel liegen die beiden Zeitpunkte 100 Tage auseinander.

Es ist auch möglich, zu einer `datetime.date`-Instanz ein `datetime.timedelta`-Objekt zu addieren oder es davon abzuziehen. In diesem Fall ist das Ergebnis ein `datetime.date`-Objekt:

```
>>> datetime.date(1987, 7, 26) + datetime.timedelta(100)
datetime.date(1987, 11, 3)
```

Außerdem können `datetime.date`-Instanzen mit den Vergleichsoperatoren `<` und `>` verglichen werden. Dabei wird das Datum als »kleiner« betrachtet, das in der Zeit weiter in Richtung Vergangenheit liegt:

```
>>> datetime.date(1987, 7, 26) < datetime.date(1987, 11, 3)
True
```

Die Attribute und Methoden von `datetime.date`-Instanzen

Im Folgenden sei *d* eine `datetime.date`-Instanz.

d.year

Speichert das Jahr des Datums. Dieses Attribut kann nur gelesen werden.

d.month

Speichert den Monat des Datums. Dieses Attribut kann nur gelesen werden.

d.day

Speichert den Tag des Datums. Dieses Attribut kann nur gelesen werden.

d.replace(year, month, day)

Erzeugt ein neues Datum, dessen Attribute den übergebenen Parametern entsprechen. Fehlt eine Angabe, wird das entsprechende Attribut von *d* verwendet:

```
>>> d = datetime.date(1987, 7, 26)
>>> d.replace(month=11, day=3)
datetime.date(1987, 11, 3)
```

d.timetuple()

Gibt eine `time.struct_time`-Instanz⁵ zurück, die das Datum von *d* repräsentiert. Die Elemente für die Uhrzeit werden dabei auf 0 und das `tm_isdst`-Attribut wird auf -1 gesetzt:

```
>>> d = datetime.date(2007, 7, 6)
>>> d.timetuple()
time.struct_time(tm_year=2007, tm_mon=7, tm_mday=6, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=4, tm_yday=187, tm_isdst=-1)
```

5 Siehe dazu Abschnitt Elementare Zeitfunktionen – `time`, »Elementare Zeitfunktionen – `time`«.

d.weekday()

Gibt den Wochentag als Zahl zurück, wobei Montag als 0 und Sonntag als 6 angegeben werden.

d.isoweek()

Gibt den Wochentag als Zahl zurück, wobei Montag den Wert 0 und Sonntag den Wert 7 ergibt.

Siehe dazu auch `d.isocalendar()`.

d.isocalendar()

Gibt ein Tupel zurück, das drei Elemente enthält: (ISO year, ISO week number, ISO weekday).

Die Angaben in dem Tupel erfolgen dabei im Format des sogenannten *ISO-Kalenders*, der eine Variante des gregorianischen Kalenders ist. Im ISO-Kalender wird ein Jahr in 52 oder 53 Wochen geteilt. Jede der Wochen beginnt mit einem Montag und endet mit einem Sonntag. Die erste Woche eines Jahres, deren Donnerstag in diesem Jahr liegt, erhält im ISO-Kalender die Wochennummer 1.

Die drei Elemente des zurückgegebenen Tupels bedeuten:

(Jahr, Wochennummer, Tagesnummer).

Beispielsweise war der 01.01.2008 ein Dienstag, weshalb der 31.12.2007 der erste Tag im Jahr 2008 des ISO-Kalenders war:

```
>>> d = datetime.date(2007, 12, 31)
>>> d.isocalendar()
(2008, 1, 1)
```

d.isoformat()

Gibt einen String zurück, der den von *d* repräsentierten Tag im ISO-8601-Format enthält. Dieses Standardformat sieht folgendermaßen aus: YYYY-MM-DD, wobei die »Y« (engl. *year*) für die Ziffern der Jahreszahl, die »M« (engl. *month*) für die Ziffern der Monatszahl und die »D« (engl. *day*) für die Ziffern des Tages im Monat stehen.

```
>>> d = datetime.date(2007, 6, 18)
>>> d.isoformat()
'2007-06-18'
```

Achtung

Die Methode `isoformat` hat nichts mit dem ISO-Kalender zu tun, den die Methoden `isoweekday` und `isocalendar` verwenden.

d.ctime()

Gibt einen String in einem 24-Zeichen-Format aus, der den von *d* gespeicherten Tag repräsentiert. Die Platzhalter für Stunde, Minute und Sekunde werden dabei auf "00" gesetzt:

```
>>> d = datetime.date(2007, 10, 23)
>>> d.ctime()
'Tue Oct 23 00:00:00 2007'
```

d.strftime(format)

Gibt den von *d* repräsentierten Tag formatiert aus, wobei der Parameter *format* die Beschreibung des gewünschten Ausgabeformats enthält.

Nähere Informationen können Sie in Abschnitt 16.1, »Elementare Zeitfunktionen – time«, unter `time.strftime` nachschlagen.

16.2.2 datetime.time

In diesem Abschnitt werden wir uns mit den Methoden und Attributen des Datentyps `datetime.time` beschäftigen.

Objekte des Typs `datetime.time` dienen dazu, Tageszeiten anhand von Stunde, Minute, Sekunde und auch Mikrosekunde zu verwalten.

In dem Attribut `tzinfo` können `datetime.time`-Instanzen Informationen zur lokalen Zeitzone speichern und ihre Werte damit an die Lokalzeit anpassen. Dadurch ist es möglich, sowohl naive als auch bewusste `datetime.time`-Instanzen zu erzeugen.

Konstruktor von datetime.time

Ein neues `datetime.time`-Objekt erzeugen Sie mit dem folgenden Konstruktor:

`datetime.time([hour[, minute[, second[, microsecond[, tzinfo]]]])`

Die vier ersten Parameter legen den Zeitpunkt fest und müssen folgende Bedingungen erfüllen, wobei nur Ganzzahlen zugelassen sind:

- ▶ `0 <= hour < 24`
- ▶ `0 <= minute < 60`
- ▶ `0 <= second < 60`
- ▶ `0 <= microsecond < 1000000`

Die Standardbelegung für *hour*, *minute*, *second* und *microsecond* ist der Wert 0.

Für den letzten Parameter namens *tzinfo* können Informationen über die lokale Zeitzone in Form einer `datetime.tzinfo`-Instanz übergeben werden. Wie das genau geht, können Sie der Python-Dokumentation entnehmen.

Klassen-Member von `datetime.time`

Der Datentyp `datetime.time` verfügt über folgende Klassen-Member:

`datetime.time.min`

Der früheste Zeitpunkt, der gespeichert werden kann, in der Regel `datetime.time(0, 0)`.

`datetime.time.max`

Der späteste darstellbare Zeitpunkt, in der Regel `datetime.time(23, 59, 59, 999999)`.

`datetime.time.resolution`

Der minimale Unterschied zwischen zwei unterschiedlichen `datetime.time`-Objekten. Diese kleinstmögliche Zeiteinheit wird auch *Auflösung* genannt.

Attribute und Methoden von `datetime.time`-Instanzen

Nachfolgend wird angenommen, dass `t` eine Instanz des Datentyps `datetime.time` ist.

`t.hour`

Speichert den Stundenanteil des Zeitpunkts `t`.

`t.minute`

Speichert den Minutenanteil des Zeitpunkts `t`.

`t.second`

Speichert den Sekundenanteil des Zeitpunkts `t`.

`t.microsecond`

Speichert den Mikrosekundenanteil (Mikrosekunde = Millionstelsekunde) des Zeitpunkts `t`.

`t.tzinfo`

Information zur lokalen Zeitzone. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.

t.replace([hour[, minute[, second[, microsecond[, tzinfo]]]])

Analog zur `datetime.date.replace`-Methode.

Eignet sich sehr gut dazu, aus einem naiven ein bewusstes `time`-Objekt zu machen, ohne dass die Daten-Member angepasst werden. Dazu rufen Sie `replace` mit `tzinfo` als einzigem Parameter auf.

t.isoformat()

Gibt einen String zurück, der den Zeitpunkt `t` im ISO-8601-Format enthält. Das Format ist folgendermaßen aufgebaut, wobei die »H« (engl. *hour*) für die Ziffern der Stunde, die »M« für die Ziffern der Minute, die »S« für die Ziffern der Sekunden und die »m« für die Ziffern der Mikrosekunden stehen: »HH:MM:SS:mmmmmm«.

Ist das `microseconds`-Attribut von `t` gleich 0, entfallen die Mikrosekunden, und das Format verkürzt sich auf »HH:MM:SS«.

t.strftime(format)

Erzeugt einen String, der den Zeitpunkt `t` nach der Formatbeschreibung in *format* enthält. Näheres dazu können Sie unter `time.strftime` nachlesen.

t.utcoffset()

Wenn `t` ein bewusstes Objekt ist, also `t.tzinfo` nicht den Wert `None` hat, gibt `t.utcoffset` den Wert zurück, der von `t.tzinfo.utcoffset(None)` erzeugt wird.

Dies sollte die Verschiebung der Lokalzeit relativ zur UTC in Sekunden sein.

t.tzname()

Gibt den Namen der Zeitzone zurück, wenn `t.tzinfo` nicht den Wert `None` hat. Ist `t.tzinfo` gleich `None`, wird stattdessen `None` zurückgegeben.

(Der Wert wird dadurch bestimmt, dass intern `t.tzinfo.tzname(None)` aufgerufen wird.)

16.2.3 datetime.datetime

In den meisten Fällen werden die Fähigkeiten der Datentypen `datetime.date` und `datetime.time` jeweils einzeln nicht ausreichen, um Zeitpunkte zu verwalten, da Zeitangaben in der Regel aus einem Datum und der Uhrzeit an dem jeweiligen Tag bestehen.

Der Datentyp `datetime.datetime` ist genau das, was sein Name vermuten lässt: ein Typ zum Speichern von Datums- und Uhrzeitangaben. Er vereint dazu die Fähigkeiten von `datetime.date` und `datetime.time` in einem Datentyp.

Konstruktoren von `datetime.datetime`

Es gibt acht Konstruktoren, um neue `datetime.datetime`-Instanzen zu erzeugen:

`datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]])`

Die Parameter haben die gleiche Bedeutung wie die gleichnamigen Elemente der Konstruktoren von `datetime.date` und `datetime.time`, weshalb wir hier auf eine Wiederholung verzichten.

```
>>> bescherung = datetime.datetime(2007, 12, 24, 18, 30)
>>> bescherung
datetime.datetime(2007, 12, 24, 18, 30)
```

`datetime.datetime.today()`

Erzeugt eine `datetime.datetime`-Instanz, die die aktuelle Lokalzeit speichert. Das `tzinfo`-Attribut wird dabei immer auf `None` gesetzt.

```
>>> datetime.datetime.today()
datetime.datetime(2009, 1, 20, 13, 10, 27, 21335)
```

Achtung

Auch wenn der Name der Methode `today` (dt. »heute«) darauf schließen lassen könnte, dass nur die Attribute für das Datum und nicht die für die Zeit gesetzt werden, erzeugt `datetime.today` ein `datetime.datetime`-Objekt, das auch die Uhrzeit enthält.

`datetime.now([tz])`

Erzeugt eine `datetime.datetime`-Instanz mit dem aktuellen Datum und der aktuellen Zeit. Wird die Methode ohne Parameter aufgerufen, erzeugt sie das gleiche Ergebnis wie `datetime.datetime.today`.

Mit dem optionalen Parameter `tz` können Informationen zur Lokalzeit übergeben werden. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.

`datetime.utcnow()`

Gibt die aktuelle koordinierte Weltzeit (UTC) zurück, wobei das `tzinfo`-Attribut der resultierenden `datetime.datetime`-Instanz den Wert `None` hat.

datetime.fromtimestamp(timestamp[, tz])

Erzeugt eine `datetime.datetime`-Instanz, die den gleichen Zeitpunkt wie der für `timestamp` übergebene Unix-Zeitstempel repräsentiert.

Übergeben Sie für `tz` keinem Wert oder `None`, ist der Rückgabewert ein naives Zeitobjekt.

Wie Sie mit dem Parameter `tz` Informationen zur Zeitzone übergeben, erläutert die Python-Dokumentation.

datetime.utcfromtimestamp(timestamp)

Wandelt den übergebenen Unix-Timestamp in ein `datetime.datetime`-Objekt um, das die koordinierte Weltzeit (UTC) speichert. Der Unix-Zeitstempel wird dabei als lokale Zeit interpretiert. Deshalb wird bei der Umwandlung nach UTC die Zeitverschiebung berücksichtigt:

```
>>> import time
>>> t = time.time()
>>> datetime.datetime.fromtimestamp(t)
datetime.datetime(2009, 1, 20, 13, 13, 40, 548336)
>>> datetime.datetime.utcfromtimestamp(t)
datetime.datetime(2009, 1, 20, 12, 13, 40, 548336)
```

Wie Sie sehen, liegen die von `fromtimestamp` und `utcfromtimestamp` gelieferten `datetime.datetime`-Objekte um genau eine Stunde auseinander. Dies rührt daher, dass das Beispiel auf einem Computer mit deutscher Lokalzeit (UTC+1) während der Winterzeit ausgeführt wurde.

datetime.combine(date, time)

Erzeugt ein `datetime.datetime`-Objekt, das aus der Kombination von *date* und *time* hervorgeht. Der Parameter *date* muss eine `datetime.date`-Instanz enthalten, und der Parameter *time* muss auf ein `datetime.time`-Objekt verweisen.

Alternativ können Sie für *date* auch ein `datetime.datetime`-Objekt übergeben. In diesem Fall wird die in *date* enthaltene Uhrzeit ignoriert und nur das Datum betrachtet.

datetime.strptime(date_string, format)

Interpretiert den String, der als Parameter *date_string* übergeben wurde, gemäß der Formatbeschreibung aus *format* als Zeitinformation und gibt ein entsprechendes `datetime.datetime`-Objekt zurück.

Für die Formatbeschreibung gelten die gleichen Regeln wie bei `time.strptime`.

Operatoren für `datetime.datetime`

Der Datentyp `datetime.datetime` überlädt die Operatoren für die Subtraktion und Addition, so dass mit Zeitangaben gerechnet werden kann.

Dabei sind folgende Summen und Differenzen möglich, wobei `d1` und `d2` jeweils `datetime.datetime`-Instanzen sind und `t` ein `datetime.timedelta`-Objekt referenziert:

Ausdruck	Hinweise
<code>d2 = d1 + t</code>	Der von <code>d2</code> beschriebene Zeitpunkt ergibt sich, indem in der Zeit von <code>d1</code> aus um die von <code>t</code> beschriebene Zeitspanne in die Zukunft oder die Vergangenheit gegangen wird, je nachdem, ob der Wert von <code>t</code> positiv oder negativ ist. Das <code>datetime.datetime</code>-Objekt <code>d2</code> übernimmt außerdem das <code>tzinfo</code>-Attribut von <code>d1</code>.
<code>d2 = d1 - t</code>	Wie bei der Addition, außer dass nun bei positivem <code>t</code> in Richtung Vergangenheit und bei negativem <code>t</code> in Richtung Zukunft gegangen wird.
<code>t = d1 - d2</code>	Das <code>datetime.timedelta</code>-Objekt <code>t</code> beschreibt den zeitlichen Abstand zwischen den Zeitpunkten <code>d1</code> und <code>d2</code>. Dabei wird <code>t</code> so gewählt, dass <code>d1 = d2 + t</code> gilt. Diese Operation kann nur durchgeführt werden, wenn <code>d1</code> und <code>d2</code> bewusst oder beide naiv sind. Ist dies nicht der Fall, wird ein <code>TypeError</code> erzeugt. Die Details zu naiven und bewussten Zeitobjekten entnehmen Sie bitte der Python-Dokumentation.

Tabelle 16.3 Rechnen mit `datetime.datetime`

Es ist auch möglich, zwei `datetime.datetime`-Instanzen mit den Vergleichsoperatoren `<` und `>` zu vergleichen. Dabei gilt das Zeitobjekt als »kleiner«, das in der Zeit weiter in Richtung Vergangenheit liegt.

Beispiele für die Verwendung dieser Operatoren können Sie im Abschnitt über `datetime.date` nachlesen, da die Verwendung für `datetime.datetime` analog erfolgt.

Statische und dynamische Attribute von `datetime.datetime`

Der Datentyp `datetime.datetime` besitzt die gleichen Member wie die Datentypen `datetime.date` und `datetime.time`: `min`, `max`, `resolution`, `year`, `month`, `day`, `hour`, `minute`, `second` und `microsecond`.

Die Bedeutung der einzelnen Member können Sie in den Abschnitten zu `datetime.date` und `datetime.time` nachlesen.

Methoden von `datetime.datetime`-Instanzen

Im Folgenden wird davon ausgegangen, dass `d` eine Instanz des Datentyps `datetime.datetime` ist.

`d.date()`

Gibt ein `datetime.date`-Objekt zurück, das die gleichen `year`-, `month`- und `day`-Attribute wie `d` hat.

`d.time()`

Gibt ein `datetime.time`-Objekt zurück, das die gleichen `hour`-, `minute`-, `second`- und `microsecond`-Attribute wie `d` hat.

`d.timetz()`

Wie `d.time`, aber es wird zusätzlich das `tzinfo`-Attribut mitkopiert.

`d.replace( [year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]])`

Erzeugt eine neue `datetime.datetime`-Instanz, die aus `d` hervorgeht, indem die Attribute, die der `replace`-Methode übergeben wurden, durch die neuen Werte ersetzt werden.

`d.utcoffset()`

Wenn `d` ein bewusstes Objekt ist, also `d.tzinfo` nicht den Wert `None` hat, gibt `d.utcoffset` den Wert zurück, der von `d.tzinfo.utcoffset(None)` erzeugt wird. Dies sollte die Verschiebung der Lokalzeit relativ zur UTC in Sekunden sein.

`d.tzname()`

Gibt den Namen der Zeitzone zurück, wenn `d.tzinfo` nicht den Wert `None` hat. Ist `d.tzinfo` gleich `None`, wird stattdessen `None` zurückgegeben.

(Der Wert wird dadurch, indem intern `d.tzinfo.tzname(None)` aufgerufen wird.)

`d.timetuple()`

Gibt ein `time.struct_time`-Objekt zurück, das den von `d` beschriebenen Zeitpunkt enthält.

`d.utctimetuple()`

Wenn `d` ein *naives* Zeitobjekt ist, also wenn `d.tzinfo` den Wert `None` hat, verhält sich `d.utctimetuple` genau wie `d.timetuple`.

Ist `d` ein *bewusstes* Zeitobjekt, wird sein Wert erst in die globale Weltzeit umgerechnet und dann als `time.struct_time`-Instanz zurückgegeben.

d.weekday()

Gibt den Wochentag als Zahl zurück, wobei Montag als 0 und Sonntag als 6 betrachtet wird.

d.isoweekday()

Gibt den Wochentag als Zahl zurück, wobei Montag den Wert 1 und Sonntag den Wert 7 ergibt.

d.isocalendar()

Gibt ein Tupel mit drei Elementen zurück, das den von *d* beschriebenen Tag als Datum im ISO-Kalender ausdrückt.

Näheres dazu finden Sie unter der Methode `isocalendar` des Datentyps `datetime.date`.

d.isoformat()

Gibt den von *d* beschriebenen Zeitpunkt im ISO-8601-Format zurück. Das Format ist folgendermaßen aufgebaut:

YYYY-MM-DDTHH:MM:SS.mmmmmmm

Die »Y« stehen für die Ziffern der Jahreszahl, die »M« für die Ziffern der Monatszahl und die »D« für die Ziffern des Tages. Das große »T« ist ein Trennzeichen, das zwischen Datums- und Zeitangabe steht. In der Zeitangabe stehen die »H« für die Ziffern der Stunde, die »M« für die Ziffern der Minute und die »S« für die Ziffern der Sekunden.

Ist das `microseconds`-Attribut von *d* von 0 verschieden, werden die Mikrosekunden, durch einen Punkt abgetrennt, an das Ende des Strings geschrieben (in der Formatbeschreibung durch die »m« angedeutet). Ansonsten entfällt der Mikrosekundenteil inklusive Punkt.

d.ctime()

Gibt einen String zurück, der den von *d* repräsentierten Zeitpunkt beschreibt:

```
>>> datetime.datetime(1987, 07, 26, 10, 15, 00).ctime()
'Sun Jul 26 10:15:00 1987'
```

d.strftime()

Erzeugt einen String, der den von *d* beschriebenen Zeitpunkt formatiert enthält.

Genauereres können Sie unter `time.strftime` nachlesen.

»But I can only show you the door, you're the one that has to walk through it. – Tank, load the jump program.«
– Morpheus in »The Matrix«

17 Schnittstelle zum Betriebssystem

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

Die Schnittstelle wird durch das Modul `os` implementiert, mit dem wir uns im nächsten Abschnitt beschäftigen werden.

17.1 Funktionen des Betriebssystems – `os`

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

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

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

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

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

Wichtig

Seit Python 3.0 wird streng zwischen Text und Daten durch die Datentypen `str` und `bytes` unterschieden, wie Sie in Abschnitt 8.5, »Sequentielle Datentypen«, gelernt haben. Alle Methoden und Funktionen, die von `os` bereitgestellt werden und `str`-Objekte als Parameter akzeptieren, können stattdessen auch mit `bytes`-Objekten gerufen werden. Allerdings ändert sich damit auch der Rückgabewert entsprechend, denn anstelle von Strings werden dann `bytes`-Objekte zurückgegeben.

Kurz: `str` rein – `str` raus; `bytes` rein – `bytes` raus.

17.1.1 Zugriff auf den eigenen Prozess und andere Prozesse

`os.environ`

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

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

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

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

`os.getpid()`

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

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

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

os.system(cmd)

Mit `os.system` können Sie beliebige Kommandos des Betriebssystems aus, so als ob Sie es in einer echten Konsole tun würden. Beispielsweise lassen wir uns mit folgendem Beispiel einen neuen Ordner mit dem Namen `test_ordner` über das `mkdir`-Kommando anlegen:

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

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

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

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

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

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

```
>>> ausgabe = os.popen("dir /B C:\\")
>>> dateien = [zeile.strip() for zeile in ausgabe]
>>> dateien
['AUTOEXEC.BAT', 'CONFIG.SYS', 'Dokumente und Einstellungen',
'Programme', 'Python30', 'WINDOWS']
```

Die genaue Bedeutung von `mode` und `bufsize` können Sie in Kapitel 9, »Dateien«, nachlesen.

17.1.2 Zugriff auf das Dateisystem

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

Sie werden oft einen sogenannten *Pfad* (engl. *path*) als Parameter an die beschriebenen Funktionen übergeben können. Dabei unterscheiden wir zwischen abso-

luten und relativen Pfaden, wobei Letztere sich auf das aktuelle Arbeitsverzeichnis beziehen.

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

os.access(path, mode)

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

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

Konstante	Bedeutung
<code>os.F_OK</code>	Prüft, ob der Pfad überhaupt existiert.
<code>os.R_OK</code>	Prüft, ob der Pfad gelesen werden darf.
<code>os.W_OK</code>	Prüft, ob der Pfad geschrieben werden darf.
<code>os.X_OK</code>	Prüft, ob der Pfad ausführbar ist.

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

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

```
>>> os.access("C:\\Python30\\python.exe", os.F_OK | os.X_OK)
True
```

Der Python-Interpreter unter der Adresse `C:\\Python30\\python.exe` existiert und ist natürlich ausführbar.

os.chdir(path)

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

os.getcwd()

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

os.getcwdb()

Wie `os.getcwd`, gibt aber eine `bytes`-Instanz anstelle der `str`-Instanz zurück.

os.chmod(path, mode)

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

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

Wert	Beschreibung
1	ausführen
2	schreiben
4	lesen

Tabelle 17.2 Zugriffsflags für os.chmod

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

```
>>> os.chmod("eine_datei", 0o640)
```

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

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

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

os.listdir(path)

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

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

os.mkdir(path[, mode])

Legt einen neuen Ordner in dem mit *path* übergebenen Pfad an. Der optionale Parameter *mode* gibt dabei eine Bitmaske an, die die Zugriffsrechte für den neuen Ordner festlegt. Standardmäßig wird für *mode* die Oktalzahl `0o777` verwendet (siehe zu *mode* auch `os.chmod`).

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

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

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

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

os.makedirs(path[, mode])

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

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

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

os.remove(path)

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

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

os.removedirs(path)

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

Wenn Sie beispielsweise

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

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

`os.rename(src, dst)`

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

Achtung

Auf Unix-Systemen wird eine bereits unter dem Pfad `dst` erreichbare Datei ohne Meldung überschrieben, wenn Sie `os.rename` aufrufen. Bei bereits existierenden Ordnern wird aber weiterhin eine Exception erzeugt.

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

`os.renames(src, dst)`

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

`os.rmdir(path)`

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

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

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

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

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

Um dies genau zu verstehen, betrachten wir einen Beispielverzeichnisbaum:

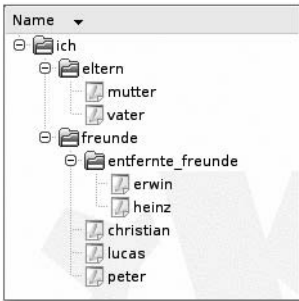


Abbildung 17.1 Beispielverzeichnisbaum

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

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

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

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

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

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

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

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

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

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

Wichtig

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

Achtung

Wenn Sie wie in unserem Beispiel einen relativen Pfadnamen angeben, dürfen Sie das aktuelle Arbeitsverzeichnis nicht während des Durchlaufens mittels `os.walk` verändern. Wenn Sie es dennoch tun, kann dies zu nicht definiertem Verhalten führen.

17.2 Umgang mit Pfaden – os.path

Verschiedene Plattformen – verschiedene Pfadnamenskonventionen. Während beispielsweise Windows-Betriebssysteme bei absoluten Pfadnamen das Laufwerk erwarten, auf das sich der Pfad bezieht, wird unter Unix ein einfacher Slash vorangestellt. Außerdem unterscheiden sich auch die Trennzeichen für einzelne Ordner innerhalb des Pfadnamens, denn Microsoft hat sich im Gegensatz zur Unix-Welt, in der der Slash üblich ist, für den Backslash entschieden.

Als Programmierer für plattformübergreifende Software stehen Sie nun vor dem Problem, dass Ihre Programme mit diesen verschiedenen Konventionen und auch denen dritter Betriebssysteme zurechtkommen müssen.

Damit dafür keine programmtechnischen Verrenkungen notwendig werden, wurde das Modul `os.path` entwickelt, mit dem Sie Pfadnamen komfortabel verwenden können.

Sie können das Modul auf zwei verschiedene Arten nutzen:

- ▶ Sie importieren erst `os` und greifen dann über `os.path` darauf zu.
- ▶ Sie importieren `os.path` direkt.

Bevor wir mit der Beschreibung der Funktionen dieses Moduls beginnen, möchten wir Sie darauf hinweisen, dass unter Windows nicht alle Funktionen korrekt mit UNC-Pfadnamen¹ umgehen können. Nur für `splitunc` und `ismount` wird garantiert, dass sie mit solchen Pfaden richtig verfahren können.

`os.path.abspath(path)`

Gibt zu einem relativen Pfad den dazugehörigen absoluten und normalisierten Pfad (siehe dazu `os.normpath`) zurück. Das folgende Beispiel verdeutlicht die Arbeitsweise:

```
>>> os.path.abspath(".")
'Z:\\beispiele\\os'
```

In diesem Fall haben wir mithilfe des relativen Pfads `"."` auf das aktuelle Verzeichnis herausgefunden, dass unser Script unter `'Z:\\beispiele\\os'` gespeichert ist.

`os.path.basename(path)`

Gibt den sogenannten *Basisnamen* des Pfads zurück. Der Basisname eines Pfads ist der Teil hinter dem letzten Ordnertrennzeichen, wie zum Beispiel `\` oder `/`. Diese Funktion eignet sich sehr gut, um den Dateinamen aus einem vollständigen Pfad zu extrahieren:

```
>>> os.path.basename(r"C:\Windows\System32\ntoskrnl.exe")
'ntoskrnl.exe'
```

Wichtig

Diese Funktion unterscheidet sich von dem Unix-Kommando *basename* dadurch, dass sie einen leeren String zurückgibt, wenn der String mit einem Ordnertrennzeichen endet:

```
>>> os.path.basename(r"/usr/lib/compiz/")
''
```

1 Uniform/Universal Naming Convention (UNC) ist ein Standard, um Ressourcen in einem Netzwerk anzusprechen.

Im Gegensatz dazu sieht die Ausgabe des gleichnamigen Unix-Kommandos so aus:

```
$ basename /usr/lib/compiz/
compiz
```

os.path.commonprefix(list)

Gibt einen möglichst langen String zurück, mit dem alle Elemente der als Parameter übergebenen Pfadliste *list* beginnen:

```
>>> os.path.commonprefix([r"C:\Windows\System32\ntoskrnl.exe",
                           r"C:\Windows\System\TAPI.dll",
                           r"C:\Windows\system32\drivers"])
'C:\\Windows\\'
```

Es ist aber nicht garantiert, dass der resultierende String auch ein gültiger und existierender Pfad ist, da die Pfade als einfache Strings betrachtet werden.

os.path.dirname(path)

Gibt den Ordnerpfad zurück, den *path* enthält:

```
>>> os.path.dirname(r"C:\Windows\System\TAPI.dll")
'C:\\Windows\\System'
```

Genau wie bei `os.path.basename` müssen Sie auch hier das abweichende Verhalten bei Pfaden beachten, die mit einem Ordnertrennzeichen enden:

```
>>> os.path.dirname(r"/usr/lib/compiz")
'/usr/lib'
>>> os.path.dirname(r"/usr/lib/compiz/")
'/usr/lib/compiz'
```

os.path.exists(path)

Gibt `True` zurück, wenn der angegebene Pfad auf eine existierende Datei oder ein vorhandenes Verzeichnis verweist, ansonsten `False`.

os.path.getatime(path)

Gibt den Unix-Zeitstempel des letzten Zugriffs auf den übergebenen Pfad zurück. Kann auf die übergebene Datei oder den Ordner nicht zugegriffen werden oder ist sie bzw. er nicht vorhanden, führt dies zu einem `os.error`.

Unix-Zeitstempel sind Ganzzahlen, die die Sekunden seit Beginn der Unix-Epoche, also dem 01.01.1970, angeben.

os.path.getmtime(path)

Gibt einen Unix-Zeitstempel zurück, der angibt, wann die Datei oder der Ordner unter *path* zum letzten Mal verändert wurde. Existiert der übergebene Pfad nicht im Dateisystem, wird `os.error` geworfen.

Unix-Zeitstempel sind Zahlen, die die Sekunden seit Beginn der Unix-Epoche, also dem 01.01.1970 um 00:00 Uhr, angeben.

os.path.getsize(path)

Gibt die Größe der unter *path* zu findenden Datei in Bytes zurück. Der Rückgabewert ist dabei immer eine `long`-Instanz.

os.path.isabs(path)

Der Rückgabewert ist `True`, wenn es sich bei *path* um eine absolute Pfadangabe handelt, sonst `False`.

os.path.isfile(path)

Gibt `True` zurück, wenn *path* auf eine Datei verweist, sonst `False`. Die Funktion folgt dabei gegebenenfalls symbolischen Links.

os.path.isdir(path)

Wenn der übergebene Pfad auf einen Ordner verweist, wird `True` zurückgegeben, ansonsten `False`.

os.path.islink(path)

Gibt `True` zurück, wenn unter *path* ein symbolischer Link zu finden ist, sonst `False`.

os.path.join(path1[, path2[, ...]])

Fügt die übergebenen Pfadangaben zu einem einzigen Pfad zusammen, indem sie verkettet werden:

```
>>> os.path.join(r"C:\Windows", r"System\ntoskrnl.exe")
'C:\Windows\System\ntoskrnl.exe'
```

Wird ein absoluter Pfad als zweites oder späteres Argument übergeben, ignoriert `os.path.join` alle übergebenen Pfade vor dem absoluten:

```
>>> os.path.join(r"Das\wird\ignoriert", r"C:\Windows",
                 r"System\ntoskrnl.exe")
'C:\Windows\System\ntoskrnl.exe'
```

os.path.normcase(path)

Auf Betriebssystemen, die bei Pfaden nicht hinsichtlich Groß- und Kleinschreibung unterscheiden (z. B. Windows), werden alle Großbuchstaben durch ihre kleinen Entsprechungen ersetzt. Außerdem werden unter Windows alle Slashes durch Backslashes ausgetauscht:

```
>>> os.path.normcase(r"C:\Windows\System32\ntoskrnl.exe")
'c:\windows\system32\ntoskrnl.exe'
```

Unter Unix wird der übergebene Pfad ohne Änderung zurückgegeben.

os.path.realpath(path)

Gibt einen zu *path* äquivalenten Pfad zurück, der keine Umwege über symbolische Links enthält.

os.path.split(path)

Teilt den übergebenen Pfad in den Namen des Ordners oder der Datei, die er beschreibt, und den Pfad zu dem direkt übergeordneten Verzeichnis und gibt ein Tupel zurück, das die beiden Teile enthält:

```
>>> os.path.split(r"C:\Windows\System32\ntoskrnl.exe")
('C:\Windows\System32', 'ntoskrnl.exe')
```

Wichtig

Wenn der Pfad mit einem Slash oder Backslash endet, ist das zweite Element des Tupels ein leerer String:

```
>>> os.path.split("/home/revelation/")
('/home/revelation', '')
```

os.path.splitdrive(path)

Teilt den übergebenen Pfad in die Laufwerksangabe und den Rest, sofern die Plattform Laufwerksangaben unterstützt:

```
>>> os.path.splitdrive(r"C:\Windows\System32\ntoskrnl.exe")
('C:', '\\Windows\System32\ntoskrnl.exe')
```

os.path.splitext(path)

Teilt den *path* in den Pfad zu der Datei und die Dateierweiterung. Beide Elemente werden in einem Tupel zurückgegeben:

```
>>> os.path.splitext(r"C:\Windows\System32\notepad.exe")
('C:\Windows\System32\notepad', '.exe')
```

17.3 Zugriff auf die Laufzeitumgebung – sys

Das Modul `sys` der Standardbibliothek stellt Konstanten und Funktionen zur Verfügung, die sich auf den Python-Interpreter selbst beziehen oder eng mit diesem zusammenhängen. So können Sie über das Modul `sys` beispielsweise die Versionsnummer des Interpreters oder des Betriebssystems abfragen. Das Modul stellt dem Programmierer eine Reihe von Informationen zur Verfügung, die mitunter sehr nützlich sein können. Es lohnt sich also, sich einen Überblick über die Funktionalität von `sys` zu verschaffen, allein schon, um einen Begriff davon zu bekommen, an welche Informationen Sie durch dieses Modul gelangen können.

Um die Beispiele dieses Abschnitts ausführen zu können, muss zuvor das Modul `sys` eingebunden werden:

```
>>> import sys
```

17.3.1 Konstanten

Das Modul `sys` enthält eine ganze Reihe von Konstanten, die mitunter sehr nützliche Informationen bereitstellen. Die wichtigsten dieser Konstanten sollen im Folgenden erklärt werden.

sys.argv

Die Liste `sys.argv` enthält die Kommandozeilenparameter, mit denen das Python-Programm aufgerufen wurde. `sys.argv[0]` ist der Name des Programms selbst. Im interaktiven Modus hat `sys.argv` die Länge 0. Bei dem Programmaufruf

```
programm.py -bla 0 -blubb abc
```

würde `sys.argv` folgende Liste referenzieren:

```
['programm.py', '-bla', '0', '-blubb', 'abc']
```

Verwenden Sie das Modul `optparse`, wenn Sie Kommandozeilenparameter komfortabel verwalten möchten.

sys.byteorder

Diese Konstante spezifiziert die Byte-Order des aktuellen Systems. Der Wert ist entweder `"big"` für ein *Big-Endian*-System, bei dem das signifikanteste Byte an erster Stelle gespeichert wird, oder `"little"` für ein *Little-Endian*-System, bei dem das am wenigsten signifikante Byte zuerst gespeichert wird.

sys.executable

Dies ist ein String, der den vollen Pfad zur ausführbaren Datei des Python-Interpreters angibt.

```
>>> sys.executable
'C:\\Python25\\pythonw.exe'
```

sys.hexversion

Diese Konstante enthält die Versionsnummer des Python-Interpreters als ganze Zahl. Wenn sie durch Aufruf der Built-in Function `hex` als Hexadezimalzahl geschrieben wird, wird der Aufbau der Zahl deutlich:

```
>>> hex(sys.hexversion)
'0x30000f0'
```

In diesem Fall wurde Python 3.0.0 verwendet. Es ist garantiert, dass `hexversion` mit jeder Python-Version immer größer wird, dass Sie also mit den Operatoren `<` und `>` testen können, ob die verwendete Version des Interpreters aktueller ist als eine bestimmte, die für die Ausführung des Programms mindestens vorausgesetzt wird.

sys.maxunicode

Diese Konstante enthält den größtmöglichen Zeichencode, den ein Unicode-Zeichen haben kann. Dieser Wert hängt davon ab, welche Unicode-Darstellung intern verwendet wird.

sys.modules

Das Dictionary `sys.modules` enthält die Namen aller momentan eingebundenen Module als Schlüssel und die dazugehörigen Namespaces als jeweiligen Wert.

sys.path

Die Liste `sys.path` enthält eine Reihe von Pfadangaben, die beim Einbinden eines Moduls der Reihe nach vom Interpreter durchsucht werden. Das zuerst gefundene Modul mit dem in einer `import`-Anweisung angegebenen Namen wird eingebunden.

Es steht dem Programmierer frei, die Liste so zu modifizieren, dass das Einbinden eines Moduls nach seinen Wünschen erfolgt.

```
>>> import sys
>>> sys.path
['', 'C:Python30\\Lib\\idlelib', 'C:\\WINDOWS\\system32\\python25.zip',
```



```
'C:\\Python30\\DLLs',
'C:\\Python30\\lib',
'C:\\Python30\\lib\\plat-win',
'C:\\Python30\\lib\\lib-tk',
'C:\\Python30',
'C:\\Python30\\lib\\site-packages']
```

sys.platform

Dieser String enthält eine Kennung des zugrundeliegenden Betriebssystems. Der Wert ist beispielsweise "win32" für Windows oder "linux2" für Linux.

Diese Kennung können Sie beispielsweise dazu verwenden, plattformspezifische Pfade an `sys.path` anzuhängen.

sys.stdin, sys.stdout, sys.stderr

Dies sind die Dateiobjekte, die für Ein- und Ausgaben des Interpreters verwendet werden. Dabei steht `sys.stdin` für *Standard Input* und entspricht dem Dateiobjekt, aus dem die Benutzereingaben beim Aufruf von `input` oder `raw_input` gelesen werden. In das Dateiobjekt `sys.stdout` (*Standard Output*) werden alle Ausgaben des Python-Programms geschrieben, während Ausgaben des Interpreters, beispielsweise Tracebacks, in `sys.stderr` (*Standard Error*) geschrieben werden.

Das Überschreiben dieser vorbelegten Dateiobjekte mit eigenen Dateiobjekten erlaubt es, Ein- und Ausgaben auf andere Streams, beispielsweise in eine Datei, umzulenken. Beachten Sie dabei, dass `sys.stdin` stets ein vollwertiges Dateiobjekt sein muss, während für `sys.stdout` und `sys.stderr` eine Instanz reicht, die eine Methode `write` implementiert.

Die ursprünglichen Streams von `sys.stdin`, `sys.stdout` und `sys.stderr` werden in `sys.__stdin__`, `sys.__stdout__` und `sys.__stderr__` gespeichert, so dass sie jederzeit wiederhergestellt werden können.

sys.version

Ein String, der die Versionsnummer des Python-Interpreters und einige weitere Informationen, wie beispielsweise das Datum seiner Kompilierung und den verwendeten Compiler, enthält.

```
>>> sys.version
'3.0 (r30:67503, Dec 7 2008, 04:54:04) \n[GCC 4.3.2]'
```

Beachten Sie, dass es bei `sys.version` im Gegensatz zu `sys.hexversion` nicht garantiert ist, dass die Versionsnummern mit den Operatoren `>` und `<` sinnvoll miteinander verglichen werden können.

sys.version_info

Ein Tupel, das die einzelnen Komponenten der Versionsnummer des Interpreters enthält.

```
>>> sys.version_info
(2, 5, 1, 'final', 0)
```

17.3.2 Exceptions

Das Modul `sys` enthält einige Funktionen, die speziell dazu gedacht sind, Zugriff auf geworfene Exceptions zu erhalten oder anderweitig mit Exceptions zu arbeiten.

Näheres dazu, wie Sie das in diesem Kapitel angesprochene *Traceback-Objekt* verwenden können, erfahren Sie in Abschnitt 21.6.

sys.exc_info()

Diese Funktion ermöglicht es, Zugriff auf eine momentan abgefangene Exception zu erlangen. Momentan abgefangen bedeutet, dass sich der Kontrollfluss innerhalb eines `except`-Zweiges einer `try/except`-Anweisung befinden muss, damit diese Funktion einen sinnvollen Wert zurückgibt.

Die Funktion `exc_info` gibt ein Tupel zurück, das drei Werte enthält: den Exception-Typ, die geworfene Instanz des Exception-Typs und das entsprechende Traceback-Objekt.

Beachten Sie, dass die Informationen über die aktuell abgefangene Exception nicht erhalten bleiben, sondern nur innerhalb des `except`-Zweiges verwendbar sind. Falls Sie Informationen über die zuletzt geworfene Exception außerhalb eines `except`-Zweiges benötigen, sollten Sie entweder `last_type`, `last_value` oder `last_traceback` verwenden.

sys.last_type, sys.last_value, sys.last_traceback

Diese Funktionen erlauben es, Zugriff auf die zuletzt geworfene Exception zu erlangen. Die drei Informationen entsprechen denen, die von `sys.exc_info` zurückgegeben werden.

Beachten Sie, dass diese Konstanten auch außerhalb eines `except`-Zweiges Gültigkeit haben, da sie stets Informationen über die zuletzt geworfene Exception enthalten.

sys.tracebacklimit

Diese ganze Zahl kennzeichnet die maximale Tiefe, bis zu der ein Traceback Informationen über die Funktionshierarchie liefern soll. Initial ist dieser Wert auf

1000 gesetzt. Ein Wert von 0 veranlasst, dass ein Traceback nur aus dem Exception-Typ und der Fehlermeldung besteht.

17.3.3 Hooks

Das Modul `sys` erlaubt den Zugriff auf sogenannte *Hooks* (dt. »Haken«). Das sind Funktionen, die bei gewissen Aktionen des Python-Interpreters aufgerufen werden. Durch Überschreiben dieser Funktionen kann sich der Programmierer in den Interpreter »einhaken« und so die Funktionsweise des Interpreters verändern.

`sys.displayhook(value)`

Diese Funktion wird immer dann aufgerufen, wenn das Ergebnis eines Ausdrucks im interaktiven Modus ausgegeben werden soll, also beispielsweise in der folgenden Situation:

```
>>> 42
42
```

Durch Überschreiben von `sys.displayhook` mit einer eigenen Funktion lässt sich dieses Verhalten ändern. Im folgenden Beispiel möchten wir erreichen, dass bei einem eingegebenen Ausdruck nicht das Ergebnis selbst, sondern die Identität des Ausdrucks ausgegeben wird:

```
>>> def f(value):
...     print(id(value))
...
>>> sys.displayhook = f
>>> 42
134536524
>>> 97 + 32
134537456
>>> "Hallo Welt"
3083420560
```

Beachten Sie, dass `sys.displayhook` nicht aufgerufen wird, wenn eine Ausgabe mittels `print` erfolgt:²

```
>>> print("Hallo Welt")
Hallo Welt
```

2 Das wäre auch sehr ungünstig, da wir im Hook selbst ja eine `print`-Ausgabe tätigen. Riefe eine `print`-Ausgabe wieder den Hook auf, befänden wir uns in einer endlosen Rekursion.

Das ursprüngliche Funktionsobjekt von `sys.displayhook` können Sie über `sys.__displayhook__` erreichen und somit die ursprüngliche Funktionsweise wiederherstellen:

```
>>> sys.displayhook = sys.__displayhook__
```

sys.excepthook(type, value, traceback)

Diese Funktion wird immer dann aufgerufen, wenn eine nicht abgefangene Exception auftritt. Sie ist dafür verantwortlich, den Traceback auszugeben. Durch Überschreiben dieser Funktion mit einem eigenen Funktionsobjekt lässt sich zum Beispiel die Ausgabe eines Tracebacks verändern.

Die drei Parameter der Funktion entsprechen denen, die von `sys.exc_info` zurückgegeben werden, und enthalten Informationen über die Exception.

Im folgenden Beispiel möchten wir einen Hook einrichten, damit bei einer nicht abgefangenen Exception kein dröger Traceback mehr ausgegeben wird, sondern ein hämischer Kommentar:

```
>>> def f(type, value, traceback):
...     print("gnahahaha: '{0}'".format(value))
...
>>> sys.excepthook = f
>>> abc
gnahahaha: 'name 'abc' is not defined'
```

Das ursprüngliche Funktionsobjekt von `sys.excepthook` können Sie über `sys.__excepthook__` erreichen und somit die ursprüngliche Funktionsweise wiederherstellen.

17.3.4 Sonstige Funktionen

Neben den bereits besprochenen Konstanten sowie den exception- bzw. hook-bezogenen Funktionen stellt das Modul `sys` einige weitere Funktionen bereit, um an Informationen über den Interpreter oder das Betriebssystem zu gelangen oder mit dem System zu interagieren.

sys.exit([arg])

Wirft eine `SystemExit`-Exception. Diese hat, sofern sie nicht abgefangen wird, zur Folge, dass das Programm ohne Traceback-Ausgabe beendet wird.

Als optionalen Parameter `arg` können Sie, wenn es sich um eine ganze Zahl handelt, einen *Exit Code* ans Betriebssystem übergeben. Ein Exit Code von 0 steht im Allgemeinen für ein erfolgreiches Beenden des Programms, und ein Exit Code ungleich 0 repräsentiert einen Programmabbruch aufgrund eines Fehlers.

Wenn Sie eine andere Instanz für `arg` übergeben haben, beispielsweise einen String, wird diese nach `sys.stderr` ausgegeben, bevor das Programm mit dem Exit Code 0 beendet wird.

sys.getrefcount(object)

Gibt den aktuellen *Reference Count* für die übergebene Instanz `object` zurück. Der Reference Count ist eine ganze Zahl und entspricht der Anzahl von Referenzen, die auf eine Instanz bestehen. Wenn eine Instanz einen Reference Count von 0 hat, wird sie vom Garbage Collector entsorgt.

Beachten Sie, dass es dem Interpreter bei Instanzen unveränderlicher Datentypen freisteht, eine neue Instanz zu erzeugen oder eine bereits bestehende neu zu referenzieren. Aus diesem Grund kann es vorkommen, dass zum Beispiel Instanzen ganzer Zahlen einen hohen Reference Count haben.

sys.getrecursionlimit(), setrecursionlimit(limit)

Mit diesen Funktionen wird die maximale Rekursionstiefe ausgelesen oder verändert. Die maximale Rekursionstiefe ist mit 1000 vorbelegt und bricht endlos rekursive Funktionsaufrufe ab, bevor diese zu einem Speicherüberlauf führen können.

sys.getwindowsversion()

Erlaubt es, die Details über die Version des aktuell verwendeten Windows-Betriebssystems auszulesen. Die Funktion gibt ein Tupel zurück, dessen erste drei Elemente ganze Zahlen sind und die Versionsnummer beschreiben. Das vierte Element ist ebenfalls eine ganze Zahl und steht für die verwendete Plattform. Folgende Werte sind hier gültig:

Plattform	Bedeutung
0	Windows 3.1 (32-Bit)
1	Windows 95/98/ME
2	Windows NT/2000/XP/2003/Vista
3	Windows CE

Tabelle 17.3 Windows-Plattformen

Das letzte Element des Tupels ist ein String, der weiterführende Informationen enthält.

```
>>> sys.getwindowsversion()
(5, 1, 2600, 2, 'Service Pack 2')
```

Unter anderen Betriebssystemen als Microsoft Windows ist die Funktion `sys.getwindowsversion` nicht verfügbar.

17.4 Informationen über das System – platform

Das Modul `platform` der Standardbibliothek stellt Informationen über das Betriebssystem bzw. die zugrundeliegende Hardware bereit. Diese Informationen sind teilweise deckungsgleich mit denen, auf die Sie über das Modul `sys` zugreifen. Aus diesem Grund werden wir hier nur die wichtigsten Funktionen erläutern.

17.4.1 Funktionen

platform.machine()

Gibt die Prozessorarchitektur des PCs als String zurück. Bei aktuellen Prozessoren ist dies `i686`.

platform.node()

Gibt den Netzwerknamen des PCs als String zurück.

platform.processor()

Gibt einen String zurück, der den Typ und den Hersteller des Prozessors enthält.

platform.system()

Gibt einen String zurück, der den Namen des Betriebssystems, beispielsweise also »Linux« oder »Windows«, enthält.

17.5 Kommandozeilenparameter – optparse

Im vorletzten Abschnitt haben wir gesagt, dass Sie über `sys.argv` auf die Kommandozeilenparameter zugreifen können, die beim Aufruf des Programms übergeben wurden. Das ist richtig und funktioniert. Das Modul `optparse` erlaubt Ihnen jedoch einen wesentlich komfortableren Umgang mit Kommandozeilenparametern.

Doch zunächst möchten wir uns allgemein mit der Thematik der Kommandozeilenparameter befassen. Bislang wurden hier ausschließlich Konsolenprogramme behandelt, das heißt Programme, die eine rein textbasierte Schnittstelle zum Be-

nutzer haben. Solche Programme werden üblicherweise aus einer *Konsole*, auch *Shell* genannt, gestartet. Eine Konsole ist beispielsweise die Eingabeaufforderung unter Windows.

Unter Windows wird ein Python-Programm aus der Eingabeaufforderung heraus gestartet, indem in das Programmverzeichnis gewechselt und dann der Name der Programmdatei eingegeben wird. Hinter dem Namen können jetzt zum einen sogenannte *Optionen* und zum anderen sogenannte *Argumente* übergeben werden:

- Ein Argument wird einfach hinter den Namen der Programmdatei geschrieben. Um einen Vergleich zu Funktionsparametern zu ziehen, könnte man von Positional Arguments sprechen. Das bedeutet vor allem, dass die Argumente anhand ihrer Reihenfolge zugeordnet werden. Ein Programmaufruf mit drei Argumenten könnte beispielsweise folgendermaßen aussehen:

```
programm.py karl 1337 heinz
```

- Neben den Argumenten können Sie Optionen übergeben. Optionen sind, wie der Name sagt, optional und deshalb Keyword Arguments. Das bedeutet, dass jede Option einen Namen hat und über diesen angesprochen wird. Beim Programmaufruf müssen Optionen vor den Argumenten geschrieben und jeweils durch einen Bindestrich eingeleitet werden. Dann folgen der Optionsname, ein Leerzeichen und der gewünschte Wert. Ein Programmaufruf mit Optionen und Argumenten könnte also folgendermaßen aussehen:

```
programm.py -a karl -b heinz -c 1337 hallo welt
```

In diesem Fall existieren drei Optionen namens *a*, *b* und *c* mit den Werten "karl", "heinz" und 1337. Zudem wurden zwei Argumente angegeben, die Strings "hallo" und "welt".

Neben diesen parameterbehafteten Optionen gibt es parameterlose Optionen, die mit einem Flag vergleichbar sind. Das bedeutet, dass sie entweder vorhanden (aktiviert) oder nicht vorhanden (deaktiviert) sind:

```
programm.py -a -b 1 hallo welt
```

In diesem Fall handelt es sich bei *a* um eine parameterlose Option.

Im Weiteren soll die Verwendung des Moduls `optparse` anhand zweier Beispiele besprochen werden.

17.5.1 Taschenrechner – ein einfaches Beispiel

Das erste Beispiel soll ein einfacher Taschenrechner sein, bei dem sowohl die Rechenoperation als auch die Operanden über Kommandozeilenparameter angegeben werden. Das Programm soll folgendermaßen aufgerufen werden können:

```
calc.py -o plus 7 5
calc.py -o minus 13 29
calc.py -o mal 4 11
calc.py -o geteilt 3 2
```

Das bedeutet, dass über die Option `-o` eine Rechenoperation festgelegt werden kann, die auf die beiden folgenden Argumente angewendet wird. Wenn die Option nicht angegeben wurde, sollen die Argumente addiert werden.

Zu Beginn des Programms muss die Klasse `OptionParser` des Moduls `optparse` eingebunden und instantiiert werden:

```
from optparse import OptionParser
parser = OptionParser()
```

Jetzt können durch die Methode `add_option` der `OptionParser`-Instanz erlaubte Optionen hinzugefügt werden. In unserem Fall ist es nur eine:

```
parser.add_option("-o", "--operation", dest="operation")
```

Der erste Parameter der Methode gibt den Kurznamen der Option an. Jede Option ist auch mit einer ausgeschriebenen Version des Namens verwendbar, sofern diese Alternative durch Angabe des zweiten Parameters gegeben ist. In diesem Fall wären die Optionen `-o` und `--operation` gleichbedeutend. Der letzte Parameter, ein Keyword Argument wohlgeemerkt, gibt an, unter welchem Namen der Wert der Option später im Programm verfügbar gemacht werden soll.

Nachdem alle Optionen hinzugefügt worden sind, wird die Methode `parse_args` aufgerufen, die die Kommandozeilenparameter ausliest und in der gewünschten Form aufbereitet.

```
(optionen, args) = parser.parse_args()
```

Die Methode gibt ein Tupel mit zwei Werten zurück: zum einen eine Instanz, die alle übergebenen Optionen enthält (`optionen`), und zum anderen eine Liste mit allen weiteren Argumenten (`args`).

Um korrekt arbeiten zu können, müssen dem Taschenrechner-Programm exakt zwei Argumente übergeben worden sein, was wir an dieser Stelle im Quelltext überprüfen. Wenn die Anzahl der Argumente ungleich zwei ist, kann keine Berechnung durchgeführt werden, und das Programm beendet sich:

```
if len(args) != 2:
    parser.error("Es werden exakt zwei Argumente erwartet")
```

Für Fehler, die aufgrund falscher oder fehlender Kommandozeilenparameter auftreten, eignet sich die Methode `error` der `OptionParser`-Instanz, die eine entsprechende Fehlermeldung ausgibt und das Programm beendet.

Als Nächstes legen wir ein Dictionary an, das alle möglichen Rechenoperationen als Schlüssel und die dazugehörigen Berechnungsfunktionen als jeweiligen Wert enthält. Die Schlüssel sind dieselben, die über die Option `-o` angegeben werden können, so dass wir anhand des bei der Option übergebenen Strings direkt auf die zu verwendende Berechnungsfunktion schließen können:

```
calc = {
    "plus" : lambda a, b: a + b,
    "minus" : lambda a, b: a - b,
    "mal" : lambda a, b: a * b,
    "geteilt" : lambda a, b: a / b,
    None : lambda a, b: a + b
}
```

Prinzipiell muss jetzt nur noch der Wert ausgelesen werden, der mit der Option `-o` übergeben wurde. Der Zugriff auf eine Option ist anhand der von `parse_args` zurückgegebenen Instanz `optionen` relativ einfach, da jede Option unter ihrem gewählten Namen als Attribut dieser Instanz verfügbar ist. Der von uns gewählte Name für die Option `-o` war `operation`.

```
op = optionen.operation
if op in calc:
    print("Ergebnis:", calc[op](float(args[0]), float(args[1])))
else:
    parser.error("{0} ist keine Operation".format(op))
```

Beachten Sie, dass im Falle einer nicht angegebenen Option das entsprechende Attribut nicht etwa nicht vorhanden ist, sondern lediglich `None` referenziert. Da `None` im Dictionary `calc` als Schlüssel geführt wird und auf die Berechnungsfunktion der Addition verweist, werden die beiden Argumente in einem solchen Fall schlicht zusammengezählt.

17.5.2 Weitere Verwendungsmöglichkeiten

In diesem Abschnitt soll das Beispielprogramm des letzten Abschnitts dahingehend erweitert werden, dass weitere Verwendungsmöglichkeiten des Moduls `optparse` hervorgehoben werden. Hier sehen Sie zunächst den Quellcode des veränderten Beispielprogramms:

```
from optparse import OptionParser

parser = OptionParser("calc2.py [Optionen] Operand1 Operand2")
parser.add_option("-o", "--operation", dest="operation",
                  help="Rechenoperation")
```

```

parser.add_option("-v", "--verbose", action="store_true",
                  dest="verbose", default=False,
                  help="Schwafelmodus")

(optionen, args) = parser.parse_args()
if len(args) != 2:
    parser.error("Es werden exakt zwei Argumente erwartet")

calc = {
    "plus" : lambda a, b: a + b,
    "minus" : lambda a, b: a - b,
    "mal" : lambda a, b: a * b,
    "geteilt" : lambda a, b: a / b,
    None : lambda a, b: a + b
}

if optionen.verbose:
    print("Das Ergebnis wird berechnet")
op = optionen.operation
if op in calc:
    print("Ergebnis:", calc[op](float(args[0]), float(args[1])))
else:
    parser.error("{0} ist keine Operation".format(op))

```

Zunächst einmal werden Sie feststellen, dass bei der Instantiierung von Option-Parser ein String übergeben wurde. Zusätzlich haben auch die Aufrufe der Methode `add_option` ein weiteres Keyword Argument namens `help` spendiert bekommen. Diese Angaben sind zwar nicht notwendig, sollten jedoch erfolgen, da die OptionParser-Instanz aus den dort übergebenen Strings automatisch eine Hilfeseite generiert, wenn das Programm mit den Optionen `-h` oder `--help` gestartet wird. In dieser Hilfeseite wird die Verwendung des Programms kurz umrissen. Dazu gehört eine Auflistung aller möglichen Optionen, jeweils mit einem kurzen erläuternden Satz. Für das obige Beispiel sieht der Hilfetext folgendermaßen aus:

```
Usage: calc2.py [Optionen] Operand1 Operand2
```

Options:

```

-h, --help          show this help message and exit
-o OPERATION, --operation=OPERATION
                    Rechenoperation
-v, --verbose       Schwafelmodus

```

Zusätzlich zu der bereits im ursprünglichen Beispielprogramm vorhandenen Option `-o` wurde eine weitere Option namens `-v` bzw. `--verbose` angelegt. Viele bekannte Programme verwenden die Option `-v` als Schalter, um das Programm in eine Art geschwätzigen Zustand zu versetzen. Das bedeutet, dass auch nicht es-

senzielle Statusmeldungen auf dem Bildschirm ausgegeben werden. Diese Funktionalität soll auch unser Beispielprogramm bekommen. Der geschwätzige Modus soll aktiviert werden, wenn `-v` oder `--verbose` angegeben wurden, und sonst deaktiviert bleiben. Programmintern sollte die Option daher als boolescher Wert ankommen.

Dazu werden der Methode `add_option` zwei weitere Keyword Arguments übergeben. Zum einen wird der Parameter `action` auf `"store_true"` gesetzt, was bedeutet, dass das dazugehörige Attribut `verbose` auf `True` gesetzt wird, wenn die Option `-v` vorhanden ist. Analog dazu wäre auch `"store_false"` für den umgekehrten Fall möglich gewesen.

Der zweite neue Parameter ist die Angabe eines Default-Wertes. Das ist der Wert, den das dazugehörige Attribut `verbose` annimmt, wenn die Option `-v` nicht vorhanden ist.

Der resultierende boolesche Wert `optionen.verbose` wird im Programm abgefragt, und dann wird eventuell eine zugegebenermaßen sinnlose Statusmeldung ausgegeben. Die Ausgabe des Programms sieht bei Angabe der Option `-v` folgendermaßen aus:

```
Das Ergebnis wird berechnet
Ergebnis: 19.0
```

Damit wäre der grundlegende Funktionsumfang von `optparse` erläutert.

17.6 Kopieren von Instanzen – copy

Wie Sie bereits wissen, wird in Python bei einer Zuweisung nur eine neue Referenz auf ein und dieselbe Instanz erzeugt, anstatt eine Kopie der Instanz zu erzeugen. Im folgenden Beispiel verweisen `s` und `t` auf dieselbe Liste, wie der Vergleich mit `is` offenbart:

```
>>> s = [1, 2, 3]
>>> t = s
>>> t is s
True
```

Dieses Vorgehen ist nicht immer erwünscht, weil Änderungen an der von `s` referenzierten Liste über Seiteneffekte auch `t` betreffen und umgekehrt.

Wenn beispielsweise eine Methode einer Klasse eine Liste zurückgibt, die auch innerhalb der Klasse verwendet wird, kann die Liste auch über die zurückgegebene Referenz verändert werden, womit das Kapselungsprinzip verletzt wäre:

```

class MeineKlasse:
    def __init__(self):
        self.__Liste = [1, 2, 3]

    def getListe(self):
        return self.__Liste

    def zeigeListe(self):
        print(self.__Liste)

```

Wenn wir uns nun mit der `getListe`-Methode eine Referenz auf die Liste zurückgeben lassen, können wir über einen Seiteneffekt das private Attribut `__Liste` der Instanz verändern:

```

>>> instanz = MeineKlasse()
>>> liste = instanz.getListe()
>>> liste.append(1337)
>>> instanz.zeigeListe()
[1, 2, 3, 1337]

```

Um dies zu verhindern, sollte die Methode `getListe` anstelle der Liste selbst eine Kopie derselben zurückgeben.

An dieser Stelle kommt das Modul `copy` ins Spiel, das dazu gedacht ist, echte Kopien einer Instanz zu erzeugen. Für diesen Zweck bietet `copy` zwei Funktionen an: `copy.copy` und `copy.deepcopy`. Beide Methoden erwarten als Parameter die zu kopierende Instanz und geben eine Referenz auf eine Kopie von ihr zurück:³

```

>>> import copy
>>> s = [1, 2, 3]
>>> t = copy.copy(s)
>>> t
[1, 2, 3]
>>> t is s
False

```

Das Beispiel zeigt, dass `t` zwar die gleichen Elemente wie `s` enthält, aber trotzdem nicht auf dieselbe Instanz wie `s` referenziert, so dass der Vergleich mit `is` negativ ausfällt.

Der Unterschied zwischen `copy.copy` und `copy.deepcopy` besteht darin, wie mit Referenzen umgegangen wird, die die zu kopierenden Instanzen enthalten. Die Funktion `copy.copy` erzeugt zwar eine neue Liste, aber die Referenzen innerhalb

3 Natürlich kann eine Liste auch per Slicing kopiert werden. Das Modul `copy` erlaubt aber das Kopieren beliebiger Instanzen.

der Liste verweisen trotzdem auf dieselben Elemente. Mit `copy.deepcopy` hingegen wird die Instanz selbst kopiert und anschließend rekursiv auch alle von ihr referenzierten Instanzen.

Wir veranschaulichen diesen Unterschied anhand einer Liste, die eine weitere Liste enthält:

```
>>> liste = [1, [2, 3]]
>>> liste2 = copy.copy(liste)
>>> liste2.append(4)
>>> liste2
[1, [2, 3], 4]
>>> liste
[1, [2, 3]]
```

Wie erwartet verändert sich beim Anhängen des neuen Elements 4 an `liste2` nicht die von `liste` referenzierte Instanz. Wenn wir aber die innere Liste `[2, 3]` verändern, betrifft dies sowohl `liste` als auch `liste2`:

```
>>> liste2[1].append(1337)
>>> liste2
[1, [2, 3, 1337], 4]
>>> liste
[1, [2, 3, 1337]]
```

Der `is`-Operator zeigt uns den Grund für dieses Verhalten: Bei `liste[1]` und `liste2[1]` handelt es sich um dieselbe Instanz:

```
>>> liste[1] is liste2[1]
True
```

Arbeiten wir stattdessen mit `copy.deepcopy`, wird die Liste inklusive aller enthaltenen Elemente kopiert:

```
>>> liste = [1, [2, 3]]
>>> liste2 = copy.deepcopy(liste)
>>> liste2[1].append(4)
>>> liste2
[1, [2, 3, 4]]
>>> liste
[1, [2, 3]]
>>> liste[1] is liste2[1]
False
```

Sowohl die Manipulation von `liste2[1]` als auch der `is`-Operator zeigen, dass es sich bei `liste2[1]` und `liste[1]` um verschiedene Instanzen handelt.

Es gibt allerdings auch Datentypen, die sowohl von `copy.copy` als auch von `copy.deepcopy` nicht wirklich kopiert, sondern nur ein weiteres Mal referenziert werden. Dazu zählen unter anderem Modul-Objekte, Methoden, `file`-Objekte, `socket`-Instanzen und `traceback`-Instanzen.

Hinweis

Beim Kopieren einer Instanz mithilfe des `copy`-Moduls wird das Objekt ein weiteres Mal im Speicher erzeugt. Dies kostet erheblich mehr Speicherplatz und Rechenzeit als eine einfache Zuweisung. Deshalb sollten Sie `copy` wirklich nur dann benutzen, wenn Sie tatsächlich eine echte Kopie brauchen.

17.7 Zugriff auf das Dateisystem – shutil

Das Modul `shutil` ist als Ergänzung zu `os` und `os.path` anzusehen und definiert abstrakte Funktionen, die insbesondere das Kopieren und Entfernen von Dateien betreffen, ohne dass man die dazu erforderlichen plattformabhängigen Programme wie beispielsweise `copy` unter Windows oder `cp` auf Unix-Maschinen kennen muss.

Folgende Funktionen werden von `shutil` implementiert, wobei die Parameter *src* und *dst* jeweils Strings sind, die den Pfad der Quell- bzw. der Zieldatei enthalten:

shutil.copyfile(src, dst)

Kopiert die Datei unter *src* nach *dst*. Wenn die Datei unter *dst* bereits existiert, wird sie überschrieben.

Dabei muss der Pfad *dst* schreibbar sein. Ansonsten wird ein `IOError` geworfen.

shutil.copyfileobj(fsrc, fdst[, length])

Kopiert den Inhalt des zum Lesen geöffneten Dateiobjekts *fsrc* in das zum Schreiben geöffnete *fdst*-Objekt.

Mit dem optionalen Parameter *length* können Sie dabei die zu verwendende Zwischenspeichergröße in Bytes angeben. Ist *length* positiv, wird die *fsrc* portionsweise ausgelesen und nach *fdst* geschrieben, während bei negativem *length* zuerst der gesamte Inhalt von *fsrc* in den Speicher gelesen und dann in einem Rutsch nach *fdst* geschrieben wird. Standardmäßig wird ein positiver Wert für *length* verwendet, den das System wählt.

shutil.copymode(src, dst)

Kopiert die Zugriffsrechte vom Pfad *src* auf den Pfad *dst*. Dabei bleiben der Inhalt von *dst* sowie der Besitzer und die Gruppe unangetastet.

Beide Pfade müssen bereits im Dateisystem existieren.

shutil.copystat(src, dst)

Wie `shutil.copymode`, aber es werden zusätzlich die Zeiten für den letzten Zugriff in die letzte Modifikation kopiert.

shutil.copy(src, dst)

Kopiert die Datei unter dem Pfad *src* nach *dst*. Der Parameter *dst* kann dabei einen Pfad zu einer Datei enthalten, die dann erzeugt oder überschrieben wird. Verweist *dst* auf einen Ordner, wird eine neue Datei mit dem Dateinamen von *src* im Ordner *dst* erzeugt oder gegebenenfalls überschrieben.

Die Zugriffsrechte werden dabei mitkopiert.

shutil.copy2(src, dst)

Genau wie `shutil.copy`, aber es werden zusätzlich die Zeiten des letzten Zugriffs und der letzten Änderung kopiert.

shutil.copytree(src, dst[, symlinks])

Kopiert die gesamte Verzeichnisstruktur unter *src* nach *dst*. Der Pfad *dst* darf dabei nicht auf einen bereits existierenden Ordner verweisen, und es werden alle fehlenden Verzeichnisse des Pfads *dst* erzeugt.

Die Rechte der erzeugten Ordner und Dateien werden mittels `shutil.copystat` gesetzt, und Dateien werden mit `shutil.copy2` kopiert.

Der optionale Parameter *symlinks* gibt an, wie mit symbolischen Links verfahren werden soll. Hat *symlinks* den Wert `False` oder wird *symlinks* nicht angegeben, werden die verlinkten Dateien oder Ordner selbst in die kopierte Verzeichnisstruktur eingefügt. Bei einem *symlinks*-Wert von `True` werden nur die Links kopiert.

shutil.rmtree(src[, ignore_errors[, onerror]])

Löscht die gesamte Verzeichnisstruktur unter *src*. Für *ignore_errors* kann ein Wahrheitswert übergeben werden, der bestimmt, ob beim Löschen auftretende Fehler ignoriert oder von der Funktion, die für *onerror* übergeben wurde, behandelt werden sollen. Wird *ignore_errors* nicht angegeben, ruft jeder auftretende Fehler eine `Exception` hervor.

Wenn Sie *onerror* angeben, muss es eine Funktion sein, die drei Parameter erwartet:

- ▶ *function* – eine Referenz auf die Funktion, die den Fehler verursacht hat. Dies können `os.listdir`, `os.remove` oder `os.rmdir` sein.
- ▶ *path* – der Pfad, für den der Fehler auftrat
- ▶ *excinfo* – der Rückgabewert von `sys.exc_info` im Kontext des Fehlers

Achtung

Exceptions, die von der Funktion *onerror* geworfen werden, werden nicht abgefangen.

shutil.move(src, dst)

Verschiebt rekursiv die Datei oder den Ordner von *src* nach *dst*

17.8 Das Programmende – atexit

Mit dem Modul *atexit* lassen sich Funktionen registrieren, die nach Programmende aufgerufen werden sollen. Dies kann nützlich sein, um Daten zu sichern, Netzwerkverbindungen zu trennen oder sonstige Aufräumarbeiten durchzuführen.

Zu diesem Zweck implementiert *atexit* eine Funktion namens *register*, die als Parameter eine Referenz auf die Funktion erwartet, die am Programmende aufgerufen werden soll.

Im folgenden Beispiel wird eine einfache Funktion registriert, die eine Nachricht auf dem Bildschirm ausgibt:

```
import atexit
print("Programm gestartet")

def amEnde():
    print("Programm beendet")

atexit.register(amEnde)
```

Ein Programmlauf erzeugt nachstehende Ausgabe:

```
Programm gestartet
Programm beendet
```

Als zusätzliche Argumente können Sie der *register*-Funktion beliebig viele Parameter übergeben, die beim Aufruf der registrierten Funktion an diese weiter-

gereicht werden. Sie können positionsbezogene Parameter und Schlüsselwortparameter mischen.

Das folgende Beispiel lässt den Benutzer so lange neue Zeilen eintippen, bis er den String "exit" eingibt. Alle Eingaben werden in einer Liste verwaltet, die am Ende des Programms mit einer durch `atexit.register` registrierten Funktion in einer Datei gesichert werden:

```
import atexit
eingaben = []

def sichereEingaben(liste):
    open("eingaben.txt", "w").writelines("\n".join(liste))

atexit.register(sichereEingaben, eingaben)

while True:
    zeile = raw_input()
    if zeile == "exit":
        break
    eingaben += [zeile]
```

Es ist auch möglich, mehrere Funktionen per `atexit.register` zu registrieren, indem `atexit.register` für jede dieser Funktionen aufgerufen wird. Diese werden dann am Programmende nacheinander aufgerufen.

Achtung

Es kann vorkommen, dass die von `atexit` registrierten Funktionen nicht aufgerufen werden: zum einen, wenn das Programm nicht normal, sondern durch eine nicht behandelte Ausnahme abgestürzt ist, oder zum anderen, wenn es durch ein Systemsignal direkt beendet wurde.

Zum anderen kann es Probleme geben, wenn das Programm selbst oder ein Modul die Funktion `sys.exitfunc` überschreibt. Wenn Sie selbst Module entwickeln, sollten Sie immer `atexit.register` anstelle von `sys.exitfunc` benutzen, um zu verhindern, dass Ihr Modul die Aufräumarbeiten des einbindenden Programms behindert.

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

18 Parallele Programmierung

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

18.1 Prozesse, Multitasking und Threads

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

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



Abbildung 18.1 Mehrere Prozesse laufen gleichzeitig ab.

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

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

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

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

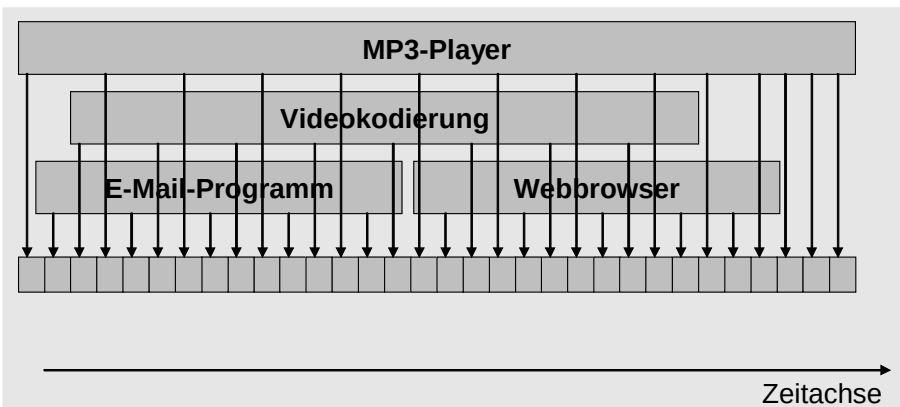


Abbildung 18.2 Die Prozesse wechseln sich ab und laufen nicht gleichzeitig.

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

Es ist zwar möglich, die Beschränkung auf nur eine Operation zur selben Zeit dadurch zu umgehen, dass weitere Prozesse erzeugt werden. Allerdings müssen dann Daten zwischen verschiedenen Prozessen ausgetauscht werden, wofür relativ viel Aufwand nötig ist, weil jeder Prozess seine eigenen Variablen hat, die von den anderen Prozessen abgeschirmt sind.¹

Eine befriedigende Lösung für das Problem liefern sogenannte *Threads*. Ein Thread (dt. »Faden«) ist ein Ausführungsstrang innerhalb eines Prozesses. Standardmäßig besitzt jeder Prozess genau einen Thread, der eben die Ausführung des Prozesses organisiert.

Nun kann ein Prozess aber auch mehrere Threads starten, die dann durch das Betriebssystem wie Prozesse scheinbar gleichzeitig ausgeführt werden. Der Vorteil von Threads gegenüber Prozessen besteht darin, dass sich die Threads eines Prozesses denselben Speicherbereich für globale Variablen teilen. Wenn also in einem Thread eine globale Variable verändert wird, ist der neue Wert auch sofort für alle anderen Threads des Prozesses sichtbar.² Demgegenüber hat jeder Thread seine eigenen lokalen Variablen. Außerdem ist die Verwaltung von Threads für das Betriebssystem weniger aufwendig als die Verwaltung von Prozessen. Deshalb werden Threads auch *Leichtgewichtprozesse* genannt.

Die Threads in einem Prozess können Sie sich vorstellen wie in Abbildung 18.3 illustriert.

1 Seit Python 3.0 gibt es das Modul `multiprocessing`, das die komfortable Nutzung mehrerer Prozesse und auch deren Synchronisation ermöglicht.

2 Um Fehler zu vermeiden, müssen solche Zugriffe in mehreren Threads speziell mit sogenannten *Critical Sections* abgesichert werden. Wir werden diese Thematik im Laufe dieses Abschnitts noch ausführlicher behandeln.

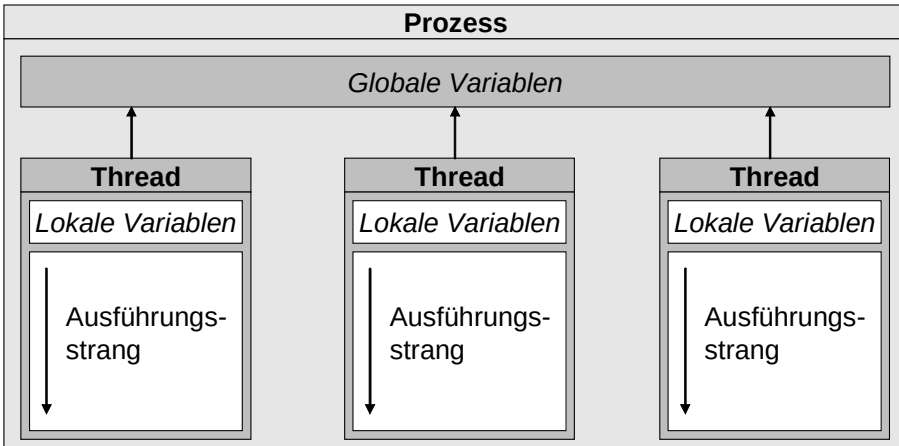


Abbildung 18.3 Ein Prozess mit drei Threads

In Python gibt es leider keine Möglichkeit, verschiedene Threads auf verschiedenen Prozessoren oder Prozessorkernen auszuführen. Dies hat zur Folge, dass selbst Python-Programme, die intensiv auf Threading setzen, nur einen einzigen Prozessor oder Prozessorkern nutzen können. Wenn Sie sehr rechenintensive Programme schreiben, die die gesamte Rechenpower des Computers ausschöpfen sollen, werfen Sie einen Blick auf das `multiprocessing`-Modul, mit dessen Hilfe mehrere Prozesse verwaltet werden können, die auch echt parallel auf verschiedenen Prozessoren laufen.

Nach dieser theoretischen Einführung wenden wir uns der Programmierung mit Threads in Python zu.

18.2 Die Thread-Unterstützung in Python

Python bietet zwei Module für den Umgang mit Threads an: `_thread` und `threading`.

Das erste Modul namens `_thread` ist die einfachere Variante und sieht jeden Thread als Funktion. Mit `threading` wird ein objektorientierter Ansatz implementiert, bei dem jeder Thread ein eigenes Objekt darstellt.

Wir werden uns mit beiden Ansätzen beschäftigen, wobei wir mit dem einfacheren Modul `_thread` beginnen werden.

18.3 Das Modul thread

Das Modul `_thread` kann einzelne Funktionen in einem separaten Thread ausführen. Dazu dient die Funktion `_thread.start_new_thread`, die mindestens zwei Parameter erwartet:

`thread.start_new_thread(function, args[, kwargs])`

Der Parameter *function* muss dabei eine Referenz auf die Funktion enthalten, die ausgeführt werden soll. Mit *args* muss eine `tuple`-Instanz übergeben werden, die die Parameter für *function* enthält.

Mit dem optionalen Parameter *kwargs* kann ein Dictionary übergeben werden, das zusätzliche Schlüsselwortparameter für die Funktion *function* bereitstellt.

Als Rückgabewert gibt `_thread.start_new_thread` eine Zahl zurück, die den erzeugten Thread eindeutig identifiziert.

Nachdem *function* verlassen wurde, wird der Thread automatisch gelöscht.

Parallele Berechnung von Pi

Als Beispiel für das Multithreading werden wir eine Funktion entwickeln, die die Kreiszahl π mithilfe des Wallis'schen Produkts berechnet, das der englische Mathematiker John Wallis (1616–1703) im Jahre 1655 entdeckte:

$$\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \cdot \dots = \frac{\pi}{2}$$

Im Zähler stehen dabei immer gerade Zahlen, die sich bei jedem zweiten Faktor um 2 erhöhen. Der Nenner enthält nur ungerade Zahlen, die sich mit Ausnahme des ersten Faktors ebenfalls alle zwei Faktoren um 2 erhöhen.

Die Funktion `naehere_pi_an`, die als Parameter die Anzahl der zu berücksichtigenden Faktoren erhält, kann damit folgendermaßen definiert werden:

```
def naehere_pi_an(n):
    pi_halbe = 1
    zaehler, nenner = 2.0, 1.0

    for i in range(n):
        pi_halbe *= zaehler / nenner
        if i % 2:
            zaehler += 2
        else:
            nenner += 2

    print("Annäherung mit {0} Faktoren: {1:.16f}".format(
        n, 2*pi_halbe))
```

Wenn für n der Wert 1000 übergeben wird, erzeugt die Funktion folgende Ausgabe, bei der nur die ersten beiden Nachkommastellen korrekt sind:

```
>>> naehere_pi_an(1000)
Annaeherung mit 1000 Faktoren: 3.1400238186005862
```

Wirklich brauchbare Näherungen werden erst für recht große n erzielt, was aber auch mit mehr Rechenzeit bezahlt werden muss. Beispielsweise benötigte ein Aufruf mit $n = 10000000$ auf unserem Testrechner ca. sieben Sekunden.

Im nächsten Programm werden wir mithilfe von `_thread.start_new_thread` mehrere Threads erzeugen, die die Funktion `naehere_pi_an` für verschiedene n aufrufen.

```
import _thread

_thread.start_new_thread(naehere_pi_an, (10000000,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (99999999,))
_thread.start_new_thread(naehere_pi_an, (123456789,))
_thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while True:
    pass
```

Die Endlosschleife am Ende des Programms ist notwendig, damit der Thread des Hauptprogramms auf die anderen Threads wartet und nicht sofort beendet wird. Alle Threads eines Programms werden nämlich sofort abgebrochen, wenn das Hauptprogramm sein Ende erreicht hat.

Eine Endlosschleife für diesen Zweck zu benutzen, ist natürlich sehr unschön, weil sie Rechenleistung sinnlos vergeudet und das Programm mit `[Strg]+[C]` beendet werden muss. Wir werden erst bei dem Modul `threading` bessere Methoden kennenlernen, um einen Thread auf das Ende eines anderen warten zu lassen.

Das Interessante an diesem Programm ist die Reihenfolge der Ausgabe, die nicht mit der Reihenfolge der Aufrufe übereinstimmt:

```
Annaeherung mit 1337 Faktoren: 3.1427668611489281
Annaeherung mit 10000 Faktoren: 3.1414355935898644
Annaeherung mit 100000 Faktoren: 3.1415769458226377
Annaeherung mit 1234569 Faktoren: 3.1415939259321926
Annaeherung mit 11111111 Faktoren: 3.1415927949601699
```

Je größer das übergebene n war, desto länger musste auf die Ausgabe der dazugehörigen Annäherung von π gewartet werden, ganz egal, wann die Funktion ge-

startet wurde. Offensichtlich liefen alle Berechnungen parallel ab, wie wir es erwartet hatten.

Im letzten Beispiel hatte jeder Thread seine eigenen Variablen und musste keine Daten mit anderen Threads austauschen. Im nächsten Abschnitt werden wir uns mit dem Datenaustausch zwischen Threads beschäftigen.

18.3.1 Datenaustausch zwischen Threads – locking

Threads haben gegenüber Prozessen den Vorteil, dass sie sich dieselben globalen Variablen teilen und deshalb sehr einfach Daten austauschen können. Trotzdem gibt es ein paar Stolperfallen, die Sie beim Zugriff auf dieselbe Variable durch mehrere Threads beachten müssen.³

Würde man beispielsweise unser vorhergehendes Beispiel um einen Zähler erweitern, der die Anzahl der zurzeit aktiven Threads enthält, damit das Programm nach dem Beenden aller Berechnungen von selbst terminiert, könnte man ganz naiv folgende Implementation vorschlagen:

```
import _thread

anzahl_threads = 0

def naehere_pi_an(n):
    global anzahl_threads
    anzahl_threads += 1

    # hier wurde der Berechnungscode zur Übersicht ausgelassen

    anzahl_threads -= 1

_thread.start_new_thread(naehere_pi_an, (10000000,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (99999999,))
_thread.start_new_thread(naehere_pi_an, (123456789,))
_thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while anzahl_threads > 0:
    pass
```

3 Die eigentliche Kunst bei der Programmierung mit Threads ist es, diese Stolperfallen zu umgehen. Es ist oft sehr schwierig, die Abläufe in parallelen Programmen zu überblicken, weswegen sich leicht Fehler einschleichen.

Dieses Programm hat zwei schwerwiegende Fehler: Erstens funktioniert es nicht immer, weil möglicherweise die `while`-Schleife erreicht ist, bevor überhaupt ein Thread gestartet werden konnte. In diesem Fall hat `anzahl_threads` den Wert 0, und damit wird die Schleife gar nicht durchlaufen, sondern das Programm beendet.

Aber selbst wenn dieses Problem bereits gelöst wäre, verhält sich das Programm unter Umständen fehlerhaft. Die Gefahr lauert in den beiden Zeilen, die den Wert der globalen Variable `anzahl_threads` verändern:

Es ist theoretisch möglich, dass das Zeitfenster eines Threads genau während der Veränderung von `anzahl_threads` endet, denn Zuweisungen bestehen intern aus mehreren Schritten. Zuerst muss der Wert von `anzahl_threads` gelesen werden, dann muss eine neue Instanz mit dem um eins vergrößerten bzw. verringerten Wert erzeugt werden, die im letzten Schritt mit der Referenz `anzahl_threads` verknüpft wird.

Wenn ein Thread A nun beim Erhöhen von `anzahl_threads` während der Erzeugung der neuen Instanz schlafen gelegt wird, könnte ein anderer Thread B aktiviert werden, der ebenfalls `anzahl_threads` erhöhen möchte. Weil aber Thread A seinen neuen Wert von `anzahl_threads` noch nicht berechnet und auch nicht mit der Referenz verknüpft hat, würde der neu aktivierte Thread B den alten Wert von `anzahl_threads` lesen und erhöhen. Wird dann später der Thread A wieder aktiv, erhöht er den schon vorher eingelesenen Wert um eins und weist ihn `anzahl_threads` zu. Das Ende vom Lied wäre ein um eins zu kleiner Wert von `anzahl_threads`, wodurch die Schleife im Hauptprogramm endlos laufen würde.

Die folgende Tabelle soll das beschriebene Szenario veranschaulichen:

Zeitfenster	Thread A	Thread B
1	Wert von <code>anzahl_threads</code> einlesen, beispielsweise 2.	<i>schläft</i>
----- Zeitfenster von A endet, und Thread B wird aktiviert. -----		
2	<i>schläft</i>	Wert von <code>anzahl_threads</code> einlesen, in diesem Fall 2. Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 3. Die neue Instanz an die Referenz <code>anzahl_threads</code> knüpfen. Damit weist <code>anzahl_threads</code> auf den Wert 3.

Tabelle 18.1 Problemszenario beim gleichzeitigen Zugriff auf eine globale Variable

Zeitfenster	Thread A	Thread B
----- Zeitfenster von B endet, und Thread A wird aktiviert. -----		
3	Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 3. Die neue Instanz an die Referenz <code>anzahl_threads</code> knüpfen. Damit verweist <code>anzahl_threads</code> auf den Wert 3.	<i>schläft</i>

Tabelle 18.1 Problemszenario beim gleichzeitigen Zugriff auf eine globale Variable (Forts.)

Im Beispiel wurde `anzahl_threads` also nur um 1 erhöht, obwohl zwei neue Threads gestartet wurden.

Um solche Probleme zu vermeiden, kann ein Programm Stellen markieren, die nicht parallel in mehreren Threads laufen dürfen. Man bezeichnet solche Stellen auch als *Critical Sections* (dt. »kritische Abschnitte«).

Critical Sections werden durch sogenannte *Lock-Objekte* (von engl. *to lock* = »sperren«) realisiert. Die parameterlose Funktion `_thread.allocate_lock` erzeugt ein neues Lock-Objekt:

```
lock_objekt = _thread.allocate_lock()
```

Lock-Objekte haben die beiden wichtigen Methoden `acquire` und `release`, die jeweils beim Betreten bzw. beim Verlassen einer Critical Section aufgerufen werden müssen. Wenn die `acquire`-Methode eines Lock-Objekts aufgerufen wurde, ist es *gesperrt*. Ruft ein Thread die `acquire`-Methode eines gesperrten Lock-Objekts auf, muss er so lange warten, bis das Lock-Objekt wieder mit `release` freigegeben worden ist. Diese Technik verhindert, dass eine Critical Section von mehreren Threads gleichzeitig ausgeführt werden kann.

Wir können unser Beispielprogramm folgendermaßen um Critical Sections erweitern, wobei wir außerdem einen Schalter namens `thread_gestartet` einfügen, damit das Hauptprogramm mindestens so lange wartet, bis die Threads gestartet worden sind. Der Zugriff auf die Variablen `anzahl_threads` und `thread_gestartet` wird durch das Lock-Objekt `lock` gesichert:

```
import _thread

anzahl_threads = 0
thread_gestartet = False
```

```

lock = _thread.allocate_lock()

def naehere_pi_an(n):
    global anzahl_threads, thread_gestartet

    lock.acquire()
    anzahl_threads += 1
    thread_gestartet = True
    lock.release()

    # hier wurde der Berechnungscode zur Übersicht ausgelassen

    lock.acquire()
    anzahl_threads -= 1
    lock.release()

_thread.start_new_thread(naehere_pi_an, (100000,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (1111111,))
_thread.start_new_thread(naehere_pi_an, (1234569,))
_thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while not thread_gestartet:
    pass

while anzahl_threads > 0:
    pass

```

Am Anfang des Programms wird der Schalter `_thread_gestartet` auf `False` gesetzt, und mit `_thread.allocate_lock()` wird ein neues Lock-Objekt erzeugt. Innerhalb von `naehere_pi_an` gibt es dann eine Critical Section, in der `anzahl_threads` an die Anzahl der laufenden Threads angepasst bzw. die Variable `thread_gestartet` auf `True` gesetzt wird.

Die erste `while`-Schleife des Hauptprogramms sorgt nun dafür, dass auf jeden Fall so lange gewartet wird, bis ein Thread gestartet worden ist und den Wert von `thread_gestartet` auf `True` gesetzt hat. Die zweite Schleife gewährleistet wie gehabt, dass das Programm so lange läuft, wie noch Threads ausgeführt werden.

Um die Wirkungsweise eines Lock-Objekts zu verdeutlichen, zeigt Ihnen die folgende Tabelle, wie unser Problemszenario durch die Critical Sections gelöst wird:

Zeitfenster	Thread A	Thread B
1	Das Lock-Objekt mit <code>lock.acquire()</code> sperren. Wert von <code>anzahl_threads</code> einlesen, beispielsweise 2.	<i>schläft</i>
----- Zeitfenster von A endet, und Thread B wird aktiviert. -----		
2	<i>schläft</i>	<code>lock.acquire</code> wird aufgerufen, aber das Lock-Objekt ist bereits gesperrt. Deshalb wird B schlafen gelegt.
--- B wurde durch <code>lock.acquire</code> schlafen gelegt. A wird weiter ausgeführt. ---		
3	Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 3. Die neue Instanz an die Referenz <code>anzahl_threads</code> knüpfen. Damit verweist <code>anzahl_threads</code> auf den Wert 3. Das Lock-Objekt wird mit <code>lock.release()</code> wieder freigegeben.	<i>schläft</i>
----- Zeitfenster von A endet, und Thread B wird aktiviert. -----		
4	<i>schläft</i>	Das Lock-Objekt wird automatisch gesperrt, da B <code>lock.acquire</code> aufgerufen hat. Wert von <code>anzahl_threads</code> einlesen, in diesem Fall 3. Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 4. Die neue Instanz an die Referenz <code>anzahl_threads</code> knüpfen. Damit verweist <code>anzahl_threads</code> auf den Wert 4. Das Lock-Objekt wird mit <code>lock.release()</code> wieder freigegeben.

Tabelle 18.2 Lösung des `anzahl_threads`-Problems mit einem Lock-Objekt

Sie sollten darauf achten, dass Sie in Ihren eigenen Programmen alle Stellen, in denen Probleme durch Zugriffe von mehreren Threads vorkommen können, durch Critical Sections schützen.

Unzureichend abgesicherte Programme mit mehreren Threads können sehr schwer reproduzierbare und lokalisierbare Fehler produzieren. Die Herausforderung beim Umgang mit Threads besteht deshalb darin, solche Probleme zu umgehen.

Achtung

Wenn Sie mehrere Lock-Objekte verwenden, kann es passieren, dass sich ein Programm in einem sogenannten *Deadlock* aufhängt, weil zwei gelockte Threads gegenseitig aufeinander warten.

Beispiel:

Es gebe zwei Threads *A* und *B* mit zwei Lock-Objekten *L* und *M*. Nun sperrt *A* das Lock-Objekt *L* und wird schlafen gelegt. In der Zwischenzeit wird der Thread *B* aufgerufen, der das Lock-Objekt *M* sperrt. Anschließend ruft er die *acquire*-Methode von *L* und wird schlafen gelegt, da *L* ja zuvor vom Thread *A* gesperrt wurde. Wenn nun *A* wieder aufgeweckt wird und die *acquire*-Methode von *M* ruft, hängt das Programm auf ewig fest, da die beiden Threads gegenseitig aufeinander warten. Ein *Deadlock* hat sich eingestellt.

18.4 Das Modul `threading`

Mit dem Modul `threading` wird eine objektorientierte Schnittstelle für Threads angeboten.

Jeder Thread ist dabei eine Instanz einer Klasse, die von `threading.Thread` erbt. Da die Klasse selbst ein Teil des globalen Namensraums ist, eignen sich ihre statischen Member sehr gut, um Daten zwischen den Threads auszutauschen. Natürlich muss auch hier der Zugriff auf die von mehreren Threads genutzten Variablen durch Critical Sections gesichert werden.

Wir wollen ein Programm schreiben, das in mehreren Threads parallel prüft, ob vom Benutzer eingegebene Zahlen Primzahlen⁴ sind. Zu diesem Zweck definieren wir eine Klasse `PrimzahlThread`, die von `threading.Thread` erbt und als Parameter für den Konstruktor die zu überprüfende Zahl erwartet.

Die Klasse `threading.Thread` besitzt eine Methode namens `start`, die den Thread ausführt. Was genau ausgeführt werden soll, bestimmt die `run`-Methode, die wir mit unserer Primzahlberechnung überschreiben. Im ersten Schritt soll der Benutzer in einer Eingabeaufforderung Zahlen eingeben können, die dann überprüft werden. Ist die Überprüfung abgeschlossen, wird das Ergebnis auf dem

⁴ Eine Primzahl ist eine natürliche Zahl, die genau zwei Teiler besitzt. Die ersten sechs Primzahlen sind demnach 2, 3, 5, 7, 11 und 13.

Bildschirm ausgegeben. Das Programm inklusive der Klasse `PrimzahlThread` sieht dann folgendermaßen aus.⁵

```
import threading

class PrimzahlThread(threading.Thread):
    def __init__(self, zahl):
        threading.Thread.__init__(self)
        self.Zahl = zahl

    def run(self):
        i = 2
        while i*i <= self.Zahl:
            if self.Zahl % i == 0:
                print("{0} ist nicht prim, "
                      "da {1} = {2} * {3}".format( self.Zahl,
                                                  self.Zahl, i, self.Zahl / i))
                return
            i += 1
        print("{0} ist prim".format(self.Zahl))

meine_threads = []

eingabe = input("> ")
while eingabe != "ende":
    thread = PrimzahlThread(int(eingabe))
    meine_threads.append(thread)
    thread.start()
    eingabe = input("> ")

for t in meine_threads:
    t.join()
```

Innerhalb der Schleife wird die Eingabe vom Benutzer eingelesen, und es wird geprüft, ob es sich um das Schlüsselwort "ende" zum Beenden des Programms handelt. Wurde etwas anderes als "ende" eingegeben, wird eine neue Instanz der Klasse `PrimzahlThread` mit der Benutzereingabe als Parameter erzeugt und mit der `start`-Methode gestartet.

Das Programm verwaltet außerdem eine Liste namens `meine_threads`, in der alle Threads gespeichert werden. Nach dem Verlassen der Eingabeschleife wird über `meine_threads` iteriert und für jeden Thread die `join`-Methode aufgerufen. Die Methode `join` sorgt dafür, dass das Hauptprogramm so lange wartet, bis alle ge-

⁵ Der verwendete Algorithmus für die Primzahlprüfung ist sehr primitiv und dient hier nur als Beispiel für irgendeine rechenintensive Funktion.

starteten Threads beendet worden sind, denn `join` unterbricht die Programmausführung so lange, bis der Thread, für den es aufgerufen wurde, terminiert wurde.

Diese Methode, auf das Ende aller Threads zu warten, ist wesentlich eleganter als die im letzten Abschnitt verwendete Endlosschleife, da mit `join` keine Rechenzeit verschwendet und das Programm automatisch beendet wird, sobald kein Thread mehr läuft.

Ein Programmlauf könnte dann so aussehen, wobei die teils verzögerten Ausgaben zeigen, dass tatsächlich parallel gerechnet wurde:

```
> 737373737373737
> 5672435793
5672435793 ist nicht prim, da  $5672435793 = 3 * 1890811931$ 
> 909091
909091 ist prim
> 100000000000037
> 5643257
5643257 ist nicht prim, da  $5643257 = 23 * 245359$ 
> 4567
4567 ist prim
100000000000037 ist prim
737373737373737 ist prim
> ende
```

18.4.1 Locking im threading-Modul

Genau wie das Modul `_thread` bietet auch `threading` Methoden an, um den Zugriff auf Variablen abzusichern, die in mehreren Threads verwendet werden. Die dazu benutzten Lock-Objekte lassen sich dabei genauso wie die von `thread.allocate_lock` zurückgegebenen Objekte verwenden.

Um den Umgang mit Lock-Objekten zu zeigen, werden wir das Primzahlprogramm des letzten Abschnitts verbessern. Eine Schwachstelle des Programms bestand darin, dass, während der Benutzer gerade die nächste Zahl zur Prüfung eingibt, ein Thread im Hintergrund seine Arbeit beendet hat und sein Ergebnis auf den Bildschirm schreibt. Dadurch verliert der Benutzer unter Umständen die Übersicht, was er schon eingegeben hat, und es sieht äußerst unschön aus, wie das folgende Beispiel zeigt:

```
> 100000000000037
> 5610000000000037 ist prim
547
56547 ist nicht prim, da  $56547 = 3 * 18849$ 
> ende
```

In diesem Fall hat der Benutzer die Zahl 10000000000037 auf ihre Primzahleigenenschaft hin untersuchen wollen. Unglücklicherweise wurde der Thread, der die Überprüfung übernahm, genau dann fertig, als der Benutzer bereits die ersten beiden Ziffern, 56, der nächsten zu prüfenden Zahl, 56547, eingegeben hatte. Dies führte zu einer hässlichen »Zerstückelung« der Eingabe und sollte vermieden werden.

Wir werden zu diesem Zweck die Klasse `PrimzahlThread` mit einem statischen Attribut namens `Ergebnis` versehen, das in einem Dictionary die Ergebnisse der Berechnungen speichert. Dabei wird jeder zu prüfenden Zahl der Status bzw. das Ergebnis der Berechnung zugewiesen, wobei der Wert "in Arbeit" dafür steht, dass aktuell noch gerechnet wird, und der String "prim" anzeigt, dass es sich bei der Zahl um eine Primzahl handelt. Für Nicht-Primzahlen werden wir das gefundene Teilerprodukt in dem Dictionary speichern. Eine Momentaufnahme von `PrimzahlThread.Ergebnis` sähe dann folgendermaßen aus:

```
{
    737373737373737 : "in Arbeit",
    5672435793 : "3 * 1890811931",
    909091 : "prim",
    10000000000037 : "in Arbeit",
    5643257 : "23 * 245359"
}
```

In dem Beispiel befinden sich die Zahlen 737373737373737 und 10000000000037 noch in der Prüfung, während für 909091 bereits nachgewiesen werden konnte, dass sie eine Primzahl ist. 5672435793 und 5643257 sind keine Primzahlen, da sie sich über die angegebenen Produkte berechnen lassen.

In dem neuen Programm wird der Benutzer wie bisher Zahlen eingeben und das Programm durch die Eingabe von "ende" terminieren können. Zusätzlich wird es einen Befehl "status" geben, der den aktuellen Berechnungsstand, eben den Inhalt von `PrimzahlThread.Ergebnis`, ausgibt.

Da die Threads zum Setzen der jeweiligen Ergebnisse alle `PrimzahlThread.Ergebnis` verändern müssen, ist es notwendig, den Zugriff auf das Dictionary mit einer Critical Section abzusichern. Das dazu erforderliche Lock-Objekt speichern wir in der statischen Variable `PrimzahlThread.ErgebnisLock`. Das neue Programm sieht damit wie folgt aus:

```
import threading

class PrimzahlThread(threading.Thread):
    Ergebnis = {}
    ErgebnisLock = threading.Lock()
```



```

def __init__(self, zahl):
    threading.Thread.__init__(self)
    self.Zahl = zahl

    PrimzahlThread.ErgebnisLock.acquire()
    PrimzahlThread.Ergebnis[zahl] = "in Arbeit"
    PrimzahlThread.ErgebnisLock.release()

def run(self):
    i = 2
    while i*i < self.Zahl + 1:
        if self.Zahl % i == 0:
            ergebnis = "{0} * {1}".format(i, self.Zahl / i)

            PrimzahlThread.ErgebnisLock.acquire()
            PrimzahlThread.Ergebnis[self.Zahl] = ergebnis
            PrimzahlThread.ErgebnisLock.release()

            return
        i += 1

    PrimzahlThread.ErgebnisLock.acquire()
    PrimzahlThread.Ergebnis[self.Zahl] = "prim"
    PrimzahlThread.ErgebnisLock.release()

meine_threads = []

eingabe = input("> ")
while eingabe != "ende":

    if eingabe == "status":
        print("----- Aktueller Status -----")
        PrimzahlThread.ErgebnisLock.acquire()
        for z, e in PrimzahlThread.Ergebnis.items():
            print("{0} = {1}".format(z, e))
        PrimzahlThread.ErgebnisLock.release()
        print("-----")

    elif int(eingabe) not in PrimzahlThread.Ergebnis:
        thread = PrimzahlThread(int(eingabe))
        meine_threads.append(thread)
        thread.start()

    eingabe = input("> ")

```

```
for t in meine_threads:
    t.join()
```

Wie Sie sehen, sind alle schreibenden Zugriffe auf `PrimzahlThread.Ergebnis` durch die Aufrufe von `acquire` und `release` umgeben, wodurch das Dictionary gefahrlos in verschiedenen Threads verändert werden kann. Da sich ein Dictionary nicht verändern darf, während darüber iteriert wird, muss auch die Statusausgabe durch eine Critical Section gesichert werden.

In der Schleife für die Verarbeitung der Benutzerdaten ist neben der Ausgabe des aktuellen Status noch eine Abfrage hinzugekommen, die verhindert, dass dieselbe Zahl unnötigerweise mehr als einmal überprüft wird.

Ein Beispiellauf des Programms könnte dann so aussehen:

```
> 100000000000037
> 5643257
> 909091
> 737373737373737
> 56547
> status
----- Aktueller Status -----
5643257 = 5643257 * 245359
909091 = prim
737373737373737 = in Arbeit
100000000000037 = in Arbeit
56547 = 56547 * 18849
-----
> status
----- Aktueller Status -----
5643257 = 5643257 * 245359
909091 = prim
737373737373737 = in Arbeit
100000000000037 = prim
56547 = 56547 * 18849
-----
> status
----- Aktueller Status -----
5643257 = 5643257 * 245359
909091 = prim
737373737373737 = prim
100000000000037 = prim
56547 = 56547 * 18849
-----
> ende
```

Mit dieser Version des Programms werden die angesprochenen Probleme zufriedenstellend beseitigt. Allerdings kann immer noch ein kleiner Schönheitsfehler auftreten: Wenn der Benutzer sehr viele sehr große Zahlen eingibt, rechnet das Programm unter Umständen eine lange Zeit, bevor das erste Ergebnis erzielt wird. Das rührt daher, dass sich die Threads gegenseitig ausbremsen, weil zwar alle Threads gleichzeitig ausgeführt werden, aber durch ihre große Anzahl nur wenig Rechenleistung für den einzelnen Thread übrigbleibt.

Um auch diese Unschönheit zu beseitigen, werden wir im nächsten Abschnitt eine Technik kennenlernen, mit der wir die Anzahl der Threads sinnvoll begrenzen können.

18.4.2 Worker-Threads und Queues

In unseren bisherigen Programmen haben wir immer für jede Aufgabe einen neuen Thread gestartet, so dass es theoretisch beliebig viele Threads geben konnte. Wie am Ende des letzten Abschnitts angemerkt wurde, kann dies zu Geschwindigkeitsproblemen führen, wenn sehr viele Threads gleichzeitig laufen.

Dies lässt sich an einem Beispiel veranschaulichen: Wären wir ein Unternehmen, das für seine Kunden Zahlen daraufhin untersucht, ob sie Primzahlen sind,⁶ könnten wir uns unser Vorgehen so vorstellen, dass wir für jede Zahl, die wir überprüfen möchten, einen separaten Mathematiker einstellen, der mit den nötigen Berechnungen betraut wird. Hat der Mathematiker sein Werk vollendet, gibt er uns als Arbeitgeber Rückmeldung über das Ergebnis und wird entlassen.

In einem realen Unternehmen ist es nicht denkbar, für jede neue Aufgabe einen neuen Arbeiter einzustellen und ihn nach der Fertigstellung seiner Tätigkeit wieder zu entlassen. Vielmehr gibt es eine relativ konstante Anzahl von Arbeitern, denen die Aufgaben zugeteilt werden. Damit auch in diesem Modell eine beliebige Anzahl von Berechnungen durchgeführt werden kann, gibt es in unserer Firma einen Briefkasten, in den die Kunden die zu prüfenden Zahlen einwerfen. Die Arbeiter holen sich dann selbstständig neue Aufgaben aus dem Briefkasten, sobald sie ihre vorherige Arbeit vollendet haben. Ist der Briefkasten einmal leer, warten die Arbeiter so lange, bis neue Zahlen eingeworfen werden.

In der Programmierung spricht man statt von Arbeitern von sogenannten *Worker-Threads* (von engl. *to work* = »arbeiten«). Der Briefkasten wird *Queue* (dt. *Warteschlange*) genannt.

Python hat ein eigenes Modul namens `queue`, um mit Warteschlangen zu arbeiten. Der Konstruktor von `queue` erwartet eine ganze Zahl als Parameter, die an-

⁶ Ob dieses Geschäftsmodell sehr erfolgreich wäre, sei einmal dahingestellt.

gibt, wie viele Elemente maximal in der Warteschlange stehen können. Ist der Parameter kleiner oder gleich 0, ist die Länge der Queue nicht begrenzt.

Queue-Instanzen verfügen im Wesentlichen über drei wichtige Methoden: `put`, `get` und `task_done`.

Mit der `put`-Methode werden neue Aufträge in die Warteschlange gestellt. Sie wird in unserem Beispiel vom Hauptprogramm benutzt werden, um neue Zahlen in den »Briefkasten« zu werfen.

Die Methode `get` liefert die nächste Aufgabe der Queue. Befindet sich gerade kein Arbeitsauftrag in der Warteschlange, blockiert `get` den Thread so lange, bis der nächste Auftrag verfügbar ist.

Hat ein Thread die Prüfung einer Zahl abgeschlossen, muss er dies der Queue mitteilen, indem er `task_done` aufruft. Die Warteschlange kümmert sich dabei selbstständig darum, dass das fertig verarbeitete Element entfernt wird.

Das folgende Beispiel wird fünf Worker-Threads einsetzen, die sich alle eine Queue teilen:

```
import threading
import queue

class Mathematiker(threading.Thread):
    Ergebnis = {}
    ErgebnisLock = threading.Lock()

    Briefkasten = queue.Queue()

    def run(self):
        while True:
            zahl = Mathematiker.Briefkasten.get()
            ergebnis = self.istPrimzahl(zahl)

            Mathematiker.ErgebnisLock.acquire()
            Mathematiker.Ergebnis[zahl] = ergebnis
            Mathematiker.ErgebnisLock.release()

            Mathematiker.Briefkasten.task_done()

    def istPrimzahl(self, zahl):
        i = 2
        while i*i < zahl + 1:
            if zahl % i == 0:
                return "{0} * {1}".format(zahl, zahl / i)
```

```

        i += 1

    return "prim"

meine_threads = [Mathematiker() for i in range(5)]
for thread in meine_threads:
    thread.setDaemon(True)
    thread.start()

eingabe = input("> ")
while eingabe != "ende":
    if eingabe == "status":
        print("----- Aktueller Status -----")
        Mathematiker.ErgebnisLock.acquire()
        for z, e in Mathematiker.Ergebnis.items():
            print("{0}: {1}".format(z, e))
        Mathematiker.ErgebnisLock.release()
        print("-----")

    elif int(eingabe) not in Mathematiker.Ergebnis:
        Mathematiker.ErgebnisLock.acquire()
        Mathematiker.Ergebnis[int(eingabe)] = "in Arbeit"
        Mathematiker.ErgebnisLock.release()

        Mathematiker.Briefkasten.put(int(eingabe))

    eingabe = input("> ")

Mathematiker.Briefkasten.join()

```

Die neben dem Einbau der Queue wichtigen Änderungen im Vergleich zum letzten Programm sind zum einen die `run`-Methode, die jetzt in einer Endlosschleife immer wieder neue Zahlen aus dem Briefkasten nimmt und mit der `istPrimzahl`-Methode überprüft, und zum anderen die Initialisierung und der Abschluss des Programms. Zu Anfang werden die fünf Worker-Threads in einer List Comprehension erzeugt und in der `for`-Schleife gestartet. Durch den Aufruf von `thread.setDaemon(True)` werden die Threads als sogenannte *Dämon-Threads* markiert. Der wesentliche Unterschied zwischen Dämon-Threads und normalen Threads besteht darin, dass ein Programm beendet wird, wenn nur noch Dämon-Threads laufen. Bei normalen Threads kann das Programm so lange laufen, bis auch der letzte Thread beendet worden ist.

Im Beispiel benötigen wir die Dämon-Threads deshalb, weil wir am Ende des Programms nicht wie bisher auf die Terminierung jedes Threads warten, sondern die

`join`-Methode der `Queue` aufrufen. Die Methode `join` unterbricht den Hauptprogramm-Thread so lange, bis alle noch in der Warteschlange stehenden Zahlen verarbeitet worden sind. Ist die Warteschlange leer, wird das Programm inklusive aller Worker-Threads beendet. Dass die Worker-Threads dabei nicht den Programmabbruch behindern können, wird durch `setDaemon` sichergestellt.

Falls Sie sich wundern, warum wir die Zugriffe auf die `Queue` nicht durch Critical Sections abgesichert haben, obwohl alle Threads auf `Mathematiker.Briefkasten` zugreifen, wundern Sie sich zu Recht: Normalerweise wäre es erforderlich, jedes Mal ein Lock-Objekt zu sperren und wieder zu entsperren. Allerdings nimmt uns das `queue`-Modul von Python diese lästige Arbeit ab, was die Arbeit mit Warteschlangen wesentlich komfortabler macht.

Wir werden uns jetzt noch zwei Klassen zuwenden, die für sehr spezielle Zwecke im Zusammenhang mit Threads dienen.

18.4.3 Ereignisse definieren – `threading.Event`

Mit der Klasse `threading.Event` können sogenannte *Ereignisse* (engl. *events*) definiert werden, um Threads bis zum Eintritt eines bestimmten Ereignisses zu unterbrechen.

Ein Thread, der die `wait`-Methode eines frisch erzeugten `threading.Event`-Objekts aufruft, wird so lange unterbrochen, bis ein anderer Thread das Event mit `set` auslöst.

Ausführliche Informationen über `threading.Event` finden Sie in der Python-Dokumentation.

18.4.4 Eine Funktion zeitlich versetzt ausführen – `threading.Timer`

Das `threading`-Modul bietet eine praktische Klasse namens `threading.Timer`, um Funktionen nach dem Verstreichen einer gewissen Zeit aufzurufen.

`threading.Timer(interval, function, args=[], kwargs={})`

Der Parameter *interval* des Konstruktors gibt die Zeit in Sekunden an, die gewartet werden soll, bis die für *function* übergebene Funktion aufgerufen wird. Dabei können Sie für *interval* sowohl Ganzzahlen als auch `float`-Instanzen übergeben. Für *args* und *kwargs* kann eine Liste bzw. ein Dictionary übergeben werden, das die Parameter enthält, mit denen *function* aufgerufen werden soll.

Wir werden `threading.Timer` im nächsten Beispiel verwenden, um exemplarisch einen Wecker zu programmieren:

```
>>> import time, threading
>>> def wecker(gestellt):
    print("RIIIIIIIING!!!")
    print("Der Wecker wurde um {0} Uhr gestellt.".format(
                                                gestellt))
    print("Es ist nun {0} Uhr".format(
        time.strftime("%H:%M:%S")))
>>> timer = threading.Timer(30, wecker,
                             [time.strftime("%H:%M:%S")])
>>> timer.start()
```

(30 Sekunden später)

```
>>> RIIIIIIIIING!!!
Der Wecker wurde um 03:11:26 Uhr gestellt.
Es ist nun 03:11:58 Uhr
```

Mit der Methode `start` beginnt der `Timer` zu laufen und ruft dann – wie Sie der vorhergehenden Ausgabe entnehmen können – nach der festgelegten Zeitspanne die übergebene Funktion auf. Die Differenz von 2 Sekunden rührt daher, dass zwischen dem Erstellen des `Timer`-Objekts und dem Aufrufen der `start`-Methode 2 Sekunden vergangen sind.

Nachdem die `start`-Methode aufgerufen wurde, kann der `Timer` außerdem mit der parameterlosen `cancel`-Methode wieder abgebrochen werden.

»Gauß wusste alles.«
– Ulrich Kaiser

19 Datenspeicherung

In den folgenden Abschnitten werden wir uns mit der permanenten Speicherung von Daten in den verschiedensten Formaten befassen. Das schließt unter anderem komprimierte Archive, XML-Dateien und Datenbanken ein.

19.1 Komprimierte Dateien lesen und schreiben – gzip

Mit dem Modul `gzip` der Standardbibliothek können Sie auf sehr einfache Weise Dateien verarbeiten, die mit der *zlib*-Bibliothek¹ erstellt wurden. Außerdem können Sie damit *zlib*-komprimierte Dateien erzeugen.

Das Modul stellt eine Funktion namens `open` bereit, die sich in ihrer Verwendung an die Built-in Function `open` anlehnt:

`gzip.open(filename[, mode[, compresslevel]])`

Die Funktion `gzip.open` gibt ein Objekt zurück, das wie ein ganz normales Dateiobjekt verwendet werden kann.

Die Parameter *filename* und *mode* sind gleichbedeutend mit denen der Built-in Function `open`.

Mit dem letzten Parameter, *compresslevel*, können Sie angeben, wie stark die Daten beim Schreiben in die Datei komprimiert werden sollen. Erlaubt sind Ganzzahlen von 0 bis 9, wobei 0 für die schlechteste und 9 für die beste Kompressionsstufe steht. Je höher die Kompressionsstufe ist, desto mehr Rechenzeit ist auch für das Komprimieren der Daten erforderlich. Wird der Parameter *compresslevel* nicht angegeben, verwendet `gzip` standardmäßig die beste Kompression.

¹ Die *zlib* ist eine quelloffene Kompressionsbibliothek, die unter anderem vom Unix-Programm `gzip` verwendet wird. Nähere Informationen finden Sie auf der Website der Bibliothek unter <http://www.zlib.net>.


```
>>> import gzip
>>> f = gzip.open("testdatei.gz", "wb")
>>> f.write(b"Hallo Welt")
>>> f.close()
>>> g = gzip.open("testdatei.gz")
>>> g.read()
b'Hallo Welt'
```

In dem Beispiel schreiben wir einen einfachen `bytes`-String in die Datei *testdatei.gz* und lesen ihn anschließend wieder aus.

Andere Module für den Zugriff auf komprimierte Daten

Es existieren in der Standardbibliothek von Python weitere Module, die den Zugriff auf komprimierte Daten erlauben. Aus Platzgründen müssen wir hier auf eine ausführliche Besprechung verzichten.

Die folgende Tabelle gibt einen Überblick über alle Module, die komprimierte Daten verwalten:

Modul	Beschreibung
zlib	Eine Low-Level-Bibliothek, die direkten Zugriff auf die Funktionen der zlib ermöglicht. Mit ihr ist es unter anderem möglich, Strings zu komprimieren oder zu entpacken. Das Modul <code>gzip</code> greift intern auf das Modul <code>zlib</code> zurück.
gzip	Beschreibung siehe oben.
bz2	Bietet komfortablen Zugriff auf Daten, die mit dem bzip2-Algorithmus komprimiert wurden, und ermöglicht es, neue komprimierte Dateien zu erzeugen. Auch <code>bz2</code> implementiert ein Dateiobjekt, das genauso zu handhaben ist wie die Objekte, die die Built-in Function <code>open</code> zurückgibt. In der Regel ist die Kompression von bzip2 der von <code>zlib</code> in puncto Kompressionsrate überlegen.
zipfile	Ermöglicht den Zugriff auf ZIP-Archive, wie sie beispielsweise von dem bekannten Programm <i>WinZip</i> erstellt werden. Auch die Manipulation und Erzeugung neuer Archive ist möglich. Das Modul <code>zipfile</code> ist sehr umfangreich und mächtig und in jedem Fall einen näheren Blick wert.
tarfile	Implementiert Funktionen und Klassen, um die in der Unix-Welt weitverbreiteten tar-Archive zu lesen oder zu schreiben.

Tabelle 19.1 Übersicht über Pythons Kompressionsmodule

19.2 XML

Das Modul `xml` der Standardbibliothek erlaubt es, XML-Dateien einzulesen. XML (kurz für »Extensible Markup Language«) ist eine standardisierte Beschreibungssprache, die es ermöglicht, komplexe, hierarchisch aufgebaute Datenstrukturen in einem lesbaren Textformat abzuspeichern. XML kann daher sehr gut zum Datenaustausch bzw. zur Datenspeicherung verwendet werden.

Besonders in der Welt des Internets finden sich viele auf XML basierende Beschreibungssprachen, wie beispielsweise XHTML, RSS, MathML oder SVG.

An dieser Stelle soll eine kurze Einführung in XML gegeben werden. Dazu dient folgende einfache XML-Datei, die eine Möglichkeit aufzeigt, wie der Inhalt eines Python-Dictionaries dauerhaft abgespeichert werden könnte:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
  <eintrag>
    <schluessel>Hallo</schluessel>
    <wert>0</wert>
  </eintrag>
  <eintrag>
    <schluessel>Welt</schluessel>
    <wert>1</wert>
  </eintrag>
</dictionary>
```

Die erste Zeile der Datei ist die sogenannte *XML-Deklaration*. Diese optionale Angabe kennzeichnet die verwendete XML-Version und das Encoding, in dem die Datei gespeichert wurde. Durch Angabe des Encodings, in diesem Fall UTF-8, können auch Umlaute und andere Sonderzeichen korrekt verarbeitet werden.

Abgesehen von der XML-Deklaration besteht ein XML-Dokument aus sogenannten *Tags*. Tags können wie Klammern geöffnet und geschlossen werden und stellen damit eine Art Gruppe dar, die weitere Tags enthalten kann. Jedes Tag hat einen Namen, den sogenannten *Tag-Namen*. Um ein Tag zu öffnen, wird dieser Tag-Name in spitze Klammern geschrieben. Ein schließendes Tag besteht aus dem Tag-Namen, der zusammen mit einem Slash ebenfalls in spitze Klammern geschrieben wird. Das folgende Beispiel zeigt ein öffnendes Tag, direkt gefolgt von dem entsprechenden schließenden Tag:

```
<wert></wert>
```

Innerhalb eines Tags können sowohl Text als auch weitere Tags stehen. Auf diese Weise lässt sich eine hierarchische Struktur erstellen, die dazu in der Lage ist, auch komplexe Datensätze abzubilden.

Zudem können Sie bei einem Tag sogenannte *Attribute* angeben. Dazu wollen wir das vorherige Beispiel dahingehend erweitern, dass der Datentyp der Schlüssel und Werte des abzubildenden Dictionarys als Attribut des jeweiligen `schluessel-` bzw. `wert-`Tags gespeichert werden kann.

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
  <eintrag>
    <schluessel typ="str">Hallo</schluessel>
    <wert typ="int">0</wert>
  </eintrag>
  <eintrag>
    <schluessel typ="str">Welt</schluessel>
    <wert typ="int">1</wert>
  </eintrag>
</dictionary>
```

Ein Attribut stellt im Prinzip ein Schlüssel-Wert-Paar dar. Im Beispiel wurde jedem `schluessel-` und `wert-`Tag ein Attribut `typ` verpasst, über das der Datentyp des Schlüssels bzw. des Werts angegeben werden kann. Beachten Sie, dass der Wert eines XML-Attributs stets in Anführungszeichen zu schreiben ist.

Zum Einlesen von XML-Dateien stellt Python, wie die meisten anderen Programmiersprachen oder XML-Bibliotheken auch, zwei sogenannte *Parser* zur Verfügung. Der Begriff des Parsers ist nicht auf XML beschränkt, sondern bezeichnet ganz allgemein ein Programm, das eine Syntaxanalyse bestimmter Daten eines speziellen Formats leistet. Die beiden im Modul `xml` enthaltenen Parser heißen `dom` und `sax` und implementieren zwei unterschiedliche Herangehensweisen an das XML-Dokument. Aus diesem Grund ist es sinnvoll, beide getrennt und ausführlich zu besprechen, was in den nächsten beiden Abschnitten geschehen soll. Das Thema des dritten Abschnitts soll eine weitere Python-spezifische Herangehensweise an XML-Daten namens `ElementTree` sein.

Hinweis

Eine Besonderheit bei XML-Tags stellen sogenannte *körperlose Tags* dar. Solche Tags spielen in den Beispielen, die in diesem Buch vorgestellt werden, keine Rolle, sind jedoch in einigen Fällen durchaus sinnvoll. Ein körperloses Tag sieht folgendermaßen aus:

```
<tag attr="wert" />
```

Ein körperloses Tag ist öffnendes und schließendes Tag zugleich und darf demzufolge nur über Attribute verfügen. Ein solches Tag kann keinen Text oder weitere Tags enthalten. XML-Parser behandeln körperlose Tags, als stünde `<tag attr="wert"></tag>` in der XML-Datei.

19.2.1 DOM – Document Object Model

Das *Document Object Model*, kurz DOM, ist eine Schnittstelle, die vom *World Wide Web Consortium* (W3C) standardisiert wurde und es ermöglicht, auf einzelne Elemente einer XML-Datei zuzugreifen und diese zu modifizieren. Dazu wird die Datei vollständig eingelesen und zu einer baumartigen Struktur aufbereitet. Jedes Tag wird durch eine Klasse repräsentiert, den sogenannten *Knoten* (engl. *node*). Durch Methoden und Attribute dieser Klasse können die enthaltenen Informationen ausgelesen oder verändert werden.

Das DOM ist vor allem dann interessant, wenn ein wahlfreier Zugriff auf die XML-Daten möglich sein muss. Unter einem *wahlfreien Zugriff* versteht man den punktuellen Zugriff auf verschiedene, voneinander unabhängige Teile des Datensatzes. Das Gegenteil des wahlfreien Zugriffs wäre das sequentielle Einlesen der XML-Datei.

Da die Datei stets vollständig eingelesen wird, ist die Verwendung von DOM sehr speicherintensiv. Im Gegensatz dazu liest das Konkurrenzmodell SAX immer nur kleine Teile der XML-Daten ein und stellt sie sofort zur Weiterverarbeitung zur Verfügung. Diese Herangehensweise benötigt weniger Arbeitsspeicher und erlaubt es, Teile der gespeicherten Daten bereits zu verwenden, beispielsweise anzuzeigen, während die Datei selbst noch nicht vollständig eingelesen ist. Ein wahlfreier Zugriff auf die XML-Daten und ihre Manipulation ist mit SAX allerdings nicht möglich.

Jetzt möchten wir darauf zu sprechen kommen, wie die XML-Daten bei Verwendung eines DOM-Parsers aufbereitet werden. Betrachten Sie dazu noch einmal unser vorheriges Beispiel einer XML-Datei:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
  <eintrag>
    <schluessel typ="str">Hallo</schluessel>
    <wert typ="int">0</wert>
  </eintrag>
  <eintrag>
    <schluessel typ="str">Welt</schluessel>
    <wert typ="int">1</wert>
  </eintrag>
</dictionary>
```

Unter Verwendung eines DOM-Parsers werden die XML-Daten zu einem sogenannten *Baum* aufbereitet. Ein Baum besteht aus einzelnen Klassen, den sogenannten Knoten. Jede dieser Knotenklassen enthält verschiedene Referenzen auf benachbarte Knoten, nämlich:

- Ihr *Elternelement* (engl. *parent*). Das ist der Knoten, der im Baum direkt über diesem Knoten steht.
- Ihre *Kindelemente* (engl. *children*). Das sind alle Knoten, die im Baum direkt unter diesem Knoten stehen.
- Ihre *Geschwisterelemente* (engl. *siblings*). Das sind alle Knoten, die im Baum direkt neben diesem Knoten stehen und dasselbe Elternelement haben.

Somit enthält jeder Knoten des Baumes Referenzen zu allen umliegenden, auch verwandten Knoten. Auf diese Weise lässt sich der Baum vollständig durchlaufen und verarbeiten.

Die aus dem obigen Beispiel erzeugte Baumstruktur sieht folgendermaßen aus:

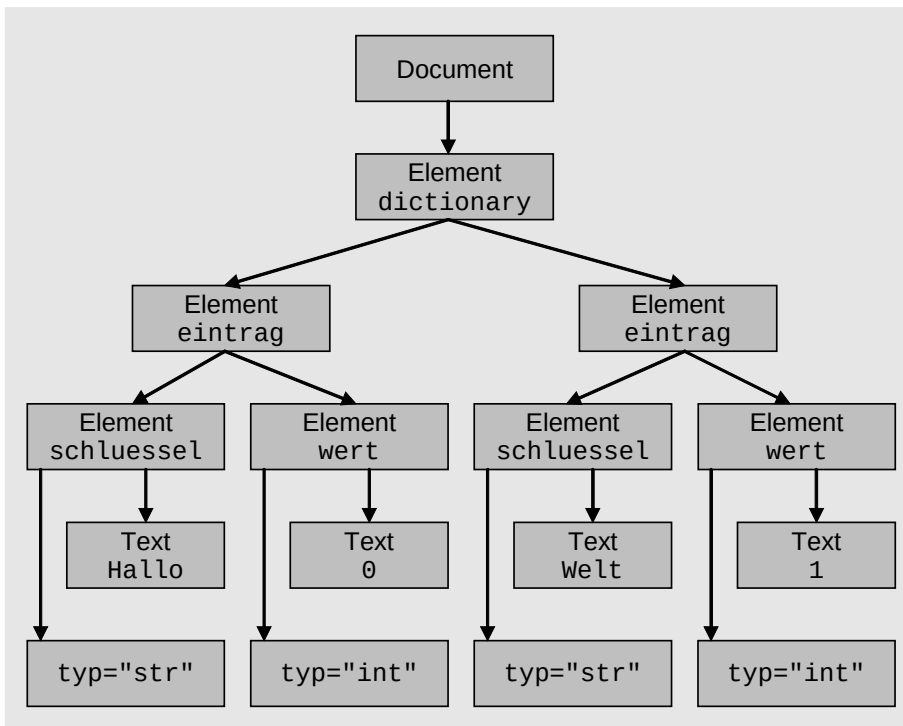


Abbildung 19.1 Vom DOM-Parser erzeugter Baum

Dabei handelt es sich bei `Document`, `Element` und `Text` um die grundlegenden Knotenklassen, aus denen ein DOM-Baum aufgebaut ist. Die `Document`-Instanz ist einmalig und entspricht der *Wurzel* des Baumes (engl. *root*). Sie enthält eine Referenz auf alle Tags erster Ordnung, wie in diesem Fall beispielsweise das Tag `dictionary`. Diesem Knoten sind mehrere Instanzen der Klasse `Element` unterge-

ordnet, die jeweils ein `eintrag`-Tag repräsentieren. Durch Attribute dieser Klasse können Informationen wie der Tag-Name, enthaltene XML-Attribute oder Ähnliches abgerufen werden.

Beachten Sie zum einen, dass in Abbildung 19.1 aus Gründen der Übersichtlichkeit keine Geschwisterbeziehungen eingezeichnet wurden, und zum anderen, dass die Attribute der Elemente `schluessel` und `wert` keine eigenständigen Instanzen einer Knotenklasse sind, sondern Teil des Elementknotens.

Neben den Klassen `Document` und `Element` existieren Instanzen einer weiteren Klasse namens `Text`. Diese Instanzen enthalten Text, der innerhalb eines Tags geschrieben wurde.

Abgesehen von den hier aufgelisteten Klassen gibt es weitere Knotenklassen, die allerdings nur in Spezialfällen im Baum vorkommen. So existiert beispielsweise die Klasse `Comment` für ein Kommentar-Tag in der XML-Datei. Wir möchten uns in diesem Kapitel auf das Wesentliche, das heißt auf die Klassen `Document`, `Element` und `Text`, beschränken.

Beispiel

An dieser Stelle soll die Verwendung von DOM an einem einfachen Beispiel gezeigt werden. Dazu rufen wir uns erneut unsere Beispieldatei ins Gedächtnis, deren Zweck es war, den Inhalt eines Python-Dictionaries abzubilden:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
  <eintrag>
    <schluessel typ="str">Hallo</schluessel>
    <wert typ="int">0</wert>
  </eintrag>
</dictionary>
```

Die Datei besteht aus einem Tag erster Ordnung namens `dictionary`, in dem mehrere `eintrag`-Tags vorkommen dürfen. Jedes `eintrag`-Tag enthält zwei untergeordnete Tags namens `schluessel` und `wert`, die gemeinsam jeweils ein Schlüssel-Wert-Paar des Dictionarys repräsentieren. Der Datentyp des Schlüssels bzw. des Wertes wird über das Attribut `typ` festgelegt, das bei den Tags `schluessel` und `wert` vorkommen muss.

Das Beispielprogramm soll dazu in der Lage sein, eine solche XML-Datei einzulesen und das entsprechende Dictionary daraus zu rekonstruieren. Im Folgenden soll der Quelltext des Beispielprogramms besprochen werden.

```
import xml.dom.minidom as dom
```

```
def knoten_auslesen(knoten):
    return eval("{0}('{1}')".format(knoten.getAttribute("typ"),
                                   knoten.firstChild.data.strip()))
```

In der ersten Zeile wird der DOM-Parser eingebunden und unter dem Namensraum `dom` verfügbar gemacht. Für dieses Beispiel wurde der Parser `xml.dom.minidom` eingebunden, der eine grundlegende und simple Implementation darstellt, die in den meisten Fällen genügen sollte. Abgesehen von dem Minidom-Parser existieren weitere spezielle DOM-Parser im Paket `xml.dom`.

Danach wird die Funktion `knoten_auslesen` definiert, deren Aufgabe es ist, aus einer `Element`-Instanz das Attribut `typ` auszulesen und den im Element enthaltenen Text in den angegebenen Datentyp zu konvertieren. Dazu wird dynamisch ein String erzeugt, der beispielsweise für den Typ `int` und den Text `"123"` zu `"int('123')"` wird. Dieser String wird mittels `eval` interpretiert und zurückgegeben. Beachten Sie, dass aus Gründen der Übersichtlichkeit alle Konsistenzprüfungen weggelassen wurden. In einem normalen Programm sollte in der Funktion `knoten_auslesen` beispielsweise geprüft werden, ob ein Attribut `typ` überhaupt existiert und ob der dort angegebene Datentyp gültig ist.

Das Auslesen eines XML-Attributs geschieht über die Methode `getAttribute` einer `Element`-Instanz. Um den vom Tag umschlossenen Text auszulesen, wird über das Attribut `firstChild` das erste Kindelement der übergebenen `Element`-Instanz angesprochen. Dabei handelt es sich um die jeweilige `Text`-Instanz. Über das Attribut `data` dieser `Text`-Instanz kann der enthaltene Text ausgelesen werden.

Beachten Sie beim Arbeiten mit `Text`-Instanzen, dass der DOM-Standard vorsieht, dass Whitespace-Zeichen, auch wenn sie nur aus Formatierungsgründen in der XML-Datei stehen, später im Baum wiederzufinden sind. Aus diesem Grund müssen wir eventuell vorkommende Whitespace-Zeichen durch Aufruf der String-Methode `strip` entfernen.

```
def lade_dict(dateiname):
    d = {}
    baum = dom.parse(dateiname)

    for eintrag in baum.firstChild.childNodes:
        if eintrag.nodeName == "eintrag":
            schluessel = wert = None

            for knoten in eintrag.childNodes:
                if knoten.nodeName == "schluessel":
                    schluessel = knoten_auslesen(knoten)
                elif knoten.nodeName == "wert":
                    wert = knoten_auslesen(knoten)
```

```

        d[schluessel] = wert
    return d

```

Danach wird die Hauptfunktion `lade_dict` definiert. Die Aufgabe dieser Funktion ist es, eine XML-Datei, deren Dateinamen sie übergeben bekommt, zu öffnen, die enthaltenen Informationen zu extrahieren, in das Dictionary `d` zu schreiben und das entstandene Dictionary zurückzugeben.

Zunächst wird durch Aufruf der Funktion `parse` des `minidom`-Parsers das XML-Dokument eingelesen und zu einer Baumstruktur aufbereitet. Der Name `baum` referenziert jetzt eine Instanz der Klasse `Document`, über die auf alle Elemente des Dokuments zugegriffen werden kann. Alternativ hätten wir auch die Methode `parseString` des `minidom`-Parsers aufrufen können, wenn die XML-Daten in Form eines Strings vorlägen.

Dann soll über alle `eintrag`-Tags iteriert und das jeweilige Schlüssel-Wert-Paar ins Dictionary `d` eingefügt werden. Dazu nutzen wir die Attribute der Klasse `Node`, von der sowohl `Document` als auch `Element` abgeleitet sind. Von der `Document`-Instanz `baum` aus erreichen wir über das Attribut `baum.firstChild` das erste Kindelement, also die `Element`-Instanz, die das `dictionary`-Tag repräsentiert. Genau genommen interessieren wir uns jedoch auch nicht für das `dictionary`-Tag, sondern für alle diesem Tag direkt untergeordneten Elemente. Diese erreichen wir über das Attribut `childNodes`, das eine Liste aller Kindelemente bereitstellt. Über diese Liste wird in einer `for`-Schleife iteriert.

Innerhalb der `for`-Schleife wird zunächst geprüft, ob es sich tatsächlich um den Knoten eines `eintrag`-Tags handelt. Dazu wird das Attribut `nodeName` verwendet, das jede `Node`-Instanz, also jeder Knoten, besitzt. Beachten Sie, wie bereits gesagt, dass laut DOM-Standard auch Whitespaces, die zur Formatierung der XML-Datei eingesetzt wurden, in Form von `Text`-Instanzen im DOM-Baum einzutragen sind. Diese `Text`-Instanzen werden hier ebenfalls herausgefiltert: ihr `nodeName`-Wert ist `"#text"`. Zudem werden zwei Referenzen namens `schluessel` und `wert` angelegt, die wir später zum Aufbau des Dictionarys verwenden.

Die darauffolgende `for`-Schleife iteriert über alle Kindelemente des `eintrag`-Tags. Je nachdem, ob es sich bei dem aktuellen Kindelement um ein `schluessel`-Tag oder um ein `wert`-Tag handelt, wird das Ergebnis des Funktionsaufrufs von `knoten_auslesen` dem Namen `schluessel` bzw. `wert` zugewiesen. Nachdem die innere Schleife durchlaufen ist, werden Schlüssel und Wert ins Dictionary `d` eingetragen.

Beachten Sie unbedingt, dass wir in diesem Beispiel davon ausgehen, dass die XML-Datei exakt unseren Ansprüchen entspricht. In einem wirklichen Programm sollten Sie grundsätzlich davon ausgehen, dass auch fehlerhafte Angaben vor-

kommen, und diese entsprechend behandeln. Auch der sorglose Umgang mit dem Attribut `typ` (direktes Übergeben nach `eval`) sollte in einem fertigen Programm so nicht vorkommen.

Dieses Beispiel sollte einen kurzen Überblick über die Verwendung des DOM-Baumes bieten. Im Folgenden werden die Klassen `Node`, `Document`, `Element` und `Text` besprochen, aus denen sich ein DOM-Baum zusammensetzt.

Die Klasse Node

Die Klasse `Node` ist die Basisklasse aller im DOM-Baum verwendeten Klassen. Das bedeutet, dass die in dieser Klasse enthaltene Funktionalität an allen Knoten des Baumes verfügbar ist. In der Klasse `Node` sind vor allem Attribute und Methoden enthalten, die Zugriff auf verwandte Knoten – das heißt Kinder, Geschwister oder den Elternknoten – ermöglichen. Im Folgenden sollen die wichtigsten Attribute der Klasse `Node` beschrieben werden. Dabei soll *n* eine Instanz der Klasse `Node` sein.

n.nodeType

Kennzeichnet den Typ des Knotens. Das Attribut referenziert eine ganze Zahl, die mit folgenden symbolischen Konstanten verglichen werden kann:

Konstante	Beschreibung
<code>Node.DOCUMENT_NODE</code>	Bei dem Knoten handelt es sich um eine <code>Document</code> -Instanz.
<code>Node.ELEMENT_NODE</code>	Bei dem Knoten handelt es sich um eine <code>Element</code> -Instanz.
<code>Node.TEXT_NODE</code>	Bei dem Knoten handelt es sich um eine <code>Text</code> -Instanz.

Tabelle 19.2 Konstanten zur Beschreibung eines Knotentyps

Wie bereits gesagt, gibt es neben den hier besprochenen `Node`-Typen noch weitere, die in ihrer Bedeutung jedoch zu speziell sind, um hier ausführlich behandelt zu werden. So existiert beispielsweise die Konstante `Node.COMMENT_NODE` für einen Kommentarknoten. Eine ausführliche Übersicht über alle Typen finden Sie in der Python Dokumentation bzw. in der dort verlinkten DOM-Spezifikation des W3C.

n.parentNode

Referenziert das Elternelement des Knotens *n*. Wenn es sich bei dem Knoten um die `Document`-Instanz handelt, referenziert dieses Attribut `None`.

n.previousSibling

Referenziert das Geschwisterelement, das in der Reihenfolge vor dem Knoten *n* steht, oder `None`, wenn dieser Knoten das erste Kind von `parentNode` ist.

n.nextSibling

Referenziert das Geschwisterelement, das in der Reihenfolge hinter dem Knoten *n* steht, oder *None*, wenn dieser Knoten das letzte Kind von *parentNode* ist.

n.firstChild

Referenziert das erste Kindelement des Knotens *n* oder *None*, wenn keine untergeordneten Knoten existieren.

n.lastChild

Referenziert das letzte Kindelement des Knotens *n* oder *None*, wenn keine untergeordneten Knoten existieren.

Abbildung 19.2 verdeutlicht die hier vorgestellten Attribute anhand der Beziehung von drei Knoten eines Baumes:

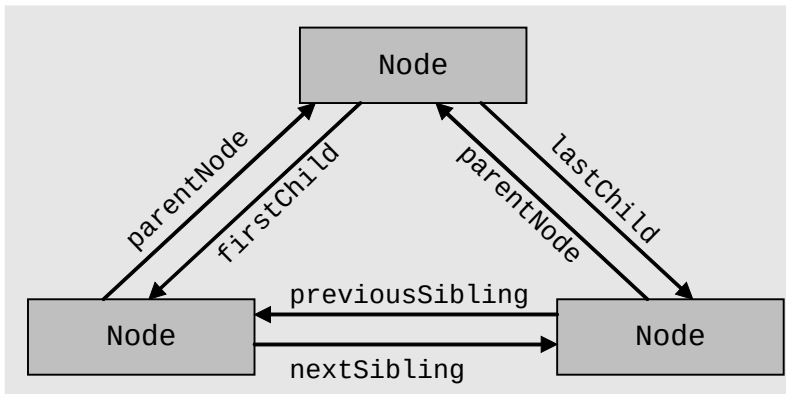


Abbildung 19.2 Verwandtschaftsbeziehungen dreier Knoten

n.childNodes

Referenziert eine Liste aller Kinder des Knotens *n*.

Dieser Auflistung der wichtigsten Attribute der Klasse *Node* folgen die wichtigsten Methoden dieser Klasse.

n.hasChildNodes()

Gibt *True* zurück, wenn der Knoten *n* über Kinder verfügt, andernfalls *False*.

n.appendChild(newChild)

Fügt die *Node*-Instanz *newChild* als Kindelement an das Ende der Liste aller Kinder von *n* ein. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.insertBefore(newChild, refChild)

Fügt die `Node`-Instanz *newChild* als Kindelement des aktuellen Knotens vor dem Kindelement *refChild* in die Liste aller Kinder von *n* ein. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.removeChild(oldChild)

Löscht das angegebene Kindelement *oldChild*. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.replaceChild(newChild, oldChild)

Ersetzt das Kindelement *oldChild* durch *newChild*. Beachten Sie, dass diese Methode den DOM-Baum verändert.

n.writexml(writer[, indent[, addindent[, newl]])

Schreibt die `Node`-Instanz *n* mitsamt all ihren Kindelementen als XML in das geöffnete Dateiojekt *writer*. Beachten Sie, dass diese Methode auch an die Klasse `Document` weitervererbt wird. Wenn sie für eine `Document`-Instanz aufgerufen wird, kann der vollständige DOM-Baum als XML-Datei gespeichert werden.

Die optionalen Parameter *indent*, *addindent* und *newl* (allesamt Strings) werden verwendet, um die Ausgabe der XML-Daten zu formatieren. Dabei steht *indent* für die Zeichen, die zur Einrückung der gesamten Ausgabe verwendet werden, *addindent* für die Zeichen, die zur Einrückung tieferer Ebenen verwendet werden, und *newl* für das zu verwendende Newline-Zeichen.

Wenn die Methode auf einer `Document`-Instanz aufgerufen wird, kann ein zusätzlicher, optionaler Schlüsselwortparameter *encoding* angegeben werden. Das hier als String übergebene Encoding wird in die XML-Deklaration eingetragen und zum Speichern der Datei verwendet.

n.toxml([encoding])

Ähnlich wie `writexml`, gibt die XML-Daten jedoch als String zurück. Optional wird über den Parameter *encoding* ein Encoding angegeben, das in die XML-Deklaration geschrieben und im zurückgegebenen String verwendet wird.

n.toprettyxml([indent[, newl]])

Ähnlich wie `toxml`, gibt die XML-Daten jedoch in einem formatierten String zurück. Um die Formatierung der Daten zu verändern, können Sie das Einrückungszeichen (*indent*, üblicherweise `\t`) und das zu verwendende Newline-Zeichen (*newl*, üblicherweise `\n`) angeben.

Ein Encoding kann wie bei der Methode `writexml` vorgegeben werden.

Die Klasse Document

Ein von einem DOM-Parser erzeugter Baum enthält als Wurzelement eine Instanz der Klasse `Document`. Dies ist die Instanz, die bei einem Aufruf der Funktion `parse` zurückgegeben wird und alle weiteren Elemente des Baumes direkt oder indirekt referenziert. Eine `Document`-Instanz verwaltet dabei immer ein vollständiges XML-Dokument.

Die Klasse `Document` erbt von der Basisklasse `Node`.

Nachfolgend sollen die wichtigsten Methoden und Attribute der Klasse `Document` erläutert werden. Dabei sei d eine Instanz der Klasse `Document`.

d.documentElement

Dieses Attribut referenziert die `Element`-Instanz des ersten Tags des XML-Dokuments d . Beachten Sie, dass ein wohlgeformtes XML-Dokument über genau ein Wurzel-Tag verfügt. Sollten mehrere sogenannte *Toplevel-Tags* vorkommen, kann auf diese über ihre Geschwisterbeziehung zu `documentElement` zugegriffen werden.

d.createElement(tagName)

Erzeugt einen neuen Elementknoten mit dem Tag-Namen $tagName$. Die Funktion gibt eine Instanz der Klasse `Element` zurück. Beachten Sie, dass der Knoten zwar erzeugt, aber nicht automatisch in den Baum eingefügt wird. Dazu können beispielsweise die Methoden `insertBefore` oder `appendChild` der Klasse `Node` verwendet werden.

d.createTextNode(data)

Erzeugt einen neuen Textknoten mit dem Inhalt $data$. Die Funktion gibt eine Instanz der Klasse `Text` zurück. Für diese Methode gilt ebenfalls der Hinweis, dass der erzeugte Knoten nicht automatisch in den DOM-Baum eingefügt wird.

d.getElementsByTagName(tagName)

Gibt eine Liste zurück, in der alle `Element`-Instanzen enthalten sind, die Tags mit dem Tag-Namen $tagName$ repräsentieren.

Die Klasse Element

Die Klasse `Element` repräsentiert ein Tag im DOM-Baum. Sie erbt von der Basisklasse `Node`.

Im Folgenden sollen die wichtigsten Attribute und Methoden der Klasse `Element` erläutert werden. Dabei sei e eine Instanz der Klasse `Element`.

e.tagName

Dieses Attribut referenziert den Tag-Namen des von *e* repräsentierten Tags.

e.getElementsByTagName(tagName)

Äquivalent zu `Document.getElementsByTagName`. Beachten Sie, dass diese Methode nur nach direkt oder indirekt untergeordneten Elementen mit dem Tag-Namen *tagName* sucht.

e.hasAttribute(name)

Gibt `True` zurück, wenn das Element *e* ein Attribut mit dem Schlüssel *name* besitzt, andernfalls `False`.

e.getAttribute(name)

Gibt den Wert des Attributs mit dem Schlüssel *name* zurück. Sollte kein Attribut *name* existieren, wird ein leerer String zurückgegeben.

e.removeAttribute(name)

Löscht das Attribut mit dem Schlüssel *name*. Beachten Sie, dass keine Exception geworfen wird, wenn kein Attribut mit dem Schlüssel *name* existiert.

e.setAttribute(name, value)

Erzeugt ein neues Attribut mit dem Schlüssel *name* und dem Wert *value* oder überschreibt ein bereits bestehendes Attribut.

Die Klasse Text

Die Klasse `Text` erbt von `Node` und fügt ein einziges Attribut hinzu:

t.data

Das Attribut `data` referenziert den String, den die `Text`-Instanz *t* repräsentiert.

Schreiben einer XML-Datei

Im vorangegangenen Beispiel wurde gezeigt, wie die in einer XML-Datei enthaltenen Daten zu einem Baum aufbereitet werden können. Zudem haben Sie soeben einige Methoden der Knotenklassen des Baums kennengelernt, die den Baum modifizieren. Der nächste logische Schritt ist es, den modifizierten Baum wieder als XML-Datei abzuspeichern.

In diesem Abschnitt besprechen wir ein Beispielprogramm, das den umgekehrten Weg des ersten Beispiels geht. Das heißt, es erzeugt aus einem Dictionary einen DOM-Baum und speichert diesen als XML-Datei ab. Diese XML-Datei soll so aufgebaut sein, dass das vorherige Beispielprogramm sie wieder auslesen kann.

Das Schreiben der XML-Datei soll durch eine Funktion `schreibe_dict` erfolgen, die das zu schreibende Dictionary `d` und den Dateinamen der Ausgabedatei als Parameter übergeben bekommt. Der Quelltext des Beispielsprogramms sieht folgendermaßen aus:

```
import xml.dom.minidom as dom

def erstelle_eintrag(schluessel, wert):
    tag_eintrag = dom.Element("eintrag")
    tag_schluessel = dom.Element("schluessel")
    tag_wert = dom.Element("wert")

    tag_schluessel.setAttribute("typ", type(schluessel).__name__)
    tag_wert.setAttribute("typ", type(wert).__name__)

    text = dom.Text()
    text.data = str(schluessel)
    tag_schluessel.appendChild(text)

    text = dom.Text()
    text.data = str(wert)
    tag_wert.appendChild(text)

    tag_eintrag.appendChild(tag_schluessel)
    tag_eintrag.appendChild(tag_wert)
    return tag_eintrag
```

Auch hier wird, ähnlich wie beim vorherigen Beispiel, zuerst eine Hilfsfunktion angelegt, die einen Schlüssel und einen Wert übergeben bekommt und daraus eine `Element`-Instanz erzeugt, die das entsprechende `eintrag`-Tag repräsentiert. Die Funktion an sich bedarf eigentlich keiner weiteren Erläuterung. Einzig erwähnenswert ist folgender Ausdruck:

```
type(schluessel).__name__
```

Dieser Ausdruck ermittelt den Namen des Datentyps der von `schluessel` referenzierten Instanz. Das wäre beispielsweise `"int"` für ganze Zahlen oder `"str"` für Strings. Jetzt folgt die Hauptfunktion des Beispielsprogramms:

```
def schreibe_dict(d, dateiname):
    baum = dom.Document()
    tag_dict = dom.Element("dictionary")
    for schluessel, wert in d.items():
        tag_eintrag = erstelle_eintrag(schluessel, wert)
        tag_dict.appendChild(tag_eintrag)

    baum.appendChild(tag_dict)

    with open(dateiname, "w") as f:
        baum.writexml(f, "", "\t", "\n")
```

Im Funktionskörper wird zunächst eine neue Instanz der Klasse `Document` angelegt. Diese Instanz soll die Wurzel des DOM-Baums werden, den wir im Laufe der Funktion erzeugen, und wird von `baum` referenziert. Danach wird das oberste Element, das `dictionary`-Tag, erzeugt und der Referenz `tag_dict` zugewiesen.

Danach durchläuft eine `for`-Schleife alle Schlüssel-Wert-Paare des Dictionarys `d`. In jedem Schleifendurchlauf wird eine neue `Element`-Instanz für das jeweilige `eintrag`-Tag mithilfe der Funktion `erstelle_eintrag` erzeugt. Die erzeugte `Element`-Instanz wird daraufhin als Kindelement in das `dictionary`-Tags eingefügt.

Am Ende der Funktion wird das `dictionary`-Tag in den DOM-Baum eingefügt, die Datei `dateiname` zum Schreiben geöffnet, und die XML-Daten werden mittels der Methode `writexml` hineingeschrieben.

Die hier vorgestellte Funktion `schreibe_dict` arbeitet perfekt mit der Funktion `lade_dict` des vorherigen Beispiels zusammen. Das bedeutet, dass eine von `schreibe_dict` erzeugte XML-Datei problemlos von `lade_dict` wieder eingelesen werden kann.

Damit wäre das Konzept des Document Object Model umrissen und anhand zweier grundlegender Beispiele erklärt. Beachten Sie, dass hier nicht alle Möglichkeiten von DOM angesprochen wurden. Fühlen Sie sich also dazu ermutigt, weiter zu recherchieren und auszuprobieren, wenn Sie weitere Details zu speziellen Features des DOM erfahren möchten.

19.2.2 SAX – Simple API for XML

Nachdem wir uns im letzten Abschnitt ausführlich der DOM-Herangehensweise an XML-Dateien gewidmet haben, möchten wir nun einen zweiten Weg vorstellen, diese Dateien zu verarbeiten. Die *Simple API for XML*, kurz *SAX*, baut im Gegensatz zu DOM kein vollständiges Abbild der XML-Datei im Speicher auf, sondern liest die Datei fortlaufend ein und setzt den Programmierer durch Aufrufen bestimmter Funktionen davon in Kenntnis, dass beispielsweise ein öffnendes oder schließendes Tag gelesen wurde. Diese Herangehensweise hat neben der Speichereffizienz einen weiteren Vorteil: Beim Laden von sehr großen XML-Dateien können bereits eingelesene Teile weiterverarbeitet werden, obwohl die Datei noch nicht vollständig eingelesen worden ist.

Allerdings sind mit der Verwendung von SAX auch einige Nachteile verbunden. So ist beispielsweise, anders als beispielsweise bei DOM, kein wahlfreier Zugriff auf einzelne Elemente der XML-Daten möglich. Außerdem sieht SAX keine Möglichkeit vor, die XML-Daten komfortabel zu verändern oder wieder zu speichern. Doch nun zur Funktionsweise von SAX.

Das Einlesen einer XML-Datei durch einen SAX-Parser, in der SAX-Terminologie auch *Reader* genannt, geschieht event-gesteuert. Das bedeutet, dass der Programmierer beim Erstellen des Readers verschiedene sogenannte *Callback-Funktionen* einrichten und mit einem bestimmten Event verknüpfen muss. Wenn beim Einlesen der XML-Datei durch den Reader dann das besagte Event auftritt, wird die damit verknüpfte Callback-Funktion aufgerufen und somit der Code ausgeführt, den der Programmierer für diesen Zweck vorgesehen hat. Ein Event könnte beispielsweise das Auffinden eines öffnenden Tags sein.

Man könnte also sagen, dass der SAX-Reader nur die Infrastruktur zum Einlesen der XML-Datei bereitstellt. Ob und in welcher Form die gelesenen Daten aufbereitet werden, entscheidet allein der Programmierer. Damit bietet SAX wesentlich mehr Flexibilität als DOM, auf Kosten eines mitunter höheren Aufwandes selbstverständlich.

Beispiel

Die Verwendung von SAX wollen wir direkt an einem einfachen Beispiel zeigen. Dazu dient uns das bereits bekannte Szenario: Ein Python-Dictionary wurde in einer XML-Datei abgespeichert und soll durch das Programm eingelesen und wieder in ein Dictionary verwandelt werden. Die Daten liegen im folgenden Format vor:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
  <eintrag>
    <schluessel typ="str">Hallo</schluessel>
    <wert typ="int">0</wert>
  </eintrag>
</dictionary>
```

Zum Einlesen dieser Datei dient folgendes Programm, das einen SAX-Reader verwendet:

```
import xml.sax as sax

class DictHandler(sax.handler.ContentHandler):

    def __init__(self):
        self.ergebnis = {}
        self.schluessel = ""
        self.wert = ""
        self.aktiv = None
        self.typ = None
```



```

def startElement(self, name, attrs):
    if name == "eintrag":
        self.schluessel = ""
        self.wert = ""
    elif name == "schluessel" or name == "wert":
        self.aktiv = name
        self.typ = eval(attrs["typ"])

def endElement(self, name):
    if name == "eintrag":
        self.ergebnis[self.schluessel] = self.typ(self.wert)
    elif name == "schluessel" or name == "wert":
        self.aktiv = None

def characters(self, content):
    if self.aktiv == "schluessel":
        self.schluessel += content
    elif self.aktiv == "wert":
        self.wert += content

```

Zunächst wird die Klasse `DictHandler` angelegt, in der wir alle interessanten Callback-Funktionen, auch *Callback Handler* genannt, in Form von Methoden implementieren. Die Klasse muss von der Basisklasse `sax.handler.ContentHandler` abgeleitet werden.

Ein Nachteil des SAX-Modells ist es, dass wir nach jedem Schritt den aktuellen Status speichern müssen, damit beim nächsten Aufruf einer der Callback-Funktionen klar ist, ob der eingelesene Text beispielsweise innerhalb eines `schluessel`- oder eines `wert`-Tags gelesen wurde. Aus diesem Grund legen wir im Konstruktor der Klasse einige Attribute an:

- ▶ `self.ergebnis` für das resultierende Dictionary
- ▶ `self.schluessel` für den Inhalt des aktuell bearbeiteten Schlüssels
- ▶ `self.wert` für den Inhalt des aktuell bearbeiteten Wertes
- ▶ `self.aktiv` für den Tag-Namen des Tags, das zuletzt eingelesen wurde
- ▶ `self.typ` für den Datentyp, der im Attribut `typ` eines `schluessel`- oder `wert`-Tags steht

Zuerst implementieren wir die Methode `startElement`, die immer dann aufgerufen wird, wenn ein öffnendes Tag eingelesen wurde. Die Methode bekommt den Tag-Namen und die enthaltenen Attribute als Parameter übergeben. In dieser Methode wird zunächst ausgelesen, um welches öffnende Tag es sich handelt. Im Falle eines `schluessel`- oder `wert`-Tags wird `self.name` entsprechend angepasst und das Attribut `typ` des Tags ausgelesen.

Die Methode `endElement` wird aufgerufen, wenn ein schließendes Tag eingelesen wurde. Auch ihr übergeben wir den Tag-Namen als Parameter. Im Falle eines schließenden `eintrag`-Tags fügen wir das eingelesene Schlüssel-Wert-Paar, das aus `self.schluesssel` und `self.wert` besteht, in das Dictionary `self.ergebnis` ein. Wenn ein schließendes `schluesssel`- oder `wert`-Tag gefunden wurde, wird das Attribut `self.aktiv` wieder auf `None` gesetzt, so dass keine weiteren Zeichen mehr verarbeitet werden.

Die letzte Methode `characters` wird aufgerufen, wenn Zeichen eingelesen wurden, die nicht zu einem Tag gehören. Beachten Sie, dass der SAX-Reader nicht garantiert, dass eine zusammenhängende Zeichenfolge auch in einem einzelnen Aufruf von `characters` resultiert. Je nachdem, welchen Namen das zuletzt eingelesene Tag hatte, werden die gelesenen Zeichen an `self.schluesssel` oder `self.wert` angehängt.

Schlussendlich fehlt noch die Hauptfunktion `lade_dict` des Beispielprogramms, in der der SAX-Parser erzeugt und gestartet wird.

```
def lade_dict(dateiname):
    handler = DictHandler()
    parser = sax.make_parser()
    parser.setContentHandler(handler)
    parser.parse(dateiname)
    return handler.ergebnis
```

Im Funktionskörper wird die Klasse `DictHandler` instantiiert und durch die Funktion `make_parser` des Moduls `xml.sax` ein SAX-Parser erzeugt. Dann wird die Methode `setContentHandler` des Parsers aufgerufen, um die `DictHandler`-Instanz mit den enthaltenen Callback-Handlern anzumelden. Zum Schluss wird der Parsing-Prozess durch Aufruf der Methode `parse` eingeleitet.

Die Klasse `ContentHandler`

Die Klasse `ContentHandler` dient als Basisklasse und implementiert alle SAX-Callback-Handler als Methoden. Um einen SAX-Parser einsetzen zu können, muss eine eigene Klasse erstellt werden, die von `ContentHandler` erbt und die benötigten Callback-Handler überschreibt. Eine Instanz einer von `ContentHandler` abgeleiteten Klasse wird von der Methode `setContentHandler` des SAX-Parsers erwartet.

Es folgt eine Auflistung der wichtigsten Callback-Handler, die in einer von `ContentHandler` abgeleiteten Klasse überschrieben werden können. Dabei sei `c` eine Instanz der Klasse `ContentHandler`.

c.startDocument()

Wird einmalig aufgerufen, wenn der SAX-Parser damit beginnt, ein XML-Dokument einzulesen.

c.endDocument()

Wird einmalig aufgerufen, wenn der SAX-Parser ein XML-Dokument vollständig eingelesen hat.

c.startElement(name, attrs)

Wird aufgerufen, wenn ein öffnendes Tag eingelesen wurde. Die Methode bekommt weitere Informationen über das Tag in Form von zwei Parametern übergeben: den Tag-Namen (*name*) und die im Tag angegebenen Attribute (*attrs*) als *Attributes*-Instanz. Auf eine solche Instanz kann wie auf ein Dictionary zugegriffen werden, um einzelne Attribute abzufragen.

c.endElement(name)

Wird aufgerufen, wenn ein schließendes Tag mit dem Tag-Namen *name* eingelesen wurde.

c.characters(content)

Wird aufgerufen, wenn ein Textabschnitt eingelesen wurde. Beachten Sie, dass es dem SAX-Parser freisteht, den gesamten Textabschnitt in einem Event zu verarbeiten oder auf mehrere Events aufzuteilen.

Über den Parameter *content* greifen Sie auf den gelesenen Text zu.

c.ignorableWhitespace(whitespace)

Wird aufgerufen, wenn Whitespace-Zeichen eingelesen wurden. Diese könnten von Bedeutung sein, sind jedoch in den meisten Fällen allein aus Gründen der Formatierung vorhanden und können ignoriert werden. Beachten Sie, dass der SAX-Parser auch hier eine Folge von mehreren Whitespace-Zeichen auf mehrere Events aufteilen kann.

Über den Parameter *whitespace* greifen Sie auf die gelesenen Zeichen zu.

So viel zur DOM- und SAX-Implementierung in Python. Diese Abschnitte sollten nicht als DOM- bzw. SAX-Referenz verstanden werden, sondern als projektorientierte Einführung in die Thematik. Bedenken Sie, dass XML, aber auch DOM und SAX standardisiert sind bzw. De-facto-Standards darstellen. Es existieren DOM- und SAX-Implementierungen für fast jede nennenswerte Programmiersprache, und dementsprechend einfach sollte es sein, weitere Informationen zu diesen Themen zu finden.

Im nun folgenden Abschnitt möchten wir uns einer dritten Herangehensweise an XML widmen: `ElementTree`.

19.2.3 `ElementTree`

Seit Python 2.5 ist im Modul `xml.etree.ElementTree` der Standardbibliothek der Datentyp `ElementTree` enthalten, der in einer gewissen Konkurrenz zu DOM steht. Der Datentyp `ElementTree` speichert ein XML-Dokument und stellt außerordentlich komfortable Möglichkeiten zur Verfügung, sich in diesem Dokument zu bewegen und Daten auszulesen. Im Gegensatz zu DOM ist `ElementTree` nicht für mehrere Sprachen verfügbar oder gar standardisiert, weswegen es spezielle Sprachfeatures von Python, beispielsweise Iteratoren, nutzen kann und sich somit perfekt in die Sprache Python integriert.

Auch eine `ElementTree`-Instanz kann, ähnlich wie bei DOM, als Baum betrachtet werden. Dieser Baum besteht aus Instanzen der Klasse `Element`, die jeweils ein Tag repräsentieren. Attribute und Textinhalt der Tags werden ebenfalls in der jeweiligen `Element`-Instanz gespeichert.

Im Folgenden sollen zunächst die im Modul `xml.etree.ElementTree` enthaltenen Funktionen und danach die Klassen `ElementTree` und `Element` erläutert werden.

Der Inhalt des Moduls `ElementTree`

In diesem Abschnitt besprechen wir die wichtigsten Funktionen, die im Modul `xml.etree.ElementTree` enthalten sind. Mit diesen Funktionen ist es beispielsweise möglich, eine XML-Datei einzulesen und zu einer `ElementTree`-Instanz aufzubereiten.

`ElementTree.parse(source[, parser])`

Liest die XML-Datei *source* ein und gibt die aufbereiteten XML-Daten in Form einer `ElementTree`-Instanz zurück. Für den Parameter *source* kann sowohl ein Dateiname als auch ein geöffnetes Dateiobjekt übergeben werden. Durch Angabe des optionalen Parameters *parser* können Sie einen eigenen XML-Parser verwenden. Ein solcher Parser muss von der Klasse `TreeBuilder` abgeleitet werden, was wir an dieser Stelle nicht näher erläutern möchten. Der Standardparser ist die Klasse `XMLTreeBuilder`.

`ElementTree.tostring(element[, encoding])`

Schreibt die `Element`-Instanz *element* mit all ihren Unterelementen als XML in einen String und gibt diesen zurück. Durch den optionalen Parameter *encoding* kann das Encoding des resultierenden Strings festgelegt werden.

Die Klasse `ElementTree`

Die Klasse `ElementTree` repräsentiert ein XML-Dokument und enthält damit den vollständigen Baum, der daraus aufgebaut worden ist. Eine Instanz der Klasse `ElementTree` stellt die folgenden Methoden bereit; im Folgenden sei *et* eine Instanz der Klasse `ElementTree`:

`et.getiterator([tag])`

Die Methode `getiterator` gibt einen Iterator zurück, der alle Elemente des Baums inklusive des Wurzelements durchläuft. Die Elemente werden dabei in der Reihenfolge durchlaufen, in der ihre öffnenden Tags in der XML-Datei vorkommen. Wenn der optionale Parameter *tag* angegeben wurde, durchläuft der zurückgegebene Iterator alle Elemente des Baums mit dem Tag-Namen *tag*.

`et.getroot()`

Gibt die `Element`-Instanz des Wurzelements zurück.

`et.write(file[, encoding])`

Speichert den vollständigen Baum als XML-Datei *file* ab. Dabei können Sie für *file* sowohl einen Dateinamen als auch ein zum Schreiben geöffnetes Dateiojekt übergeben. Über den optionalen Parameter *encoding* legen Sie das Encoding der geschriebenen Daten fest.

Die Klasse `Element`

Die Klasse `Element` repräsentiert ein Tag des XML-Dokuments im `ElementTree`-Baum. Dafür kann eine `Element`-Instanz über beliebig viele Kindelemente verfügen.

Die Klasse `Element` erbt alle Eigenschaften einer Liste. Es ist also möglich, wie bei einer Liste auf Kindelemente mit ihrem Index zuzugreifen. Außerdem können insbesondere die Methoden `append`, `insert`, `items`, `keys` und `remove` einer Liste verwendet werden. Im Folgenden sei *e* eine Instanz der Klasse `Element`.

`e.clear()`

Löscht alle Unterelemente und Attribute sowie den Text des Elements *e*.

`e.find(path)`

Gibt das erste direkte Kindelement von *e* mit dem Tag-Namen *path* zurück. Statt eines einzelnen Tag-Namens können Sie für *path*, wie der Name bereits andeutet, auch einen Pfad übergeben. So gäbe ein Aufruf von `find` mit einem Parameter *path* von `"element1/element2"` das erste Kindelement namens `element2` des ers-

ten direkten Kindelements mit dem Tag-Namen `element1` zurück. Auch das Wildcard-Zeichen `*` kann verwendet werden, um einen beliebigen Tag-Namen zu kennzeichnen.

`e.findall(path)`

Wie `find`, gibt aber eine Liste aller passenden `Element`-Instanzen zurück statt nur des zuerst gefundenen Elements.

`e.findtext(path[, default])`

Wenn für `path` ein leerer String übergeben wird, gibt die Methode `findtext` den Text als String zurück, den `e` enthält. Ansonsten kann der Parameter `path` wie bei den Methoden `find` und `findall` verwendet werden. Wenn eine `Element`-Instanz keinen Text enthält, wird `None` zurückgegeben. Sollte dies nicht Ihren Wünschen entsprechen, können Sie über den Parameter `default` festlegen, was in diesen Fällen stattdessen zurückgegeben werden soll.

Beachten Sie, dass auch Whitespace-Zeichen wie beispielsweise ein Zeilenumbruch zum Text einer `Element`-Instanz zählen.

`e.get(key[, default])`

Mithilfe der Methode `get` greifen Sie auf den Wert des Attributs `key` der `Element`-Instanz `e` zu. Wenn kein Attribut mit dem Schlüssel `key` vorhanden ist, wird `default` zurückgegeben. Der Parameter `default` ist mit `None` vorbelegt.

`e.getchildren()`

Gibt eine Liste aller Kindelemente zurück.

`e.getiterator([tag])`

Die Methode `getiterator` hat die gleiche Bedeutung wie die gleichnamige Methode der Klasse `ElementTree`, allerdings nur für alle Kindelemente von `e`.

`e.set(key, value)`

Durch Aufruf der Methode `set` wird ein neues Attribut mit dem Schlüssel `key` und dem Wert `value` im Element `e` angelegt.

Neben den soeben besprochenen Methoden verfügen alle `Element`-Instanzen über die folgenden Attribute:

`e.attrib`

Das Attribut `attrib` referenziert ein Dictionary, das alle in der `Element`-Instanz `e` vorhandenen XML-Attribute als Schlüssel-Wert-Paare enthält.

e.tag

Das Attribut `tag` enthält den Tag-Namen des Elements *e*.

e.text

Das Attribut `text` enthält den Text, der in der XML-Datei zwischen dem öffnenden Tag der Element-Instanz *e* und dem öffnenden Tag des ersten Kindelements steht. Wenn kein Kindelement existiert, enthält das Attribut `text` den vollständigen im Körper des Tags enthaltenen Text.

e.tail

Das Attribut `tail` enthält den Text, der in der XML-Datei zwischen dem schließenden Tag der Element-Instanz *e* und dem nächsten öffnenden oder schließenden Tag steht.

Beispiel

Als Beispiel für die Verwendung des Datentyps `ElementTree` soll das Beispielprogramm der vorherigen Abschnitte an diesen Datentyp angepasst werden und somit seine Stärken demonstrieren. Das Programm soll eine XML-Datei des folgenden Formats einlesen und zu einem Dictionary aufbereiten:

```
<?xml version="1.0" encoding="UTF-8"?>
<dictionary>
  <eintrag>
    <schluessel typ="str">Hallo</schluessel>
    <wert typ="int">0</wert>
  </eintrag>
</dictionary>
```

Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
import xml.etree.ElementTree as ElementTree

def lese_text(element):
    typ = element.get("typ", "str")
    return eval("{0}('{1}').format(typ, element.text))

def lade_dict(dateiname):
    d = {}
    baum = ElementTree.parse(dateiname)
    tag_dict = baum.getroot()
    for eintrag in tag_dict.getchildren():
        tag_schluessel = eintrag.find("schluessel")
        tag_wert = eintrag.find("wert")
        d[lese_text(tag_schluessel)] = lese_text(tag_wert)
    return d
```

Zunächst wird die Funktion `lese_text` implementiert, die aus der `Element`-Instanz eines `schluessel`- oder `wert`-Tags das Attribut `typ` ausliest und den vom jeweiligen Tag umschlossenen Text in den durch `typ` angegebenen Datentyp konvertiert. Dazu wird die Built-in Function `eval` wie bei den Beispielen der vorherigen Kapitel verwendet. Der Inhalt des Tags wird dann als Instanz des passenden Datentyps zurückgegeben.

Die Hauptfunktion des Beispielprogramms `lade_dict` bekommt den Dateinamen einer XML-Datei übergeben und soll die darin enthaltenen Daten zu einem Python-Dictionary aufbereiten. Dazu wird die XML-Datei zunächst mithilfe der Funktion `parse` des Moduls `xml.etree.ElementTree` zu einem Baum aufbereitet. Danach wird der Referenz `tag_dict` das Wurzelement des Baums zugewiesen, um auf diesem weiter zu operieren.

Die nun folgende Schleife iteriert über alle Kindelemente des Wurzelements, also über alle `eintrag`-Tags. In jedem Iterationsschritt werden die ersten Kindelemente mit den Tag-Namen `schluessel` und `wert` gesucht und den Referenzen `tag_schluessel` und `tag_wert` zugewiesen. Am Ende des Schleifenkörpers werden die `Element`-Instanzen der jeweiligen `schluessel`- oder `wert`-Tags durch die Funktion `lese_text` geschleust, was den im Tagkörper enthaltenen Text in eine Instanz des korrekten Datentyps konvertiert. Die resultierenden Instanzen werden als Schlüssel bzw. als Wert in das Dictionary `d` eingetragen. Schlussendlich wird das erzeugte Dictionary `d` zurückgegeben.

So viel zum Datentyp `ElementTree`. Wir beschäftigen uns weiterhin mit Datenspeicherung bei und werden uns im nächsten Abschnitt um das Thema Datenbanken kümmern.

19.3 Datenbanken

Je mehr Daten ein Programm verwalten muss und je komplexer die Struktur dieser Daten wird, desto größer wird der programmtechnische Aufwand für die dauerhafte Speicherung und Verwaltung der Daten. Außerdem gibt es eine ganze Reihe von Aufgaben wie das Lesen, Schreiben oder Aktualisieren, die in fast jedem Programm gebraucht werden, aber immer wieder neu implementiert werden müssten.

Abhilfe für diese Problematik wird geschaffen, indem man eine Abstraktionsschicht zwischen dem benutzenden Programm und dem physikalischen Massenspeicher einzieht, die sogenannte *Datenbank*. Dabei erfolgt die Kommunikation zwischen Benutzerprogramm und Datenbank über eine vereinheitlichte Schnittstelle.

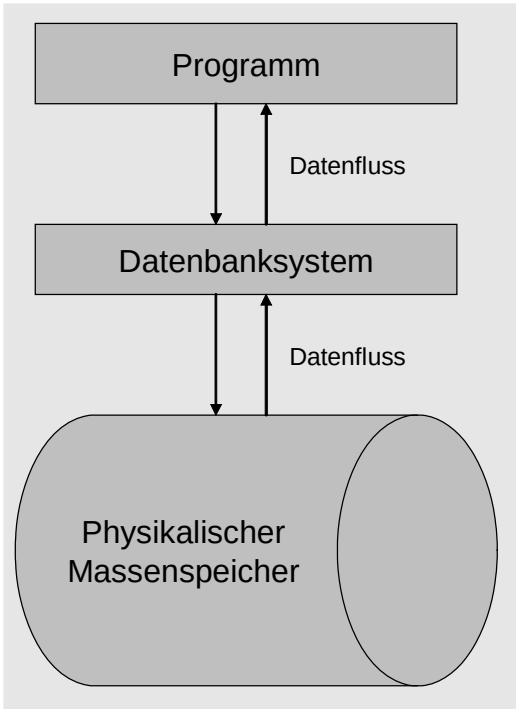


Abbildung 19.3 Die Datenbankschnittstelle

Das Datenbanksystem nimmt dabei Abfragen, sogenannte *Queries*, entgegen und gibt alle Datensätze zurück, die den Bedingungen der Abfragen genügen.

Wir werden uns in diesem Kapitel ausschließlich mit sogenannten *relationalen Datenbanken* beschäftigen, die einen Datenbestand in Tabellen organisieren.² Für die Abfragen in relationalen Datenbanken wurde eine eigene Sprache entwickelt, deren Name *SQL* (*Structured Query Language*, dt. »strukturierte Abfragesprache«) ist. SQL ist zu komplex, um es in diesem Kapitel erschöpfend zu beschreiben. Wir werden hier nur auf grundlegende SQL-Befehle eingehen, die nötig sind, um das Prinzip von Datenbanken und deren Anwendung in Python zu verdeutlichen.

SQL ist standardisiert und wird von eigentlich allen Datenbanksystemen unterstützt. Dabei ist zu beachten, dass die Systeme immer nur Teilmengen der Sprache implementieren und teilweise geringfügig abändern. Aus diesem Grund wer-

² Der Attribut »relational« geht auf den Begriff der *Relation* aus der Mathematik zurück. Vereinfacht gesagt ist eine Relation eine Zuordnung von Elementen zweier oder mehrerer Mengen.

den wir Sie hier in das SQL einführen, das von SQLite, der Standarddatenbank in Python, genutzt wird.

Neben der Abfragesprache SQL ist in Python auch die Schnittstelle der Datenbankmodule standardisiert. Dies hat für den Programmierer den angenehmen Nebeneffekt, dass sein Code mit minimalen Anpassungen auf allen Datenbanksystemen lauffähig ist, die diesen Standard implementieren. Die genaue Definition dieser sogenannten *Python Database API Specification* können Sie im Internet unter der Adresse <http://www.python.org/dev/peps/pep-0249/> nachlesen.

Bevor wir uns aber eingehend mit der Abfragesprache SQL selbst beschäftigen, werden wir eine kleine Beispieldatenbank erarbeiten und uns überlegen, welche Operationen man überhaupt ausführen könnte. Anschließend werden wir dieses Beispiel mithilfe von SQLite implementieren und dabei auf Teile der Abfragesprache SQL und die Verwendung in Python-Programmen eingehen.

Stellen wir uns vor, wir müssten das Lager eines Computerversands verwalten. Wir sind dafür verantwortlich, dass die gelieferten Teile an der richtigen Stelle im Lager aufbewahrt werden, wobei für jede Komponente der Lieferant, der Lieferzeitpunkt und die Nummer des Fachs im Lager gespeichert werden sollen. Für Kunden, die bei dem Versand ihre Rechner bestellen, werden die entsprechenden Teile reserviert, und diese sind dann für andere Kunden nicht mehr verfügbar. Außerdem sollen wir Listen mit allen Kunden und Lieferanten der Firma bereitstellen.

Um ein Datenbankmodell für dieses Szenario zu erstellen, legen wir zuerst eine Tabelle namens »Lager« an, die alle im Lager befindlichen Komponenten enthält. Wir gehen der Einfachheit halber davon aus, dass unser Lager in 100 Fächer eingeteilt ist, die fortlaufend nummeriert sind. Dabei kann jedes Fach nur ein einzelnes Computerteil aufnehmen. Eine entsprechende Tabelle mit ein paar Beispieldatensätzen für das Lager könnte dann wie folgt aussehen, wenn wir zusätzlich den Lieferanten und den Reservierungsstatus speichern wollen:

Fachnummer	Seriennummer	Komponente	Lieferant	Reserviert
1	26071987	Grafikkarte Typ 1	FC	0
2	19870109	Prozessor Typ 13	LPE	57
10	06198823	Netzteil Typ 3	FC	0
25	11198703	LED-Lüfter	FC	57
26	19880105	Festplatte 128 GB	LPE	12

Tabelle 19.3 Tabelle »Lager« für den Lagerbestand

Die Spalte »Lieferant« enthält dabei das Kürzel der liefernden Firma, und das Feld »Reserviert« ist auf »0« gesetzt, wenn der betreffende Artikel noch nicht von einem Kunden reserviert wurde. Ansonsten enthält das Feld die Kundennummer des reservierenden Kunden. In der Tabelle werden nur die belegten Fächer gespeichert, weshalb alle Fächer, für die kein Eintrag existiert, mit neuen Teilen gefüllt werden können.

Die ausführlichen Informationen zu Lieferanten und Kunden werden in zwei weiteren Tabellen namens »Lieferanten« und »Kunden« abgelegt:

Kurzname	Name	Telefonnummer
FC	FiboComputing Inc.	011235813
LPE	LettgenPetersErnesti	026741337
GC	Golden Computers	016180339

Tabelle 19.4 Tabelle »Lieferanten«

Kundennummer	Name	Anschrift
12	Heinz Elhurg	Turnhallenstr. 1, 3763 Sporthausen
57	Markus Altbert	Kämperweg 24, 2463 Duisschloss
64	Steve Apple	Podmacstr. 2, 7467 Iwarhausen

Tabelle 19.5 Tabelle »Kunden«

Damit wir als Lagerverwalter von dieser Datenbank profitieren können, müssen wir die Möglichkeit haben, den Datenbestand zu manipulieren. Wir brauchen Routinen, um neue Kunden und Lieferanten hinzuzufügen, ihre Daten beispielsweise bei einem Umzug zu aktualisieren oder sie auf Wunsch aus unserer Datenbank zu entfernen. Auch in die Tabelle »Lager« müssen wir neue Einträge einfügen und alte löschen oder anpassen. Um die Datenbank aktuell zu halten, benötigen wir also Funktionen zum *Hinzufügen* und *Löschen*.

Wirklich nützlich wird die Datenbank aber erst, wenn wir die enthaltenen Daten nach bestimmten Kriterien abfragen können. Im einfachsten Fall könnten wir beispielsweise einfach nur eine Liste aller Kunden oder Lieferanten anfordern oder uns informieren wollen, welche Fächer zurzeit belegt sind. Uns könnte aber auch interessieren, ob der Kunde mit dem Namen »Markus Altbert« Artikel reserviert hat und wenn ja, welche Artikel dies sind und wo diese gelagert werden; oder wir möchten wissen, welche Komponenten wir von dem Lieferanten mit der Telefonnummer »011235813« nachbestellen müssen, weil sie nicht mehr vorhanden oder bereits reserviert sind. Bei diesen Operationen werden immer Da-

tensätze nach bestimmten Kriterien *ausgewählt* und an das aufrufende Benutzerprogramm zurückgegeben.

Nach dieser theoretischen Vorbereitung werden wir uns der Implementation des Beispiels in einer SQLite-Datenbank zuwenden.

19.3.1 Pythons eingebaute Datenbank – sqlite3

SQLite ist ein sehr einfaches Datenbanksystem, das seine Daten in normalen Dateien abspeichert. Trotzdem ist es extrem schnell und auch für verhältnismäßig große Datenmengen geeignet.

In Python müssen Sie das Modul `sqlite3` importieren, um mit der Datenbank zu arbeiten. Anschließend können Sie eine Verbindung zu der Datenbank aufbauen, indem Sie die `connect`-Funktion, die ein `Connection`-Objekt zu der Datenbank zurückgibt, aufrufen und ihr den Dateinamen für die Datenbank übergeben:

```
import sqlite3
connection = sqlite3.connect("lagerverwaltung.db")
```

Die Dateiendung kann frei gewählt werden und hat keinerlei Einfluss auf die Funktionsweise der Datenbank. Obiger Code führt dazu, dass die Datenbank, die in der Datei *lagerverwaltung.db* im selben Verzeichnis wie das Programm liegt, eingelesen und mit dem `Connection`-Objekt `connection` verbunden wird. Wenn es noch keine Datei mit dem Namen *lagerverwaltung.db* gibt, so wird eine leere Datenbank erzeugt und die Datei angelegt.

Oft benötigt man eine Datenbank nur während des Programmlaufs, um Daten zu verwalten oder zu ordnen, ohne dass diese dauerhaft auf der Festplatte gespeichert werden müssen. Zu diesem Zweck gibt es die Möglichkeit, eine Datenbank im Arbeitsspeicher zu erzeugen, indem Sie statt eines Dateinamens den String `":memory:"` an die `connect`-Methode übergeben:

```
>>> connection = sqlite3.connect(":memory:")
```

Um mit der verbundenen Datenbank arbeiten zu können, werden sogenannte *Cursor* (dt. »Positionsanzeigen«) benötigt. Einen Cursor kann man sich ähnlich wie den blinkenden Strich in Textverarbeitungsprogrammen als aktuelle Bearbeitungsposition innerhalb der Datenbank vorstellen. Erst mit solchen Cursorn können wir Datensätze verändern oder abfragen, wobei es zu einer Datenbankverbindung beliebig viele Cursor geben kann. Ein neuer Cursor wird mithilfe der `cursor`-Methode des `Connection`-Objekts erzeugt:

```
cursor = connection.cursor()
```

Neue Tabellen anlegen

Nun können wir endlich unser erstes SQL-Statement an die Datenbank schicken, um unsere Tabellen anzulegen. Für das Anlegen unserer Tabelle »Lager« sähe das SQL-Statement folgendermaßen aus:

```
CREATE TABLE lager (  
    fachnummer INTEGER, seriennummer INTEGER, komponente TEXT,  
    lieferant TEXT, reserviert INTEGER  
)
```

Alle großgeschriebenen Wörter sind Bestandteile der Sprache SQL. Allerdings unterscheidet SQL nicht zwischen Groß- und Kleinschreibung, und deshalb hätten wir auch alles kleinschreiben können. Wegen der besseren Lesbarkeit werden wir SQL-Schlüsselwörter immer komplett groß- und von uns vergebene Namen durchgängig kleinschreiben.

Die Zeichenketten `INTEGER` und `TEXT` hinter den Spaltennamen geben den Datentyp an, der in den Spalten gespeichert werden soll. Sinnvollerweise werden die Spalten `fachnummer`, `seriennummer` und `reserviert` als Ganzzahlen und die Spalten `komponente` und `lieferant` als Zeichenketten definiert. SQLite kennt mehrere solcher Datentypen, in die Python-Datentypen beim Schreiben der Datenbank automatisch umgewandelt werden, wie es die folgende Tabelle zeigt:

Python-Datentyp	SQLite-Datentyp
None	NULL
int	INTEGER
float	REAL
bytes (UTF8-Kodiert)	TEXT
str	TEXT
bytes	BLOB

Tabelle 19.6 So konvertiert SQLite beim Schreiben der Daten.

Es ist auch möglich, andere Datentypen in SQLite-Datenbanken abzulegen, wenn entsprechende Konvertierungsfunktionen definiert wurden. Wie das genau erreicht werden kann, wird später behandelt.

Nun senden wir das SQL-Statement mithilfe der `execute`-Methode des `Cursor`-Objekts an die SQLite-Datenbank:

```
cursor.execute("""CREATE TABLE lager (  
    fachnummer INTEGER, seriennummer INTEGER,  
    komponente TEXT, lieferant TEXT, reserviert INTEGER)""")
```

Die Tabellen für die Lieferanten und Kunden erzeugen wir auf die gleiche Weise:

```
cursor.execute("""CREATE TABLE lieferanten (
    kurzname TEXT, name TEXT, telefonnummer TEXT)""")

cursor.execute("""CREATE TABLE kunden (
    kundennummer INTEGER, name TEXT, anschrift TEXT)""")
```

Daten in die Tabellen einfügen

Als Nächstes werden wir die noch leeren Tabellen mit unseren Beispieldaten füllen. Zum Einfügen neuer Datensätze in eine bestehende Tabelle dient das `INSERT`-Statement, das für den ersten Beispieldatensatz folgendermaßen aussieht:

```
INSERT INTO lager VALUES (
    1, 26071987, 'Grafikkarte Typ 1', 'FC', 0
)
```

Innerhalb der Klammern hinter `VALUES` stehen die Werte für jede einzelne Spalte in der gleichen Reihenfolge, wie auch die Spalten selbst definiert wurden. Wie bei allen anderen Datenbankabfragen auch können wir per `execute` unser Statement abschicken:

```
cursor.execute("""INSERT INTO lager VALUES (
    1, 26071987, 'Grafikkarte Typ 1', 'FC', 0)""")
```

Beim Einfügen von Datensätzen müssen Sie allerdings beachten, dass die neuen Daten nicht sofort nach Ausführen eines `INSERT`-Statements in die Datenbank `daten` geschrieben werden, sondern vorerst nur im Arbeitsspeicher liegen. Um sicherzugehen, dass die Daten wirklich auf der Festplatte landen und damit dauerhaft gespeichert sind, muss man die `commit`-Methode des `Connection`-Objekts aufrufen.³

```
connection.commit()
```

In der Regel werden die Daten, die wir in die Datenbank einfügen wollen, nicht schon vor dem Programmlauf bekannt sein und deshalb auch nicht in Form von String-Konstanten im Quellcode stehen. Stattdessen werden es Benutzereingaben oder Berechnungsergebnisse sein, die wir dann als Python-Instanzen im Speicher

³ Dies ist deshalb notwendig, damit die Datenbank transaktionssicher ist. Transaktionen sind Ketten von Operationen, die vollständig ausgeführt werden müssen, damit die Konsistenz der Datenbank erhalten bleibt. Stellen Sie sich einmal vor, bei einer Bank würde während einer Überweisung zwar das Geld von Ihrem Konto abgebucht, jedoch aufgrund eines Fehlers nicht dem Empfänger gutgeschrieben. Mit der Methode `rollback` können alle Operationen seit dem letzten `commit`-Aufruf wieder rückgängig gemacht werden, um solche Probleme zu vermeiden.

haben. Auf den ersten Blick scheint für solche Fälle die Formatierungsmethode `format` für Strings ein geeignetes Mittel zu sein, und die letzte `INSERT`-Anweisung hätte auch folgendermaßen zusammengebaut werden können:

```
>>> werte = (1, 26071987, "Grafikkarte Typ 1", "FC", 0)
>>> "INSERT INTO lager VALUES ({0}, {1}, '{2}', '{3}', {4})".format(
    *werte)
'INSERT INTO lager VALUES (1, 26071987, 'Grafikkarte Typ 1', 'FC', 0)'
```

Diese auf den ersten Blick sehr elegante Methode entpuppt sich bei genauer Betrachtung aber als gefährliche Sicherheitslücke. Betrachten wir einmal folgende `INSERT`-Anweisung, die einen neuen Lieferanten in die Tabelle »Lieferanten« einfügen soll:

```
>>> werte = ("DR", "Danger Electronics",
    "666"); Hier kann Schadcode stehen")
>>> "INSERT INTO lieferanten VALUES ('{0}', '{1}', '{2}')".format(
    *werte)
'INSERT INTO lager VALUES ('DR', 'Danger Electronics', '666'); Hier
kann Schadcode stehen')'
```

Wie Sie sehen, haben wir dadurch, dass der Wert für die Telefonnummer den String `''');"` enthält, die `SQL`-Abfrage verunstaltet, so dass der Versuch, sie auszuführen, zu einem Fehler führen und damit unser Programm zum Absturz bringen würde. Durch den außerdem enthaltenen Text `"Hier kann Schadcode stehen"` haben wir angedeutet, dass es unter Umständen sogar möglich ist, eine Abfrage so zu manipulieren, dass wieder gültiger `SQL`-Code dabei herauskommt, wobei jedoch eine andere Operation als beabsichtigt (zum Beispiel das Auslesen von Benutzerdaten) ausgeführt wird.⁴

Verwenden Sie deshalb niemals die String-Formatierung zur Übergabe von Parametern in `SQL`-Abfragen!

Um sichere Parameterübergaben durchzuführen, schreibt man in den Query-String an die Stelle, an der der Parameter stehen soll, ein Fragezeichen und übergibt der `execute`-Methode ein Tupel mit den entsprechenden Werten als zweiten Parameter:

```
werte = ("DR", "Danger Electronics",
    "666"); Hier kann Schadcode stehen")
sql = "INSERT INTO lieferanten VALUES (?, ?, ?)"
cursor.execute(sql, werte)
```

4 Man nennt diese Form des Angriffs auf verwundbare Programme auch *SQL Injection*.

In diesem Fall kümmert sich SQLite darum, dass die übergebenen Werte korrekt umgewandelt werden und es zu keinen Sicherheitslücken durch böswillige Parameter kommen kann.

Analog zur String-Formatierung gibt es auch hier die Möglichkeit, den übergebenen Parametern Namen zu geben und statt der `tuple`-Instanz mit einem `Dictionary` zu arbeiten. Dazu schreiben Sie im Query-String statt des Fragezeichens einen Doppelpunkt, gefolgt von dem symbolischen Namen des Parameters, und übergeben das passende `Dictionary` als zweiten Parameter an `execute`:

```
werte = {"kurz" : "DR", "name" : "Danger Electronics",
        "telefon" : "123456"}
sql = "INSERT INTO lieferanten VALUES (:kurz, :name, :telefon)"
cursor.execute(sql, werte)
```

Mit diesem Wissen können wir unsere Tabellen elegant und sicher mit Daten füllen:

```
for row in ((1, "2607871987", "Grafikkarte Typ 1", "FC", 0),
            (2, "19870109", "Prozessor Typ 13", "LPE", 57),
            (10, "06198823", "Netzteil Typ 3", "FC", 0),
            (25, "11198703", "LED-Lüfter", "FC", 57),
            (26, "19880105", "Festplatte 128 GB", "LPE", 12)):
    cursor.execute("INSERT INTO lager VALUES (?, ?, ?, ?, ?)", row)
```

Im Gegensatz zu früheren Versionen von Python nimmt Ihnen Python 3.0 einen Großteil der lästigen Arbeit mit Stringkodierungen ab. Deshalb ist es auch problemlos möglich, den String "LED-Lüfter" ohne Sonderbehandlung zu übergeben, obwohl er einen deutschen Umlaut enthält.⁵

Strukturen wie die obige `for`-Schleife, die die gleiche Datenbankoperation sehr oft für jeweils andere Daten durchführen, kommen häufig vor und bieten großes Optimierungspotenzial. Aus diesem Grund haben `cursor`-Instanzen zusätzlich die Methode `executemany`, die als zweiten Parameter eine Sequenz oder ein anderes iterierbares Objekt erwartet, das die Daten für die einzelnen Operationen enthält. Wir nutzen `executemany`, um unsere Tabellen »Lieferanten« und »Kunden« mit Daten zu füllen:

```
lieferanten = (("FC", "FiboComputing Inc.", "011235813"),
               ("LPE", "LettgenPetersErnesti", "026741337"),
               ("GC", "Golden Computers", "016180339"))
cursor.executemany("INSERT INTO lieferanten VALUES (?, ?, ?)",
                  lieferanten)
```

⁵ In Python-Versionen vor 3.0 mussten Sie diesen String als eine `unicode`-Instanz übergeben und auch im Quellcode explizit als solche kennzeichnen.


```
kunden = ((12, "Heinz Elhurg",
            "Turnhallenstr. 1, 3763 Sporthausen"),
           (57, "Markus Altbart",
            "Kämperweg 24, 2463 Duisschloss"),
           (64, "Steve Apple",
            "Podmacstr 2, 7467 Iwarhausen"))
cursor.executemany("INSERT INTO kunden VALUES (?, ?, ?)", kunden)
```

Nun haben wir gelernt, wie man Datenbanken und Tabellen anlegt und diese mit Daten füllt. Im nächsten Schritt wollen wir uns mit dem Abfragen von Daten beschäftigen.

Daten abfragen

Um Daten aus der Datenbank abzufragen, dient das `SELECT`-Statement. `SELECT` erwartet als Parameter durch Kommata getrennt die Spalten, die uns von den Datensätzen interessieren, und den Tabellennamen der Tabelle, die wir abfragen wollen. Standardmäßig werden alle Zeilen aus der abgefragten Tabelle zurückgegeben. Mit einer `WHERE`-Klausel können wir nur bestimmte Datensätze auswählen, indem wir Bedingungen für die Auswahl angeben. Stark vereinfacht ist ein `SELECT`-Statement folgendermaßen aufgebaut:

```
SELECT <spaltenliste> FROM <tabellenname> [WHERE <bedingung>]
```

Wie durch die eckigen Klammern angedeutet wird, ist die `WHERE`-Klausel optional und kann entfallen.

Wenn wir beispielsweise alle belegten Fachnummern und die dazugehörigen Komponenten abfragen wollen, formulieren wir das folgende Statement:

```
SELECT fachnummer, komponente FROM lager
```

Auch bei Datenabfragen benutzen wir die `execute`-Methode des Cursor-Objekts, um der Datenbank unser Anliegen mitzuteilen. Anschließend können wir uns mit `cursor.fetchall` alle Datensätze zurückgeben lassen, die unsere Abfrage ergeben hat:

```
>>> cursor.execute("SELECT fachnummer, komponente FROM lager")
>>> cursor.fetchall()
[(1, 'Grafikkarte Typ 1'), (2, 'Prozessor Typ 13'),
 (10, 'Netzteil Typ 3'), (25, 'LED-Lüfter'),
 (26, 'Festplatte 128 GB')]
```

Der Rückgabewert von `fetchall` ist eine Liste, die für jeden Datensatz ein Tupel mit den Werten der angeforderten Spalten enthält.

Mit einer passenden `WHERE`-Klausel können wir die Auswahl auf die Computerteile beschränken, die noch nicht reserviert sind:

```
>>> cursor.execute("""
    SELECT fachnummer, komponente FROM lager WHERE reserviert=0
""")
>>> cursor.fetchall()
[(1, 'Grafikkarte Typ 1'), (10, 'Netzteil Typ 3')]
```

Wir können auch mehrere Bedingungen mittels logischer Operatoren wie `AND` und `OR` zusammenfassen. Damit ermitteln wir beispielsweise, welche Artikel, die von der Firma »FiboComputing Inc.« geliefert wurden, schon reserviert worden sind:

```
>>> cursor.execute("""
    SELECT fachnummer, komponente FROM lager
    WHERE reserviert!=0 AND lieferant='FC'
""")
>>> cursor.fetchall()
[(25, 'LED-Lüfter')]
```

Da es lästig ist, immer die auszuwählenden Spaltennamen anzugeben und man sehr oft Abfragen über alle Spalten machen möchte, gibt es dafür eine verkürzte Schreibweise, bei der die Spaltenliste durch ein Sternchen ersetzt wird:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[(12, 'Heinz Elhurg', 'Turnhallenstr. 1, 3763 Sporthausen'),
 (57, 'Markus Altbert', 'Kämperweg 24, 2463 Duisschloss'),
 (64, 'Steve Apple', 'Podmacstr 2, 7467 Iwarhausen')]
```

Die Reihenfolge der Spaltenwerte richtet sich danach, in welcher Reihenfolge die Spalten der Tabelle mit `CREATE` definiert wurden.

Als letzte Ergänzung zum `SELECT`-Statement wollen wir uns mit den Abfragen über mehrere Tabellen, den sogenannten *Joins* (dt. »Verbindungen«), beschäftigen. Wir möchten zum Beispiel abfragen, welche Komponenten des Lieferanten mit der Telefonnummer »011235813« zurzeit im Lager vorhanden sind und in welchen Fächern sie liegen.

Eine Abfrage über mehrere Tabellen unterscheidet sich von einfachen Abfragen dadurch, dass anstelle des einfachen Tabellennamens eine durch Kommata getrennte Liste angegeben wird, die alle an der Abfrage beteiligten Tabellen enthält. Wenn auf Spalten, zum Beispiel in der `WHERE`-Bedingung, verwiesen wird, muss der jeweilige Tabellename mit angegeben werden. Das gilt auch für die auszuwählenden Spalten direkt hinter `SELECT`. Unsere Beispielabfrage betrifft nur die

Tabellen »Lager« und »Lieferanten« und lässt sich als Join folgendermaßen formulieren:

```
SELECT lager.fachnummer, lager.komponente, lieferanten.name
FROM lager, lieferanten
WHERE lieferanten.telefonnummer='011235813' AND
      lager.lieferant=lieferanten.kurzname
```

Sie können sich die Verarbeitung eines solchen Joins so vorstellen, dass die Datenbank jede Zeile der Tabelle »Lager« mit jeder Zeile der Tabelle »Lieferanten« zu neuen Datensätzen verknüpft und aus der dadurch entstehenden Liste alle Zeilen zurückgibt, bei denen die Spalte `lieferanten.telefonnummer` den Wert `'011235813'` hat und die Spalten `lager.lieferant` und `lieferanten.kurzname` übereinstimmen.

Führen wir die Abfrage mit SQLite aus, erhalten wir die erwartete Ausgabe:

```
>>> sql = """
SELECT lager.fachnummer, lager.komponente, lieferanten.name
FROM lager, lieferanten
WHERE lieferanten.telefonnummer='011235813' AND
      lager.lieferant=lieferanten.kurzname"""
>>> cursor.execute(sql)
>>> cursor.fetchall()
[(1, 'Grafikkarte Typ 1', 'FiboComputing Inc.'),
 (10, 'Netzteil Typ 3', 'FiboComputing Inc.'),
 (25, 'LED-Lüfter', 'FiboComputing Inc.')]

```

Bis hierher haben wir nach einer Abfrage immer mit `cursor.fetchall` direkt alle Ergebnisse der Abfrage aus der Datenbank geladen und dann gesammelt ausgegeben. Diese Methode eignet sich allerdings nur für relativ kleine Datenmengen, da erstens das Programm so lange warten muss, bis die Datenbank alle Ergebnisse ermittelt und zurückgegeben hat, und zweitens das Resultat komplett als Liste im Speicher gehalten wird. Dass dies bei sehr umfangreichen Ergebnissen eine Verschwendung von Speicherplatz darstellt, bedarf keiner weiteren Erklärung. Aus diesem Grund gibt es die Möglichkeit, die Daten zeilenweise, also immer in kleinen Portionen, abzufragen. Wir erreichen durch dieses Vorgehen, dass wir nicht mehr auf die Berechnung der kompletten Ergebnismenge warten müssen, sondern schon währenddessen mit der Verarbeitung beginnen können. Außerdem müssen nicht mehr alle Datensätze zeitgleich im Arbeitsspeicher verfügbar sein.

Mit der Methode `fetchone` der `cursor`-Klasse fordern wir jeweils ein Ergebnis-Tupel an. Wurden bereits alle Datensätze der letzten Abfrage ausgelesen, gibt `fetchone` `None` zurück. Damit lassen sich auch große Datenmengen speichereffi-

zient auslesen, auch wenn unser Beispiel mangels einer großen Datenbank nur drei Zeilen ermittelt:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> row = cursor.fetchone()
>>> while row:
    print(row)
    row = cursor.fetchone()
(12, 'Heinz Elhurg', 'Turnhallenstr. 1, 3763 Sporthausen')
(57, 'Markus Altbert', 'Kämperweg 24, 2463 Duisschloss')
(64, 'Steve Apple', 'Podmacstr 2, 7467 Iwarhausen')
```

Diese Methode führt durch die `while`-Schleife zu etwas holprigem Code und wird deshalb seltener eingesetzt. Eine wesentlich elegantere Methode bietet die Iterator-Schnittstelle der `cursor`-Klasse, die es uns erlaubt, wie bei einer Liste mithilfe von `for` über die Ergebniszeilen zu iterieren:

```
>>> for row in cursor:
    print(row)
(12, 'Heinz Elhurg', 'Turnhallenstr. 1, 3763 Sporthausen')
(57, 'Markus Altbert', 'Kämperweg 24, 2463 Duisschloss')
(64, 'Steve Apple', 'Podmacstr 2, 7467 Iwarhausen')
```

Aufgrund des wesentlich besser lesbaren Programmtextes ist die Iterator-Methode für solche Anwendungen der Methode `fetchone` vorzuziehen. Sie sollten `fetchone` nur dann benutzen, wenn Sie gezielt jede Ergebniszeile separat und auf eine andere Weise verarbeiten wollen.

Der Umgang mit Datentypen bei SQLite

Aus dem einleitenden Teil dieses Abschnitts kennen Sie bereits das Schema, nach dem SQLite Daten beim Schreiben der Datenbank konvertiert. Die entsprechende Rückübersetzung von SQLite-Datentypen zu Python-Datentypen beschreibt folgende Tabelle:

SQLite-Datentyp	Python-Datentyp
NULL	None
INTEGER	int
REAL	float
TEXT	str
BLOB	bytes

Tabelle 19.7 Typumwandlung beim Lesen von SQLite-Datenbanken

Im Wesentlichen wirft diese Tabelle nur zwei Fragen auf: Wie werden andere Datentypen, beispielsweise Listen oder eigene Klassen, in der Datenbank gespeichert, wenn doch nur diese Typen unterstützt werden? Und wie können wir in den Rückübersetzungsprozess eingreifen, um beispielsweise alle Strings vor dem Auslesen in Großbuchstaben umzuwandeln?

Wir werden zuerst die zweite Frage beantworten.

Connection.text_factory

Jede von `sqlite3.connect` erzeugte `Connection`-Instanz hat ein Attribut `text_factory`, das eine Referenz auf eine Funktion enthält, die immer dann aufgerufen wird, wenn `TEXT`-Spalten ausgelesen werden. Im Ergebnis-Tupel der Datenbankabfrage steht dann der Rückgabewert dieser Funktion. Standardmäßig ist das `text_factory`-Attribut auf die Built-in Funktion `str` gesetzt.

```
>>> connection = sqlite3.connect("lagerverwaltung.db")
>>> connection.text_factory
<class 'str'>
```

Um unser Ziel zu erreichen, `str`-Instanzen für `TEXT`-Spalten zu erhalten, in denen alle Buchstaben groß sind, können wir eine eigene `text_factory`-Funktion angeben. Diese Funktion muss einen Parameter erwarten und den konvertierten Wert zurückgeben. Der Parameter ist ein `bytes`-String, der die Rohdaten aus der Datenbank mit UTF-8 kodiert enthält. In unserem Fall reicht also eine einfache Funktion aus, die den ausgelesenen Wert erst in einen String umwandelt und anschließend mit der `upper`-Methode alle Buchstaben zu Großbuchstaben macht:

```
>>> def my_text_factory(value):
        return str(value, "utf-8", "ignore").upper()
```

Nun müssen wir nur noch das Attribut `text_factory` unseres `Connection`-Objektes auf unsere neue Funktion setzen und können uns über das erwartete Ergebnis freuen:

```
>>> connection.text_factory = my_text_factory
>>> cursor = connection.cursor()
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[(12, 'HEINZ ELHURG', 'TURNHALLENSTR. 1, 3763 SPORTHAUSEN'),
 (57, 'MARKUS ALTBERT', 'KÄMPERWEG 24, 2463 DUISCHLOSS'),
 (64, 'STEVE APPLE', 'PODMACSTR 2, 7467 IWARHAUSEN')]
```

Es ist noch interessant zu wissen, dass `sqlite3` schon über eine alternative `text_factory`-Funktion verfügt: `sqlite3.OptimizedUnicode`. Diese erkennt automatisch, ob es sich bei dem gerade aus der Datenbank gelesenen `bytes`-

String um gültiges UTF-8 oder um binäre Daten handelt. Davon abhängig entscheidet `sqlite3.OptimizedUnicode` dann, ob ein `str`-Objekt oder ein `bytes`-String zurückgegeben werden soll. Um das Verhalten von `sqlite3.OptimizedUnicode` zu demonstrieren, legen wir eine Datenbank im Arbeitsspeicher an und erzeugen in dieser eine Tabelle »test«. Anschließend schreiben wir einen normalen String und einen UTF-16-kodierten String in die Tabelle »test«.

```
>>> connection1 = sqlite3.connect(":memory:")
>>> connection1.text_factory = sqlite3.OptimizedUnicode
>>> cursor1 = connection1.cursor()
>>> cursor1.execute("CREATE TABLE test (spalte TEXT)")
>>> cursor1.execute("INSERT INTO test VALUES('Hallo Welt')")
>>> cursor1.execute("INSERT INTO test VALUES(?)",
                    ("foo".encode("UTF-16"),))
```

Da wir "foo" mit UTF-16 kodieren, sieht `sqlite3` diesen Eintrag als Binärdatum. Nun lesen wir die beiden Zeilen wieder aus und stellen fest, dass tatsächlich im ersten Fall eine `str`-Instanz und im zweiten Fall ein `bytes`-String zurückgeliefert wird:

```
>>> cursor1.execute("SELECT * FROM test")
>>> cursor1.fetchall()
[('Hallo Welt',), (b'\xff\xfe\x00o\x00o\x00',)]
```

Der Name `OptimizedUnicode` kommt nicht von ungefähr, denn diese Funktion ist auf Geschwindigkeit optimiert.

Connection.row_factory

Ein ähnliches Attribut wie `text_factory` für `TEXT`-Spalten existiert auch für ganze Zeilen. In dem Attribut `row_factory` kann eine Referenz auf eine Funktion gespeichert werden, die Zeilen für das Benutzerprogramm aufbereitet. Standardmäßig wird die Funktion `tuple` benutzt. Wir wollen beispielhaft eine Funktion implementieren, die uns auf die Spaltenwerte eines Datensatzes über die Namen der jeweiligen Spalten zugreifen lässt. Das Ergebnis soll dann folgendermaßen aussehen:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[{'anschrift': 'Turnhallenstr. 1, 3763 Sporthausen', 'kundennummer': 12, 'name': 'Heinz Elhurg'},
 {'anschrift': 'K\xc3\xa4mperweg 24, 2463 Duisschloss', 'kundennummer': 57, 'name': 'Markus Altbert'},
 {'anschrift': 'Podmacstr 2, 7467 Iwarhausen', 'kundennummer': 64, 'name': 'Steve Apple'}]
```

Um dies zu bewerkstelligen, benötigen wir noch das Attribut `description` der `Cursor`-Klasse, das uns Informationen zu den Spaltennamen der letzten Abfrage liefert. `description` enthält dabei eine Sequenz, die für jede Spalte ein Tupel mit sieben Elementen bereitstellt, von denen uns aber nur das erste, nämlich der Spaltenname, interessiert:

```
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.description
(('kundennummer', None, None, None, None, None, None),
 ('name', None, None, None, None, None, None),
 ('anschrift', None, None, None, None, None, None))
```

Die `row_factory`-Funktion erhält als Parameter eine Referenz auf den Cursor, der für die Abfrage verwendet wurde, und die Ergebniszeile als Tupel.

Mit diesem Wissen können wir jetzt unsere `row_factory`-Funktion namens `zeilen_dict` wie folgt implementieren:

```
def zeilen_dict(cursor, zeile):
    ergebnis = {}
    for spaltennr, spalte in enumerate(cursor.description):
        ergebnis[spalte[0]] = zeile[spaltennr]
    return ergebnis
```

Zur Erinnerung: `enumerate` erzeugt einen Iterator, der für jedes Element der übergebenen Sequenz ein Tupel zurückgibt, das den Index des Elements in der Sequenz und seinen Wert enthält.

In der Praxis arbeitet unsere `row_factory` wie folgt:

```
>>> connection.row_factory = zeilen_dict
>>> cursor = connection.cursor()
>>> cursor.execute("SELECT * FROM kunden")
>>> cursor.fetchall()
[{'anschrift': 'Turnhallenstr. 1, 3763 Sporthausen', 'kundennummer': 12, 'name': 'Heinz Elhurg'},
 {'anschrift': 'Kämperweg 24, 2463 Duisschloss', 'kundennummer': 57, 'name': 'Markus Altburt'},
 {'anschrift': 'Podmacstr 2, 7467 Iwarhausen', 'kundennummer': 64, 'name': 'Steve Apple'}]
```

Pythons `sqlite3`-Modul liefert schon eine erweiterte `row_factory` namens `sqlite3.Row` mit, die die Zeilen in ähnlicher Weise verarbeitet wie unsere `zeilen_dict`-Funktion. Da `sqlite3.Row` sehr stark optimiert ist und außerdem der Zugriff auf die Spaltenwerte über den jeweiligen Spaltennamen unabhängig von Groß- und Kleinschreibung erfolgen kann, sollten Sie die eingebaute Funk-

tion unserem Beispiel vorziehen und nur dann eine eigene `row_factory` implementieren, wenn Sie etwas ganz anderes erreichen möchten.

Nach diesem kleinen Ausflug zu den `factory`-Funktionen wenden wir uns endlich der ersten unserer beiden Fragen zu: Wie können wir beliebige Datentypen in SQLite-Datenbanken speichern?

Adapter und Konvertierer

Wie Sie bereits wissen, unterstützt SQLite nur eine beschränkte Menge von Datentypen. Als Folge davon müssen wir alle anderen Datentypen, die wir in der Datenbank ablegen möchten, durch die vorhandenen abbilden. Aufgrund ihrer Flexibilität eignen sich die `TEXT`-Spalten am besten, um beliebige Daten aufzunehmen, weshalb wir uns im Folgenden auf sie beschränken.

Analog zur String-Kodierung, bei der wir `str`-Instanzen mittels ihrer `encode`-Methode in gleichwertige `bytes`-Instanzen umformen und die ursprünglichen Unicode-Daten mithilfe der `decode`-Methode wiederherstellen konnten, brauchen wir nun Operationen, um beliebige Datentypen erst in Strings zu transformieren und anschließend die Ursprungsdaten wieder aus dem String zu extrahieren.

Das Umwandeln von beliebigen Datentypen in einen String wird *Adaption* genannt, und die Rückgewinnung der Daten aus diesem String heißt *Konvertierung*. Abbildung 19.4 veranschaulicht diesen Zusammenhang am Beispiel der Klasse `Kreis`, die als Attribute die Koordinaten des Kreismittelpunktes `Mx` und `My` sowie die Länge des Radius `R` besitzt:

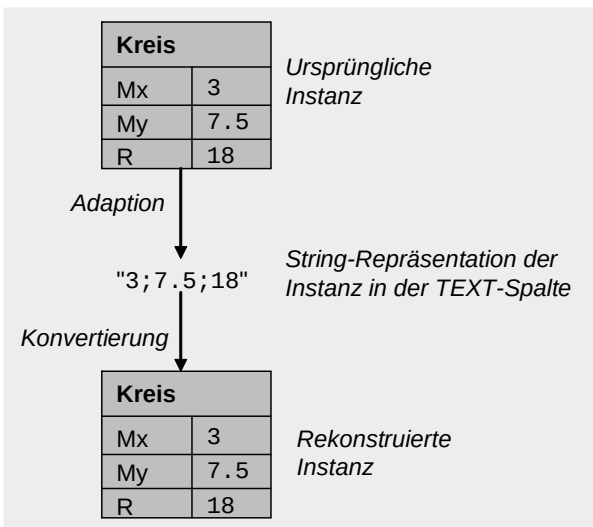


Abbildung 19.4 Schema der Adaption und Konvertierung

Eine entsprechende `Kreis`-Klasse lässt sich folgendermaßen definieren:

```
class Kreis:
    def __init__(self, mx, my, r):
        self.Mx = mx
        self.My = my
        self.R = r
```

Nun müssen wir eine Adapterfunktion erstellen, die aus unseren `Kreis`-Instanzen Strings macht. Die Umwandlung nehmen wir so vor, dass wir einen String erstellen, der durch Semikola getrennt die drei Attribute des Kreises enthält:

```
def kreisadapter(k):
    return "{0};{1};{2}".format(k.Mx, k.My, k.R)
```

Damit die Datenbank weiß, dass wir die Kreise mit dieser Funktion adaptieren möchten, muss sie registriert und mit dem Datentyp `Kreis` verknüpft werden. Dies geschieht durch den Aufruf der `sqlite3.register_adapter`-Methode, die als ersten Parameter den zu adaptierenden Datentyp und als zweiten Parameter die Adapterfunktion erwartet:

```
>>> sqlite3.register_adapter(Kreis, kreisadapter)
```

Durch diese Schritte ist es uns möglich, Kreise in `TEXT`-Spalten abzulegen. Wirklich nützlich wird das Ganze aber erst dann, wenn beim Auslesen auch automatisch wieder `Kreis`-Instanzen generiert werden.

Deshalb müssen wir noch die Umkehrfunktion von `kreisadapter`, den Konverter, definieren, der aus dem String die ursprüngliche `Kreis`-Instanz wiederherstellt. In unserem Beispiel erweist sich das als sehr einfach:

```
def kreiskonverter(bytestring):
    mx, my, r = bytestring.split(b";")
    return Kreis(float(mx), float(my), float(r))
```

Unerwartet ist wohl nur, dass wir beim Aufruf von `bytestring.split` das Trennzeichen als `bytestring` übergeben. Dies ist deshalb erforderlich, da `bytestring` ein Objekt von Typ `bytes` ist und das Trennzeichen vom selben Typ sein muss.

Genau wie der Adapter muss auch die Konverterfunktion bei SQLite registriert werden, was wir mit der Methode `sqlite3.register_converter()` erreichen:

```
>>> sqlite3.register_converter("KREIS", kreiskonverter)
```

Anders als `register_adapter` erwartet `register_convert` dabei einen String als ersten Parameter, der dem zu konvertierenden Datentyp einen Namen innerhalb von SQLite zuweist. Dadurch haben wir einen neuen SQLite-Datentyp namens `KREIS` definiert, den wir genau wie die eingebauten Typen für die Spalten unserer

Tabellen verwenden können. Allerdings müssen wir SQLite beim Verbinden zu der Datenbank mitteilen, dass wir von uns definierte Typen verwenden möchten. Dazu übergeben wir der `connect`-Methode einen entsprechenden Wert als Schlüsselwortparameter `detect_types`:

```
>>> connection = sqlite3.connect(":memory:",
                                detect_types=sqlite3.PARSE_DECLTYPES)
```

Nachfolgend demonstrieren wir die Definition und Verwendung unseres neuen Datentyps `kreis` in einem Miniprogramm:

```
import sqlite3

class Kreis:
    def __init__(self, mx, my, r):
        self.Mx = mx
        self.My = my
        self.R = r

def kreisadapter(k):
    return "{0};{1};{2}".format(k.Mx, k.My, k.R)

def kreiskonverter(bytestring):
    mx, my, r = bytestring.split(b";")
    return Kreis(float(mx), float(my), float(r))

# Adapter und Konverter registrieren
sqlite3.register_adapter(Kreis, kreisadapter)
sqlite3.register_converter("KREIS", kreiskonverter)

# Hier wird eine Beispieldatenbank im Arbeitsspeicher mit
# einer einspaltigen Tabelle für Kreise definiert
connection = sqlite3.connect(":memory:",
                             detect_types=sqlite3.PARSE_DECLTYPES)
cursor = connection.cursor()
cursor.execute("CREATE TABLE kreis_tabelle(k KREIS)")

# Kreis in die Datenbank schreiben
kreis = Kreis(1, 2.5, 3)
cursor.execute("INSERT INTO kreis_tabelle VALUES (?)", (kreis,))

# Kreis wieder auslesen
cursor.execute("SELECT * FROM kreis_tabelle")

gelesener_kreis = cursor.fetchall()[0][0]
```

```
print(type(gelesener_kreis))
print(gelesener_kreis.Mx, gelesener_kreis.My, gelesener_kreis.R)
```

Die Ausgabe dieses Programms ergibt sich wie folgt und zeigt, dass `gelesener_kreis` tatsächlich eine Instanz unserer `Kreis`-Klasse mit den korrekten Attributen ist:

```
<class '__main__.Kreis'>
1.0 2.5 3.0
```

Einschränkungen

Das Datenbanksystem SQLite ist in bestimmten Punkten eingeschränkt. Beispielsweise wird eine Datenbank beim Verändern oder Hinzufügen von Datensätzen für Lesezugriffe gesperrt, was besonders bei Webanwendungen sehr hinderlich ist: In der Regel werden mehrere Besucher eine Internetseite gleichzeitig aufrufen, und wenn jemand beispielsweise einen neuen Foreneintrag erstellt, wollen die anderen Besucher nicht länger auf die Anzeige der Seite warten müssen.

Deshalb gibt es andere Systeme, die auch mit den Anforderungen größerer Projekte zurechtkommen, wie zum Beispiel MySQL. Da zum Zeitpunkt der Drucklegung dieses Buches leider noch keine mit Python 3.0 lauffähige Anbindung an MySQL verfügbar war, müssen wir Sie an dieser Stelle mit einem Link auf die Website des Projektes vertrösten. Dort wird in naher Zukunft eine aktualisierte Version erscheinen, die auch mit Python 3.0 funktioniert:

<http://sourceforge.net/projects/mysql-python>

Allerdings können wir Sie beruhigen: Die in diesem Kapitel erworbenen Kenntnisse werden Sie ohne große Umstellungen auch auf MySQL anwenden können.

Außerdem befindet sich auf der Buch-CD die erste Auflage dieses Buches, die ein Kapitel über die Verwendung von MySQL mit Python 2.5 enthält.

19.4 Serialisierung von Instanzen – pickle

Das Modul `pickle` (dt. »pökeln«) bietet komfortable Funktionen für das *Serialisieren* von Objekten. Beim Serialisieren eines Objekts wird ein `bytes`-Objekt erzeugt, das alle Informationen des Objekts speichert, so dass es später wieder durch das sogenannte *Deserialisieren* rekonstruiert werden kann.

Besonders für die dauerhafte Speicherung von Daten in Dateien ist `pickle` sehr gut geeignet. Folgende Datentypen können mithilfe von `pickle` serialisiert bzw. deserialisiert werden:

- ▶ `None`, `True`, `False`
- ▶ numerische Datentypen (`int`, `float`, `complex`, `bool`)
- ▶ `str`, `bytes`
- ▶ sequentielle Datentypen (`tuple`, `list`), Mengen (`set`, `frozenset`) und Dictionaries (`dict`), solange alle ihre Elemente auch von `pickle` serialisiert werden können
- ▶ globale Funktionen
- ▶ Built-in Functions
- ▶ globale Klassen
- ▶ Klasseninstanzen, deren Attribute serialisiert werden können

Bei Klassen und Funktionen müssen Sie beachten, dass solche Objekte beim Serialisieren nur mit ihrem Klassennamen gespeichert werden. Der Code einer Funktion oder die Definition der Klasse und ihre Attribute werden nicht gesichert. Wenn Sie also beispielsweise eine Instanz einer selbstdefinierten Klasse deserialisieren möchten, muss die Klasse in dem aktuellen Kontext genauso wie bei der Serialisierung definiert sein. Ist das nicht der Fall, wird ein `UnpicklingError` erzeugt.

Es gibt drei verschiedene Formate, in denen `pickle` seine Daten speichern kann. Jedes dieser Formate hat eine Zahl, um es zu identifizieren:

Nummer	Beschreibung
0	Der resultierende String besteht nur aus ASCII-Zeichen und kann deshalb auch von Menschen beispielsweise zu Debug-Zwecken gelesen werden.
1	Dieses Protokoll erzeugt einen Binärstring, der die Daten im Vergleich zur ASCII-Variante platzsparender speichert. Auch das Protokoll 1 ist abwärtskompatibel mit Python-Versionen vor 2.3.
2	Neues Binärformat, das besonders für Klasseninstanzen optimiert wurde. Objekte, die mit dem Protokoll 2 serialisiert wurden, können nur von Python-Versionen ab 2.3 gelesen werden.
3	Ein neues Protokoll, das mit Python 3.0 eingeführt wurde und unter anderem auch den neuen <code>bytes</code> -Typ unterstützt. Die Daten können nicht mehr mit älteren Python-Versionen als 3.0 rekonstruiert werden. Trotzdem ist das Protokoll 3 das empfohlene und wird im <code>pickle</code> -Modul als Standard verwendet.

Tabelle 19.8 Die pickle-Protokolle

Das Modul `pickle` bietet seine Funktionalität über zwei Schnittstellen an: eine imperative über die Funktionen `dump` und `load` und eine objektorientierte mit den Klassen `Pickler` und `Unpickler`.

Um `pickle` verwenden zu können, muss das Modul importiert werden:

```
>>> import pickle
```

Die imperative Schnittstelle

`pickle.dump(obj, file[, protocol])`

Schreibt die Serialisierung von *obj* in das Dateiobjekt *file*. Das übergebene Dateiobjekt muss dabei für den Schreibzugriff geöffnet worden sein.

Mit dem Parameter *protocol* können Sie das Protokoll für die Speicherung übergeben. Der Standardwert für *protocol* ist 3. Geben Sie ein Binärformat an, so muss das für *file* übergebene Dateiobjekt im binären Schreibmodus geöffnet worden sein.

```
>>> f = open("pickle-test.dat", "bw")
>>> pickle.dump([1, 2, 3], f)
```

Für *file* können Sie neben echten Dateiobjekten jedes Objekt übergeben, das eine `write`-Methode mit einem String-Parameter implementiert, zum Beispiel `StringIO`-Instanzen.

`pickle.load(file)`

Lädt das nächste serialisierte Objekt aus dem geöffneten Dateiobjekt, das für *file* übergeben wurde. Dabei erkennt *load* selbstständig, in welchem Format die Daten gespeichert wurden.

Das folgende Beispiel setzt voraus, dass im aktuellen Arbeitsverzeichnis eine Datei mit dem Namen *pickle-test.dat* existiert, die eine serialisierte Liste enthält:

```
>>> f = open("pickle-test.dat", "rb")
>>> pickle.load(f)
[1, 2, 3]
```

Auch hier müssen Sie darauf achten, die Dateien im Binärmodus zu öffnen, wenn Sie andere `pickle`-Protokolle als 0 verwenden.

`pickle.dumps(obj[, protocol])`

Gibt die serialisierte Repräsentation von *obj* als `bytes`-String zurück, wobei der Parameter *protocol* angibt, welches der drei Serialisierungsprotokolle verwendet werden soll. Standardmäßig wird das Protokoll mit der Kennung 3 benutzt.

```
>>> pickle.dumps([1, 2, 3])
b'\x80\x03q\x00(K\x01K\x02K\x03e.'
```

pickle.loads(string)

Stellt das in *string* serialisierte Objekt wieder her. Das verwendete Protokoll wird dabei automatisch erkannt, und überflüssige Zeichen am Ende des Strings werden ignoriert:

```
>>> s = pickle.dumps([1, 2, 3])
>>> pickle.loads(s)
[1, 2, 3]
```

Die objektorientierte Schnittstelle

Gerade dann, wenn viele Objekte in dieselbe Datei serialisiert werden sollen, ist es lästig und schlecht für die Lesbarkeit, jedes Mal das Dateiojekt und das zu verwendende Protokoll bei den Aufrufen von `dump` mit anzugeben.

Neben den schon vorgestellten Modulfunktionen gibt es deshalb noch die beiden Klassen `Pickler` und `Unpickler`.

`Pickler` und `Unpickler` haben außerdem den Vorteil, dass Klassen von ihnen erben und so die Serialisierung anpassen können.

pickle.Pickler(file[, protocol])

Die Parameter *file* und *protocol* haben die gleiche Bedeutung wie bei der `pickle.dump`-Funktion. Das resultierende `Pickler`-Objekt hat eine Methode namens `dump`, die als Parameter ein Objekt erwartet, das serialisiert werden soll.

Alle an die `load`-Methode gesendeten Objekte werden in das beim Erzeugen der `Pickler`-Instanz übergebene Dateiojekt geschrieben.

```
>>> p = pickle.Pickler(open("eine_datei.dat", "wb"), 2)
>>> p.dump({"vorname" : "Peter", "nachname" : "Kaiser"})
>>> p.dump([1, 2, 3, 4])
```

pickle.Unpickler(file)

Das Gegenstück zu `Pickler` ist `Unpickler`, um aus der übergebenen Datei die ursprünglichen Daten wiederherzustellen. `Unpickler`-Instanzen besitzen eine parameterlose Methode namens `load`, die jeweils das nächste Objekt aus der Datei liest.

Das folgende Beispiel setzt voraus, dass die im Beispiel zur `Pickler`-Klasse erzeugte Datei *eine_datei.dat* im aktuellen Arbeitsverzeichnis liegt:

```
>>> u = pickle.Unpickler(open("eine_datei.dat", "rb"))
>>> u.load()
{'nachname': 'Kaiser', 'vorname': 'Peter'}
>>> u.load()
[1, 2, 3, 4]
```

Das optimierte pickle

Das Modul `pickle` ist komplett in Python implementiert und deshalb für sehr große Datenmengen langsam. Seit Python 3.0 existiert ein weiteres Modul namens `_pickle`, das eine optimierte Version von `pickle` bereitstellt. Die gute Nachricht: Sie brauchen sich nicht im Geringsten darum zu kümmern, da Python automatisch prüft, ob statt der langsamen Standardimplementation das schnellere Modul genutzt werden kann.⁶

19.5 Das Tabellenformat CSV – csv

Ein sehr verbreitetes Import- und Exportformat für Datenbanken und Tabellenkalkulationen ist das *CSV-Format* (CSV steht für *Comma Separated Values*). CSV-Dateien sind Textdateien, die zeilenweise Datensätze enthalten. Innerhalb der Datensätze sind die einzelnen Werte durch ein Trennzeichen wie beispielsweise das Komma voneinander getrennt, daher auch der Name.

Eine CSV-Datei, die Informationen zu Personen speichert und das Komma als Trennzeichen nutzt, könnte beispielsweise so aussehen:

```
vorname,nachname,geburtsdatum,wohntort,haarfarbe
Markus,Maier,19.05.1986,Gaggenau,Braun
David,Schönauer,10.09.1988,Aachen,Braun
Sebastian,Sentner,06.09.1987,Sydney,Dunkelblond
Jan,Fitzke,13.09.1987,Köln,Schwarz
Lucas,Hövelmann,25.03.1988,Canberra,Hellrot
```

Die erste Zeile enthält die jeweiligen Spaltenköpfe, und alle folgenden Zeilen enthalten die eigentlichen Datensätze.

Leider existiert kein Standard für CSV-Dateien, so dass sich beispielsweise das Trennzeichen von Programm zu Programm unterscheiden kann. Dieser Umstand erschwert es, CSV-Dateien von verschiedenen Quellen zu lesen, da immer auf das besondere Format der exportierenden Anwendung eingegangen werden muss.

Um trotzdem mit CSV-Dateien der verschiedensten Formate umgehen zu können, stellt Python das Modul `csv` zur Verfügung. Das `csv`-Modul implementiert `reader`- und `writer`-Klassen, die den Lese- bzw. Schreibzugriff auf CSV-Daten kapseln. Mithilfe sogenannter *Dialekte* kann dabei das Format der Datei angegeben werden. Standardmäßig gibt es vordefinierte Dialekte für die CSV-Dateien, die von Micro-

⁶ In den Versionen vor Python 3.0 gab es zu diesem Zweck das Modul `cPickle`, das aber explizit benutzt werden musste.

soft Excel generiert werden. Außerdem stellt das Modul eine Klasse namens `Sniffer` (dt. »Schnüffler«) bereit, die den Dialekt einer Datei erraten kann.

Eine Liste aller definierten Dialekte erhalten Sie mit `csv.list_dialects`:

```
>>> import csv
>>> csv.list_dialects()
['excel-tab', 'excel']
```

reader-Objekte

Mithilfe von `reader`-Objekten können CSV-Dateien gelesen werden. Der Konstruktor sieht dabei folgendermaßen aus:

`csv.reader(csvfile[, dialect][, fmtparam])`

Der Parameter *csvfile* muss eine Referenz auf ein für den Lesezugriff geöffnetes Dateiojekt sein, aus dem die Daten gelesen werden sollen.

Mit *dialect* können Sie angeben, in welchem Format die zu lesende Datei geschrieben wurde. Dazu übergeben Sie als *dialect* einen String, der in der Liste enthalten ist, die `csv.list_dialects` zurückgibt. Alternativ geben Sie eine Instanz der Klasse `Dialect` an, die wir später besprechen werden. Standardmäßig wird der Wert "excel" für *dialect* verwendet, wobei die damit kodierte Dateien das Komma als Trennzeichen verwenden.

Der Platzhalter *fmtparam* steht nicht für einen einzelnen Parameter, sondern für Schlüsselwortparameter, die übergeben werden können, um den Dialekt ohne Umweg über die `Dialect`-Klasse festzulegen. Ein Beispiel, bei dem wir auf diese Weise das Semikolon als Trennzeichen zwischen den einzelnen Werten festlegen, sieht folgendermaßen aus:

```
>>> reader = csv.reader(open("datei.csv"), delimiter=";")
```

Wir werden uns später ausführlich mit Dialekten beschäftigen.

Die `reader`-Instanzen implementieren das Iterator-Protokoll und lassen sich deshalb zum Beispiel komfortabel mit einer `for`-Schleife verarbeiten. Im folgenden Beispiel lesen wir die CSV-Datei mit den Personen:

```
>>> reader = csv.reader(open("namen.csv"))
>>> for row in reader:
    print(row)
['vorname', 'nachname', 'geburtsdatum', 'wohntort', 'haarfarbe']
['Markus', 'Maier', '19.05.1986', 'Gaggenau', 'Braun']
['David', 'Schönauer', '10.09.1988', 'Aachen', 'Braun']
['Sebastian', 'Sentner', '06.09.1987', 'Sydney', 'Dunkelblond']
```



```
['Jan', 'Fitzke', '13.09.1987', 'Köln', 'Schwarz']
['Lucas', 'Hövelmann', '25.03.1988', 'Canberra', 'Hellrot']
```

Wie Sie sehen, gibt uns der `reader` für jede Zeile eine Liste mit den Werten der einzelnen Spalten zurück. Wichtig ist dabei, dass die Spaltenwerte immer als Strings zurückgegeben werden.

Neben dem Standard-`reader`, der Listen zurückgibt, existiert noch der sogenannte `DictReader`, der für jede Zeile ein Dictionary erzeugt, das den Spaltenköpfen die Werte der jeweiligen Zeile zuordnet.

Unser letztes Beispiel verändert sich durch die Verwendung von `DictReader` wie folgt, wobei wir nur die ersten beiden Datensätze ausgeben, um Platz zu sparen:

```
>>> reader = csv.DictReader(open("namen.csv"))
>>> for row in reader:
    print(row)
{'nachname': 'Maier', 'geburtsdatum': '19.05.1986', 'wohntort':
 'Gaggenau', 'vorname': 'Markus', 'haarfarbe': 'Braun'}
{'nachname': 'Schönauer', 'geburtsdatum': '10.09.1988', 'wohntort':
 'Aachen', 'vorname': 'David', 'haarfarbe': 'Braun'}
```

writer-Objekte

Der Konstruktor der `writer`-Klasse erwartet die gleichen Parameter wie der Konstruktor der `reader`-Klasse, mit der Ausnahme, dass das für *csvfile* übergebene Dateiobjekt für den Schreibzugriff geöffnet worden sein muss.

csv.reader(csvfile[, dialect][, fmtparam])

Das resultierende `writer`-Objekt hat die beiden Methoden `writerow` und `writerows`, mit denen sich einzelne bzw. mehrere Zeilen auf einmal in die CSV-Datei schreiben lassen:

```
>>> writer = csv.writer(open("autos.csv", "wb"))
>>> writer.writerow(["marke", "modell", "leistung_in_ps"])
>>> daten = (
    ["Volvo", "P245", "130"], ["Ford", "Focus", "90"],
    ["Mercedes", "CLK", "250"], ["Audi", "A6", "350"],
)
>>> writer.writerows(daten)
```

In dem Beispiel erzeugen wir eine neue CSV-Datei mit dem Namen `"autos.csv"`. Mit der `writerow`-Methode schreiben wir die Spaltenköpfe in die erste Zeile der neuen Datei und mit `writerows` anschließend vier Beispieldatensätze.

Analog zur `DictReader`-Klasse existiert auch eine `DictWriter`-Klasse, die sich fast genauso wie die normale `writer`-Klasse erzeugen lässt, außer dass Sie neben dem Dateiojekt noch eine Liste mit den Spaltenköpfen übergeben müssen. Für ihre `writerow`- und `writerows`-Methoden erwarten `DictWriter`-Instanzen Dictionaries als Parameter. Das folgende Beispiel erzeugt die gleiche CSV-Datei wie das letzte:

```
>>> writer = csv.DictWriter(open("autos.csv", "w"),
                             ["marke", "modell", "leistung_in_ps"])
>>> writer.writerow({"marke" : "marke", "modell" : "modell",
                     "leistung_in_ps" : "leistung_in_ps"})
>>> daten = ({ "marke" : "Volvo", "modell" : "P245",
                "leistung_in_ps" : "130"},
              { "marke" : "Ford", "modell" : "Focus",
                "leistung_in_ps" : "90"},
              { "marke" : "Mercedes", "modell" : "CLK",
                "leistung_in_ps" : "250"},
              { "marke" : "Audi", "modell" : "A6",
                "leistung_in_ps" : "350"})
>>> writer.writerows(daten)
```

Die merkwürdige erste Zeile mit `writerow` ist notwendig, um die Spaltenköpfe zu schreiben, da dies nicht automatisch geschieht.

Dialect-Objekte

Die Instanzen der Klasse `csv.Dialect` dienen dazu, den Aufbau von CSV-Dateien zu beschreiben. Sie sollten `Dialect`-Objekte nicht direkt erzeugen, sondern stattdessen die Funktion `csv.register_dialect` verwenden. Mit `register_dialect` erzeugen Sie einen neuen Dialekt und versehen ihn mit einem Namen. Dieser Name kann dann später als Parameter an die Konstruktoren der `reader`- und `writer`-Klassen übergeben werden. Außerdem ist jeder registrierte Name in der von `csv.get_dialects` zurückgegebenen Liste enthalten.

Die Funktion `register_dialect` hat folgende Schnittstelle:

`csv.register_dialect(name[, dialect][, fmtparam])`

Der Parameter *name* muss dabei ein String sein, der den neuen Dialekt identifiziert. Mit *dialect* kann ein bereits bestehendes `Dialect`-Objekt übergeben werden, das dann mit dem entsprechenden Namen verknüpft wird.

Am wichtigsten ist der Platzhalter *fmtparam*, der für eine Reihe optionaler Schlüsselwortparameter steht, die den neuen Dialekt beschreiben. Es sind die in der folgenden Tabelle aufgeführten Parameter erlaubt:

Name	Bedeutung
delimiter	Trennzeichen zwischen den Spaltenwerten. Der Standardwert ist das Komma „ <code>,</code> “.
quotechar	Ein Zeichen, um Felder zu umschließen, die besondere Zeichen wie das Trennzeichen oder den Zeilenumbruch enthalten. Der Standardwert sind die doppelten Anführungszeichen „ <code>“</code> “.
doublequote	Ein boolescher Wert, der angibt, wie das für <code>quotechar</code> angegebene Zeichen innerhalb von Feldern selbst maskiert werden soll. Hat <code>doublequote</code> den Wert <code>True</code>, so wird <code>quotechar</code> zweimal hintereinander eingefügt. Ist der Wert von <code>doublequote</code> <code>False</code>, wird stattdessen das für <code>escapechar</code> angegebene Zeichen vor <code>quotechar</code> geschrieben. Standardmäßig hat <code>doublequote</code> den Wert <code>True</code>.
escapechar	Ein Zeichen, das benutzt wird, um das Trennzeichen innerhalb von Spaltenwerten zu maskieren, sofern <code>quoting</code> den Wert <code>QUOTE_NONE</code> hat. Bei einem <code>doublequote</code>-Wert von <code>False</code> wird <code>escapechar</code> außerdem für die Maskierung <code>quotechar</code> verwendet. Standardmäßig ist die Maskierung deaktiviert und <code>escapechar</code> hat den Wert <code>None</code>.
lineterminator	Zeichen, das zum Trennen der Zeilen benutzt wird. Standardmäßig ist es auf „<code>\r\n</code>“ gesetzt. Bitte beachten Sie, dass diese Einstellung nur den Writer betrifft. Alle <code>reader</code>-Objekte bleiben von der <code>lineterminator</code>-Einstellung unbeeinflusst und verwenden immer „<code>\r</code>“, „<code>\n</code>“ oder die Kombination aus beiden als Zeilentrennzeichen.
quoting	Gibt an, ob und wann Spaltenwerte mit <code>quotechar</code> umschlossen werden sollen. Gültige Werte sind: <code>QUOTE_ALL</code>: Alle Spaltenwerte werden umschlossen. <code>QUOTE_MINIMAL</code>: Nur die Felder mit speziellen Zeichen wie Zeilenvorschüben oder dem Trennzeichen für Spaltenwerte werden umschlossen. <code>QUOTE_NONNUMERIC</code>: Beim Schreiben werden alle nicht-numerischen Felder von <code>quotechar</code> umschlossen. Beim Lesen werden alle nicht umschlossenen Felder automatisch nach <code>float</code> konvertiert. <code>QUOTE_NONE</code>: Keine Umschließung mit <code>quotechar</code> wird vorgenommen. Standardmäßig ist <code>quoting</code> auf <code>QUOTE_MINIMAL</code> eingestellt.

Tabelle 19.9 Schlüsselwortparameter für `register_dialect`

Name	Bedeutung
skipinitialspace	Ein boolescher Wert, der angibt, wie mit führenden Whitespaces in einem Spaltenwert verfahren werden soll. Eine Einstellung auf <code>True</code> bewirkt, dass alle führenden Whitespaces ignoriert werden; bei einem Wert von <code>False</code> wird der komplette Spalteninhalt gelesen und zurückgegeben. Der Standardwert ist <code>False</code> .

Tabelle 19.9 Schlüsselwortparameter für `register_dialect` (Forts.)

Wir wollen als Beispiel einen neuen Dialekt namens `"mein_dialekt"` registrieren, der als Trennzeichen den Tabulator verwendet und alle Felder mit Anführungszeichen umschließt:

```
>>> csv.register_dialect("mein_dialekt", delimiter="\t",
                        quoting=csv.QUOTE_ALL)
```

Diesen neuen Dialekt können wir nun dem Konstruktor unserer `reader`- und `writer`-Klassen übergeben und auf diese Weise unsere eigenen CSV-Dateien schreiben und lesen.

19.6 Temporäre Dateien – tempfile

Wenn Ihre Programme sehr umfangreiche Daten verarbeiten müssen, ist es oft nicht sinnvoll, alle Daten auf einmal im Arbeitsspeicher zu halten. Für diesen Zweck existieren temporäre Dateien, die es Ihnen erlauben, gerade nicht benötigte Daten vorübergehend auf die Festplatte auszulagern. Für die dauerhafte Speicherung der Daten eignen sich temporäre Dateien nicht.

Für den komfortablen Umgang mit temporären Dateien stellt Python das Modul `tempfile` bereit.

Die wichtigste Funktion dieses Moduls ist `TemporaryFile`, die ein geöffnetes Dateiobjekt zurückgibt, das mit einer neuen temporären Datei verknüpft ist. Die Datei wird für Lese- und Schreibzugriffe im Binärmodus (`"w+b"`) geöffnet. Wir als Benutzer der Funktion brauchen uns dabei um nichts weiter als das Lesen und Schreiben unserer Daten zu kümmern. Das Modul sorgt dafür, dass die temporäre Datei angelegt wird, und löscht sie auch wieder, wenn das Dateiobjekt von der Garbage Collection entsorgt wird.

Das Auslagern von Daten eines Programms auf die Festplatte ist ein Sicherheitsrisiko, weil andere Programme die Daten auslesen und damit unter Umständen

Zugriff auf sicherheitsrelevante Informationen erhalten könnten. Deshalb versucht `TemporaryFile`, die Datei sofort nach ihrer Erzeugung aus dem Dateisystem zu entfernen, um sie vor anderen Programmen zu verstecken, falls dies vom Betriebssystem unterstützt wird. Außerdem wird für den Dateiname ein zufälliger String benutzt, der aus sechs Zeichen besteht, wodurch es für andere Programme schwierig wird herauszufinden, zu welchem Programm eine temporäre Datei gehört.

Auch wenn Sie `TemporaryFile` in den meisten Fällen ohne Parameter aufrufen werden, wollen wir die vollständige Schnittstelle besprechen:

`TemporaryFile([mode[, bufsize[, suffix[, prefix[, dir]]]])`

Die Parameter *mode* und *bufsize* entsprechen den gleichnamigen Argumenten der Built-in Function `open` (nachzulesen in Kapitel 9, »Dateien«). Mit *suffix* und *prefix* passen Sie bei Bedarf den Namen der neuen temporären Datei an. Das, was Sie für *prefix* übergeben, wird vor den automatisch erzeugten Dateinamen gesetzt, und der Wert für *suffix* wird hinten an den Dateinamen angehängt. Zusätzlich können Sie mit dem Parameter *dir* angeben, in welchem Ordner die Datei erzeugt werden soll. Standardmäßig kümmert sich `TemporaryFile` auch automatisch um einen Speicherort für die Datei.

Zur Veranschaulichung der Nutzung von `TemporaryFile` folgt ein kleines Beispiel, das erst einen String in einer temporären Datei ablegt und ihn anschließend wieder einliest:

```
>>> import tempfile
>>> tmp = tempfile.TemporaryFile()
>>> tmp.write(b"Hallo Zwischenspeicher")
>>> tmp.seek(0)
>>> data = tmp.read()
>>> data
b'Hallo Zwischenspeicher'
```

Beachten Sie in obigem Beispiel, dass wir einen `bytes`-String übergeben mussten, weil die temporäre Datei im Binärmodus geöffnet wurde. Möchten Sie `str`-Objekte in temporäre Dateien schreiben, müssen Sie die Datei im Textmodus `"w"` öffnen oder die Strings beim Speichern mithilfe der `encode`-Methode in ein `bytes`-Objekt umwandeln.

Falls Sie nicht wünschen, dass die temporäre Datei verborgen wird, benutzen Sie die Funktion `NamedTemporaryFile`, die die gleiche Schnittstelle wie `TemporaryFile` hat und sich auch ansonsten bis auf das Verstecken genauso verhält.

tempfile.mkdtemp([suffix='', prefix='tmp', dir=None])

Mithilfe von `tempfile.mkdtemp` ist es außerdem möglich, temporäre Ordner anzulegen, wobei alle vom Betriebssystem angebotenen Mittel genutzt werden, um unberechtigte Zugriffe auf die temporären Daten zu unterbinden. Die Schnittstelle von `tempfile.mkdtemp` ist analog zu `tempfile.TemporaryFile` zu verwenden.

Als Rückgabewert erhalten Sie den absoluten Pfadnamen des temporären Ordners:

```
>>> tempfile.mkdtemp()
'/tmp/tmpFvqxTh'
```


»Alle reden von Kommunikation,
aber die wenigsten haben sich etwas mitzuteilen.«
– Hans Magnus Enzensberger

20 Netzwerkkommunikation

Nachdem wir uns ausführlich mit der Speicherung von Daten in Dateien verschiedener Formate oder Datenbanken beschäftigt haben, folgt nun ein Kapitel, das sich mit einer weiteren interessanten Programmierdisziplin beschäftigt: mit der Netzwerkprogrammierung.

Grundsätzlich lässt sich das Themenfeld der Netzwerkkommunikation in mehrere sogenannte *Protokollebenen* (engl. *layer*) aufteilen. Abbildung 20.1 zeigt eine stark vereinfachte Version des *OSI-Schichtenmodells*, das die Hierarchie der verschiedenen Protokollebenen veranschaulicht.

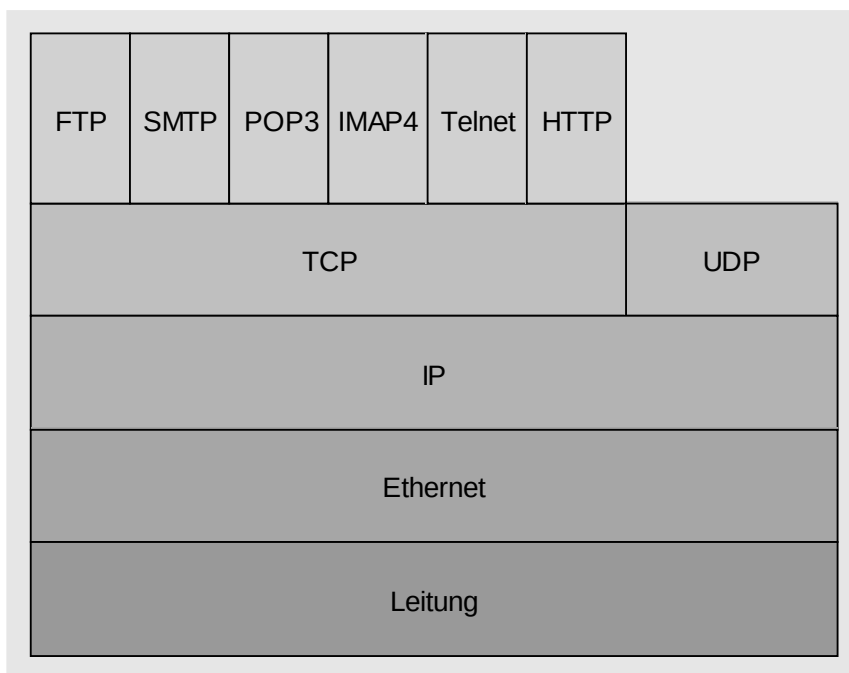


Abbildung 20.1 Netzwerkprotokolle

Das rudimentärste Protokoll steht in der Grafik ganz unten. Dabei handelt es sich um die blanke Leitung, über die die Daten in Form von elektrischen Impulsen übermittelt werden. Darauf aufbauend existieren etwas abstraktere Protokolle wie Ethernet und IP. Doch der für Anwendungsprogrammierer eigentlich interessante Teil fängt erst oberhalb des IP-Protokolls an, nämlich bei den Transportprotokollen TCP und UDP. Beide Protokolle werden wir ausführlich im Zusammenhang mit Sockets im nächsten Abschnitt besprechen.

Die Protokolle, die auf TCP aufbauen, sind am weitesten abstrahiert und deshalb für uns ebenfalls sehr interessant. In diesem Buch werden wir folgende Protokolle ausführlich behandeln:

Protokoll	Beschreibung	Modul	Abschnitt
UDP	grundlegendes verbindungsloses Netzwerkprotokoll	socket	20.1.2
TCP	grundlegendes verbindungsorientiertes Netzwerkprotokoll	socket	20.1.3
HTTP	Übertragen von Textdateien, beispielsweise Webseiten	urllib	20.2
FTP	Dateiübertragung	ftplib	20.3
SMTP	Versenden von E-Mails	smtplib	20.4.1
POP3	Abholen von E-Mails	poplib	20.4.2
IMAP4	Abholen von E-Mails	imaplib	20.4.3
Telnet	Terminalemulation	telnetlib	20.5

Tabelle 20.1 Netzwerkprotokolle

Beachten Sie, dass es auch abstrakte Protokolle gibt, die auf UDP aufbauen, beispielsweise NFS (*Network File System*). Wir werden in diesem Buch aber ausschließlich auf TCP basierende Protokolle behandeln.

Wir werden im ersten Unterabschnitt zunächst eine ganz grundlegende Einführung in das systemnahe Modul `socket` bringen. Es lohnt sich absolut, einen Blick in dieses Modul zu riskieren, denn es bietet viele Möglichkeiten der Netzwerkprogrammierung, die bei den anderen, abstrakteren Modulen verlorengehen. Außerdem lernen Sie den Komfort, den die abstrakten Schnittstellen bieten, erst wirklich zu schätzen, wenn Sie das `socket`-Modul kennengelernt haben.

Nachdem wir uns mit der Socket API beschäftigt haben, folgen einige spezielle Module, die beispielsweise mit bestimmten Protokollen wie HTTP oder FTP umgehen können.

20.1 Socket API

Das Modul `socket` der Standardbibliothek bietet grundlegende Funktionalität zur Netzwerkkommunikation. Es bildet dabei die standardisierte *Socket API* ab, die so oder in ganz ähnlicher Form auch für viele andere Programmiersprachen implementiert ist.

Hinter der Socket API steht die Idee, dass das Programm, das Daten über die Netzwerkschnittstelle senden oder empfangen möchte, dies beim Betriebssystem anmeldet und von diesem einen sogenannten *Socket* (dt. »Steckdose«) bekommt. Über diesen Socket kann das Programm jetzt eine Netzwerkverbindung zu einem anderen Socket aufbauen. Dabei spielt es keine Rolle, ob sich der Zielsocket auf demselben Rechner, einem Rechner im lokalen Netzwerk oder einem Rechner im Internet befindet.

Zunächst ein paar Worte dazu, wie ein Rechner in der komplexen Welt eines Netzwerks adressiert werden kann. Jeder Rechner besitzt in einem Netzwerk, auch dem Internet, eine eindeutige sogenannte *IP-Adresse*, über die er angesprochen werden kann. Eine IP-Adresse ist ein String der folgenden Struktur:

```
"192.168.1.23"
```

Dabei repräsentiert jeder der vier Zahlenwerte ein Byte und kann somit zwischen 0 und 255 liegen. In diesem Fall handelt es sich um eine IP-Adresse eines lokalen Netzwerks, was an der Anfangssequenz 192.168 zu erkennen ist.

Damit ist es jedoch noch nicht getan, denn auf einem einzelnen Rechner könnten mehrere Programme laufen, die gleichzeitig Daten über die Netzwerkschnittstelle senden und empfangen möchten. Aus diesem Grund wird eine Netzwerkverbindung zusätzlich an einen sogenannten *Port* gebunden. Der Port ermöglicht es, ein bestimmtes Programm anzusprechen, das auf einem Rechner mit einer bestimmten IP-Adresse läuft.

Bei einem Port handelt es sich um eine 16-Bit-Zahl – grundsätzlich sind also 65.535 verschiedene Ports verfügbar. Allerdings sind viele dieser Ports für Protokolle und Anwendungen registriert und sollten nicht verwendet werden. Beispielsweise sind für HTTP- und FTP-Server die Ports 80 bzw. 21 registriert. Grundsätzlich können Sie Ports ab 49152 bedenkenlos verwenden.

Beachten Sie, dass beispielsweise eine Firewall oder ein Router bestimmte Ports blockieren kann. Sollten Sie also auf Ihrem Rechner einen Server betreiben wollen, zu dem sich Clients über einen bestimmten Port verbinden können, müssen Sie diesen Port gegebenenfalls mit der entsprechenden Software freischalten.

20.1.1 Client-Server-Systeme

Die beiden Kommunikationspartner einer Netzwerkkommunikation haben in der Regel verschiedene Aufgaben. So existiert zum einen ein *Server*, der bestimmte Dienstleistungen anbietet, und zum anderen ein *Client* (dt. »Kunde«), der diese Dienstleistungen in Anspruch nimmt.

Ein Server ist unter einer bekannten Adresse im Netzwerk erreichbar und operiert passiv, das heißt, er wartet auf eingehende Verbindungen. Sobald eine Verbindungsanfrage eines Clients eintrifft, wird, sofern der Server die Anfrage akzeptiert, ein neuer Socket erzeugt, über den die Kommunikation mit diesem speziellen Client läuft. Wir werden uns zunächst mit sogenannten *seriellen Servern* befassen, das sind Server, bei denen die Kommunikation mit dem vorherigen Client abgeschlossen sein muss, bevor eine neue Verbindung akzeptiert werden kann. Dem gegenüber stehen die Konzepte der *parallelen Server* und der *multiplexenden Server*, auf die wir auch noch zu sprechen kommen werden.

Der Client stellt den aktiven Kommunikationspartner dar. Das heißt, er sendet eine Verbindungsanfrage an den Server und nimmt dann aktiv dessen Dienstleistungen in Anspruch.

Die Stadien, in denen sich ein serieller Server und ein Client vor, während und nach der Kommunikation befinden, verdeutlicht das Flussdiagramm in Abbildung 20.2. Sie können es als eine Art Bauplan für einen seriellen Server und den dazu gehörigen Client auffassen.

Zunächst wird im Serverprogramm der sogenannte *Verbindungssocket* erzeugt. Das ist ein Socket, der ausschließlich dazu gedacht ist, auf eingehende Verbindungen zu horchen und diese gegebenenfalls zu akzeptieren. Über den Verbindungssocket läuft keine Kommunikation. Durch Aufruf der Methoden `bind` und `listen` wird der Verbindungssocket an eine Netzwerkadresse gebunden und dazu instruiert, nach einkommenden Verbindungsanfragen zu lauschen.

Nachdem eine Verbindungsanfrage eingetroffen ist und mit `accept` akzeptiert wurde, wird ein neuer Socket, der sogenannte *Kommunikationssocket*, erzeugt. Über einen solchen Kommunikationssocket wird die vollständige Kommunikation zwischen Server und Client über Methoden wie `send` oder `recv` abgewickelt. Beachten Sie, dass ein Kommunikationssocket immer nur für einen verbundenen Client zuständig ist.

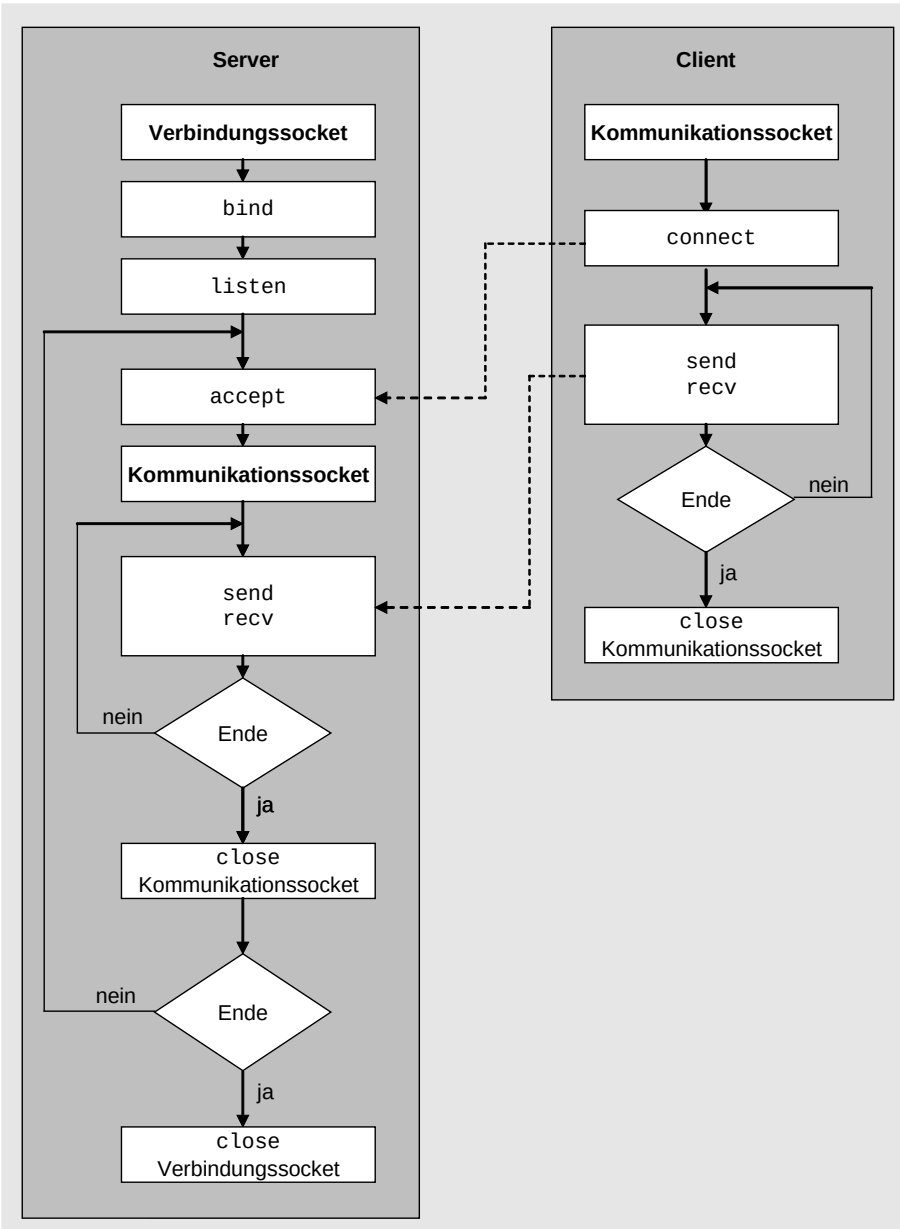


Abbildung 20.2 Das Client-Server-Modell

Sobald die Kommunikation beendet ist, wird das Kommunikationsobjekt geschlossen und eventuell eine weitere Verbindung eingegangen. Beachten Sie, dass Verbindungsanfragen, die nicht sofort akzeptiert werden, keineswegs verloren sind, sondern gepuffert werden. Sie befinden sich in der sogenannten *Queue* und können somit nacheinander abgearbeitet werden. Zum Schluss wird auch der Verbindungssocket geschlossen.

Die Struktur des Clients ist vergleichsweise einfach. So gibt es beispielsweise nur einen Kommunikationssocket, über den mithilfe der Methode `connect` eine Verbindungsanfrage an einen bestimmten Server gesendet werden kann. Danach erfolgt, ähnlich wie beim Server, die tatsächliche Kommunikation über Methoden wie `send` oder `recv`. Nach dem Ende der Kommunikation wird der Verbindungssocket geschlossen.

Grundsätzlich kann für die Datenübertragung zwischen Server und Client aus zwei verfügbaren Netzwerkprotokollen gewählt werden: UDP und TCP. In den folgenden beiden Abschnitten sollen kleine Beispielserver und -clients für beide dieser Protokolle implementiert werden.

Beachten Sie, dass sich das hier vorgestellte Flussdiagramm auf das verbindungsbehaftete und üblichere TCP-Protokoll bezieht. Die Handhabung des verbindungslosen UDP-Protokolls unterscheidet sich davon in einigen wesentlichen Punkten. Näheres dazu finden Sie im folgenden Abschnitt.

20.1.2 UDP

Das Netzwerkprotokoll *UDP (User Datagram Protocol)* wurde 1977 als Alternative zu TCP für die Übertragung menschlicher Sprache entwickelt. Charakteristisch ist, dass UDP verbindungslos und nicht-zuverlässig ist. Diese beiden Begriffe gehen miteinander einher und bedeuten zum einen, dass keine explizite Verbindung zwischen den Kommunikationspartnern aufgebaut wird, und zum anderen, dass UDP weder garantiert, dass gesendete Pakete in der Reihenfolge ankommen, in der sie gesendet wurden, noch dass sie überhaupt ankommen. Aufgrund dieser Einschränkungen können mit UDP jedoch vergleichsweise schnelle Übertragungen stattfinden, da beispielsweise keine Pakete neu angefordert oder gepuffert werden müssen.

Damit eignet sich UDP insbesondere für Multimedia-Anwendungen wie VoIP, Audio- oder Videostreaming, bei denen es auf eine schnelle Übertragung der Daten ankommt und kleinere Übertragungsfehler toleriert werden können.

Das im Folgenden entwickelte Beispielprojekt besteht aus einem Server- und einem Clientprogramm. Der Client schickt eine Textnachricht per UDP an eine

bestimmte Adresse. Das dort laufende Serverprogramm nimmt die Nachricht entgegen und zeigt sie an. Betrachten wir zunächst den Quellcode des Clients:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

ip = input("IP-Adresse: ")
nachricht = input("Nachricht: ")

s.sendto(bytes(nachricht, "utf-8"), (ip, 50000))
s.close()
```

Zunächst erzeugt der Aufruf der Funktion `socket` eine Socket-Instanz. Dabei können zwei Parameter übergeben werden: zum einen der zu verwendende Adresstyp und zum anderen das zu verwendende Netzwerkprotokoll. Die Konstanten `AF_INET` und `SOCK_DGRAM` stehen dabei für Internet/IPv4 und UDP.

Danach werden zwei Angaben vom Benutzer eingelesen: die IP-Adresse, an die die Nachricht zu schicken ist, und die Nachricht selbst.

Zum Schluss wird die Nachricht unter Verwendung der Socket-Methode `sendto` zur angegebenen IP-Adresse geschickt, wozu der Port 50000 verwendet wird. Da die zu versendende Nachricht keineswegs ein String sein muss, sondern vielmehr eine beliebige Folge von Bytes enthalten darf, wird an der Schnittstelle von `s.sendto` kein String erwartet, sondern eine `bytes`- oder `bytearray`-Instanz. Im Beispiel muss der eingelesene String also zuvor in eine `bytes`-Instanz überführt werden.

Das Clientprogramm allein ist so gut wie wertlos, solange es kein dazu passendes Serverprogramm auf der anderen Seite gibt, das die Nachricht entgegennehmen und verwerten kann. Beachten Sie, dass UDP verbindungslos ist und sich die Implementation daher etwas vom Flussdiagramm eines Servers aus Abschnitt 20.1.1, »Client-Server-Systeme«, unterscheidet.

Der Quelltext des Servers sieht folgendermaßen aus:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

try:
    s.bind(("", 50000))
    while True:
        daten, addr = s.recvfrom(1024)
        print("[{0}] {1}".format(addr[0], daten.decode()))
finally:
    s.close()
```

Auch hier wird zunächst eine Socket-Instanz erstellt. In der darauffolgenden `try/finally`-Anweisung wird dieser Socket durch Aufruf der Methode `bind` an eine

Adresse gebunden. Beachten Sie, dass diese Methode ein Adressobjekt als Parameter übergeben bekommt. Immer wenn im Zusammenhang mit Sockets von einem Adressobjekt die Rede ist, ist damit schlicht ein Tupel mit zwei Elementen gemeint: einer IP-Adresse als String und einer Portnummer als ganze Zahl.

Das Binden eines Sockets an eine Adresse legt fest, über welche interne Schnittstelle der Socket Pakete empfangen kann. Wenn keine IP-Adresse angegeben wurde, bedeutet dies, dass Pakete über alle dem Server zugeordneten Adressen empfangen werden können, beispielsweise also auch über `127.0.0.1` oder `localhost`.

Nachdem der Socket an eine Adresse gebunden worden ist, können Daten empfangen werden. Dazu wird die Methode `recvfrom` (für *receive from*) in einer Endlosschleife aufgerufen. Die Methode wartet so lange, bis ein Paket eingegangen ist, und gibt die gelesenen Daten mitsamt den Absenderinformationen als Tupel zurück. Beachten Sie, dass die empfangenen Daten ebenfalls in Form einer `bytes`-Instanz zurückgegeben werden.

Der Parameter von `recvfrom` kennzeichnet die maximale Paketgröße und sollte eine Zweierpotenz sein.

An dieser Stelle wird auch der Sinn der `try/finally`-Anweisung deutlich: Das Programm wartet in einer Endlosschleife auf eintreffende Pakete und kann daher nur mit einem Programmabbruch durch Tastenkombination, also durch eine `KeyboardInterrupt`-Exception, beendet werden. In einem solchen Fall muss der Socket trotzdem noch mit `close` geschlossen werden.

20.1.3 TCP

TCP (Transmission Control Protocol) ist kein Konkurrenzprodukt zu UDP, sondern füllt mit seinen Möglichkeiten die Lücken auf, die UDP offen lässt. So ist TCP vor allem verbindungsorientiert und zuverlässig. Verbindungsorientiert bedeutet, dass nicht, wie bei UDP, einfach Datenpakete an bestimmte IP-Adressen geschickt werden, sondern dass zuvor eine Verbindung aufgebaut wird und auf Basis dieser Verbindung weitere Operationen durchgeführt werden. Zuverlässig bedeutet, dass es mit TCP nicht, wie bei UDP, vorkommen kann, dass Pakete verlorengehen, fehlerhaft oder in falscher Reihenfolge ankommen. Solche Vorkommnisse korrigiert das TCP-Protokoll intern, indem es beispielsweise unvollständige oder fehlerhafte Pakete neu anfordert.

Aus diesem Grund ist TCP zumeist die erste Wahl, wenn es um eine Netzwerkschnittstelle geht. Bedenken Sie aber unbedingt, dass jedes Paket, das neu angefordert werden muss, Zeit kostet und die Latenz der Verbindung somit steigen

kann. Außerdem sind fehlerhafte Übertragungen in einem LAN äußerst selten, weswegen Sie gerade dort die Performance von UDP und die Verbindungsqualität von TCP gegeneinander abwägen sollten.

Im Folgenden wird auch die Verwendung von TCP anhand eines kleinen Beispielprojekts erläutert: Es soll ein rudimentäres Chatprogramm entstehen, bei dem der Client eine Nachricht an den Server sendet, auf die der Server wieder antworten kann. Die Kommunikation soll also immer abwechselnd erfolgen. Der Quelltext des Servers sieht folgendermaßen aus:

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(("", 50000))
s.listen(1)

try:
    while True:
        komm, addr = s.accept()
        while True:
            data = komm.recv(1024)

            if not data:
                komm.close()
                break

            print("[{0}] {1}".format(addr[0], data.decode()))
            nachricht = input("Antwort: ")
            komm.send(bytes(nachricht, "utf-8"))
finally:
    s.close()
```

Bei der Erzeugung des Verbindungssockets unterscheidet sich TCP von UDP nur in den zu übergebenden Werten. In diesem Fall wird `AF_INET` für das IPv4-Protokoll und `SOCK_STREAM` für die Verwendung von TCP übergeben. Damit ist allerdings nur der Socket in seiner Rohform instantiiert. Auch bei TCP muss der Socket an eine IP-Adresse und einen Port gebunden werden. Beachten Sie, dass `bind` ein Adressobjekt als Parameter erwartet – die Angaben von IP-Adresse und Port also noch in ein Tupel gefasst sind. Auch hier werden wieder alle IP-Adressen des Servers genutzt.

Danach wird der Server durch Aufruf der Methode `listen` in den passiven Modus geschaltet und instruiert, nach Verbindungsanfragen zu horchen. Beachten Sie, dass diese Methode noch keine Verbindung herstellt. Der übergebene Parameter bestimmt die maximale Anzahl von zu puffernden Verbindungsversuchen und sollte mindestens 1 sein.

In der darauffolgenden Endlosschleife wartet die aufgerufene Methode `accept` des Verbindungssockets nun auf eine eingehende Verbindungsanfrage und akzeptiert diese. Zurückgegeben wird ein Tupel, dessen erstes Element der Kommunikationssocket ist, der zur Kommunikation mit dem verbundenen Client verwendet werden kann. Das zweite Element des Tupels ist das Adressobjekt des Verbindungspartners.

Nachdem eine Verbindung hergestellt wurde, wird eine zweite Endlosschleife eingeleitet, deren Schleifenkörper im Prinzip aus zwei Teilen besteht: Zunächst wird immer eine Nachricht per `komm.recv` vom Verbindungspartner erwartet und ausgegeben. Sollte von `komm.recv` ein leerer String zurückgegeben werden, so bedeutet dies, dass der Verbindungspartner die Verbindung beendet hat. In einem solchen Fall wird die innere Schleife abgebrochen. Wenn eine wirkliche Nachricht angekommen ist, erlaubt es der Server dem Benutzer, eine Antwort einzugeben, und verschickt diese per `komm.send`.

Jetzt soll der Quelltext des Clients besprochen werden:

```
import socket

ip = input("IP-Adresse: ")
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, 50000))

try:
    while True:
        nachricht = input("Nachricht: ")
        s.send(bytes(nachricht, "utf-8"))
        antwort = s.recv(1024)
        print("[{0}] {1}".format(ip, antwort.decode()))
finally:
    s.close()
```

Auf der Clientseite wird der instantiierte Socket `s` durch Aufruf der Methode `connect` mit dem Verbindungspartner verbunden. Die Methode `connect` verschickt genau die Verbindungsanfrage, die beim Server durch `accept` akzeptiert werden kann. Wenn die Verbindung abgelehnt wurde, wird eine Exception geworfen.

Die nachfolgende Endlosschleife funktioniert ähnlich wie die des Servers, mit dem Unterschied, dass zuerst eine Nachricht eingegeben und abgeschickt und danach auf eine Antwort des Servers gewartet wird. Damit wären Client und Server in einen Rhythmus gebracht, bei dem der Server immer dann auf eine Nachricht wartet, wenn beim Client eine eingegeben wird und umgekehrt.

Genau dieser Rhythmus ist aber auch der größte Knackpunkt des Beispielpjekts, denn es ist für einen der Kommunikationspartner schlicht unmöglich, zwei

Nachrichten direkt hintereinander abzusetzen. Für den praktischen Einsatz hätte das Programm also allenfalls Unterhaltungswert. Das Ziel war es auch nicht, eine möglichst perfekte Chat-Applikation zu schreiben, sondern eine einfache und kurze Beispielimplementierung einer Client-Server-Kommunikation über TCP zu erstellen.

Betrachten Sie es also als Herausforderung, Client und Server, beispielsweise durch Threads, zu einem brauchbaren Chat-Programm zu erweitern. Das könnte so aussehen, dass ein Thread jeweils `s.recv` abhört und eingehende Nachrichten anzeigt und ein zweiter Thread es ermöglicht, dass die Benutzer Nachrichten per `input` eingeben, und diese dann verschickt.

20.1.4 Blockierende und nicht-blockierende Sockets

Wenn ein Socket erstellt wird, befindet er sich standardmäßig im sogenannten *blockierenden Modus* (engl. *blocking mode*). Das bedeutet, dass alle Methodenaufrufe warten, bis die von ihnen angestoßene Operation durchgeführt wurde. So würde ein Aufruf der Methode `recv` eines Sockets so lange das komplette Programm blockieren, bis tatsächlich Daten eingegangen sind und aus dem internen Puffer des Sockets gelesen werden können.

In vielen Fällen ist dieses Verhalten durchaus gewünscht, doch möchte man bei einem Programm, in dem viele verbundene Sockets verwaltet werden, beispielsweise nicht, dass einer dieser Sockets mit seiner `recv`-Methode das komplette Programm blockiert, nur weil noch keine Daten eingegangen sind, während an einem anderen Socket Daten zum Lesen bereitstehen. Um solche Probleme zu umgehen, lässt sich der Socket in den sogenannten *nicht-blockierenden Modus* (engl. *non-blocking mode*) versetzen. Dies wirkt sich folgendermaßen auf diverse Socket-Operationen aus:

- ▶ Die Methoden `recv` und `recvfrom` des Socket-Objekts geben nur noch ankommende Daten zurück, wenn sich diese bereits im internen Puffer des Sockets befinden. Sobald die Methode auf weitere Daten zu warten hätte, wirft sie eine `socket.error`-Exception und gibt damit den Kontrollfluss wieder an das Programm ab.
- ▶ Die Methoden `send` und `sendto` versenden die angegebenen Daten nur, wenn sie direkt in den Ausgangspuffer des Sockets geschrieben werden können. Gelegentlich kommt es vor, dass dieser Puffer voll ist und `send` bzw. `sendto` zu warten hätten, bis der Puffer weitere Daten aufnehmen kann. In einem solchen Fall wird im nicht-blockierenden Modus eine `socket.error`-Exception geworfen und der Kontrollfluss damit an das Programm zurückgegeben.

- Die Methode `connect` sendet eine Verbindungsanfrage an den Zielsocket und wartet nicht, bis diese Verbindung zustande kommt. Wenn `connect` aufgerufen wird und die Verbindungsanfrage noch läuft, wird eine `socket.error-Exception` mit der Fehlermeldung »Operation now in progress« geworfen. Durch mehrmaligen Aufruf von `connect` lässt sich feststellen, ob die Operation immer noch durchgeführt wird.

Alternativ kann im nicht-blockierenden Modus die Methode `connect_ex` für Verbindungsanfragen verwendet werden. Diese wirft keine `socket.error-Exception`, sondern zeigt eine erfolgreiche Verbindung mit einem Rückgabewert von 0 an. Bei echten Fehlern, die bei der Verbindung auftreten, wirft auch `connect_ex` eine Exception.

Ein Socket lässt sich durch Aufruf seiner Methode `setblocking` in den nicht-blockierenden Zustand versetzen:

```
s.setblocking(False)
```

In diesem Fall würden sich Methodenaufrufe des Sockets `s` wie oben beschrieben verhalten. Der Parameter `True` würde den Socket wieder in den ursprünglichen blockierenden Modus versetzen.

Socket-Operationen werden im Falle des blockierenden Modus auch *synchrone Operationen* und im Falle des nicht-blockierenden Modus *asynchrone Operationen* genannt.

Es ist durchaus möglich, auch während des Betriebs zwischen dem blockierenden und dem nicht-blockierenden Modus eines Sockets umzuschalten. So könnten Sie beispielsweise die Methode `connect` blockierend und anschließend die Methode `read` nicht-blockierend verwenden.

20.1.5 Verwendung des Moduls

Da die Funktionen des Moduls oder die Methoden des Socket-Objekts in den beiden vorherigen Abschnitten vielleicht etwas zu kurz gekommen sind, möchten wir an dieser Stelle noch einmal die wichtigsten dieser Funktionen und Methoden auflisten. Wir beginnen mit den Funktionen und Konstanten des Moduls `socket`.

Funktionen

socket.getfqdn([name])

Gibt den vollständigen Domainnamen (FQDN, *Fully qualified Domain Name*) der Domain *name* zurück. Wenn *name* weggelassen wird, wird der vollständige Domainname des lokalen Hosts zurückgegeben.

```
>>> socket.getfqdn()
'HOSTNAME.localdomain'
```

socket.gethostbyname(hostname)

Gibt die IPv4-Adresse des Hosts *hostname* als String zurück.

```
>>> socket.gethostbyname("HOSTNAME")
'192.168.1.23'
```

socket.gethostname()

Gibt den Hostnamen des Systems als String zurück.

```
>>> socket.gethostname()
'HOSTNAME'
```

socket.getservbyname(servicename[, protocolname])

Gibt den Port für den Service *servicename* mit dem Netzwerkprotokoll *protocolname* zurück. Bekannte Services wären beispielsweise "http" oder "ftp" mit den Portnummern 80 bzw. 21. Der Parameter *protocolname* sollte entweder "tcp" oder "udp" sein.

```
>>> socket.getservbyname("http", "tcp")
80
```

socket.getservbyport(port[, protocolname])

Diese Funktion ist das Gegenstück zu `getservbyname`.

```
>>> socket.getservbyport(21)
'ftp'
```

socket.socket([family[, type[, proto]]])

Erzeugt einen neuen Socket. Der erste Parameter *family* kennzeichnet dabei die Adressfamilie und sollte entweder `socket.AF_INET` für den IPv4-Namensraum oder `socket.AF_INET6` für den IPv6-Namensraum sein.

Der zweite Parameter *type* kennzeichnet das zu verwendende Netzwerkprotokoll und sollte entweder `socket.SOCK_STREAM` für TCP oder `socket.SOCK_DGRAM` für UDP sein.

Der optionale Parameter *proto* ist sehr speziell, weswegen wir ihn hier nicht besprechen möchten. Nähere Informationen dazu finden Sie beispielsweise in den Linux Manpages (<http://linux.die.net/man/>) unter dem Stichwort »socket«.

socket.getdefaulttimeout(), socket.setdefaulttimeout(timeout)

Gibt in Form einer Gleitkommazahl die maximale Anzahl an Sekunden zurück, die beispielsweise die Methode `recv` eines Socket-Objekts auf ein eingehendes Paket wartet. Durch die Funktion `setdefaulttimeout` kann dieser Wert für alle neu erzeugten Socket-Instanzen verändert werden.

Die Socket-Klasse

Nachdem durch die Funktion `socket` des Moduls `socket` eine neue Instanz der Klasse `Socket` erzeugt wurde, stellt diese natürlich weitere Funktionalität bereit, um sich mit einem zweiten Socket zu verbinden oder Daten an den Verbindungspartner zu übermitteln. Die Methoden der `Socket`-Klasse sollen im Folgenden beschrieben werden.

Beachten Sie, dass sich das Verhalten der Methoden im blockierenden und nicht-blockierenden Modus unterscheidet. Näheres dazu finden Sie in Abschnitt 20.1.4, »Blockierende und nicht-blockierende Sockets«. Im Folgenden sei `s` eine Instanz der Klasse `socket.Socket`.

s.accept()

Wartet auf eine eingehende Verbindungsanfrage und akzeptiert diese. Die `Socket`-Instanz muss zuvor durch Aufruf der Methode `bind` an eine bestimmte Adresse und einen Port gebunden worden sein und Verbindungsanfragen erwarten. Letzteres geschieht durch Aufruf der Methode `listen`.

Die Methode `accept` gibt ein Tupel zurück, dessen erstes Element eine neue `Socket`-Instanz, auch `Connection`-Objekt genannt, ist, über die die Kommunikation mit dem Verbindungspartner erfolgen kann. Das zweite Element des Tupels ist ein weiteres Tupel, das IP-Adresse und Port des verbundenen Sockets enthält.

Diese Methode ist für TCP gedacht.

s.bind(address)

Bindet den Socket an die Adresse `address`. Der Parameter `address` muss ein Tupel der Form sein, wie es `accept` zurückgibt.

Nachdem ein Socket an eine bestimmte Adresse gebunden wurde, kann er, im Falle von TCP, in den passiven Modus geschaltet werden oder, im Falle von UDP, direkt Datenpakete empfangen.

s.close()

Schließt den Socket. Das bedeutet, dass keine Daten mehr über ihn gesendet oder empfangen werden können.

s.connect(address)

Verbindet zu einem Server mit der Adresse *address*. Beachten Sie, dass dort ein Socket existieren muss, der auf dem gleichen Port auf Verbindungsanfragen wartet, damit die Verbindung zustande kommen kann. Der Parameter *address* muss im Falle des IPv4-Protokolls ein Tupel sein, das aus der IP-Adresse und der Portnummer besteht.

Diese Methode ist für TCP gedacht.

s.connect_ex(address)

Unterscheidet sich von `connect` nur darin, dass im nicht-blockierenden Modus keine Exception geworfen wird, wenn die Verbindung nicht sofort zustande kommt. Der Verbindungsstatus wird über einen ganzzahligen Rückgabewert angezeigt. Ein Rückgabewert von 0 bedeutet, dass der Verbindungsversuch erfolgreich durchgeführt wurde.

Beachten Sie, dass bei echten Fehlern, die beim Verbindungsversuch auftreten, weiterhin Exceptions geworfen werden, beispielsweise wenn der Zielsocket nicht erreicht werden konnte.

Diese Methode ist für TCP gedacht.

s.getpeername()

Gibt das Adressobjekt des mit diesem Socket verbundenen Sockets zurück. Das Adressobjekt ist ein Tupel, das IP-Adresse und Portnummer enthält.

Diese Methode ist für TCP gedacht.

s.getsockname()

Gibt das Adressobjekt zurück, über das dieser Socket mit dem verbundenen Socket kommuniziert.

s.listen(backlog)

Versetzt einen Serversocket in den sogenannten Listen-Modus, das heißt, der Socket achtet auf Sockets, die sich mit ihm verbinden wollen. Nachdem diese Methode aufgerufen worden ist, können eingehende Verbindungswünsche mit `accept` akzeptiert werden.

Der Parameter *backlog* legt die maximale Anzahl an gepufferten Verbindungsanfragen fest und sollte mindestens 1 sein. Den größtmöglichen Wert für *backlog* legt das Betriebssystem fest, meistens liegt er bei 5.

Diese Methode ist für TCP gedacht.

s.recv(bufsize[, flags])

Liest beim Socket eingegangene Daten. Durch den Parameter *bufsize* wird die maximale Anzahl von zu lesenden Bytes festgelegt. Die gelesenen Daten werden in Form eines Strings zurückgegeben.

Über den optionalen Parameter *flags* kann das Standardverhalten von `recv` geändert werden. Diese Veränderungen werden allerdings nur in seltenen Fällen benötigt, weswegen wir sie hier nicht besprechen möchten.¹ Dasselbe gilt für den gleichnamigen Parameter der folgenden Methoden.

Diese Methode ist für TCP gedacht.

s.recvfrom(bufsize[, flags])

Unterscheidet sich von `recv` in Bezug auf den Rückgabewert. Dieser ist bei `recvfrom` ein Tupel, das als erstes Element die gelesenen Daten als String und als zweites Element das Adressobjekt des Verbindungspartners enthält.

Diese Methode ist für UDP gedacht.

s.send(string[, flags])

Sendet den String *string* zum verbundenen Socket. Die Anzahl der gesendeten Bytes wird zurückgegeben. Beachten Sie, dass unter Umständen die Daten nicht vollständig gesendet wurden. In einem solchen Fall ist die Anwendung dafür verantwortlich, die verbleibenden Daten erneut zu senden.

Diese Methode ist für TCP gedacht.

s.sendall(string[, flags])

Unterscheidet sich von `send` darin, dass `sendall` so lange versucht, die Daten zu senden, bis entweder der vollständige Datensatz *string* versendet wurde oder ein Fehler aufgetreten ist. Im Fehlerfall wird eine entsprechende Exception geworfen.

Diese Methode ist für TCP gedacht.

s.sendto(string[, flags], address)

Versendet den Datensatz *string* an einen Socket mit der Adresse *address*. Da der Verbindungspartner explizit angegeben wird, brauchen die beiden Sockets nicht untereinander verbunden zu sein. Der Parameter *address* muss ein Adressobjekt sein.

Diese Methode ist für UDP gedacht.

1 Eine nähere Erläuterung des Parameters *flags* finden Sie zum Beispiel in den Linux Manpages (<http://linux.die.net/man/>) unter dem jeweiligen Methodennamen.

s.setblocking(flag)

Wenn für *flag* `False` übergeben wird, wird der Socket in den nicht-blockierenden Modus versetzt, sonst in den blockierenden Modus.

Im blockierenden Modus warten Methoden wie `send` oder `recv`, bis Daten versendet bzw. gelesen werden konnten. Im nicht-blockierenden Modus würde ein Aufruf von `recv` beispielsweise eine Exception verursachen, wenn keine Daten eingegangen sind, die gelesen werden könnten.

s.settimeout(value), gettimeout()

Setzt einen Timeout-Wert für diesen Socket. Dieser Wert bestimmt im blockierenden Modus, wie lange auf das Eintreffen bzw. Versenden von Daten gewartet werden soll. Dabei können Sie für *value* die Anzahl an Sekunden in Form einer Gleitkommazahl oder `None` übergeben.

Über die Methode `gettimeout` kann der Timeout-Wert ausgelesen werden.

Wenn ein Aufruf von beispielsweise `send` oder `recv` die maximale Wartezeit überschreitet, wird eine `socket.timeout`-Exception geworfen.

20.1.6 Netzwerk-Byte-Order

Das Schöne an standardisierten Protokollen wie TCP oder UDP ist, dass Computer verschiedenster Bauart eine gemeinsame Schnittstelle haben, über die sie miteinander kommunizieren können. Allerdings hören diese Gemeinsamkeiten hinter der Schnittstelle unter Umständen wieder auf. So ist beispielsweise die sogenannte *Byte-Order* ein signifikanter Unterschied zwischen diversen Systemen. Diese Byte-Order legt die Speicherreihenfolge von Zahlen fest, die mehr als ein Byte Speicher benötigen.

Bei der Übertragung von Binärdaten führt es zu Problemen, wenn diese ohne Konvertierung zwischen zwei Systemen mit verschiedener Byte-Order ausgetauscht werden. Das Protokoll TCP garantiert dabei nur, dass die gesendeten Bytes in der Reihenfolge ankommen, in der sie abgeschickt wurden.

Solange Sie sich bei der Netzwerkkommunikation auf reine ASCII-Strings beschränken, können keine Probleme auftreten, da ASCII-Zeichen nie mehr als ein Byte Speicher benötigen. Außerdem sind Verbindungen zwischen zwei Computern derselben Plattform problemlos. So können beispielsweise Binärdaten zwischen zwei x86er-PCs übertragen werden, ohne Probleme befürchten zu müssen.

Allerdings möchte man bei einer Netzwerkverbindung in der Regel Daten übertragen, ohne sich über die Plattform des verbundenen Rechners Gedanken zu machen. Dazu hat man die sogenannte *Netzwerk-Byte-Order* definiert. Das ist die

Byte-Order, die für Binärdaten im Netzwerk zu verwenden ist. Um diese Netzwerk-Byte-Order sinnvoll umzusetzen, enthält das Modul `socket` vier Funktionen, die entweder Daten von der Host-Byte-Order in die Netzwerk-Byte-Order (»hton«) oder umgekehrt (»ntoh«) konvertieren. Die folgende Tabelle listet diese Funktionen auf und erläutert ihre Bedeutung:

Alias	Bedeutung
<code>socket.ntohl(x)</code>	Konvertiert eine 32-Bit-Zahl von der Netzwerk- in die Host-Byte-Order.
<code>socket.ntohs(x)</code>	Konvertiert eine 16-Bit-Zahl von der Netzwerk- in die Host-Byte-Order.
<code>socket.htonl(x)</code>	Konvertiert eine 32-Bit-Zahl von der Host- in die Netzwerk-Byte-Order.
<code>socket.htons(x)</code>	Konvertiert eine 16-Bit-Zahl von der Host- in die Netzwerk-Byte-Order.

Tabelle 20.2 Konvertierung von Binärdaten

Der Aufruf dieser Funktionen ist möglicherweise überflüssig, wenn das entsprechende System bereits die Netzwerk-Byte-Order verwendet. Der gebräuchliche x86er-PC verwendet diese übrigens nicht.

An dieser Stelle möchten wir noch einmal darauf hinweisen, dass eine Konvertierung von Binärdaten in einem professionellen Programm selbstverständlich dazugehört. Solange Sie jedoch im privaten Umfeld kleinere Netzwerkanwendungen schreiben, die Binärdaten ausschließlich zwischen x86er-PCs austauschen, brauchen Sie sich über die Byte-Order keine Gedanken zu machen. Zudem können ASCII-Zeichen, wie gesagt, problemlos auch zwischen Systemen mit verschiedener Byte-Order ausgetauscht werden, so dass auch in diesem Fall keine explizite Konvertierung nötig ist.

20.1.7 Multiplexende Server – select

Ein Server ist in den meisten Fällen nicht dazu gedacht, immer nur einen Client zu bedienen, wie es in den bisherigen Beispielen vereinfacht angenommen wurde. In der Regel muss ein Server eine ganze Reihe von verbundenen Clients verwalten, die sich in verschiedenen Phasen der Kommunikation befinden. Es stellt sich die Frage, wie so etwas sinnvoll in einem Prozess, also ohne den Einsatz von Threads, durchgeführt werden kann.

Selbstverständlich könnte man alle verwendeten Sockets in den nicht-blockierenden Modus schalten und die Verwaltung selbst in die Hand nehmen. Das ist aber nur auf den ersten Blick eine Lösung, denn der blockierende Modus besitzt einen

unschätzbaren Vorteil: Ein blockierender Socket veranlasst, dass das Programm bei einer Netzwerkoperation so lange schlafen gelegt wird, bis die Operation durchgeführt werden kann. Auf diese Weise kann die Prozessorauslastung reduziert werden.

Im Gegensatz dazu müssten wir beim Einsatz von nicht-blockierenden Sockets in einer Schleife ständig über alle verbundenen Sockets iterieren und prüfen, ob sich etwas getan hat, also ob beispielsweise Daten zum Auslesen bereitstehen. Dieser Ansatz, auch *Busy Waiting* genannt, ermöglicht uns zwar das quasi-parallele Auslesen mehrerer Sockets, das Programm lastet den Prozessor aber wesentlich mehr aus, da es über den gesamten Zeitraum aktiv ist.

Das Modul `select` ermöglicht es, im gleichen Prozess mehrere blockierende Sockets zu verwalten, so dass die Vorteile blockierender Sockets erhalten bleiben. Ein Server, der `select` verwendet, wird *multiplexender Server* genannt. Im Modul ist im Wesentlichen die Funktion `select` enthalten, die im Folgenden besprochen werden soll.

`select.select(rlist, wlist, xlist[, timeout])`

Im einfachsten Fall bekommt die Funktion `select` als ersten Parameter `rlist` eine Liste von Sockets übergeben, mit denen eine Leseoperation durchgeführt werden soll. Nehmen wir einmal an, für die weiteren Parameter `wlist` und `xlist` würde jeweils eine leere Liste übergeben. In diesem Fall würde die Funktion `select` das Programm so lange schlafen legen, bis an mindestens einem der übergebenen Sockets Daten vorliegen, die ausgelesen werden können.

Ähnlich verhält es sich mit dem zweiten Parameter, `wlist`. Hier wird eine Liste von Sockets übergeben, mit denen eine Schreiboperation durchgeführt werden soll. Die Funktion `select` weckt das Programm auf, sobald einer der hier übergebenen Sockets zum Schreiben bereit ist.

Für den dritten Parameter, `xlist`, wird eine Liste von Sockets übergeben, bei denen möglicherweise sogenannte *Out-of-Band Data* eingegangen sind. Das sind TCP-Pakete, die als besonders dringend (engl. *urgent*) eingestuft sind und somit privilegiert übertragen werden. Mithilfe solcher Nachrichten kann ein Programm wichtige Ausnahmefälle signalisieren. Dennoch werden wir hier nicht näher darauf eingehen, da solche OOB-Pakete so gut wie nie verwendet werden.

Als vierter, optionaler und letzter Parameter kann ein Timeout-Wert in Sekunden angegeben werden. Dieser veranlasst die Funktion `select`, das Programm nach einer gewissen Zeit aufzuwecken, auch wenn sich bei keinem der übergebenen Sockets etwas getan hat. Wenn ein Timeout-Wert von `0.0` übergeben wird, gibt `select` nur die Sockets zurück, die beim Aufruf schon bereit zum Lesen bzw. Schreiben sind.

Es ist möglich, für *rlist*, *wlist* oder *xlist* leere Listen zu übergeben, vor allem, weil in der Regel nur der erste dieser Parameter benötigt wird, denn es ist das klassische Anwendungsgebiet von `select`, auf eintreffende Daten zu warten. Der zweite Parameter ist deshalb weniger wichtig, weil ein Socket in der Regel zu jeder Zeit zum Versenden von Daten bereitsteht. Und in den seltenen Fällen, bei denen dies nicht der Fall ist, ist die »Verstopfung« des Ausgangspuffers nur von kurzer Dauer und ein blockierender Aufruf von `send` somit nicht weiter tragisch.

Die Funktion `select` gibt in jedem Fall ein Tupel zurück, das aus drei Listen besteht. Diese Listen enthalten jeweils die Sockets, bei denen entweder Daten gelesen oder geschrieben werden können oder, wie erwähnt, dringende Pakete vorliegen. Beachten Sie, dass dieselbe Socket-Instanz beim Aufruf von `select` durchaus in mehreren der übergebenen Listen vorkommen darf.

Im folgenden Beispiel soll ein Server geschrieben werden, der Verbindungen von beliebig vielen Clients akzeptiert und verwaltet. Diese Clients sollen dazu in der Lage sein, dem Server mehrere Nachrichten zu schicken, die von diesem dann am Bildschirm angezeigt werden. Aus Gründen der Einfachheit verzichten wir auf eine Antwortmöglichkeit des Servers.

```
import socket
import select

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("", 50000))
server.listen(1)

clients = []

try:
    while True:
        lesen, schreiben, oob = select.select([server] + clients,
                                              [], [])

        for sock in lesen:
            if sock is server:
                client, addr = server.accept()
                clients.append(client)
                print("+++ Client {} verbunden".format(addr[0]))
            else:
                nachricht = sock.recv(1024)
                ip = sock.getpeername()[0]
                if nachricht:
                    print("[{}] {}1".format(ip,
                                             nachricht.decode()))
```

```

        else:
            print("+++ Verbindung zu {0}"
                  "beendet".format(ip))
            sock.close()
            clients.remove(sock)

    finally:
        for c in clients:
            c.close()
        server.close()

```

Zunächst wird ein Socket `server` erzeugt, der dazu gedacht ist, eingehende Verbindungsanfragen zu akzeptieren. Zudem wird die leere Liste `clients` angelegt, die später alle verbundenen Client-Sockets enthalten soll. Die darauf folgende `try/except`-Anweisung hat die Aufgabe, alle verbundenen Sockets ordnungsgemäß durch Aufruf von `close` zu schließen, wenn das Programm beendet wird.

Viel interessanter ist aber die Endlosschleife innerhalb des `try`-Zweiges, in der zunächst die Funktion `select` aufgerufen wird. Dabei werden alle geöffneten Sockets, inklusive des Serversockets, als erster Parameter übergeben. Die von `select` zurückgegebenen Listen werden von `lesen`, `schreiben` und `oob` referenziert, wobei wir uns nur für die Liste `lesen` näher interessieren.

Nach dem Aufruf von `select` wird über die zurückgegebene Liste `lesen` iteriert und in jedem Iterationsschritt überprüft, ob es sich bei dem betrachteten Socket um den Serversocket handelt. Wenn das der Fall ist, wenn also beim Serversocket Daten zum Einlesen bereitstehen, bedeutet dies, dass eine Verbindungsanfrage vorliegt. Wir akzeptieren die Verbindung, fügen den neuen Socket in die Liste `clients` ein und geben eine entsprechende Meldung aus.

Wenn Daten bei einem Client-Socket eingegangen sind, bedeutet dies, dass entweder eine Nachricht von diesem eingetroffen ist oder dass die Verbindung beendet wurde. Um zu testen, welcher der beiden Fälle eingetreten ist, lesen wir die vorhandenen Daten mit `recv` aus. Wenn die Verbindung seitens des Clients beendet wurde, gibt `recv` einen leeren String zurück. In diesem Fall löschen wir diesen Socket aus der Liste `clients` und geben eine entsprechende Meldung aus.

Der Vollständigkeit halber folgt hier noch der Quelltext des zu diesem Server passenden Clients:

```

import socket

ip = input("IP-Adresse: ")
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, 50000))

```

```

try:
    while True:
        nachricht = input("Nachricht: ")
        s.send(bytes(nachricht, "utf-8"))
finally:
    s.close()

```

Dabei handelt es sich tatsächlich um reine Socket-Programmierung, wie wir sie bereits in den vorherigen Abschnitten behandelt haben. Beachten Sie, dass der Client, abgesehen von eventuell auftretenden Latenzen, nicht bemerkt, ob er von einem seriellen oder einem multiplexenden Server bedient wird.

20.1.8 socketserver

Sie werden festgestellt haben, dass das Schreiben eines Servers unter Verwendung des Moduls `socket` mitunter eine komplexe Aufgabe sein kann. Aus diesem Grund enthält Pythons Standardbibliothek das Modul `socketserver`, das es erleichtern soll, einen Server zu schreiben, der in der Lage ist, mehrere Clients zu bedienen.

Im folgenden Beispiel soll der Chat-Server des vorherigen Abschnitts mit dem Modul `socketserver` nachgebaut werden. Dazu muss zunächst ein sogenannter *Request Handler* erstellt werden. Das ist eine Klasse, die von der Basisklasse `socketserver.BaseRequestHandler` abgeleitet wird. Im Wesentlichen muss in dieser Klasse die Methode `handle` überschrieben werden, in der die Kommunikation mit einem Client ablaufen soll:

```

import socketserver

class ChatRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        addr = self.client_address[0]
        print("[{0}] Verbindung hergestellt".format(addr))
        while True:
            s = self.request.recv(1024)
            if s:
                print("[{0}] {1}".format(addr, s.decode()))
            else:
                print("[{0}] Verbindung"
                      " geschlossen".format(addr))
                break

```

Hier wurde die Klasse `ChatRequestHandler` erzeugt, die von `BaseRequestHandler` erbt. Später erzeugt die `socketserver`-Instanz bei jeder hergestellten Verbindung eine neue Instanz dieser Klasse und ruft die Methode `handle` auf. In dieser Methode läuft dann die Kommunikation mit dem verbundenen Client ab. Zusätzlich zur Methode `handle` können noch die Methoden `setup` und `finish` überschrieben werden, die entweder vor (`setup`) oder nach (`finish`) dem Aufruf von `handle` aufgerufen werden.

In unserem Beispiel werden innerhalb der Methode `handle` in einer Endlosschleife eingehende Daten eingelesen. Wenn ein leerer String eingelesen wurde, wird die Verbindung vom Kommunikationspartner geschlossen. Andernfalls wird der gelesene String ausgegeben.

Damit ist die Arbeit am Request Handler beendet. Was jetzt noch fehlt, ist der Server, der eingehende Verbindungen akzeptiert und daraufhin den Request Handler instantiiert:

```
server = socketserver.ThreadingTCPServer(("", 50000),
                                         ChatRequestHandler)
server.serve_forever()
```

Um den tatsächlichen Server zu erzeugen, erzeugen wird eine Instanz der Klasse `ThreadingTCPServer`. Dem Konstruktor übergeben wir dabei ein Adress-Tupel und die soeben erstellte Request-Handler-Klasse `ChatRequestHandler`. Durch Aufruf der Methode `serve_forever` der `ThreadingTCPServer`-Instanz instruieren wir den Server, eine unbestimmte Anzahl an Verbindungen einzugehen.

Beachten Sie, dass der Programmierer selbst Verantwortung für eventuell von mehreren Threads gemeinsam genutzte Ressourcen trägt. Diese müssen gegebenenfalls durch Critical Sections abgesichert werden.

Neben der Klasse `ThreadingTCPServer` können auch andere Server-Klassen instantiiert werden, je nachdem, wie sich der Server verhalten soll. Die Schnittstelle der Konstruktoren ist immer dieselbe.

socketserver.TCPServer, socketserver.UDPServer

Ein einfacher TCP- bzw. UDP-Server. Beachten Sie, dass diese Server immer nur eine Verbindung gleichzeitig eingehen können. Aus diesem Grund ist die Klasse `TCPServer` für unser Beispielprogramm nicht einsetzbar.

socketserver.ThreadingTCPServer, socketserver.ThreadingUDPServer

Diese Klassen implementieren einen TCP- bzw. UDP-Server, der jede Anfrage eines Clients in einem eigenen Thread behandelt, so dass der Server mit mehre-

ren Clients gleichzeitig in Kontakt sein kann. Die Klasse `ThreadingTCPServer` ist somit ideal für unser obiges Beispiel.

`socketserver.ForkingTCPServer`, `socketserver.ForkingUDPServer`

Diese Klassen implementieren einen TCP- bzw. UDP-Server, der jede Anfrage eines Clients in einem eigenen Prozess behandelt, so dass der Server mit mehreren Clients gleichzeitig in Kontakt sein kann. Beachten Sie dabei, dass die Methode `handle` des Request Handlers in einem eigenen Prozess ausgeführt wird, also nicht auf Instanzen des Hauptprozesses zugreifen kann.

Die Server-Klassen

An dieser Stelle erläutern wir die wichtigsten Attribute und Methoden der eben vorgestellten Server-Klassen. Im Folgenden sei `s` eine Instanz einer solchen Server-Klasse.

`s.address_family`

Dieses Attribut referenziert ein Adress-Tupel, das die IP-Adresse und die Portnummer enthält, auf denen der Server `s` nach eingehenden Verbindungsanfragen horcht.

`s.socket`

Dieses Attribut referenziert die von dem Server verwendete `Socket`-Instanz.

`s.fileno()`

Gibt den Dateideskriptor des Serversockets zurück.

`s.handle_request()`

Instruiert den Server, genau eine Verbindungsanfrage zu akzeptieren und zu behandeln.

`s.serve_forever([poll_intervall])`

Instruiert den Server, eine unbestimmte Anzahl von Verbindungsanfragen zu akzeptieren und zu behandeln. Der Parameter `poll_intervall` bestimmt, in welchem Zeitintervall (in Sekunden) der Server prüfen soll, ob die Methode `shutdown` aufgerufen wurde. Der Parameter ist mit `0.5` vorbelegt.

`s.shutdown()`

Beendet den Server.

Die Klasse `BaseRequestHandler`

Die Klasse `BaseRequestHandler` bietet einige Methoden und Attribute, die Sie überschreiben oder verwenden können. Beachten Sie, dass eine Instanz der Klasse `BaseRequestHandler` immer für einen verbundenen Client zuständig ist. Im Folgenden sei *rh* eine Instanz der Klasse `BaseRequestHandler`.

rh.request

Über das Attribut `request` können Sie Informationen über die aktuelle Anfrage eines Clients herausfinden. Bei einem TCP-Server referenziert `request` die Socket-Instanz, die zur Kommunikation mit dem Client verwendet wird. Mit ihr können Daten gesendet oder empfangen werden. Bei Verwendung des verbindungslosen UDP-Protokolls referenziert `request` einen String, der die gesendeten Daten enthält.

rh.client_address

Das Attribut `client_address` referenziert ein Adress-Tupel, das die IP-Adresse und die Portnummer des Clients enthält, dessen Anfrage mit dieser `BaseRequestHandler`-Instanz behandelt wird.

rh.server

Das Attribut `server` referenziert den verwendeten Server, also eine Instanz der Klassen `TCPServer`, `UDPServer`, `ThreadingTCPServer`, `ThreadingUDPServer`, `ForkingTCPServer` oder `ForkingUDPServer`.

rh.handle()

Diese Methode sollte überschrieben werden. Wenn der Server eine Verbindungsanfrage eines Clients akzeptiert hat, wird eine neue Instanz der Request-Handler-Klasse erzeugt und diese Methode aufgerufen.

rh.setup()

Diese Methode kann überschrieben werden und wird stets vor dem Aufruf von `handle` aufgerufen.

rh.finish()

Diese Methode kann überschrieben werden und wird stets nach dem Aufruf von `handle` aufgerufen.

20.2 URLs

Das Paket `urllib` bietet eine komfortable Schnittstelle zum Umgang mit Ressourcen im Internet. Dazu enthält `urllib` die folgenden Module:

Modul	Beschreibung
<code>urllib.request</code>	Enthält Funktionen und Klassen zum Zugriff auf eine Ressource im Internet.
<code>urllib.response</code>	Enthält die von im <code>urllib</code> Paket verwendeten speziellen Datentypen.
<code>urllib.parse</code>	Enthält Funktionen zum komfortablen Einlesen, Verarbeiten und Erstellen von URLs.
<code>urllib.error</code>	Enthält die im <code>urllib</code> Paket verwendeten Exception-Klassen.
<code>urllib.robotparser</code>	Enthält eine Klasse, die die <i>robots.txt</i> -Datei einer Website interpretiert.

Tabelle 20.3 Module des Pakets `urllib`

In den folgenden Abschnitten sollen die Module `request` und `parse` des Pakets `urllib` erläutert werden

20.2.1 Zugriff auf Ressourcen im Internet – `urllib.request`

Das Modul `urllib.request` bietet eine komfortable Schnittstelle, um auf Dateien im Internet zuzugreifen. Die zentrale Funktion dieser Bibliothek ist `urlopen`, die der Funktion `open` ähnelt, bis auf die Tatsache, dass statt eines Dateinamens eine URL übergeben wird. Außerdem können auf dem resultierenden Dateiobjekt aus naheliegendem Grunde keine Schreiboperationen durchgeführt werden.

Um die Beispiele nachzuvollziehen, muss das Modul `request` des Pakets `urllib` eingebunden werden:

```
import urllib.request
```

Im Folgenden sollen die wichtigsten im Modul `urllib.request` enthaltenen Funktionen detailliert besprochen werden.

`urllib.request.urlopen(url[, data][, timeout])`

Die Funktion `urlopen` greift auf die durch *url* adressierte Netzwerkressource zu und gibt ein geöffnetes Dateiobjekt auf dieser Ressource zurück. Damit ermöglicht die Funktion es beispielsweise, den Quelltext einer Website herunterzuladen und wie eine lokale Datei einzulesen.

Wenn bei der URL kein Protokoll wie beispielsweise `http://` oder `ftp://` angegeben wurde, wird angenommen, dass die URL auf eine Ressource der lokalen Festplatte verweist. Für Zugriffe auf die lokale Festplatte können Sie außerdem das Protokoll `file://` angeben.

Wenn kein Zugriff auf die Ressource erlangt werden kann, weil die Ressource beispielsweise nicht existiert oder der entsprechende Server nicht erreichbar ist, wird eine `IOError`-Exception geworfen.

Das von der Funktion `urlopen` zurückgegebene Dateiojekt ist ein dateiähnliches Objekt (engl. *file-like object*), da es nur eine Untermenge der Funktionalität eines echten Dateiojekts bereitstellt. Die folgende Tabelle zeigt die verfügbaren Methoden des zurückgegebenen dateiähnlichen Objekts mit einer kurzen Beschreibung.

Methode	Beschreibung
<code>read([size])</code>	Liest <code>size</code> Byte aus der Ressource aus. Wenn <code>size</code> nicht angegeben wurde, wird der komplette Inhalt ausgelesen. Die gelesenen Daten werden als String zurückgegeben.
<code>readline([size])</code>	Liest eine Zeile aus der Ressource aus. Wenn <code>size</code> angegeben wurde, werden maximal <code>size</code> Byte gelesen. Die gelesenen Daten werden als String zurückgegeben.
<code>readlines([sizehint])</code>	Liest die Ressource zeilenweise aus und gibt sie in Form einer Liste von Strings zurück. Wird <code>sizehint</code> angegeben, so werden Zeilen nur so lange eingelesen, bis die Gesamtgröße der gelesenen Zeilen <code>sizehint</code> überschreitet.
<code>fileno()</code>	Gibt den Dateideskriptor der geöffneten Ressource als ganze Zahl zurück.
<code>close()</code>	Schließt das geöffnete Objekt. Nach Aufruf dieser Methode sind keine weiteren Operationen mehr möglich.
<code>info()</code>	Gibt ein dictionary-ähnliches Objekt zurück, das Metainformationen der heruntergeladenen Seite enthält. Im Anschluss an diese Tabelle werden wir uns eingehend mit der von <code>info</code> zurückgegebenen Instanz beschäftigen.
<code>geturl()</code>	Gibt einen String mit der URL der Ressource zurück.

Tabelle 20.4 Methoden des dateiähnlichen Objekts

Die Methode `info` des von `urlopen` zurückgegebenen dateiähnlichen Objekts stellt eine Instanz bereit, die verschiedene Informationen über die Netzwerkressource enthält. Auf diese Informationen kann wie bei einem Dictionary zugegriffen werden. Dazu folgendes Beispiel:

```
>>> f = urllib.request.urlopen("http://www.galileo-press.de")
>>> d = f.info()
>>> d
<http.client.HTTPMessage object at 0xb7689e8c>
>>> d.keys()
['Date', 'Server', 'Content-Length', 'Content-Type',
'Cache-Control', 'Expires', 'Connection']
```

Im Beispiel wurde auf die Internetressource *http://www.galileo-press.de* zugegriffen und durch Aufruf der Methode `info` das dictionary-ähnliche Objekt erzeugt, das Informationen zu der Website enthält. Durch die Methode `keys` eines Dictionaries lassen sich alle enthaltenen Schlüssel anzeigen. Welche Informationen enthalten sind, hängt vom verwendeten Protokoll ab. Beim HTTP-Protokoll enthält das dictionary-ähnliche Objekt alle vom Server gesendeten Informationen. So können Sie beispielsweise über die Schlüssel "Content-Length" und "Server" die Größe der heruntergeladenen Datei in Byte bzw. den Identifikationsstring der Serversoftware auslesen:

```
>>> d["Content-Length"]
'27395'
>>> d["Server"]
'Zope/(Zope 2.7.6-final, python 2.3.5, linux2) ZServer/1.1'
```

Beachten Sie, dass es sich bei dem in diesem Fall verwendeten Server *Zope* um einen Webserver für Python handelt.

Wenn das verwendete Protokoll `http` ist, dient der optionale Parameter *data* dazu, POST-Parameter an die Ressource zu übermitteln. Für den Parameter *data* müssen diese POST-Werte speziell aufbereitet werden. Dazu wird die Funktion `urlencode` des Moduls `urllib.parse` verwendet:

```
>>> prm = urllib.parse.urlencode({"prm1" : "wert1",
                                   "prm2" : "wert2"})
>>> f = urllib.request.urlopen("http://www.beispiel.de", prm)
```

Näheres zur Funktion `urlencode` finden Sie weiter unten im Zusammenhang mit dem Modul `urllib.parse`.

Beachten Sie, dass neben POST eine weitere Methode zur Parameterübergabe an eine Website existiert: GET. Bei GET werden die Parameter direkt in die URL geschrieben:

```
>>> f = urllib.request.urlopen("http://www.beispiel.de?prm=wert")
```

Über den dritten optionalen Parameter *timeout* wird der Timeout festgelegt, der beim Zugriff auf eine Internetressource berücksichtigt werden soll, das heißt die Zeit, die `urlopen` auf eine Antwort des Servers wartet, bis er als unerreichbar ver-

standen wird. Wenn dieser Parameter nicht übergeben wird, wird ein Standardwert als Timeout verwendet.

urllib.request.urlretrieve(url[, filename[, reporthook[, data]])

Macht den Inhalt der Ressource, auf die die URL *url* verweist, unter einem lokalen Dateinamen verfügbar. Dazu wird der Inhalt der Ressource heruntergeladen oder kopiert, sofern dies notwendig ist. Wenn sich die Ressource bereits auf der lokalen Festplatte befindet, wird sie nicht kopiert. Die Funktion `urlretrieve` gibt ein Tupel mit zwei Elementen zurück: dem Dateinamen der lokalen Datei und dem Rückgabewert der `info`-Methode des dateiähnlichen Objekts:

```
>>> urllib.request.urlretrieve("http://www.galileo-press.de")
('/tmp/tmpYger_7',
<http.client.HTTPMessage object at 0xb768a30c>)
```

Normalerweise werden heruntergeladene Ressourcen als temporäre Dateien im entsprechenden Verzeichnis des Betriebssystems gespeichert. Durch Angabe eines Dateinamens als zweiten Parameter können Sie jedoch festlegen, wohin die heruntergeladene Ressource kopiert werden soll. Wenn dieser Parameter angegeben wurde, werden auch lokale Ressourcen kopiert.

Als dritter Parameter kann ein Funktionsobjekt übergeben werden. Diese Funktion wird aus `urlretrieve` heraus einmal aufgerufen, wenn die Verbindung zur Netzwerkressource hergestellt wurde, und dann öfter, wenn ein Block der Ressource heruntergeladen wurde. Der Callback-Funktion werden drei Parameter übergeben: die Anzahl der bisher übertragenen Blöcke, die Größe eines Blocks in Byte und die Gesamtgröße der Ressource in Byte.

Der vierte Parameter, *data*, entspricht dem Parameter *data* der Funktion `urlopen` und wird auch so verwendet.

urllib.request.FancyURLopener([proxies], **args)

Die bisher besprochenen Funktionen des Moduls `urllib.request` eignen sich für den schnellen Zugriff auf Ressourcen über eine URL, bieten aber wenig Tiefgang. Was ist beispielsweise, wenn der Server eine Authentifizierung verlangt oder die Verbindung über einen Proxy laufen soll? Für solche Fälle kann eine Instanz der Klasse `FancyURLopener` erzeugt werden. Diese stellt ein Interface bereit, das dem des Moduls `urllib.request` ähnelt. So verfügt die Instanz über die Methoden `open` und `retrieve`, die analog zu den gleichnamigen Funktionen aus `urllib.request` funktionieren.

Über den optionalen Parameter *proxies* können Sie Proxy-Server angeben, über die die Verbindung laufen soll. Dabei müssen Sie ein Dictionary übergeben, dessen Schlüssel-Wert-Paare jeweils ein Protokoll und einen Proxy-Server einander zu-

ordnen. Das Dictionary {"http": "http://proxy.beispiel.de:8080/"} würde beispielsweise den Server proxy.beispiel.de als http-Proxy einrichten.

Zusätzlich können Sie bei der Instantiierung von `FancyURLopener` die Schlüsselwortparameter `key_file` und `cert_file` übergeben, die Schlüssel und Zertifikat bei einer SSL-verschlüsselten Verbindung beschreiben.

Im Allgemeinen interessanter ist jedoch die Möglichkeit von `FancyURLopener`, auch mit Servern zu kommunizieren, die eine Authentifikation verlangen. Dazu folgendes Beispiel:

```
import urllib.request

class MyURLopener(urllib.request.FancyURLopener):
    def __init__(self, proxies=None, **args):
        urllib.request.FancyURLopener.__init__(self, proxies,
                                                **args)

    def prompt_user_passwd(self, host, realm):
        return ("username", "password")

opener = MyURLopener()
f = opener.open("http://www.beispiel.de")
print(f.read())
```

Es wurde eine eigene, von `FancyURLopener` abgeleitete Klasse `MyURLopener` erstellt, die, abgesehen vom Konstruktor, nur die Methode `prompt_user_passwd` von `FancyURLopener` überschreibt. Diese Methode wird bei einer Anfrage des Servers nach Benutzernamen und Passwort gerufen und soll diese beiden Informationen in Form eines Tupels zurückgeben. Die Methode bekommt dabei den Host und den Authentication Realm der aufgebauten Verbindung übergeben.

20.2.2 Verarbeiten einer URL – `urllib.parse`

Das Modul `urllib.parse` enthält Funktionen, die es ermöglichen, eine URL komfortabel in ihre Bestandteile zu zerlegen oder diese wieder zu einer gültigen URL zusammenzufügen.

Um die Beispiele ausführen zu können, muss zuvor das Modul `urllib.parse` eingebunden worden sein:

```
>>> import urllib.parse
```

`urllib.parse.urlparse(urlstring[, default_scheme[, allow_fragments]])`

Die Funktion `urlparse` liest die URL *urlstring* ein und bricht sie in mehrere Teile auf. Dabei kann eine URL grundsätzlich aus sechs Teilen bestehen:

`scheme://netloc/path;params?query#fragment`

Der `netloc`-Bereich der URL wird außerdem in vier weitere Bereiche unterteilt:

`username:password@host:port`

Die meisten der angegebenen URL-Teile sind optional und können in URLs weggelassen werden.

Die sechs Bestandteile der URL werden in Form eines tupelähnlichen Objekts mit sechs Elementen zurückgegeben. Diese am meisten verwendeten Teile der URL lassen sich wie bei einem echten Tupel über die Indizes 0 bis 5 ansprechen. Zusätzlich – und das unterscheidet die zurückgegebene Instanz von einem Tupel – kann auf alle Teile der URL über Attribute der Instanz zugegriffen werden. Beachten Sie, dass Sie über Attribute auch auf die vier Unterbereiche des `netloc`-Teils zugreifen können, die nicht über einen Index erreichbar sind.

Die folgende Tabelle listet alle Attribute des Rückgabewertes der Funktion `url-parse` auf und erläutert sie jeweils mit einem kurzen Satz. Zusätzlich ist der entsprechende Index angegeben, sofern sich das entsprechende Attribut auch über einen Index ansprechen lässt. Die Attributnamen entsprechen den Namen der Bereiche, wie sie in den obigen URL-Beispielen verwendet wurden.

Attribut	Index	Beschreibung
<code>scheme</code>	0	das Protokoll der URL, beispielsweise <code>http</code> oder <code>file</code>
<code>netloc</code>	1	Die <i>Network Location</i> besteht üblicherweise aus einem Domainnamen mit Subdomain und TLD, z.B. <code>www.galileo-press.de</code> . Optional können auch Benutzername, Passwort und Portnummer in <code>netloc</code> enthalten sein.
<code>path</code>	2	eine Pfadangabe, die einen Unterordner der Network Location kennzeichnet
<code>params</code>	3	Parameter für das letzte Element des Pfades
<code>query</code>	4	Über den <i>Query-String</i> können zusätzliche Informationen an ein serverseitiges Script übertragen werden.
<code>fragment</code>	5	Das Fragment, auch <i>Anker</i> genannt. Ein geläufiges Beispiel für einen Anker ist eine Sprungmarke innerhalb einer HTML-Datei.
<code>username</code>		der in der URL angegebene Benutzername, sofern vorhanden
<code>password</code>		Das in der URL angegebene Passwort, sofern vorhanden.
<code>hostname</code>		der Domainname der URL, z.B. <code>www.galileo-press.de</code>
<code>port</code>		die in der URL angegebene Portnummer, sofern vorhanden

Tabelle 20.5 Teile einer URL

Über den optionalen Parameter *default_scheme* ist es möglich, ein Protokoll anzugeben, das in die resultierende Instanz eingetragen wird, wenn in der URL kein Protokoll angegeben wurde.

Der optionale Parameter *allow_fragments* legt fest, ob Fragmente, auch *Anker* genannt, in der URL vorkommen dürfen. Wenn hier `False` übergeben wird, referenziert das Attribut `fragment` der zurückgegebenen Instanz immer `None`, egal, ob ein Fragment übergeben wurde oder nicht. Der Parameter ist mit `True` vorbelegt.

Im folgenden Beispiel soll die URL

```
http://www.beispiel.de/pfad/zur/datei.py?prm=abc
```

in ihre Bestandteile zerlegt werden:

```
>>> url = "http://www.beispiel.de/pfad/zur/datei.py?prm=abc"
>>> teile = urllib.parse.urlparse(url)
>>> teile.scheme
'http'
>>> teile.netloc
'www.beispiel.de'
>>> teile.path
'/pfad/zur/datei.py'
>>> teile.params
''
>>> teile.query
'prm=abc'
>>> teile.fragment
''
>>> teile.hostname
'www.beispiel.de'
```

**`urllib.parse.parse_qs(qs[, keep_blank_values[, strict_parsing]]),`
`urllib.parse.parse_qsl(qs[, keep_blank_values[, strict_parsing]])`**

Die Funktion `parse_qs` ermöglicht das Zerlegen des Query-Strings einer URL in seine Bestandteile. Die im Query-String enthaltenen Schlüssel und Werte werden zu einem Dictionary aufbereitet und zurückgegeben. Die Funktion `parse_qsl` funktioniert ganz ähnlich, gibt die aufbereiteten Daten jedoch nicht als Dictionary, sondern als Liste von Schlüssel-Wert-Paaren zurück. Dazu folgendes Beispiel:

```
>>> url = "http://www.beispiel.de?hallo=welt&bla=blubb&xyz=12"
>>> teile = urllib.parse.urlparse(url)
>>> urllib.parse.parse_qs(teile.query)
{'hallo': ['welt'], 'xyz': ['123'], 'bla': ['blubb']}
```

```
>>> urllib.parse.parse_qs(teile.query)
[('hallo', 'welt'), ('bla', 'blubb'), ('xyz', '123')]
```

Über den mit `False` vorbelegten Parameter *keep_blank_values* lässt sich steuern, ob Schlüssel-Wert-Paare mit einem leeren Wert in das Ergebnis mit aufgenommen werden sollen oder nicht. Der dritte Parameter *strict_parsing* legt fest, ob kleinere Fehler im Query-String toleriert werden oder zu einer `ValueError-Exception` führen sollen. Er ist mit `False` vorbelegt.

urllib.parse.urlunparse(parts)

Die Funktion `urlunparse` ist das Gegenstück zu `urlparse`. Sie erzeugt aus einem Tupel mit sechs Elementen einen URL-String. Statt eines reinen Tupels können Sie ein beliebiges iterierbares Objekt mit sechs Elementen, unter anderem beispielsweise auch der Rückgabewert von `urlparse`, übergeben.

```
>>> url = ("http", "beispiel.de", "/pfad/datei.py", "", "", "")
>>> urllib.parse.urlunparse(url)
'http://beispiel.de/pfad/datei.py'
```

Beachten Sie, dass der Ausdruck

```
urllib.parse.urlunparse(urllib.parse.urlparse(url)) == url
```

nicht immer `True` ergibt, da überflüssige Angaben, wie beispielsweise ein leeres Fragment am Ende einer URL, beim Aufruf von `urlparse` verlorengehen.

urllib.parse.urlsplit(urlstring[, default_scheme[, allow_fragments]])

Die Funktion `urlsplit` funktioniert ähnlich wie `urlparse`, mit dem Unterschied, dass das Attribut `params` in der zurückgegebenen Instanz nicht vorhanden ist. Die Parameter werden dem Pfad zugeordnet und sind damit im Attribut `path` enthalten. Die Funktion `urlsplit` sollte dann verwendet werden, wenn die neuere URL-Syntax erlaubt sein soll, die es ermöglicht, Parameter an jedes Element des Pfades anzuhängen.

Ansonsten ist die Schnittstelle von `urlsplit` mit der von `urlparse` identisch.

urllib.parse.urlunsplit(parts)

Die Funktion `urlunsplit` ist das Gegenstück zu `urlsplit` und funktioniert damit genauso wie `urlunparse` in Bezug auf `urlparse`.

urllib.parse.urljoin(base, url[, allow_fragments])

Die Funktion `urljoin` kombiniert die Basis-URL *base* und die relative URL *url* zu einer absoluten Pfadangabe. Der optionale Parameter *allow_fragments* hat dieselbe Bedeutung wie bei `urlparse`.


```
>>> base = "http://www.test.de"
>>> relativ = "pfad/zur/datei.py"
>>> urllib.parse.urljoin(base, relativ)
'http://www.test.de/pfad/zur/datei.py'
>>> base = "http://www.test.de/hallo/welt.py"
>>> relativ = "du.py"
>>> urllib.parse.urljoin(base, relativ)
'http://www.test.de/hallo/du.py'
```

Beachten Sie, dass `urljoin` die beiden übergebenen Pfade nicht einfach aneinanderhängt, sondern, wie im Beispiel zu sehen ist, Dateinamen am Ende der Basis-URL abschneidet.

`urllib.parse.urldefrag(url)`

Die Funktion `urldefrag` spaltet den Anker einer URL, sofern vorhanden, von der URL selbst ab. Die Funktion gibt ein Tupel zurück, dessen erstes Element die URL abzüglich des Ankers ist. Der Anker selbst ist als zweites Element im Tupel enthalten.

```
>>> urllib.parse.urldefrag("http://www.test.de#frag")
('http://www.test.de', 'frag')
```

`urllib.parse.quote(string[, safe[, encoding[, errors]]])`

Ersetzt Sonderzeichen, die in einer URL nicht als solche vorkommen dürfen, durch Escape-Sequenzen der Form `%xx`, wie sie in URLs erlaubt sind. Durch den optionalen Parameter *safe*, einen String, geben Sie Zeichen an, die nicht in eine Escape-Sequenz umgewandelt werden sollen.

```
>>> urllib.parse.quote("www.test.de/hallo welt.html")
'www.test.de/hallo%20welt.html'
```

Über die optionalen Parameter *encoding* und *errors* legen Sie fest, wie mit Unicode-Zeichen im String *string* zu verfahren ist. Die Parameter haben dabei die gleiche Bedeutung wie bei der `encode`-Methode eines Strings und sind mit "utf-8" bzw. "strict" vorbelegt.

`urllib.parse.quote_plus(string[, safe[, encoding[, errors]]])`

Die Funktion `quote_plus` funktioniert wie `quote`, mit dem Unterschied, dass ein Leerzeichen in *string* durch ein `+` ersetzt wird. Dies ist insbesondere im Zusammenhang mit HTML-Formulardaten interessant.

`urllib.parse.quote_from_bytes(bytes[, safe])`

Funktioniert wie `quote`, mit dem Unterschied, dass der betrachtete Text als `bytes`-Instanz und nicht als String vorliegt. Damit erübrigen sich auch die beiden zusätzlichen Parameter der Funktion `quote`.

urllib.parse.unquote(string[, encoding[, errors]])

Die Funktion `unquote` ist das Gegenstück von `quote`. Escape-Sequenzen der Form `%xx` im String *string* werden durch das Sonderzeichen ersetzt, und der resultierende String wird zurückgegeben.

```
>>> urllib.parse.unquote("www.test.de/hallo%20welt.html")
'www.test.de/hallo welt.html'
```

Beachten Sie, dass auch hier ganz analog die Funktionen `unquote_plus` und `unquote_to_bytes` als Gegenstücke der zuvor besprochenen Funktionen `quote_plus` bzw. `quote_from_bytes` existieren.

urllib.parse.urlencode(query[, doseq])

Erzeugt aus den Schlüssel-Wert-Paaren des Dictionarys *query* einen String des folgenden Formats:

```
>>> urllib.parse.urlencode({"abc" : 1, "def" : "ghi"})
'abc=1&def=ghi'
```

Ein solcher String enthält Parameter, die per POST oder GET an ein serverseitiges Script übergeben werden können. Der Rückgabewert der Funktion `urlencode` kann als Parameter *data* der Funktionen `urlopen` und `urlretrieve` übergeben werden.

Wenn das übergebene Dictionary Sequenzen als Werte enthält und der optionale Parameter *doseq* `True` ist, werden diese Sequenzen zu eigenen Schlüssel-Wert-Paaren aufgebrochen. Dabei wird als Parametername der Schlüssel der jeweiligen Sequenz im Dictionary verwendet:

```
>>> urllib.urlencode({"abc" : [1,2,3], "def" : "ghi"}, True)
'abc=1&abc=2&abc=3&def=ghi'
>>> urllib.urlencode({"abc" : [1,2,3], "def" : "ghi"}, False)
'abc=%5B1%2C2%2C3%5D&def=ghi'
```

Wenn für *doseq* `False` übergeben wird, werden eventuell vorhandene Sequenzen als Text in den Parameterstring eingetragen.

20.3 FTP – ftplib

Das Modul `ftplib` ermöglicht es einer Anwendung, sich mit einem FTP-Server zu verbinden und Operationen auf diesem durchzuführen. *FTP* steht für *File Transfer Protocol* und bezeichnet ein Netzwerkprotokoll, das für Dateiübertragungen in TCP/IP-Netzwerken entwickelt wurde. Gerade im Internet ist FTP sehr verbreitet. So erfolgen beispielsweise Dateiübertragungen auf einen Webserver üblicherweise via FTP.

Das Protokoll FTP ist sehr einfach aufgebaut und besteht aus einer umfangreichen Anzahl von Befehlen, die auch von Menschen gelesen werden können. Im Prinzip könnte man also auch direkt mit dem FTP-Server kommunizieren, ohne eine abstrahierende Bibliothek zwischenschalten. Die folgende Tabelle listet die wichtigsten FTP-Befehle auf und erläutert kurz ihre Bedeutung. Sie werden sehen, dass sich das Modul `ftplib` sehr stark an diese Befehle anlehnt und man deshalb gut beraten ist, sich zumindest einen Überblick über die FTP-Befehle zu verschaffen.

Befehl	Beschreibung
OPEN	Baut eine Verbindung zu einem FTP-Server auf.
USER	Überträgt einen Benutzernamen zum Login an den FTP-Server.
PASS	Überträgt ein Passwort zum Login an den FTP-Server.
CWD	Ändert das aktuelle Arbeitsverzeichnis auf dem FTP-Server. (CWD steht für <i>change working directory</i> .)
PWD	Gibt das aktuelle Arbeitsverzeichnis auf dem FTP-Server zurück. (PWD steht für <i>print working directory</i> .)
DELE	Löscht eine Datei auf dem FTP-Server. (DELE steht für <i>delete</i> .)
LIST	Überträgt eine Liste aller im Arbeitsverzeichnis enthaltenen Dateien und Ordner. Die Liste wird über den Datenkanal übermittelt.
MKD	Erstellt ein Verzeichnis auf dem FTP-Server. (MKD steht für <i>make directory</i> .)
RMD	Löscht ein Verzeichnis auf dem FTP-Server. (RMD steht für <i>remove directory</i> .)
RETR	Überträgt eine Datei vom FTP-Server. (RETR steht für <i>retrieve</i> .)
STOR	Überträgt eine Datei vom Client an den FTP-Server. (STOR steht für <i>store</i> .)
QUIT	Beendet die Verbindung zwischen Server und Client.

Tabelle 20.6 FTP-Befehle

Die Kommunikation mit einem FTP-Server läuft auf zwei Kanälen ab: auf dem Steuerkanal zum Senden von Befehlen an den Server und auf dem Datenkanal zum Empfangen von Daten. Diese Trennung von Kommando- und Übertragungsebene ermöglicht es, dass auch während einer laufenden Datenübertragung Befehle, beispielsweise zum Abbruch der Übertragung, an den Server gesendet werden können. Grundsätzlich kann eine Datenübertragung in zwei Modi ablaufen: im sogenannten *aktiven Modus* fordert der Client eine Datei an und öffnet gleichzeitig einen Port, über den dann die Übertragung der Datei ablaufen soll. Dem gegenüber steht der *passive Modus*, bei dem der Client den Server instruiert, einen Port zu öffnen, um die Datenübertragung durchzuführen. Das hat den Vorteil, dass auch Datenübertragungen mit Clients stattfinden können, die für den Server nicht direkt adressierbar sind, weil sie beispielsweise hinter einem Router oder einer Firewall stehen.

So viel zu den theoretischen Grundlagen. Ab jetzt werden wir behandeln, wie das Modul `ftplib` zur Kommunikation mit einem FTP-Server verwendet werden kann. Das Modul `ftplib` stellt die Klasse `FTP` zur Verfügung, die es einer Anwendung ermöglicht, sich mit einem FTP-Server zu verbinden und die dort unterstützten Operationen auszuführen. Mit diesem Modul können Sie also einen vollwertigen FTP-Client implementieren.

Bereits beim Instantiieren der Klasse `FTP` kann eine Verbindung mit einem FTP-Server hergestellt werden. Dazu muss dem Konstruktor mindestens die Adresse des FTP-Servers als String übergeben werden. Der Konstruktor der Klasse `FTP` hat folgende Schnittstelle:

```
FTP([host[, user[, passwd[, acct[, timeout]]]])
```

Der Konstruktor erzeugt eine Instanz der Klasse `FTP`, die mit dem FTP-Server *host* verbunden ist. Bei der Anmeldung an diesem Server werden der Benutzername *user*, das Passwort *passwd* und, sofern notwendig, der Account *acct* verwendet. Über den optionalen Parameter *timeout* wird ein Timeout-Wert in Sekunden für die Verbindungsanfrage eingestellt. Wenn Sie *timeout* nicht angeben, wird ein Systemdefault verwendet.

Die Klasse `FTP`

Im Folgenden erläutern wir die wichtigsten Methoden einer `FTP`-Instanz. Um die folgenden Beispiele ausführen zu können, müssen Sie sowohl das Modul `ftplib` importieren als auch eine `FTP`-Instanz `ftp` erzeugen, die mit einem FTP-Server Ihrer Wahl verbunden ist:

```
>>> import ftplib
>>> ftp = ftplib.FTP("ftp.test.de")
```

Im Folgenden sei *f* eine Instanz der Klasse `ftplib.FTP`.

f.connect(host[, port[, timeout]])

Verbindet zu dem FTP-Server *host* unter Verwendung des Ports *port*. Diese Methode sollte nicht aufgerufen werden, wenn bei der Instantiierung der Klasse `FTP` bereits die Adresse des FTP-Servers übergeben wurde. Der optionale Parameter *timeout* hat dieselbe Bedeutung wie beim Konstruktor der Klasse `FTP`.

f.getwelcome()

Gibt die Willkommensnachricht des verbundenen FTP-Servers als String zurück.

```
>>> ftp.getwelcome()
'220 Welcome to xyz FTP server. Please login...'
```

f.login([user[, passwd[, acct]])

Loggt sich auf dem verbundenen FTP-Server ein. Beachten Sie, dass die Methode `connect` unbedingt aufgerufen werden muss, bevor ein Login durchgeführt werden kann.

Die Parameter haben dieselbe Bedeutung wie die des Konstruktors der Klasse `FTP`.

```
>>> ftp.login("Benutzername", "Passwort")
'230 User Benutzername logged in'
```

Wenn die Methode `login` aufgerufen wird, obwohl der Client bereits eingeloggt ist, wird eine `ftplib.error_perm`-Exception geworfen.

f.abort()

Unterbricht einen laufenden Dateitransfer. Beachten Sie, dass eine solche Unterbrechung je nach Server nicht zu jedem Zeitpunkt durchgeführt werden kann.

f.sendcmd(command)

Schickt den Kommandostring *command* an den verbundenen FTP-Server und gibt dessen Antwort ebenfalls als String zurück.

```
>>> ftp.sendcmd("PWD")
'257 "/" is the current directory'
```

f.retrbinary(command, callback[, maxblocksize[, rest]])

Leitet einen Datentransfer im Binärmodus ein. Dazu muss als erster Parameter ein entsprechendes FTP-Kommando übergeben werden, aufgrund dessen der Server einen Datentransfer über den Datenkanal startet. Für einen simplen Dateitransfer dient das Kommando `RETR dateiname`.

An zweiter Stelle muss ein Funktionsobjekt übergeben werden. Die dahinterstehende Funktion muss exakt einen Parameter akzeptieren. Nach jedem erfolgreich übermittelten Block wird die Funktion *callback* aufgerufen. Die übertragenen Binärdaten werden dabei als Parameter in Form eines Strings übergeben.

Der Parameter *maxblocksize* bestimmt die maximale Größe der Blöcke, in die die Datei zum Herunterladen aufgeteilt wird.

Über den vierten, optionalen Parameter *rest* wird ein Offset in der zu übertragenden Datei angegeben, ab dem der Server den Dateinhalt senden soll. Dies ist zum Beispiel nützlich, um abgebrochene Downloads wiederaufzunehmen, ohne dabei Teile der Datei doppelt herunterladen zu müssen. In den meisten Fällen spielt dieser Parameter jedoch keine Rolle und soll hier deshalb nicht weiter behandelt werden.

Zur Verwendung von `retrbinary` nun folgendes Beispiel; beachten Sie, dass das Ergebnis die Daten von `retrbinary` als `bytes`-String an die Funktion *callback* übermittelt werden:

```
bild = bytes()
def f(data):
    global bild
    bild += data
ftp.retrbinary("RETR bild.jpg", f)
```

Das Beispielprogramm lädt die Bilddatei *bild.jpg* aus dem aktuellen Arbeitsverzeichnis des FTP-Servers herunter und speichert die Binärdaten im `bytes`-String `bild`.

Alternativ könnte auch ein `LIST`-Kommando abgesetzt werden, aufgrund dessen der Verzeichnisinhalt ebenfalls über den Datenkanal geschickt wird:

```
>>> def f(data):
...     print(data.decode())
...
>>> ftp.retrbinary("LIST", f)
drwxr-xr-x  11 user  group      360 Sep  5 02:45 .
drwxr-xr-x  11 user  group      360 Sep  5 02:45 ..
drwxr-xr-x   4 user  group       96 Jun 20 2006 ordner1
[...]
```

f.retrlines(command[, callback])

Leitet einen Dateitransfer im ASCII-Modus ein. Dazu müssen Sie als ersten Parameter ein entsprechendes FTP-Kommando übergeben. Für einen simplen Dateitransfer wäre dies `RETR dateiname`. Möglich wäre aber beispielsweise auch, den Inhalt des Arbeitsverzeichnisses durch ein `LIST`-Kommando zu übertragen.

Eine Dateiübertragung im ASCII-Modus geschieht zeilenweise. Das heißt, die Callback-Funktion *callback* wird nach jeder vollständig übertragenen Zeile aufgerufen. Sie bekommt dabei die gelesene Zeile als Parameter übergeben. Beachten Sie, dass das abschließende Newline-Zeichen nicht mit übergeben wird.

Wenn Sie keine Callback-Funktion angegeben haben, werden die übertragenen Daten ausgegeben.

```
text = ""
def f(data):
    global text
    text = "".join((text, data, "\n"))
ftp.retrlines("RETR text.txt", f)
```

Dieses Beispielprogramm lädt die Textdatei *text.txt* zeilenweise herunter und fügt die heruntergeladenen Zeilen im String `text` wieder zu einem Gesamttext zusammen.

f.set_pasv(boolean)

Übergeben Sie für *boolean* `False`, wird die FTP-Instanz in den sogenannten aktiven Zustand versetzt. Ein Wert von `True` versetzt sie zurück in den passiven Zustand. Beachten Sie, dass der Client im aktiven Zustand für den Server erreichbar sein muss, sich also nicht hinter einer Firewall oder einem Router befinden darf.

f.storbinary(command, file[, blocksize[, callback]])

Leitet einen Datei-Upload ein. Dabei muss als erster Parameter ein entsprechender FTP-Befehl in Form eines `bytes`-Strings übergeben werden. Für einen simplen Datei-Upload lautet dieser Befehl `STOR datei`, wobei *datei* der Zielname der Datei auf dem FTP-Server ist. Als zweiten Parameter müssen Sie ein im Binärmodus geöffnetes Dateiojekt übergeben, dessen Inhalt hochgeladen werden soll.

Optional kann in Form des dritten Parameters, *blocksize*, die maximale Größe der Datenblöcke angegeben werden, in denen die Datei hochgeladen wird.

Wenn für den vierten Parameter *callback* das Funktionsobjekt einer Funktion mit einem Parameter übergeben wird, so wird diese Funktion nach jedem gesendeten Block gerufen. Dabei bekommt sie die gesendeten Daten als `bytes`-String übergeben.

Das folgende Beispielprogramm führt einen binären Datei-Upload durch:

```
f = open("bla.jpg", "rb")
ftp.storbinary("STOR hallo.jpg", f)
f.close()
```

Beachten Sie, dass die Datei im lokalen Arbeitsverzeichnis *bla.txt* heißt, auf den Server jedoch unter dem Namen *hallo.txt* hochgeladen wird.

f.storlines(command, file[, callback])

Verhält sich ähnlich wie `storbinary` mit dem Unterschied, dass die Datei im ASCII-Modus zeilenweise hochgeladen wird. Die Parameter *command*, *file* und *callback* lassen sich wie bei `storbinary` verwenden.

Beachten Sie, dass Sie das für *file* übergebene Dateiojekt wie bei `storbinary` auch im Binärmodus geöffnet haben müssen.

f.nlst(argument[, ...])

Gibt eine Liste mit dem Inhalt des aktuellen Arbeitsverzeichnisses auf dem FTP-Server zurück. Über den optionalen Parameter *dirname* wird ein Unterverzeichnis angegeben, dessen Inhalt aufgelistet werden soll:

```
>>> ftp.nlst()
['.', '..', 'ordner1', 'ordner2', 'hallo.txt']
>>> ftp.nlst("ordner1")
[' ordner1/.', ' ordner1/..', ' ordner1/test.py']
```

Neben den im Arbeitsverzeichnis existierenden Ordnern und Dateien sind die Verweise auf das aktuelle Verzeichnis (.) und das übergeordnete Verzeichnis (..) in der Liste enthalten.

f.dir(argument[, ...])

Gibt den Inhalt des aktuellen Arbeitsverzeichnisses auf dem FTP-Server in Form einer Aufzählung auf dem Bildschirm aus, wie sie vom FTP-Befehl LIST erzeugt würde. Optional kann über den Parameter *dirname* ein Unterverzeichnis angegeben werden, dessen Inhalt ausgegeben werden soll.

Außerdem kann eine Callback-Funktion übergeben werden, die anstelle einer Bildschirmausgabe aufgerufen wird. Die Callback-Funktion *callback* muss über die gleiche Schnittstelle verfügen wie die, die bei *retrlines* angegeben werden kann.

```
>>> ftp.dir()
drwxr-xr-x  11 user   group      360 Sep  5 02:45 .
drwxr-xr-x  11 user   group      360 Sep  5 02:45 ..
[...]
```

f.rename(fromname, toname)

Benennt die Datei *fromname* auf dem FTP-Server in *toname* um.

```
>>> ftp.rename("ordner", "ordner2")
'250 Rename successful'
```

Es erübrigt sich zu sagen, dass der Ordner, der umbenannt werden soll, existieren muss. Ist dies nicht der Fall, wird eine `ftplib.error_perm`-Exception geworfen.

f.delete(filename)

Löscht die Datei *filename* auf dem FTP-Server.

```
>>> ftp.delete("hallo.txt")
'250 DELE command successful'
```


Auch hier wird eine `ftplib.error_perm`-Exception geworfen, wenn die zu löschende Datei nicht existiert.

f.cwd(pathname)

Ändert das aktuelle Arbeitsverzeichnis auf dem FTP-Server in *pathname*. Sollte das Verzeichnis *pathname* nicht existieren, wird eine `ftplib.error_perm`-Exception geworfen.

```
>>> ftp.cwd("ordner")
'250 CWD command successful'
```

f.mkd(pathname)

Erzeugt das Verzeichnis *pathname* auf dem FTP-Server. Die Methode `mkd` gibt den Pfad zu dem neu erstellten Verzeichnis zurück.

```
>>> ftp.mkd("ordner")
'/ordner'
```

Beachten Sie, dass kein Verzeichnis dieses Namens bereits existieren darf. In einem solchen Fall würde eine `ftplib.error_perm`-Exception geworfen.

f.pwd()

Gibt den Pfad des aktuellen Arbeitsverzeichnisses auf dem FTP-Server zurück.

```
>>> ftp.pwd()
'/'
```

f.rmd(dirname)

Löscht das Verzeichnis *dirname* auf dem FTP-Server. Beachten Sie, dass das Verzeichnis *dirname* vorhanden und leer sein muss, damit es erfolgreich gelöscht werden kann. Im Fehlerfall wird eine `ftplib.error_perm`-Exception geworfen.

```
>>> ftp.rmd("ordner")
'250 RMD command successful'
```

f.size(filename)

Ermittelt die Dateigröße der Datei *filename* auf dem FTP-Server. Wenn sie sich ermitteln ließ, wird die Dateigröße als ganze Zahl zurückgegeben, andernfalls ist der Rückgabewert `None`. Beachten Sie, dass das dieser Methode zugrundeliegende FTP-Kommando `SIZE` nicht standardisiert ist und somit nicht von allen FTP-Servern unterstützt wird.

f.quit()

Beendet die Verbindung zum FTP-Server, indem ihm ein `QUIT`-Befehl gesendet wird. Dies ist die saubere Art, die Verbindung zu kappen, könnte aber eine Exception verursachen, falls der Server mit einem Fehlercode antwortet.

Der Aufruf von `quit` erübrigt einen weiteren Aufruf von `close`.

f.close()

Beendet die Verbindung, ohne den FTP-Server davon in Kenntnis zu setzen. Beachten Sie, dass dieselbe FTP-Instanz nach Aufruf dieser Funktion nicht wieder per `login` mit einem FTP-Server verbunden werden kann. Dazu sollte eine neue Instanz erzeugt werden.

20.4 E-Mail

In diesem Abschnitt werden wir Module der Standardbibliothek vorstellen, die es ermöglichen, mit einem E-Mail-Server zu kommunizieren, das heißt E-Mails von diesem abzuholen bzw. E-Mails über den Server zu versenden.

Das Versenden einer E-Mail geschieht über einen sogenannten SMTP-Server, mit dem über ein gleichnamiges Protokoll kommuniziert werden kann. Im ersten Unterabschnitt werden wir deshalb das Modul `smtplib` der Standardbibliothek vorstellen, das genau dieses Kommunikationsprotokoll implementiert.

Für das Herunterladen einer empfangenen E-Mail gibt es zwei verbreitete Möglichkeiten: das POP3- und das IMAP4-Protokoll. Beide können mit dem jeweiligen Modul `poplib` bzw. `imaplib` verwendet werden.

Im letzten Abschnitt wird das Modul `email` der Standardbibliothek besprochen, das es über die MIME-Kodierung ermöglicht, beliebige Dateien (üblicherweise Bilder oder Dokumente) mit der E-Mail zu versenden.

20.4.1 SMTP – smtplib

Das sogenannte *SMTP-Protokoll* (für *Simple Mail Transfer Protocol*) wird zum Versenden einer E-Mail über einen SMTP-Server verwendet. Das SMTP-Protokoll ist ähnlich wie FTP ein textbasiertes, lesbares Protokoll. Ursprünglich bot das SMTP-Protokoll keine Möglichkeit zur Authentifizierung des angemeldeten Benutzers beispielsweise durch Benutzername und Passwort. Dies war bei der rasanten Entwicklung des Internets ziemlich schnell nicht mehr tragbar, und so wurde das SMTP-Protokoll um den *ESMTP-Standard* (*Extended SMTP*) erweitert.

Ähnlich wie in Abschnitt 20.3, »FTP – ftplib«, möchten wir hier zunächst eine Übersicht über die wichtigsten SMTP-Befehle geben. Die Befehle sind in der folgenden Tabelle in der Reihenfolge ihrer Benutzung in einer SMTP-Sitzung aufgelistet und werden jeweils mit einem kurzen Satz erklärt.

Befehl	Beschreibung
HELO	Startet eine SMTP-Sitzung.
EHLO	Startet eine ESMTP-Sitzung.
MAIL FROM	Leitet das Absenden einer E-Mail ein. Diesem Kommando wird die Absenderadresse beigelegt.
RCPT TO	Fügt einen Empfänger der E-Mail hinzu. (RCPT steht für <i>Recipient</i> , dt. »Empfänger«.)
DATA	Mit diesem Kommando wird der Inhalt der E-Mail angegeben und die Mail schlussendlich verschickt.
QUIT	Beendet die SMTP- bzw. ESMTP-Sitzung.

Tabelle 20.7 SMTP-Befehle

Wie schon die `ftplib` enthält das Modul `smtpplib` im Wesentlichen nur eine Klasse namens `SMTP`. Über diese Klasse läuft, nachdem sie instantiiert wurde, alle weitere Kommunikation mit dem Server. Der Konstruktor der Klasse `SMTP` hat folgende Schnittstelle:

`smtpplib.SMTP([host[, port[, local_hostname[, timeout]]]])`

Erzeugt eine Instanz der Klasse `SMTP`. Optional können hier bereits die Verbindungsdaten zum SMTP-Server übergeben werden. Beachten Sie, dass Sie den Port nur explizit anzugeben brauchen, wenn er sich vom SMTP-Standardport 25 unterscheidet.

Als dritter Parameter kann der Domainname des lokalen Hosts übergeben werden. Dieser wird dem SMTP-Server als Identifikation im ersten gesendeten Kommando übermittelt. Wenn der Parameter `local_hostname` nicht angegeben wird, wird versucht, den lokalen Hostnamen automatisch zu ermitteln.

Für den vierten Parameter können Sie einen speziellen Timeout-Wert in Sekunden übergeben, der bei der Verbindung zum SMTP-Server berücksichtigt wird. Wenn Sie `timeout` nicht angeben, wird ein Standardwert verwendet.

Die Klasse `SMTP`

Im Folgenden sollen die wichtigsten Methoden der `SMTP`-Klasse erläutert werden. Um die Beispiele dieses Abschnitts nachvollziehen zu können, muss das Modul

`smtpplib` eingebunden werden und eine Instanz der Klasse `SMTP` mit dem Namen `s` existieren. Für die meisten der Beispiele muss die `SMTP`-Instanz zusätzlich mit einem Server verbunden und angemeldet sein.

s.connect([host[, port]])

Verbindet zum `SMTP`-Server *host* mit der Portnummer *port*. Diese Methode sollte nicht aufgerufen werden, wenn bei der Instanziierung der `SMTP`-Klasse bereits Verbindungsdaten übergeben wurden. Wenn keine Verbindung zum `SMTP`-Server aufgebaut werden kann, wird eine Exception geworfen.

```
>>> s.connect("smtp.beispiel.de")
(220, 'Die Botschaft des Servers')
```

s.login(user, password)

Diese Methode ermöglicht es, sich beim `SMTP`-Server mit dem Benutzernamen *user* und dem Passwort *password* einzuloggen, sofern der Server dies verlangt.

```
>>> s.login("Benutzername", "Passwort")
(235, '2.0.0 Authentication successful')
```

Im Fehlerfall werden folgende Exceptions geworfen:

Exception	Beschreibung
<code>SMTPHeloError</code>	Der <code>SMTP</code> -Server hat nicht oder nicht richtig auf das Begrüßungskommando <code>HELO</code> geantwortet.
<code>SMTPAuthenticationError</code>	Die angegebene Benutzername-Passwort-Kombination wurde vom <code>SMTP</code> -Server nicht akzeptiert.
<code>SMTPError</code>	Es wurde keine Möglichkeit gefunden, eine Authentifizierung bei diesem <code>SMTP</code> -Server durchzuführen.

Tabelle 20.8 Mögliche Exceptions beim Login

s.sendmail(from_addr, to_addr, msg[, mail_options, rctp_options])

Durch Aufruf der Methode `sendmail` wird eine E-Mail über den `SMTP`-Server versendet. Beachten Sie, dass die `SMTP`-Instanz dafür an einem `SMTP`-Server angemeldet und zumeist auch authentifiziert sein muss.

Die ersten beiden Parameter enthalten die E-Mail-Adressen des Absenders (*from_addr*) bzw. eine Liste der E-Mail-Adressen der Empfänger (*to_addr*). Als E-Mail-Adresse wird dabei ein String des folgenden Formats bezeichnet:

```
Vorname Nachname <em@il.addr>
```

Alternativ kann auch nur die E-Mail-Adresse an sich im String stehen.

Als dritten Parameter, *msg*, übergeben Sie den Text der E-Mail. Beachten Sie, dass weitere Angaben wie beispielsweise der Betreff der E-Mail innerhalb des E-Mail-Bodys definiert werden. Wie so etwas genau aussieht und welche Möglichkeiten Python bietet, diesen Header komfortabel zu erzeugen, erfahren Sie in Abschnitt 20.4.4, »Erstellen komplexer E-Mails – email«.

Die Methode `sendmail` gibt stets ein Dictionary zurück, in dem alle Empfänger, die vom SMTP-Server zurückgewiesen wurden, als Schlüssel enthalten sind und der jeweilige Error-Code mit Fehlerbezeichnung als Wert aufgeführt ist. Wenn alle Empfänger die E-Mail bekommen haben, ist das zurückgegebene Dictionary leer.

Im Fehlerfall wirft die Methode `sendmail` folgende Exceptions:

Exception	Beschreibung
SMTPHeloError	Der SMTP-Server hat nicht oder nicht richtig auf das Begrüßungskommando <code>HELO</code> geantwortet.
SMTPRecipientsRefused	Alle Empfänger wurden vom SMTP-Server zurückgewiesen. Das heißt, dass die E-Mail tatsächlich an niemanden verschickt wurde. Als Attribut enthält die Exception ein Dictionary, das demjenigen ähnelt, das von <code>sendmail</code> im Erfolgsfall zurückgegeben wird.
SMTPSenderRefused	Der angegebene Absender wurde vom SMTP-Server zurückgewiesen.
SMTPDataError	Der Server hat mit einem unerwarteten Fehler geantwortet.

Tabelle 20.9 Mögliche Exceptions beim Login

Beachten Sie, dass der Text einer E-Mail nur aus ASCII-Zeichen bestehen darf. Um auch andere Zeichen und insbesondere auch Binärdaten verschicken zu können, bedient man sich der sogenannten MIME-Kodierung, die wir in Abschnitt 20.4.4, »Erstellen komplexer E-Mails – email«, behandeln werden.

Über die optionalen Parameter *mail_options* und *rctp_options* kann je eine Liste von Strings übergeben werden, die Optionen des ESMTP-Standards (*Extended SMTP*) enthalten. Die für *mail_options* übergebenen Optionen werden dem Kommando `MAIL FROM` angefügt; hier wäre beispielsweise die Option `"8bitmime"` sinnvoll. Alle für *rctp_options* übergebenen Optionen werden dem Kommando `RCPT TO` angehängt, wo Sie beispielsweise die Option `"DSN"` verwenden können.

Welche Optionen im ESMTP-Standard definiert werden und was diese bedeuten, soll nicht Thema dieses Buches sein. Wenn Sie sich dafür interessieren, fühlen Sie sich dazu ermutigt, unter den genannten Stichwörtern im Internet zu recherchieren.

s.quit()

Beendet die Verbindung zum SMTP-Server.

Beispiel

Nachdem die wichtigsten Methoden einer SMTP-Instanz erläutert wurden, folgt nun ein kleines Beispiel, in dem zu einem SMTP-Server verbunden wird, um zwei E-Mails an verschiedene Empfänger zu verschicken:

```
>>> smtp = smtplib.SMTP("smtp.hostname.de")
>>> smtp.login("benutzername", "passwort")
(235, '2.0.0 Authentication successful')
>>> smtp.sendmail(
...     "Peter Kaiser <p@penguin-p.de>",
...     "Johannes Ernesti <je@revelation-soft.de>",
...     "Dies ist der Text")
{}
>>> smtp.sendmail(
...     "Peter Kaiser <p@penguin-p.de>",
...     ["je@revelation-soft.de", "p@penguin-p.de"],
...     "Dies ist der Text")
{}
>>> smtp.quit()
```

Bei der ersten E-Mail wurden die vollen Namen des Absenders bzw. des Empfängers angegeben. Das zweite Beispiel zeigt, dass auch die E-Mail-Adresse allein reicht, und demonstriert, wie eine E-Mail an mehrere Empfänger versandt werden kann.

20.4.2 POP3 – poplib

Nachdem anhand der `smtplib` erläutert wurde, wie E-Mails über einen SMTP-Server versandt werden können, soll das Thema dieses Abschnitts das Modul `poplib` der Standardbibliothek sein. Dieses Modul implementiert das POP3-Protokoll (*Post Office Protocol Version 3*). Bei POP3 handelt es sich um ein Protokoll, um auf einen POP3-Server zuzugreifen und dort gespeicherte E-Mails einzusehen und abzuholen. Das POP3-Protokoll steht damit in Konkurrenz zu IMAP4, dessen Benutzung mit der `imaplib` das Thema des nächsten Abschnitts sein soll. Die folgende Tabelle listet die wichtigsten POP3-Kommandos mit ihrer Bedeutung auf. Die Befehle stehen dabei in der Reihenfolge, wie sie in einer üblichen POP3-Sitzung verwendet werden.

Befehl	Beschreibung
USER	Überträgt den Benutzernamen zur Authentifizierung auf dem Server.
PASS	Überträgt das Passwort zur Authentifizierung auf dem Server.
STAT	Liefert den Status des Posteingangs, beispielsweise die Anzahl der neu eingegangenen E-Mails.
LIST	Liefert Informationen zu einer bestimmten E-Mail des Posteingangs.
RETR	Überträgt eine bestimmte E-Mail.
DELE	Löscht eine bestimmte E-Mail.
RSET	Löschvorgänge werden gepuffert und erst am Ende der Sitzung ausgeführt. Mit diesem Kommando können alle anstehenden Löschvorgänge widerrufen werden.
QUIT	Beendet die POP3-Sitzung.

Tabelle 20.10 POP3-Befehle

Wie bereits beim Modul `smtpplib` ist im Modul `poplib` im Wesentlichen die Klasse `POP3` enthalten, die instantiiert werden muss, bevor Operationen auf einem POP3-Server durchgeführt werden können. Die Schnittstelle des Konstruktors sieht folgendermaßen aus:

poplib.POP3(host[, port[, timeout]])

Erzeugt eine Instanz der Klasse `POP3`. Dem Konstruktor wird der Hostname des POP3-Servers übergeben, zu dem verbunden werden soll. Optional kann ein Port angegeben werden, wenn dieser sich vom voreingestellten Standardport 110 unterscheidet. Bei dem Parameter *timeout* handelt es sich um einen Timeout-Wert in Sekunden, der bei der Verbindung zum Server berücksichtigt wird.

Die Klasse POP3

Im Folgenden sollen die wichtigsten Methoden der Klasse `POP3` beschrieben werden. Die Funktionsnamen entsprechen im Wesentlichen den POP3-Befehlen, die sie senden. Um die in diesem Abschnitt vorgestellten Beispiele ausführen zu können, muss zum einen das Modul `poplib` eingebunden sein und zum anderen eine Instanz der Klasse `POP3` mit dem Namen `pop` existieren. Beachten Sie, dass diese Instanz für die meisten Beispiele mit einem POP3-Server verbunden und bei diesem authentifiziert sein muss.

pop.user(username)

Übermittelt den Benutzernamen *username* an den POP3-Server. Die Antwort des Servers ist in der Regel ein String, in dem er ein Passwort fordert. Das angeforderte Passwort kann jetzt durch einen Aufruf der Methode `pass_` übermittelt

werden. Beachten Sie, dass ein falscher Benutzername zunächst nicht zu einer Exception führt.

```
>>> pop.user("Benutzername")
b'+OK Password required.'
```

pop.pass_(password)

Übermittelt das Passwort *password* an den POP3-Server. Nachdem das Passwort vom Server akzeptiert worden ist, darf auf den Posteingang zugegriffen werden. Dieser ist bis zum Aufruf von `quit` für andere Login-Versuche gesperrt.

```
>>> pop.pass_("Passwort")
b'+OK logged in.'
```

Im Falle einer fehlgeschlagenen Authentifizierung wird eine `poplib.error_proto`-Exception geworfen.

pop.stat()

Gibt den Status des Posteingangs zurück. Das Ergebnis ist ein Tupel mit zwei ganzen Zahlen: der Anzahl der enthaltenen Nachrichten und der Größe des Posteingangs.

```
>>> pop.stat()
(1, 623)
```

In diesem Fall befindet sich eine E-Mail im Posteingang, und die Gesamtgröße des Posteingangs beläuft sich auf 623 Byte.

pop.list([which])

Gibt eine Liste der im Posteingang liegenden Mails zurück. Der Rückgabewert dieser Methode ist ein Tupel der folgenden Form:

```
(antwort, [b"mailID laenge", ...], datlen)
```

Dabei enthält das Tupel als erstes Element den Antwortstring des Servers und als zweites Element eine Liste von `bytes`-Strings, die je für eine E-Mail des Posteingangs stehen. Der String enthält zwei Angaben: Die Angabe `mailID` ist die laufende Nummer der Mail, eine Art Index, und `laenge` ist die Gesamtgröße der Mail in Byte. In Bezug auf den Index sollten Sie beachten, dass alle E-Mails auf dem Server fortlaufend von 1 an indiziert werden und nicht, wie beispielsweise bei einer Python-Liste, mit 0 beginnend.

Das erste Element des Tupels (`antwort`) enthält dabei nicht den vollständigen Antwortstring des Servers, denn die Informationen, die zum zweiten Element des Tupels aufbereitet wurden, wurden aus `antwort` entfernt. Um dennoch die kom-

plette Länge der Serverantwort berechnen zu können, existiert das dritte Element des Tupels (`datlen`). Dieses referenziert die Länge des Datenbereichs der Antwort des Servers. Damit entspräche `len(antwort) + datlen` der Gesamtgröße des vom Server tatsächlich gesendeten Antwortstrings.

Über den optionalen Parameter *which* kann die laufende Nummer einer E-Mail angegeben werden, über die nähere Informationen zurückgegeben werden sollen. In diesem Fall gibt die Methode einen bytes-String mit dem Format `b"+OK mailID laenge"` zurück. Es ist also mit dieser Methode nur möglich, die Länge einer bestimmten E-Mail in Byte herauszufinden, da die ID ja bereits bekannt ist. Wenn eine ungültige ID für *which* übergeben wurde, wird eine `poplib.error_proto-Exception` geworfen.

```
>>> pop.list()
(b'+OK [...].', [b'1 623'], 7)
>>> pop.list(1)
b'+OK 1 623'
```

pop.retr(which)

Greift auf die Mail mit der laufenden Nummer *which* zu und gibt ihren Inhalt in Form des folgenden Tupels zurück:

```
(antwort, zeilen, laenge)
```

Das erste Element des Tupels entspricht dem Antwortstring des Servers. An zweiter Stelle steht eine Liste von bytes-Strings, die je eine Zeile der E-Mail inklusive des E-Mail-Headers enthalten. Das letzte Element des Tupels ist die Größe der E-Mail in Byte.

```
>>> pop.retr(1)
(b'+OK 623 octets follow.', [...], 623)
```

Anstelle des Auslassungszeichens stünde eine Liste von Strings, die die Zeilen der vollständigen E-Mail enthält.

pop.dele(which)

Löscht die Mail mit der laufenden Nummer *which* vom POP3-Server. Beachten Sie, dass die meisten Server solche Befehle puffern und erst nach Aufruf der Methode `quit` tatsächlich ausführen.

```
>>> pop.dele(1)
b'+OK Deleted.'
```

pop.reset()

Ein Aufruf dieser Methode veranlasst, dass alle anstehenden Löschvorgänge verworfen werden.

```
>>> pop.reset()
b'+OK Resurrected.'
```

pop.quit()

Beendet die Verbindung zum POP3-Server. Bei den meisten Servern werden erst jetzt alle anstehenden Löschvorgänge durchgeführt.

```
>>> pop.quit()
b'+OK Bye-bye.'
```

Beispiel

Nachdem die wichtigsten Methoden einer POP3-Instanz erklärt wurden, werden wir hier in einem kleinen Beispiel das Modul `poplib` dazu verwenden, alle Mails von einem POP3-Server abzuholen und auf dem Bildschirm anzuzeigen:

```
import poplib

pop = poplib.POP3("pop.hostname.de")
pop.user("benutzername")
pop.pass_("password")

for i in range(1, pop.stat()[0]+1):
    for zeile in pop.retr(i)[1]:
        print(zeile)
    print("***")

pop.quit()
```

Zunächst wird eine Instanz der Klasse `POP3` erzeugt, und das Programm meldet sich mit den Methoden `user` und `pass_` beim POP3-Server an. Der Ausdruck `pop.stat()[0]` liefert die Zahl der Mails, die sich im Posteingang befinden. In der `for`-Schleife werden also alle Mail-Indizes durchlaufen. Beachten Sie dazu, dass die Indizes der E-Mails im Posteingang mit 1 beginnen.

In der inneren Schleife wird die jeweils aktuelle Mail mit dem Index `i` durch Aufruf der Methode `retr` heruntergeladen. Das zweite Element, also das mit dem Index 1 des von dieser Methode zurückgegebenen Tupels, enthält eine Liste mit allen Zeilen des Mail-Inhalts. Diese Liste wird in der Schleife durchlaufen, und es wird jeweils die aktuelle Zeile ausgegeben.

Beachten Sie, dass im Beispielprogramm aus Gründen der Übersichtlichkeit auf jegliche Fehlerbehandlung verzichtet wurde. In einem fertigen Programm müss-

ten Sie auf jeden Fall prüfen, ob die Verbindung zum Server hergestellt werden konnte und ob die Authentifizierung erfolgreich war.

Nachdem eine E-Mail vollständig durchlaufen worden ist, werden drei Sternchen ausgegeben, die damit als eine Art Trennlinie zwischen den Mails fungieren.

20.4.3 IMAP4 – imaplib

Das Modul `imaplib` stellt die Klasse `IMAP4` zur Verfügung, mit deren Hilfe Sie eine Verbindung zu einem IMAP4-Server herstellen und mit diesem kommunizieren. Das IMAP4-Protokoll (*Internet Message Access Protocol 4*) ist ähnlich wie das POP3-Protokoll zur Verwaltung von E-Mails auf einem Mailserver gedacht. Anders als bei dem bekannteren Protokoll POP3 verbleiben die E-Mails bei IMAP4 zumeist auf dem Mailserver, was den Vorteil hat, dass man von überall – beispielsweise auch von einem Internet-Cafe im Urlaub aus – vollen Zugriff auf alle archivierten E-Mails hat. Heutzutage bieten die meisten E-Mail-Anbieter sowohl einen POP3- als auch einen IMAP4-Zugang an. Im Vergleich zu POP3 unterstützt IMAP4 Kommandos zur komfortablen Verwaltung der Mails auf dem Server. So können beispielsweise Unterordner angelegt werden.

Im Gegensatz zu den bisherigen Protokollen wie FTP oder POP3 ist IMAP4 mit einem hohen Funktionsumfang ausgestattet, und obwohl das Protokoll immer noch auf lesbaren Textnachrichten basiert, ist es zu komplex, um es im Stil der bisherigen Abschnitte mit einem kurzen Text und einer Tabelle ausreichend zu beschreiben. Grundsätzlich kann aber gesagt werden, dass das IMAP4-Protokoll umfassende Unterstützung zur Verwaltung der E-Mails bereitstellt. So lassen sich diese beispielsweise in verschiedene sogenannte *Mailboxen* einsortieren. Dabei können Sie sich eine Mailbox als eine Art Verzeichnis vorstellen, das E-Mails enthalten kann, wie ein Ordner Dateien enthält. Die Mailbox-Struktur des verwendeten Beispielservers sieht folgendermaßen aus:



Abbildung 20.3 Mailbox-Struktur des Beispielservers

Es existieren eine übergeordnete Mailbox namens *INBOX* sowie zwei untergeordnete Mailboxen namens *INBOX.Ham* und *INBOX.Spam*.

Um eine Verbindung zu einem IMAP4-Server herzustellen, muss eine Instanz der Klasse `IMAP4` erzeugt werden. Der Konstruktor dieser Klasse hat folgende Schnittstelle:

`imaplib.IMAP4([host[, port]])`

Erzeugt eine Instanz der Klasse `IMAP4`. Optional kann direkt nach der Instantiierung automatisch eine Verbindung zu einem IMAP4-Server mit dem Hostnamen *host* unter Verwendung des Ports *port* aufgebaut werden. Wenn der Parameter *port* nicht angegeben wurde, wird der IMAP4-Standardport 143 verwendet.

Die Klasse `IMAP4`

Nachdem eine Instanz der Klasse `IMAP4` erzeugt wurde, stellt diese verschiedene Methoden bereit, um mit dem verbundenen Server zu kommunizieren. Jede Methode, die ein IMAP4-Kommando repräsentiert, gibt ein Tupel der folgenden Form zurück:

```
(Status, [Daten, ...])
```

Dabei steht im resultierenden Tupel für *Status* entweder "OK" oder "NO", je nachdem, ob die Operation erfolgreich verlaufen oder fehlgeschlagen ist. Das zweite Element des Tupels ist eine Liste, die die Daten enthält, die der Server als Antwort geschickt hat. Diese Daten können entweder ein `bytes`-String oder ein Tupel sein. Wenn es sich um ein Tupel handelt, verfügt dieses über zwei Elemente:

```
(Header, Daten)
```

Beide Elemente dieses Tupels sind `bytes`-Strings. Im Folgenden werden die wichtigsten Methoden einer `IMAP4`-Instanz erläutert. Die Beispiele setzen zumeist eine verbundene `IMAP4`-Instanz `im` voraus:

```
>>> import imaplib
>>> im = imaplib.IMAP4("imap.beispiel.de")
```

In den meisten Fällen muss die `IMAP4`-Instanz zudem beim Server eingeloggt sein, was durch Aufruf der Methode `login` geschieht.

`im.login(user, password)`

Sendet Benutzername und Passwort an den verbundenen IMAP4-Server. Beachten Sie, dass beide Informationen als `bytes`-String übergeben werden müssen.

```
>>> im.login(b"Benutzername", b"Passwort")
('OK', [b'LOGIN Ok.'])
```

im.logout()

Beendet die Verbindung zum IMAP4-Server.

```
>>> im.logout()
('BYE', [b'Courier-IMAP server shutting down'])
```

im.select([mailbox[, readonly]])

Wählt eine Mailbox aus, um weitere Operationen auf dieser durchführen zu können. Dabei übergeben Sie als ersten Parameter den Namen der auszuwählenden Mailbox. Wenn für den Parameter *readonly* 1 übergeben wird, ist die gewählte Mailbox bei diesem Zugriff schreibgeschützt und kann somit nicht verändert werden. Die Funktion `select` gibt die Anzahl der E-Mails zurück, die sich in der gewählten Mailbox befinden.

```
>>> im.select("INBOX")
('OK', [b'2'])
```

Beachten Sie, dass keine Exception geworfen wird, wenn die gewünschte Mailbox nicht existiert, sondern dass der Fehler anhand des Rückgabewertes ausgemacht werden muss:

```
>>> im.select("INBOX.NichtExistent")
('NO', [b'Mailbox does not exist, or must be subscribed to.'])
```

im.close()

Schließt die momentan ausgewählte Mailbox.

```
>>> im.close()
('OK', [b'mailbox closed.'])
```

im.list([directory[, pattern]])

Gibt die Namen aller Mailboxen zurück, die sich im Ordner *directory* befinden und auf *pattern* passen. Wenn der Parameter *directory* nicht übergeben wird, werden Mailboxen des Hauptordners zurückgegeben. Geben Sie den zweiten Parameter *pattern* nicht an, so werden alle im jeweiligen Ordner enthaltenen Mailboxen zurückgegeben. Der Parameter *pattern* muss ein String sein und enthält üblicherweise Fragmente eines Mailbox-Namens inklusive Platzhalter (*).

```
>>> im.list(".", "*Ham")
('OK', [b'(\HasNoChildren) "." "INBOX.Ham"'])
>>> im.list(".", "*am")
('OK', [b'(\HasNoChildren) "." "INBOX.Ham"',
b'(\HasNoChildren) "." "INBOX.Spam"'])
>>> im.list(".", "*")
('OK', [b'(\HasNoChildren) "." "INBOX.Ham"',
```

```
b'(\HasNoChildren) "." "INBOX.Spam"',
b'(\Unmarked \HasChildren) "." "INBOX"']])
>>> im.list(".", "NichtVorhandeneMailbox")
('OK', [None])
```

Jeder Eintrag der Liste ist ein `bytes`-String und enthält drei jeweils durch ein Leerzeichen voneinander getrennte Informationen: die sogenannten *Flags* der Mailbox in Klammern, das Verzeichnis der Mailbox und den Mailbox-Namen jeweils in doppelten Anführungsstrichen. Aus den Flags kann man beispielsweise die Information entnehmen, ob eine Mailbox untergeordnete Mailboxen enthält (`\HasChildren`) oder nicht (`\HasNoChildren`).

im.fetch(message_set, message_parts)

Lädt Teile der E-Mails vom Server herunter. Der Parameter *message_set* muss ein String sein, der die Mail-IDs der E-Mails enthält, die herunterzuladen sind. Dabei können diese entweder einzeln im String vorkommen ("1"), als Bereich ("1:4" für Mail Nr. 1 bis 4), als Liste von Bereichen ("1:4,7:9" für Mail Nr. 1 bis 4 und Nr. 7 bis 9) oder als Bereich mit unbestimmter oberer Grenze ("3:*" für alle Mails ab Mail Nr. 3).

Wenn andere Methoden der IMAP4-Klasse über einen Parameter *message_set* verfügen, so ist damit stets ein String im oben beschriebenen Format gemeint.

Der zweite Parameter *message_parts* kennzeichnet, welche Teile der angegebenen E-Mails heruntergeladen werden sollen. Ein Wert von "(RFC822)" bedeutet, die gesamte Mail, also inklusive des Mail-Headers herunterzuladen. Bei einem Wert von "(BODY[TEXT])" würde hingegen nur der Text und bei "(BODY[HEADER])" nur der Header der E-Mail heruntergeladen.

Ein Aufruf der Methode `fetch` funktioniert nur, wenn zuvor eine Mailbox mittels `select` ausgewählt wurde.

```
>>> im.fetch("1", "(BODY[TEXT])")
('OK', [(b'1 (BODY[TEXT] {29}',
b'Dies ist eine Testnachricht\r\n'), b')'])

>>> im.fetch("1:2", "(BODY[TEXT])")
('OK', [(b'1 (BODY[TEXT] {29}',
b'Dies ist eine Testnachricht\r\n'), b')',
(b'2 (BODY[TEXT] {25}',
b'Noch eine Testnachricht\r\n'), b')'])
```

Im Falle einer nicht vorhandenen Mail-ID wird keine Exception geworfen, sondern schlicht ein leeres Ergebnis zurückgegeben. Wenn die ID ungültig ist, kommt eine entsprechende Fehlermeldung zurück:

```
>>> im.fetch("100", "(BODY[TEXT])")
('OK', [None])
>>> im.fetch("KeineID", "(BODY[TEXT])")
('NO', [b'Error in IMAP command received by server.'])
```

im.create(mailbox)

Erstellt eine neue Mailbox namens *mailbox*.

```
>>> im.create("INBOX.Hallo")
('OK', [b'"INBOX.Hallo" created.'])
```

im.delete(mailbox)

Löscht die Mailbox *mailbox*.

```
>>> im.delete("INBOX.Hallo")
('OK', [b'Folder deleted.'])
```

im.rename(old_mailbox, new_mailbox)

Benennt die Mailbox *old_mailbox* in *new_mailbox* um.

```
>>> im.rename("INBOX.Hallo", "INBOX.Ciao")
('OK', [b'Folder renamed.'])
```

im.copy(message_set, new_mailbox)

Kopiert die E-Mails *message_set* in die Mailbox *new_mailbox*.

im.search(charset, criterion[, ...])

Sucht innerhalb der ausgewählten Mailbox nach E-Mails, die auf die angegebenen Kriterien passen. Als Kriterium *criterion* kann entweder der String "ALL" (alle Mails erfüllen dieses Kriterium) oder ein String des Formats "(FROM \"Johannes\")" verwendet werden. Das zweite Kriterium ist für alle Mails erfüllt, die von einem gewissen »Johannes« geschrieben wurden.

Der Parameter *charset* spezifiziert das Encoding von *criterion* in Form eines Strings. Üblicherweise wird der Parameter *charset* nicht benötigt und *None* übergeben.

Die Funktion *search* gibt die IDs der gefundenen E-Mails in Form einer Liste zurück.

```
>>> im.search(None, "(FROM \"Johannes\")")
('OK', [b'1 2 3'])
>>> im.search(None, "(FROM \"Johann\")")
('OK', [b'1 2 3'])
```

```
>>> im.search(None, "(FROM \"Johanninski\")")
('OK', [b''])
```

Beispiel

Im folgenden Beispielprogramm soll das Modul `imaplib` dazu verwendet werden, zu einem IMAP4-Server zu verbinden und alle enthaltenen E-Mails einer bestimmten Mailbox anzuzeigen. Dabei soll dem Benutzer die Möglichkeit gegeben werden, die Mailbox zu wählen.

Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
import imaplib

im = imaplib.IMAP4("imap.beispiel.de")
im.login(b"Benutzername", b"Passwort")

print("Vorhandene Mailboxen:")
for mb in im.list()[1]:
    name = mb.split(b"\".\"")[1]
    print(" - {0}".format(name.decode().strip(" \")))

mb = input("Welche Mailbox soll angezeigt werden: ")
im.select(mb)
status, daten = im.search(None, "ALL")

for mailnr in daten[0].split():
    typ, daten = im.fetch(mailnr, "(RFC822)")
    print("{0}\n+++\n".format(daten[0][1].decode()))

im.close()
im.logout()
```

Zunächst wird eine Instanz der Klasse `IMAP4` erzeugt und zu einem IMAP4-Server verbunden. Dann werden mithilfe der Methode `list` alle im Hauptordner des IMAP4-Kontos vorhandenen Mailboxen durchlaufen und die Namen der Mailboxen auf dem Bildschirm angezeigt. Dabei ist zu beachten, dass die Methode `list` die Namen der Mailboxen mit zusätzlichen Informationen zurückgibt. Diese Informationen müssen herausgefiltert werden, bevor der Mailboxname angezeigt werden kann. Nachdem die Namen angezeigt wurden, wird der Benutzer dazu aufgefordert, einen der angegebenen Mailbox-Namen auszuwählen.

Die vom Benutzer ausgewählte Mailbox wird dann mithilfe der Methode `select` auch auf dem Server ausgewählt. Der danach aufgerufenen Methode `search` übergeben wir den String `"ALL"`, was den Mailserver dazu veranlasst, Daten über alle E-Mails der ausgewählten Mailbox zurückzugeben.

Anschließend iterieren wir in einer `for`-Schleife über die Liste von Mail-IDs, die `search` zurückgegeben hat, und laden die jeweilige Mail mit `fetch` vollständig herunter. Die heruntergeladene Mail wird auf dem Bildschirm ausgegeben.

Schlussendlich schließen wir die ausgewählte Mailbox und beenden die Verbindung mit dem Server.

Beachten Sie auch bei diesem Beispielprogramm, dass keine Fehlerbehandlung durchgeführt wurde. In einem fertigen Programm sollten sowohl die Verbindungsanfrage als auch das Login und insbesondere die Benutzereingabe überprüft werden.

20.4.4 Erstellen komplexer E-Mails – email

In den vorherigen Abschnitten wurde besprochen, wie Sie E-Mails über einen SMTP-Server versenden und von einem POP3- oder IMAP4-Server heruntergeladen. Trotz alledem bleibt eine Frage weiterhin offen: Wie Sie wissen, basiert das Senden und Empfangen von E-Mails auf reinen ASCII-Protokollen. Das bedeutet vor allem, dass mit diesen Protokollen keine Binärdaten verschickt werden können. Außerdem sind Sonderzeichen, die nicht dem 7-Bit-ASCII-Standard entsprechen, problematisch.

Um also solche Zeichen oder Binärdaten verschicken zu können, wurde der sogenannte MIME-Standard (*Multipurpose Internet Mail Extension*) entwickelt, der Sonderzeichen und Binärdaten so kodiert, dass sie als eine Folge reiner ASCII-Zeichen versandt werden können. Durch eine solche Form der Kodierung steigt allerdings die Größe der zu übermittelnden Daten. Zudem definiert der MIME-Standard verschiedene Dateitypen und legt eine Syntax fest, mit der Dateianhänge einem bestimmten Dateityp zugeordnet werden, so dass die Dateien beim Empfänger leichter verarbeitet werden können.

Das `email`-Paket ist sehr mächtig, weswegen hier nur ein Teil seiner Funktionalität besprochen werden kann. Zunächst werden wir uns darum kümmern, wie eine simple ASCII-Mail mittels `email` erstellt werden kann. Darauf aufbauend werden wir zu komplexeren MIME-kodierten Mails übergehen.

Erstellen einer einfachen E-Mail

Als Basisklasse für eine neue E-Mail dient die Klasse `Message` des Moduls `email.message`. Das folgende Beispielprogramm zeigt, wie sie zu verwenden ist:

```
from email.message import Message

msg = Message()
msg.set_payload("Dies ist meine selbst erstellte E-Mail.")
```

```
msg["Subject"] = "Hallo Welt"
msg["From"] = "Donald Duck <don@ld.de>"
msg["To"] = "Onkel Dagobert <d@gobert.de>"

print(msg.as_string())
```

Zunächst erzeugen wir eine Instanz der Klasse `Message`. Der Konstruktor dieser Klasse erwartet keine Argumente. Durch die Methode `set_payload` (dt. »Nutzlast«) wird der E-Mail ein Text hinzugefügt.

Jetzt fehlt nur noch der E-Mail-Header. Um diesen hinzuzufügen, kann die `Message`-Instanz wie ein Dictionary angesprochen werden. Auf diese Weise werden die einzelnen Teile des Headers hinzugefügt. Wichtig sind dabei "Subject" für den Betreff, "From" für den Absender und "To" für den Empfänger der Mail.

Zu guter Letzt wird die entstandene E-Mail durch die Methode `as_string` in einen String geschrieben und ausgegeben. Dieser String könnte der Methode `sendmail` des Moduls `smtpplib` übergeben und somit als E-Mail verschickt werden. Die Ausgabe des Beispielprogramms, also die erzeugte E-Mail, sieht folgendermaßen aus:

```
Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>
```

Dies ist meine selbst erstellte E-Mail.

Erstellen einer E-Mail mit Anhängen

Wir haben angekündigt, dass es das Paket `email` ermöglicht, Binärdaten per E-Mail zu verschicken. Dafür ist das Modul `email.mime` zuständig. Das folgende Beispielprogramm erstellt eine E-Mail und fügt eine Bilddatei als Anhang ein:

```
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage
from email.mime.text import MIMEText

msg = MIMEMultipart()
msg["Subject"] = "Hallo Welt"
msg["From"] = "Donald Duck <don@ld.de>"
msg["To"] = "Onkel Dagobert <d@gobert.de>"

text = MIMEText("Dies ist meine selbst erstellte E-Mail.")
msg.attach(text)

f = open("buch.png", "rb")
bild = MIMEImage(f.read())
```

```
f.close()
msg.attach(bild)

print(msg.as_string())
```

Zunächst wird eine Instanz der Klasse `MIMEMultipart` erzeugt. Diese repräsentiert eine E-Mail, die MIME-kodierte Binärdaten enthalten kann. Wie im vorherigen Beispiel werden Betreff, Absender und Empfänger nach Art eines Dictionarys hinzugefügt.

Danach wird eine Instanz der Klasse `MIMEText` erzeugt, die den reinen Text der E-Mail enthalten soll. Diese Instanz wird mithilfe der Methode `attach` an die `MIMEMultipart`-Instanz angehängt.

Genauso verfahren wir mit dem Bild: Es wird eine Instanz der Klasse `MIMEImage` erzeugt und mit den Binärdaten des Bildes gefüllt. Danach wird sie mittels `attach` an die E-Mail angefügt.

Schlussendlich wird die `MIMEMultipart`-Instanz durch Aufruf der Methode `as_string` in einen String konvertiert, der so als reine ASCII-E-Mail versendet werden kann. Der angefügte Anhang wird von E-Mail-Programmen als Grafik erkannt und dann dementsprechend präsentiert.

Die Ausgabe des Beispiels sieht in etwa so aus:

```
Content-Type: multipart/mixed; boundary="=====0094312333=="
MIME-Version: 1.0
Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>

-----0094312333==
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Dies ist meine selbst erstellte E-Mail.
-----0094312333==
Content-Type: image/png
MIME-Version: 1.0
Content-Transfer-Encoding: base64

iVBORw0KGgoAAAANSUHEUgAAASwAAD8CAIAAABCvg65AAAACXBIWXMAACcQAAAnE
B3RJTUUH1wkMERU1+MujgAAIABJREFUeNrsfXecJWWV9nPet+rez2ThzDkqCBLB
[...]
-----0094312333==--
```

Sie sehen, dass sowohl der Text als auch das Bild in ähnlicher Form kodiert wurden. Die Aufbereitung der beiden Sektionen zum Textteil der E-Mail und zu einem Bild im Anhang erledigt Ihr Mail-Programm. Das `mime`-Paket bietet auch eine entsprechende Funktionalität an, auf die wir noch zu sprechen kommen werden.

Hier wurden nur `MIMEText` und `MIMEImage` besprochen. Im Folgenden sind alle verfügbaren MIME-Datentypen aufgelistet:

- ▶ `email.mime.application.MIMEApplication` für ausführbare Programme
- ▶ `email.mime.audio.MIMEAudio` für Sounddateien
- ▶ `email.mime.image.MIMEImage` für Grafikdateien
- ▶ `email.mime.message.MIMEMessage` für Message-Instanzen
- ▶ `email.mime.image.MIMEText` für reinen Text

Beim Instantiieren all dieser Klassen müssen Sie die jeweiligen Binärdaten bzw. den Text, den die entsprechende Instanz enthalten soll, als ersten Parameter des Konstruktors übergeben. Wichtig ist noch, dass alle hier vorgestellten Klassen von der Basisklasse `Message` abgeleitet sind, also über die Methoden dieser Basis-klasse verfügen.

Internationale Zeichensätze

Bisher wurde besprochen, wie der MIME-Standard dazu verwendet werden kann, Binärdaten im Anhang einer E-Mail zu versenden. Beim Text der E-Mail waren wir aber bislang auf die Zeichen des 7-Bit-ASCII-Standards beschränkt. Die Frage ist, wie ein spezielles Encoding innerhalb einer E-Mail verwendet werden kann. Auch dies ermöglicht der MIME-Standard. Das folgende Beispielprogramm erstellt eine einfache E-Mail, deren Text ein Euro-Zeichen enthält:

```
from email.mime.text import MIMEText

text = "39,90\u20AC"
msg = MIMEText(text.encode("cp1252"), _charset="cp1251")
msg["Subject"] = "Hallo Welt"
msg["From"] = "Donald Duck <don@ld.de>"
msg["To"] = "Onkel Dagobert <d@gobert.de>"

print(msg.as_string())
```

Als Erstes erzeugen wir einen String, der das Euro-Zeichen enthält, das nicht Teil des ASCII-Standards ist. Nachfolgend wird der String ins Windows-Encoding `cp1252` kodiert und bei der Instantiierung der Klasse `MIMEText` übergeben. Das verwendete Encoding muss dem Konstruktor ebenfalls über den Parameter

`_charset` bekannt gemacht werden. Der nun folgende Teil des Programms ist bereits von den anderen Beispielen her bekannt.

Der MIME-kodierte Text, den das Beispielprogramm ausgibt, sieht folgendermaßen aus:

```
Content-Type: text/plain; charset="cp1252"
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>
```

MzksOTCA

Dabei entspricht MzksOTCA der MIME-Kodierung des Texts 39,90€.

Es kann durchaus vorkommen, dass auch Einträge im Header der E-Mail Sonderzeichen enthalten. Solche können mithilfe der Klasse `Header` kodiert werden:

```
from email.mime.text import MIMEText
from email.header import Header
msg = MIMEText("Hallo Welt")
msg["Subject"] = Header("39,90\u20AC", "cp1252")
[...]
```

Eine E-Mail einlesen

Zum Schluss möchten wir noch ein kurzes Beispiel dazu geben, dass eine abgespeicherte E-Mail auch wieder eingelesen und automatisch zu der bislang besprochenen Klassenstruktur aufbereitet werden kann. Dazu folgendes Beispiel:

```
import email

mail = """Subject: Hallo Welt
From: Donald Duck <don@ld.de>
To: Onkel Dagobert <d@gobert.de>

Hallo Welt
"""

msg = email.message_from_string(mail)
print(msg["From"])
```

Im Beispielprogramm ist eine E-Mail in Form eines Strings vorhanden und wird durch die Funktion `message_from_string` eingelesen. Diese Funktion gibt eine vollwertige Message-Instanz zurück, wie die darauf folgende `print`-Ausgabe beweist:

Donald Duck <don@ld.de>

Alternativ hätten wir auch die Funktion `message_from_file` verwenden können, um die E-Mail aus einer Datei zu lesen. Dieser Funktion hätten wir dann ein geöffnetes Dateiobjekt übergeben müssen.

20.5 Telnet – telnetlib

Das Modul `telnetlib` ermöglicht die Verwendung des sogenannten *Telnet*-Netzwerkprotokolls (*Teletype Network*). Telnet wurde als möglichst einfaches bidirektionales Netzwerkprotokoll konzipiert. Häufig wird Telnet dazu verwendet, einen kommandozeilenbasierenden Zugriff auf einen entfernten Rechner zu ermöglichen. Da das Telnet-Protokoll aber keine Möglichkeit zur Verschlüsselung der übertragenen Daten bietet, wurde es nach und nach von anderen, in diesem Bereich stärkeren Protokollen wie beispielsweise SSH verdrängt.

Das Modul `telnetlib` enthält im Wesentlichen die Klasse `Telnet`, über die die weitere Kommunikation mit dem entfernten Rechner abläuft. Der Konstruktor der Klasse `Telnet` hat die folgende Schnittstelle:

`telnetlib.Telnet([host[, port[, timeout]])`

Erzeugt eine Instanz der Klasse `Telnet`. Optional übergeben Sie bereits hier den Hostnamen und den Port des Rechners übergeben, zu dem eine Verbindung hergestellt werden soll. Wenn keiner der Parameter angegeben wird, muss die erzeugte `Telnet`-Instanz explizit durch Aufruf der Methode `open` verbunden werden. Die Angabe einer Portnummer ist nur dann notwendig, wenn die Verbindung nicht über den Standardport 23 ablaufen soll.

Über den optionalen Parameter *timeout* lässt sich ein Timeout-Wert in Sekunden angeben, der beim Verbindungsversuch zum Server eingehalten werden soll. Wenn *timeout* nicht angegeben wurde, wird ein Standardwert als Timeout verwendet.

Die Klasse `Telnet`

Nachdem sie erzeugt und mit dem Zielrechner verbunden wurde, kann eine `Telnet`-Instanz zur Kommunikation mit dem verbundenen Rechner verwendet werden. Dazu enthält sie eine Reihe Methoden, von denen die wichtigsten im Folgenden erläutert werden sollen. Dabei sei *t* eine Instanz der Klasse `telnetlib.Telnet`.

t.read_until(expected[, timeout])

Liest ankommende Daten, bis der String *expected* empfangen wurde. Alternativ geben Sie einen Timeout in Sekunden als zweiten Parameter an, nach dessen Ablauf der Lesevorgang abgebrochen wird. Die gelesenen Daten werden als `bytes`-String zurückgegeben.

t.read_all()

Liest alle ankommenden Daten, bis die Verbindung geschlossen wird. Beachten Sie, dass diese Methode das Programm auch so lange blockiert. Die gelesenen Daten werden als `bytes`-String zurückgegeben.

t.open(host[, port])

Verbindet die `Telnet`-Instanz zum entfernten Rechner *host* unter Verwendung des Ports *port*. Diese Funktion sollte nur aufgerufen werden, wenn die Verbindungsdaten nicht bereits dem Konstruktor der Klasse `Telnet` übergeben wurden.

t.close()

Schließt die `Telnet`-Verbindung zum entfernten Rechner.

t.write(buffer)

Sendet den String *buffer* zum Verbindungspartner. Diese Funktion kann das Programm blockieren, wenn die Daten nicht sofort geschrieben werden können.

Beispiel

Im folgenden Beispielprogramm soll das Modul `telnetlib` dazu verwendet werden, zu einem POP3-Server zu verbinden. Dabei möchten wir auf die abstrahierte Schnittstelle des Moduls `poplib` verzichten und dem Server direkt POP3-Kommandos senden. Da das POP3-Protokoll jedoch relativ simpel ist und auf lesbaren Kommandos basiert, stellt dies kein großes Problem dar.

Das Ziel des Programms ist es, die Ausgabe des POP3-Kommandos `LIST` zu erhalten, das die Indizes aller im Posteingang liegenden Mails auflistet.

Im Programm soll die `Telnet`-Kommunikation möglichst komfortabel über eine auf POP3 zugeschnittene Klasse ablaufen:

```
import telnetlib

class POP3Telnet:

    def __init__(self, host, port):
        self.tel = telnetlib.Telnet(host, port)
        self lese_daten()
```

```

def close(self):
    self.tel.close()

def lese_daten(self):
    return self.tel.read_until(b".\r\n", 20.0)

def kommando(self, kom):
    self.tel.write("{}\r\n".format(kom)).encode()
    return self.lese_daten()

```

Dem Konstruktor der Klasse `POP3Telnet` werden `Hostname` und `Port` des POP3-Servers übergeben. Intern wird dann eine Instanz der Klasse `Telnet` erzeugt und mit diesem Server verbunden. Durch Aufruf der Methode `lese_daten` wird die Begrüßungsnachricht des Servers ausgelesen und verworfen, da sie nicht weiter von Interesse ist, aber bei späteren Lesevorgängen stören würde.

Wichtig sind die Methoden `lese_daten` und `kommando`. Die Methode `lese_daten` liest genau einen Antwortstring des POP3-Servers ein. Eine solche Antwort wird stets durch den String `".\r\n"` beendet. Der gelesene String wird zurückgegeben. Damit dieser Lesevorgang das Programm bei einem unerreichbaren Server nicht auf unbestimmte Zeit blockiert, wurde ein Timeout von 20 Sekunden festgelegt.

Die zweite wichtige Methode ist `kommando`. Sie erlaubt es, einen POP3-Befehl an den Server zu senden. Dieser Befehl wird inklusive eines abschließenden `".\r\n"` in die `Telnet`-Instanz geschrieben und von dieser an den verbundenen Rechner weitergeleitet. Schlussendlich wird die Antwort des Servers eingelesen und zurückgegeben.

Doch die Klasse ist nur der erste Teil des Beispielprogramms. Im nun folgenden zweiten Teil setzen wir die Klasse `POP3Telnet` zur Kommunikation mit einem POP3-Server ein. Dazu legen wir zunächst die Zugangsdaten für den POP3-Server fest:

```

host = b"pop.beispiel.de"
port = 110
user = "benutzername"
passwd = "passwort"

```

Jetzt erzeugen wir eine Instanz der Klasse `POP3Telnet`, die mit dem angegebenen POP3-Server verbunden ist. Dann führen wir die Anmeldeprozedur durch Senden der Kommandos `USER` und `PASS` durch.

```

pop = POP3Telnet(host, port)
pop.kommando("USER {}".format(user))
pop.kommando("PASS {}".format(passwd))

```


An dieser Stelle sind wir, wenn bei der Anmeldung alles gut gelaufen ist, dazu in der Lage, mit beliebigen POP3-Kommandos auf den Posteingang zuzugreifen. Dann schicken wir das eingangs erwähnte `LIST`-Kommando. Das `LIST`-Kommando des POP3-Protokolls liefert eine Liste aller im Posteingang enthaltenen E-Mails. Jeder Eintrag besteht dabei aus dem ganzzahligen Index der jeweiligen E-Mail und ihrer Größe in Byte.

Beachten Sie, dass der Server auf `LIST` zwei Antwortstrings sendet, von denen uns nur der zweite interessiert, da dieser die Daten über vorhandene E-Mails enthält. Aus diesem Grund müssen wir nach dem Aufruf der Methode `kommando` noch einmal den zweiten Antwortstring einlesen. Der zurückgegebene String wird ausgegeben. Im Code sieht das folgendermaßen aus:

```
pop.kommando("LIST")
print(pop lese_daten().decode())
pop.kommando("QUIT")
pop.close()
```

Zum Schluss schicken wir das Kommando `QUIT` an den Server und schließen die Telnet-Verbindung. Die Ausgabe des Beispielprogramms könnte folgendermaßen aussehen:

```
1 623
2 614
3 1387
.
```

In diesem Fall befinden sich drei E-Mails mit den Größen 623, 614 und 1387 Byte im Posteingang.

20.6 XML-RPC

Der Standard *XML-RPC (Extensible Markup Language Remote Procedure Call)* ermöglicht den entfernten Funktions- und Methodenaufruf über eine Netzwerkschnittstelle. Dabei können entfernte Funktionen aus Sicht des Programmiers aufgerufen werden, als gehörten sie zum lokalen Programm. Das Übertragen der Funktionsaufrufe und insbesondere der Parameter und des Rückgabewertes wird vollständig von der XML-RPC-Bibliothek übernommen, so dass der Programmierer die Funktionen tatsächlich nur aufzurufen braucht.

Neben XML-RPC existieren weitere mehr oder weniger standardisierte Verfahren zum entfernten Funktionsaufruf. Da aber XML-RPC auf zwei bereits bestehenden Standards, nämlich XML und HTTP, basiert und keine völlig neuen binären Pro-

tolle einführt, ist es vergleichsweise einfach umzusetzen und daher in vielen Programmiersprachen verfügbar.

Da XML-RPC unabhängig von einer bestimmten Programmiersprache entwickelt wurde, ist es durchaus möglich, Client und Server in zwei verschiedenen Sprachen zu schreiben. Aus diesem Grund musste man sich bei der XML-RPC-Spezifikation auf einen kleinsten gemeinsamen Nenner einigen, was die Eigenheiten bestimmter Programmiersprachen und besonders die verfügbaren Datentypen anbelangt. Sie werden feststellen, dass Sie bei einer Funktion mit einer XML-RPC-fähigen Schnittstelle bestimmte Einschränkungen zu beachten haben.

Im Folgenden werden wir uns zunächst damit beschäftigen, wie durch einen XML-RPC-Server bestimmte Funktionen nach außen hin aufrufbar werden. Danach widmen wir uns der Client-Seite und klären, wie solche Funktionen dann aufgerufen werden.

20.6.1 Der Server

Zum Aufsetzen eines XML-RPC-Servers wird das Modul `xmlrpc.server` benötigt. Dieses Modul enthält im Wesentlichen die Klasse `SimpleXMLRPCServer`, die einen entsprechenden Server aufsetzt und Methoden zur Verwaltung desselben bereitstellt. Der Konstruktor der Klasse hat folgende Schnittstelle:

```
SimpleXMLRPCServer(addr[, requestHandler[, logRequests[, allow_none  
[, encoding[, bind_and_activate]]]])
```

Der einzige zwingend erforderliche Parameter ist *addr*; er spezifiziert die IP-Adresse und den Port, an die der Server gebunden wird. Die Angaben müssen in einem Tupel der Form `(ip, port)` übergeben werden, wobei die IP-Adresse ein String und die Portnummer eine ganze Zahl zwischen 0 und 65535 ist. Technisch wird der Parameter an die zugrundeliegende `Socket`-Instanz weitergereicht. Der Server kann sich nur an Adressen binden, die ihm auch zugeteilt sind. Wenn für *ip* im Tupel ein leerer String angegeben wird, wird der Server an alle dem PC zugewiesenen Adressen gebunden, beispielsweise auch an `127.0.0.1` oder `localhost`.

Über den optionalen Parameter *requestHandler* legen Sie eine Art Backend fest. In den meisten Fällen reicht die Voreinstellung des Standard-Handlers `SimpleXMLRPCRequestHandler`. Die Aufgabe dieser Klasse ist es, eingehende Daten in einen Funktionsaufruf zurückzuverwandeln.

Über den Parameter *logRequest* können Sie bestimmen, ob einkommende Funktionsaufrufe protokolliert werden sollen oder nicht. Der Parameter ist mit `True` vorgelegt.

Der vierte Parameter, *allow_none*, ermöglicht es, sofern hier `True` übergeben wird, `None` in XML-RPC-Funktionen zu verwenden. Normalerweise verursacht die Verwendung von `None` eine Exception, da kein solcher Datentyp im XML-RPC-Standard vorgesehen ist. Da dies aber eine übliche Erweiterung des Standards darstellt, wird *allow_none* von vielen XML-RPC-Implementationen unterstützt.

Über den fünften Parameter *encoding* kann ein Encoding zur Datenübertragung festgelegt werden. Standardmäßig wird hier UTF-8 verwendet.

Der letzte optionale Parameter *bind_and_activate* bestimmt, ob der Server direkt nach der Instantiierung an die Adresse gebunden und aktiviert werden soll. Das ist interessant, wenn Sie die Server-Instanz vor dem Aktivieren noch manipulieren möchten, wird aber in der Regel nicht benötigt. Der Parameter ist mit `True` vorbelegt.

Für gewöhnlich reicht zur Instantiierung des XML-RPC-Servers folgender Aufruf des Konstruktors:

```
>>> from xmlrpc.server import SimpleXMLRPCServer as Server
>>> srv = Server("", 1337)
```

Die Klasse SimpleXMLRPCServer

Nachdem eine Instanz der Klasse `SimpleXMLRPCServer` erzeugt wurde, verfügt diese über Methoden, um beispielsweise Funktionen zum entfernten Aufruf zu registrieren. Die wichtigsten Methoden einer `SimpleXMLRPCServer`-Instanz werden im Folgenden erläutert.

Dazu ist noch zu sagen, dass die Klasse `SimpleXMLRPCServer` von der Klasse `socketserver` des gleichnamigen Moduls erbt. Das bedeutet insbesondere, dass ein XML-RPC-Server ebenfalls über die Methode `serve_forever` dazu instruiert wird, eine unbestimmte Anzahl von Anfragen zu beantworten. Näheres zum Modul `socketserver` erfahren Sie in Abschnitt 20.1.8.

Im Folgenden sei `s` eine Instanz der Klasse `SimpleXMLRPCServer`.

s.register_function(function[, name])

Registriert das Funktionsobjekt *function* für einen RPC-Aufruf. Das bedeutet, dass ein zu diesem Server verbundener XML-RPC-Client die Funktion *function* über das Netzwerk aufrufen kann.

Optional kann der Funktion ein anderer Name gegeben werden, über den sie für den Client zu erreichen ist. Wenn Sie einen solchen Namen angeben, kann dieser aus beliebigen Unicode-Zeichen bestehen, auch solchen, die in einem Python-Bezeichner eigentlich nicht erlaubt sind, beispielsweise ein Umlaut oder ein Punkt.

s.register_instance(instance[, allow_dotted_names])

Registriert die Instanz *instance* für den entfernten Zugriff. Wenn der verbundene Client eine Methode dieser Instanz aufruft, wird der Aufruf durch die spezielle Methode `_dispatch` geleitet. Die Methode muss folgendermaßen definiert sein:

```
def _dispatch(self, method, params):
    pass
```

Bei jedem entfernten Aufruf einer Methode dieser Instanz wird `_dispatch` aufgerufen. Der Parameter *method* enthält den Namen der aufgerufenen Methode, und *params* enthält die dabei angegebenen Parameter. Wenn die Parameter entpackt und an eine Methode weitergereicht werden sollen, kann folgender Methodenaufruf verwendet werden:

```
self.meth(*params)
```

Eine konkrete Implementierung der Methode `_dispatch`, die die tatsächliche Methode der registrierten Instanz mit dem Namen *method* aufruft und die Parameter übergibt, sähe folgendermaßen aus:

```
def _dispatch(self, method, params):
    try:
        return getattr(self, method)(*params)
    except (AttributeError, TypeError):
        return None
```

Diese Funktion gibt sowohl dann `None` zurück, wenn keine Methode mit dem Namen *method* vorhanden ist, als auch dann, wenn die Methode mit der falschen Zahl oder einem unpassenden Parameter aufgerufen wird.

Wenn Sie für den optionalen Parameter *allow_dotted_names* `True` übergeben, sind Punkte im entfernten Methodenaufruf möglich. Dadurch können Sie auch Methoden von Attributen über das Netzwerk aufrufen. Beachten Sie unbedingt, dass es damit einem Angreifer möglich gemacht wird, auf die globalen Variablen des Programms zuzugreifen und möglicherweise schädlichen Code auszuführen. Sie sollten *allow_dotted_names* nur innerhalb eines lokalen, vertrauenswürdigen Netzes auf `True` setzen, da sonst eine massive Sicherheitslücke geöffnet würde.

s.register_introspection_functions()

Registriert die Funktionen `system.listMethods`, `system.methodHelp` und `system.methodSignature` für den entfernten Zugriff. Diese Funktionen ermöglichen es einem verbundenen Client, eine Liste aller verfügbaren Funktionen und Informationen zu einzelnen dieser Funktionen zu bekommen.

Mehr zur Verwendung der Funktionen `system.listMethods`, `system.methodHelp` und `system.methodSignature` erfahren Sie in Abschnitt 20.6.2, »Der Client«.

s.register_multicall_functions()

Registriert die Funktion `system.multicall` für den entfernten Zugriff. Durch Aufruf der Funktion `system.multicall` kann der Client mehrere Methodenauf-rufe bündeln, um so Traffic zu sparen. Auch die Rückgabewerte der Methodenaufrufe werden gebündelt zurückgegeben.

Näheres zur Verwendung der Funktion `system.multicall` erläutert Abschnitt 20.6.2, »Der Client«.

Beispiel

Nachdem die wichtigsten Funktionen der Klasse `SimpleXMLRPCServer` erläutert wurden, soll an dieser Stelle ein kleines Beispielprogramm entwickelt werden. Bei dem Programm handelt es sich um einen XML-RPC-Server, der zwei mathematische Funktionen (genauer gesagt die Berechnungsfunktionen für die Fakultät und das Quadrat einer ganzen Zahl) bereitstellt, die ein verbundener Client aufrufen kann.

```
from xmlrpc.server import SimpleXMLRPCServer as Server

def fak(n):
    """ Berechnet die Fakultät der ganzen Zahl n. """
    erg = 1
    for i in range(2, n+1):
        erg *= i
    return erg

def quad(n):
    """ Berechnet das Quadrat der Zahl n. """
    return n*n

srv = Server("", 1337)
srv.register_function(fak)
srv.register_function(quad)
srv.serve_forever()
```

Zunächst werden die beiden Berechnungsfunktionen `fak` und `quad` für die Fakultät bzw. das Quadrat einer Zahl erstellt. Danach wird ein auf Port 1337 horchender XML-RPC-Server erzeugt und werden die soeben erstellten Funktionen registriert. Schlussendlich wird der Server durch Aufruf der Methode `serve_forever` gestartet und ist nun bereit, eingehende Verbindungsanfragen und Methodenauf-rufe entgegenzunehmen und zu bearbeiten.

Der hier vorgestellte Server ist natürlich nur eine Hälfte des Beispielprogramms. Im nächsten Abschnitt werden wir besprechen, wie ein XML-RPC-Client auszusehen hat, und schließlich werden wir am Ende des folgenden Abschnitts einen Client entwickeln, der mit diesem Server kommunizieren kann.

20.6.2 Der Client

Um einen XML-RPC-Client zu schreiben, wird das Modul `xmlrpc.client` der Standardbibliothek verwendet. In diesem Modul ist vor allem die Klasse `ServerProxy` enthalten, über die die Kommunikation mit einem XML-RPC-Server abläuft. Hier sehen Sie zunächst die Schnittstelle des Konstruktors der Klasse `ServerProxy`:

`ServerProxy(uri[, transport[, encoding[, verbose[, allow_none[, use_datetime]]]])`

Erzeugt eine Instanz der Klasse `ServerProxy`, die mit dem XML-RPC-Server verbunden ist, den die URI *uri* beschreibt.

An zweiter Stelle kann wie bei der Klasse `SimpleXMLRPCServer` ein Art Backend festgelegt werden. Die voreingestellten Klassen `Transport` für das HTTP-Protokoll und `SafeTransport` für das HTTPS-Protokoll dürften in den meisten Anwendungsfällen genügen.

Wenn für den vierten Parameter, *verbose*, `True` übergeben wird, gibt die `ServerProxy`-Instanz alle ausgehenden und ankommenden XML-Pakete auf dem Bildschirm aus. Dies kann besonders zur Fehlersuche hilfreich sein.

Wenn Sie für den letzten Parameter *use_datetime* `True` übergeben, wird zur Repräsentation von Datums- und Zeitangaben statt der `xmlrpc.client`-internen Klasse `DateTime` die Klasse `datetime` des gleichnamigen Moduls verwendet, die einen wesentlich größeren Funktionsumfang besitzt.

Auf die Parameter *encoding* und *allow_none* müssen wir hier nicht weiter eingehen, da sie dieselbe Bedeutung haben wie die gleichnamigen Parameter des Konstruktors der Klasse `SimpleXMLRPCServer`, der zu Beginn des letzten Abschnitts besprochen wurde.

Die Klasse `ServerProxy`

Nach der Instantiierung der Klasse `ServerProxy` ist diese mit einem XML-RPC-Server verbunden. Das bedeutet insbesondere, dass Sie alle bei diesem Server registrierten Funktionen wie Methoden der `ServerProxy`-Instanz aufrufen und verwenden können. Es ist also keine weitere Sonderbehandlung nötig.

Zusätzlich umfasst eine `ServerProxy`-Instanz drei Methoden, die weitere Informationen über die verfügbaren entfernten Funktionen bereitstellen. Beachten Sie jedoch, dass der Server diese Methoden explizit zulassen muss. Dies geschieht durch den Aufruf der Methoden `register_introspection_functions` der `SimpleXMLRPCServer`-Instanz.

Im Folgenden sei `s` eine Instanz der Klasse `ServerProxy`.

`s.system.listMethods()`

Gibt die Namen aller beim XML-RPC-Server registrierten entfernten Funktionen in Form einer Liste von Strings zurück. Die Liste enthält keine dieser Systemmethoden.

`s.system.methodSignature(name)`

Diese Methode gibt Auskunft über die Schnittstelle der registrierten Funktion mit dem Funktionsnamen *name*. Die Schnittstellenbeschreibung ist ein String im Format:

```
"string, int, int, int"
```

Dabei entspricht die erste Angabe dem Datentyp des Rückgabewertes und alle weiteren den Datentypen der Funktionsparameter. Der XML-RPC-Standard sieht vor, dass zwei verschiedene Funktionen den gleichen Namen haben dürfen, sofern sie anhand ihrer Schnittstelle unterscheidbar sind.² Aus diesem Grund gibt die Methode `system.methodSignature` nicht einen einzelnen String, sondern eine Liste von Strings zurück.

Beachten Sie, dass der Methode `system.methodSignature` nur eine tiefere Bedeutung zukommt, wenn der XML-RPC-Server in einer Sprache geschrieben wurde, bei der Funktionsparameter jeweils an einen Datentyp gebunden werden. Solche Sprachen sind beispielsweise C, C++, C# oder Java. Sollten Sie `system.methodSignature` bei einem XML-RPC-Server aufrufen, der in Python geschrieben wurde, so wird schlicht "signatures not supported" zurückgegeben.

`s.system.methodHelp(name)`

Gibt den Docstring der entfernten Funktion *name* zurück, wenn ein solcher existiert. Wenn kein Docstring gefunden werden konnte, wird ein leerer String zurückgegeben.

² Dies wird auch *Funktionsüberladung* genannt.

Beispiel

Damit wäre die Verwendung einer `ServerProxy`-Instanz beschrieben und eigentlich denkbar einfach. Das folgende Beispiel implementiert einen zu dem XML-RPC-Server des letzten Abschnitts passenden Client:

```
from xmlrpc.client import ServerProxy

cli = ServerProxy("http://ip:1337")
print(cli.fak(5))
print(cli.quad(5))
```

Sie sehen, dass das Verbinden zu einem XML-RPC-Server und das Ausführen einer entfernten Funktion nur wenige Codezeilen benötigt und damit fast so einfach ist, als befände sich die Funktion im Client-Programm selbst.

20.6.3 Multicall

Das Modul `xmlrpc.client` enthält eine Klasse namens `MultiCall`. Diese Klasse ermöglicht es, mehrere Funktionsaufrufe gebündelt an den Server zu schicken, und instruiert diesen, die Rückgabewerte ebenfalls gebündelt zurückzusenden. Auf diese Weise minimieren Sie bei häufigen Funktionsaufrufen die Netzlast.

Die Verwendung der `MultiCall`-Klasse ist denkbar einfach und soll an folgendem Beispiel verdeutlicht werden. Das Beispiel benötigt einen laufenden Server, der die Funktionen `fak` und `quad` für den entfernten Zugriff bereitstellt, also genau so einen, wie wir ihn in Abschnitt 20.6.1, »Der Server«, vorgestellt haben. Zusätzlich muss der Server den Einsatz von Multicall durch Aufruf der Methode `register_multicall_functions` erlauben.

```
from xmlrpc.client import ServerProxy, MultiCall

cli = ServerProxy("http://ip:1337")

mc = MultiCall(cli)
for i in range(10):
    mc.fak(i)
    mc.quad(i)

for ergebnis in mc():
    print(ergebnis)
```

Zunächst stellen wir wie gehabt eine Verbindung zu dem XML-RPC-Server her. Danach erzeugen wir eine Instanz der Klasse `MultiCall` und übergeben dem Konstruktor die zuvor erzeugte `ServerProxy`-Instanz.

Ab jetzt läuft die gebündelte Kommunikation mit dem Server über die `MultiCall`-Instanz. Dazu können die entfernten Funktionen `fak` und `quad` aufgerufen wer-

den, als wären es lokale Methoden der `MultiCall`-Instanz. Beachten Sie aber, dass diese Methodenaufrufe keinen sofortigen entfernten Funktionsaufruf zur Folge haben und somit auch zu dieser Zeit keinen Wert zurückgeben.

Im Beispiel werden `fak` und `quad` jeweils zehnmal mit einer fortlaufenden ganzen Zahl aufgerufen.

Durch Aufruf der `MultiCall`-Instanz (`mc()`) werden alle gepufferten entfernten Funktionsaufrufe zusammen an den Server geschickt. Als Ergebnis wird ein Iterator zurückgegeben, der über alle Rückgabewerte in der Reihenfolge des jeweiligen Funktionsaufrufes iteriert. Im Beispielprogramm nutzen wir den Iterator dazu, die Ergebnisse mit `print` auszugeben.

Gerade bei wenigen Rückgabewerten ist es sinnvoll, diese direkt zu referenzieren.

```
wert1, wert2, wert3 = mc()
```

Hier wird davon ausgegangen, dass zuvor drei entfernte Funktionsaufrufe durchgeführt wurden und dementsprechend auch drei Rückgabewerte vorliegen.

20.6.4 Einschränkungen

Der XML-RPC-Standard ist nicht auf Python allein zugeschnitten, sondern es wurde bei der Ausarbeitung des Standards versucht, einen kleinsten gemeinsamen Nenner vieler Programmiersprachen zu finden, so dass beispielsweise Server und Client auch dann problemlos miteinander kommunizieren können, wenn sie in verschiedenen Sprachen geschrieben wurden.

Aus diesem Grund bringt das Verwenden von XML-RPC einige Einschränkungen mit sich, was die komplexeren bzw. exotischeren Datentypen von Python betrifft. So gibt es im XML-RPC-Standard beispielsweise keine Repräsentation der Datentypen `complex`, `set` und `frozenset`. Auch `None` darf nur verwendet werden, wenn dies bei der Instantiierung der Server- bzw. Clientklasse explizit angegeben wurde. Das bedeutet natürlich nur, dass Instanzen dieser Datentypen nicht über die XML-RPC-Schnittstelle versendet werden dürfen. Programmintern können Sie sie weiterhin verwenden. Sollten Sie versuchen, beispielsweise eine Instanz des Datentyps `complex` als Rückgabewert einer Funktion über die XML-RPC-Schnittstelle zu versenden, so wird eine `xmlrpc.client.Fault-Exception` geworfen. Beachten Sie, dass es natürlich dennoch möglich ist, eine komplexe Zahl über eine XML-RPC-Schnittstelle zu schicken, indem Sie Real- und Imaginärteil getrennt als jeweils ganze Zahl übermitteln.

Die folgende Tabelle listet alle im XML-RPC-Standard vorgesehenen Datentypen auf und beschreibt, wie sich diese in Python verwenden lassen.

XML-RPC	Python	Anmerkungen
boolesche Werte	bool	–
ganze Zahlen	int	–
Gleitkommazahlen	float	–
Strings	str	–
Arrays	list	In der Liste dürfen als Elemente nur XML-RPC-konforme Instanzen verwendet werden.
Strukturen	dict	Alle Schlüssel müssen Strings sein. Als Werte dürfen nur XML-RPC-konforme Instanzen verwendet werden.
Datum/Zeit	DateTime	Der spezielle Datentyp <code>xmlrpc.client.DateTime</code> wird verwendet. ³
Binärdaten	Binary	Der spezielle Datentyp <code>xmlrpc.client.Binary</code> wird verwendet.

Tabelle 20.11 Erlaubte Datentypen bei XML-RPC

Es ist möglich, Instanzen von selbst erstellten Klassen zu verwenden. In einem solchen Fall wird die Instanz in ein Dictionary, also eine Struktur, umgewandelt, in der die Namen der enthaltenen Attribute als Schlüssel eingetragen werden und die jeweils referenzierten Instanzen als Werte. Dies geschieht automatisch. Beachten Sie jedoch, dass das auf der Gegenseite ankommende Dictionary nicht automatisch wieder in eine Instanz der ursprünglichen Klasse umgewandelt wird.

Die letzten beiden Datentypen in der Tabelle 20.11 sind uns noch nicht begegnet. Es handelt sich dabei um Datentypen, die im Modul `xmlrpc.client` enthalten und speziell auf die Verwendung im Zusammenhang mit XML-RPC zugeschnitten sind. Die beiden erwähnten Datentypen `DateTime` und `Binary` werden im Folgenden kurz erläutert.

Der Datentyp `DateTime`

Der Datentyp `DateTime` des Moduls `xmlrpc.client` kann verwendet werden, um Datums- und Zeitangaben über eine XML-RPC-Schnittstelle zu versenden. Statt einer `DateTime`-Instanz kann, sofern der entsprechende Parameter bei der Instanziierung der `ServerProxy`-Instanz übergeben wurde, auch direkt eine Instanz der

³ Beachten Sie, dass es sich dabei nicht um den Datentyp `datetime` aus dem Modul `datetime` der Standardbibliothek handelt.

bekannten Datentypen `datetime.date`, `datetime.time` oder `datetime.datetime` verwendet werden.

Bei der Erzeugung einer Instanz des Datentyps `DateTime` kann entweder einer der Datentypen des Moduls `datetime` übergeben werden oder ein UNIX-Timestamp als ganze Zahl:

```
>>> import xmlrpc.client
>>> import datetime
>>> xmlrpc.client.DateTime(987654321)
<DateTime '20010419T06:25:21' at 80fda4c>
>>> xmlrpc.client.DateTime(datetime.datetime.today())
<DateTime '20090107T21:57:47' at b7ad3cec>
```

Die erste `DateTime`-Instanz wurde aus einem UNIX-Timestamp erzeugt, während dem `DateTime`-Konstruktor bei der zweiten Instantiierung eine `datetime.datetime`-Instanz übergeben wurde.

Instanzen des Datentyps `DateTime` können Sie bedenkenlos in Form eines Rückgabewertes oder eines Parameters über eine XML-RPC-Schnittstelle senden.

Der Datentyp Binary

Der Datentyp `Binary` des Moduls `xmlrpclib` wird zum Versenden von Binärdaten über eine XML-RPC-Schnittstelle verwendet. Bei der Instantiierung des Datentyps `Binary` wird ein `bytes`-String übergeben, der die binären Daten enthält. Diese können auf der Gegenseite über das Attribut `data` wieder ausgelesen werden:

```
>>> import xmlrpc.client
>>> b = xmlrpc.client.Binary(b"\x00\x01\x02\x03")
>>> b.data
b'\x00\x01\x02\x03'
```

Instanzen des Datentyps `Binary` können Sie bedenkenlos in Form eines Rückgabewertes oder eines Parameters über eine XML-RPC-Schnittstelle senden.

»Intelligente Fehler zu machen, ist eine große Kunst.«
– Federico Fellini

21 Debugging

Das sogenannte *Debugging* bezeichnet das Aufspüren und Beseitigen von Fehlern, sogenannten *Bugs*, in einem Programm. Üblicherweise steht dem Programmierer dabei ein sogenannter *Debugger* zur Verfügung. Das ist ein wichtiges Entwicklerwerkzeug, das es ermöglicht, den Ablauf eines Programms zu überwachen und an bestimmten Stellen anzuhalten. Wenn der Programmablauf in einem Debugger angehalten wurde, kann der momentane Programmstatus genau analysiert werden. Auf diese Weise können Fehler sehr viel schneller gefunden werden als durch bloßes gedankliches Durchgehen des Quellcodes.

Im ersten Abschnitt widmen wir uns dem Debugging allgemein. Danach erläutern wir Module, die nicht direkt etwas mit dem Debugger zu tun haben, sondern allgemein bei der Fehlersuche in einem Programm hilfreich sind. So erklären wir beispielsweise, welche Möglichkeiten Python zum Erstellen einer Laufzeitanalyse oder automatisierter Tests bietet.

21.1 Der Debugger

Im Lieferumfang von Python ist ein Programm zum Debuggen von Python-Code enthalten, der sogenannte *PDB* (*Python Debugger*). Dieser Debugger läuft in einem Konsolenfenster und ist damit weder übersichtlich noch intuitiv. Aus diesem Grund haben wir uns dagegen entschieden, den PDB an dieser Stelle zu besprechen. Sollten Sie dennoch Interesse an diesem Debugger haben, beispielsweise gerade wegen seiner kommandozeilenbasierenden Benutzerschnittstelle, so finden Sie nähere Informationen dazu in der Python-Dokumentation.

Viele moderne Entwicklungsumgebungen¹ für Python bieten einen umfangreichen, integrierten Debugger mit grafischer Benutzeroberfläche, der die Fehlersuche in einem Python-Programm recht komfortabel gestaltet. Auch IDLE bietet einen rudimentären grafischen Debugger:

¹ Eine Auflistung der gängigsten Entwicklungsumgebungen für Python finden Sie im Anhang.

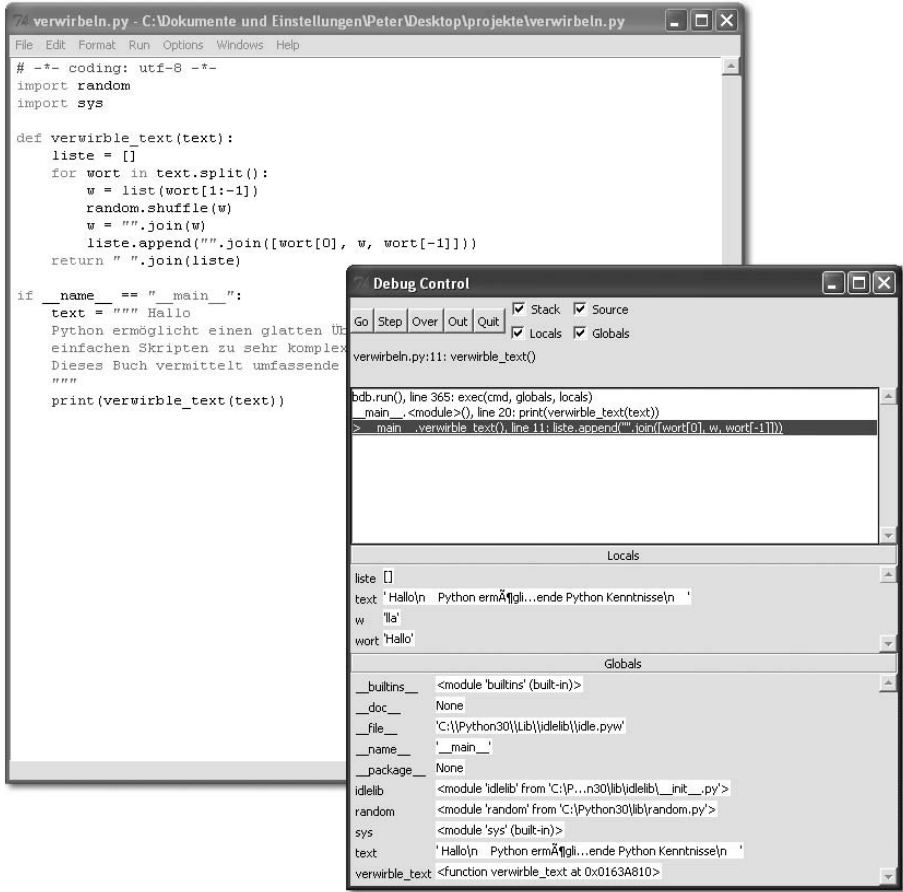


Abbildung 21.1 Der grafische Debugger von IDLE

Um den Debugger in IDLE zu aktivieren, klicken Sie in der Python-Shell auf den Menüpunkt **DEBUG • DEBUGGER** und führen dann das auf Fehler zu untersuchende Programm ganz normal per **RUN • RUN MODULE** aus. Es erscheint zusätzlich zum Editorfenster ein Fenster, in dem die aktuell ausgeführte Codezeile steht. Durch einen Doppelklick auf diese Zeile wird sie im Programmcode hervorgehoben, so dass Sie stets wissen, wo genau Sie sich im Programmablauf befinden.

Da es abgesehen von IDLE noch eine Menge weitere Python-IDEs gibt und IDLE bei Weitem nicht das Nonplusultra ist, wäre es müßig, an dieser Stelle eine detaillierte Einführung in den grafischen Debugger von IDLE zu geben. Allerdings ähneln sich die Funktionen der diversen grafischen Debugger sehr stark, so dass

wir allgemein darauf eingehen möchten, welche Funktionen ein grafischer Debugger in der Regel anbietet.

Das grundsätzliche Prinzip eines Debuggers ist es, dem Programmierer das schrittweise Ausführen eines Programms zu ermöglichen, um sich somit von Zeile zu Zeile ein genaues Bild davon zu machen, welche Änderungen sich ergeben haben und wie sich diese im Laufe des Programms auswirken. Eine sogenannte *Debugging-Session* beginnt zumeist damit, dass der Programmierer sogenannte *Breakpoints* im Programm verteilt. Beim Starten des Debuggers wird das Programm normal ausgeführt, bis der Programmfluss auf den ersten Breakpoint stößt. An dieser Stelle hält der Debugger den Programmlauf an und erlaubt das Eingreifen des Programmierers. Viele Debugger halten auch direkt nach dem Starten an der ersten Programmzeile und warten auf weitere Instruktionen des Programmierers.

Wenn das Programm angehalten wurde und der Programmfluss somit an einer bestimmten Zeile im Quellcode steht, hat der Programmierer mehrere Möglichkeiten, den weiteren Programmlauf zu steuern. Diese Möglichkeiten, im Folgenden Befehle genannt, finden Sie in einem grafischen Debugger üblicherweise an prominenter Stelle in einer Toolbar am oberen Rand des Fensters, da es sich dabei wirklich um die essentiellen Fähigkeiten eines Debuggers handelt.

- ▶ Mit dem Befehl *Step over* veranlassen Sie den Debugger dazu, zur nächsten Quellcodezeile zu springen und dort erneut zu halten.
- ▶ Der Befehl *Step into* verhält sich ähnlich wie *Step over*, mit dem Unterschied, dass bei *Step into* auch in Funktions- oder Methodenaufrufe hineingesprungen wird, während diese bei *Step over* übergangen werden.
- ▶ Der Befehl *Step out* springt aus der momentanen Unterfunktion heraus wieder dorthin, wo die Funktion aufgerufen wurde. *Step out* kann damit gewissermaßen als Umkehrfunktion zu *Step into* gesehen werden.
- ▶ Der Befehl *Run* führt das Programm weiter aus, bis der Programmfluss auf den nächsten Breakpoint stößt oder das Programmende eintritt. Einige Debugger erlauben es mit einem ähnlichen Befehl, zu einer bestimmten Quellcodezeile zu springen oder den Programmcode bis zur Cursorposition auszuführen.

Neben diesen Befehlen, mit denen sich der Programmlauf steuern lässt, stellt ein Debugger einige Hilfsmittel bereit, mit deren Hilfe der Programmierer den Zustand des angehaltenen Programms vollständig erfassen kann. Welche dieser Hilfsmittel vorhanden sind und wie sie bezeichnet werden, ist von Debugger zu Debugger verschieden, dennoch möchten wir an dieser Stelle eine Übersicht über die gebräuchlichsten Hilfsmittel geben:

- ▶ Das grundlegendste Hilfsmittel ist das Anzeigen einer Liste aller lokalen und globalen Referenzen mitsamt referenzierter Instanz, die im momentanen Programmkontext existieren. Auf diese Weise lassen sich Wertänderungen verfolgen und Fehler, die dabei entstehen, relativ leicht aufspüren.
- ▶ Zusätzlich zu den lokalen und globalen Referenzen ist der sogenannte *Stack* von Interesse. In diesem wird die momentane Funktionshierarchie aufgelistet, so dass sich genau verfolgen lässt, welche Funktion welche Unterfunktion aufgerufen hat.
- ▶ Gerade in Bezug auf die Programmiersprache Python bieten einige Debugger eine interaktive Shell, die sich im Kontext des angehaltenen Programms befindet und es dem Programmierer erlaubt, komfortabel Referenzen zu verändern, um somit in den Programmfluss einzugreifen.
- ▶ Ein sogenannter *Post-Mortem Debugger* kann in Anlehnung an den vorherigen Punkt betrachtet werden. In einem solchen Modus hält der Debugger das Programm erst an, wenn eine nicht abgefangene Exception aufgetreten ist. Im angehaltenen Zustand verfügt der Programmierer wieder über eine Shell sowie über die genannten Hilfsmittel, um dem Fehler auf die Spur zu kommen. Diese Form des Debuggens wird »post mortem« genannt, da sie erst nach dem Auftreten des tatsächlichen Fehlers, also nach dem »Tod« des Programms, aktiviert wird.

Mithilfe dieser Einführung in die Techniken des Debuggens und mit ein wenig Spieltrieb dürfte es für Sie kein Problem darstellen, den Debugger Ihrer favorisierten IDE in den Griff zu bekommen.

Abgesehen von dem eigentlichen Debugger, umfasst die Standardbibliothek Pythons noch einige Module, die speziell im Kontext des Debuggens von Bedeutung sind – sei es innerhalb der interaktiven Python-Shell eines Debuggers oder völlig losgelöst vom Debugger. Diese Module werden in den folgenden Abschnitten besprochen.

21.2 Inspizieren von Instanzen – inspect

Das Modul `inspect` stellt Funktionen bereit, über die der Programmierer detaillierte Informationen über eine Instanz erlangen kann. So könnten Sie beispielsweise den Inhalt einer Klasse oder die Parameterliste einer Funktion ermitteln. Damit eignet sich `inspect` besonders zum Erstellen von detaillierten Debug-Ausgaben.

Grundsätzlich lässt sich die Funktionalität von `inspect` in drei Teilbereiche gliedern, die im Folgenden erklärt werden sollen:

- ▶ Funktionen, die sich auf Datentypen, Attribute und Methoden einer Instanz beziehen
- ▶ Funktionen, die sich auf ein Stück des Quellcodes beziehen, das im Zusammenhang mit einer Instanz steht
- ▶ Funktionen, die sich auf Klassen- und Funktionsobjekte beziehen

Bevor Sie die Beispiele dieser Abschnitte ausführen können, müssen Sie das Modul `inspect` einbinden:

```
>>> import inspect
```

21.2.1 Datentypen, Attribute und Methoden

In diesem Abschnitt werden die wichtigsten Funktionen des Moduls `inspect` besprochen, mit deren Hilfe der Programmierer Informationen über den Datentyp, Attribute oder Methoden einer Instanz abfragen kann.

`inspect.getmembers(object[, predicate])`

Gibt alle Attribute und Methoden, auch *Member* genannt, der Instanz *object* in Form einer Liste von Tupeln zurück. Jedes Tupel enthält dabei den Namen des jeweiligen Members als erstes Element und den Wert des Members als zweites Element. Im Falle einer Methode entspricht das Funktionsobjekt dem Wert des Members. Die zurückgegebene Liste ist nach Member-Namen sortiert.

```
>>> class klasse:
...     def __init__(self):
...         self.a = 1
...         self.b = 2
...         self.c = 3
...
...     def hallo(self):
...         return "welt"
...
>>> inspect.getmembers(klasse())
[('__class__', <class '__main__.klasse'>), [...]
 ('a', 1), ('b', 2), ('c', 3), ('hallo', <bound method klasse.hallo
 of <__main__.klasse object at 0xb7a9d08c>>)]
```

Für den optionalen Parameter *predicate* kann eine Filterfunktion übergeben werden. Diese Funktion wird für jeden Member der Instanz *object* aufgerufen und bekommt diesen als einzigen Parameter übergeben. Es werden alle Member in die Ergebnisliste aufgenommen, für die die Filterfunktion `True` zurückgegeben hat.


```
>>> inspect.getmembers(klasse(), lambda x: inspect.ismethod(x))
[('__init__', <bound method klasse.__init__ of <__main__.klasse
object at 0xb7a9df2c>>), ('hallo', <bound method klasse.hallo of
<__main__.klasse object at 0xb7a9df2c>>)]
```

In diesem Fall wurde eine Lambda-Form übergeben, die `True` zurückgibt, wenn es sich bei dem Member `x` um eine Methode handelt. Dementsprechend klein ist die Ergebnisliste. Auf die verwendete Funktion `ismethod` gehen wir später noch ein.

Zusätzlich zu `getmembers` enthält das Modul `inspect` eine Menge Funktionen, deren Namen allesamt mit `is` beginnen und die testen, ob ein Objekt eine bestimmte Eigenschaft erfüllt. Eine dieser Funktionen wurde im vorangegangenen Beispiel bereits erfolgreich eingesetzt. Die folgende Tabelle listet die wichtigsten dieser Funktionen auf.

Funktion	Rückgabewert
<code>isclass(object)</code>	True, wenn object eine Klasse ist, andernfalls False
<code>ismodule(object)</code>	True, wenn object ein Modul ist, andernfalls False
<code>ismethod(object)</code>	True, wenn object eine Methode ist, andernfalls False
<code>isfunction(object)</code>	True, wenn object eine Funktion ist, andernfalls False
<code>istraceback(object)</code>	True, wenn object ein Traceback-Objekt ist, andernfalls False
<code>iscode(object)</code>	True, wenn object ein Code-Objekt ist, andernfalls False. Näheres zum Code-Objekt erfahren Sie im nächsten Abschnitt.
<code>isbuiltin(object)</code>	True, wenn object eine Built-in Function ist, andernfalls False
<code>isroutine(object)</code>	True, wenn object eine Funktion oder Methode ist, andernfalls False

Tabelle 21.1 Auf Member bezogene Funktionen des Moduls `inspect`

Grundsätzlich werden die Methoden folgendermaßen verwendet:

```
>>> def a():
...     print("Hallo Welt")
...
>>> inspect.isbuiltin(a)
False
>>> inspect.isfunction(a)
True
>>> inspect.isroutine(a)
True
```

21.2.2 Quellcode

In diesem Abschnitt werden die wichtigsten Funktionen des Moduls `inspect` behandelt, die sich direkt auf den Quellcode beispielsweise einer Funktion beziehen.

`inspect.getfile(object)`

Gibt den Namen der Datei zurück, in der das Objekt *object* definiert wurde. Dabei kann es sich sowohl um eine Quellcode-Datei als auch um eine Bytecode-Datei handeln.

Diese Funktion wirft eine `TypeError-Exception`, wenn es sich bei *object* um ein eingebautes Objekt, beispielsweise eine Built-in Function, handelt. Das liegt daran, dass Built-in Functions intern in C implementiert sind und somit keiner Quelldatei zugeordnet werden können.

```
>>> inspect.getfile(inspect.getfile)
'C:\\Python30\\lib\\inspect.py'
```

In diesem Beispiel wurde die Funktion `getfile` verwendet, um herauszufinden, in welcher Quelldatei sie selbst definiert ist.

`inspect.getmodule(object)`

Gibt die Modulinstanz des Moduls zurück, in dem das Objekt *object* definiert wurde.

```
>>> inspect.getmodule(inspect.getmodule)
<module 'inspect' from 'C:\\Python30\\lib\\inspect.py'>
```

In diesem Beispiel wurde `getmodule` dazu verwendet, den Pfad des Moduls herauszufinden, in dem die Funktion selbst definiert ist.

`inspect.getsourcefile(object)`

Gibt den Namen der Quellcode-Datei zurück, in der das Objekt *object* definiert wurde. Diese Funktion wirft eine `TypeError-Exception`, wenn es sich bei *object* um ein eingebautes Objekt, beispielsweise eine Built-in Function, handelt. Das liegt daran, dass diese Objekte entweder intern in C implementiert sind oder die Quelldatei nur als Kompilat vorliegt. Dem Objekt kann also kein tatsächlicher Quellcode zugeordnet werden. Diese Einschränkung betrifft auch viele Funktionen der Standardbibliothek.

```
>>> inspect.getsourcefile(inspect.getsourcefile)
'C:\\Python30\\lib\\inspect.py'
```

In diesem Fall wurde `getsourcefile` dazu verwendet, die Quelldatedatei herauszufinden, in der die Funktion selbst definiert ist.

inspect.getsourcelines(object)

Gibt ein Tupel mit zwei Elementen zurück. Das erste ist eine Liste von Strings, die alle dem Objekt *object* zugeordneten Quellcodezeilen enthält. Das zweite Element des zurückgegebenen Tupels ist die Zeilennummer der ersten dem Objekt *object* zugeordneten Quellcodezeile.

Für *object* kann ein Modul, eine Methode, eine Funktion, ein Traceback-Objekt oder ein Frame-Objekt übergeben werden. Näheres zum Frame-Objekt erfahren Sie im nächsten Kapitel.

Die Funktion wirft eine `IOError`-Exception, wenn der Quellcode zum Objekt *object* nicht geladen werden konnte.

```
>>> inspect.getsourcelines(inspect.getsourcelines)
(['def getsourcelines:\n', [...], 670)
```

In diesem Fall wurde die Funktion `getsourcelines` dazu verwendet, die Quellcodezeilen ihrer eigenen Definition zurückzugeben. Beachten Sie, dass das Auslassungszeichen keine rekursive Liste andeutet, sondern weiteren Quellcodezeilen entsprechen soll.

inspect.getsource(object)

Gibt die dem Objekt *object* zugeordneten Quellcodezeilen in einem einzigen String zurück. Die Funktion unterscheidet sich demzufolge nur im Rückgabewert von `getsourcelines`.

Für *object* kann ein Modul, eine Methode, eine Funktion, ein Traceback-Objekt oder ein Frame-Objekt übergeben werden.

Die Funktion wirft eine `IOError`-Exception, wenn der Quellcode zum Objekt *object* nicht geladen werden konnte.

21.2.3 Klassen und Funktionen

Dieser Abschnitt behandelt die wichtigsten Funktionen des Moduls `inspect`, die sich auf Klassen und Funktionen beziehen.

inspect.getclasstree(classes[, unique])

Gibt die Vererbungshierarchie der übergebenen Klassen in Form eines Baums² zurück. Im ersten Teil des Beispielsprogramms bauen wir daher zunächst eine Klassenhierarchie auf:

```
>>> class a:
...     pass
... 
```

```
>>> class b(a):
...     pass
...
>>> class c(a):
...     pass
...
>>> class d(b):
...     pass
...
>>> class e(b):
...     pass
...
```

Abbildung 21.1 veranschaulicht die Hierarchie.

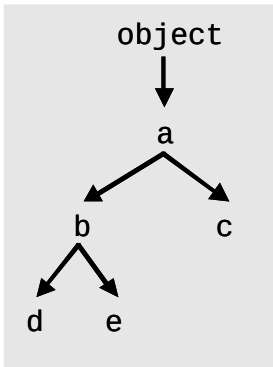


Abbildung 21.2 Die Klassenhierarchie

Die Funktion `getclasstree` bereitet Teile dieser Hierarchie zu einem Baum auf. Dazu müssen Sie beim Funktionsaufruf die Klassen übergeben, die im resultierenden Baum enthalten sein sollen. Im folgenden Beispiel wird die Hierarchie um die Klasse `b` erzeugt:

```
>>> inspect.getclasstree([b])
[
  (<class '__main__.a'>, (<class 'object'>,)),
  [
    (<class '__main__.b'>, (<class '__main__.a'>,))
  ]
]
```

- 2 Bei einem Baum handelt es sich grundsätzlich um nichts anderes als um eine verschachtelte Liste. Gerade im Zusammenhang mit `getclasstree` ist die zurückgegebene Liste ausdrücklich als Baum zu sehen.

Der resultierende Baum, hier aus Gründen der Übersichtlichkeit formatiert, besteht aus einer Liste von Tupeln, die jeweils eine Klasse repräsentieren. Ein solches Tupel ist folgendermaßen aufgebaut:

```
(Klasse, (Basisklassen))
```

Nach jedem solchen Tupel kann eine eingebettete Liste folgen, die alle Klassen enthält, die von der im vorangegangenen Tupel beschriebenen Klasse abgeleitet sind. Auch diese eingebetteten Listen bestehen aus Tupeln des obigen Formats.

Wie Sie sehen, wurde im Beispiel ein Baum erzeugt, der nur die Klasse `b` selbst und ihre Basisklasse `a` enthält.

Für den ersten Parameter *classes* von `getclasstree` muss eine Liste übergeben werden, die auch, anders als im vorangegangenen Beispiel, mehrere Klassen enthalten darf, die dann alle im resultierenden Baum vorkommen. So soll im folgenden Beispiel die Hierarchie um die Klassen `b` und `a` erstellt werden:

```
>>> inspect.getclasstree([b, a])
[
  (<class 'object'>, ()),
  [
    (<class '__main__.a'>, (<class 'object'>,)),
    [
      (<class '__main__.b'>, (<class '__main__.a'>,))
    ]
  ]
]
```

Der resultierende Baum enthält die aufgelisteten Klassen `a` und `b` sowie die Basisklasse `object` von `a`.

Wenn für den optionalen Parameter *unique* `True` übergeben wird, taucht jede Klasse auch im Falle von Mehrfachvererbung nur ein einziges Mal im Klassenbaum auf.

inspect.getmro(cls)

Gibt ein Tupel zurück, das alle Basisklassen der Klasse *cls* inklusive der Klasse *cls* selbst enthält. Die Klassen sind dabei so angeordnet, dass die allgemeinste Basisklasse als letzte im Tupel enthalten ist. Das bedeutet umgekehrt, dass die Klasse *cls* selbst als erstes Element des resultierenden Tupels geführt wird.

```
>>> inspect.getmro(b)
(
<class '__main__.b'>,
<class '__main__.a'>,
<class 'object'>
)
```

Dieses Beispiel bezieht sich auf die bei `getclasstree` angelegte Klassenhierarchie.

inspect.getargspec(func)

Gibt die Funktionsschnittstelle der Funktion *func*, also die Namen und vordefinierten Werte der Funktionsparameter, zurück. Das Ergebnis ist eine Instanz der Klasse `inspect.ArgSpec`, die im Wesentlichen über die in der folgenden Tabelle aufgelisteten Attribute verfügt. Beachten Sie, dass diese Attribute wie bei einem Tupel auch über einen Index angesprochen werden können.

Index	Attribut	Beschreibung
0	<code>args</code>	eine Liste, die die Namen aller Positionsparameter enthält
1	<code>varargs</code>	der Name, über den zusätzliche Positionsparameter funktionsintern angesprochen werden
2	<code>keywords</code>	der Name des Parameters, über den zusätzliche Schlüsselwortparameter funktionsintern angesprochen werden
3	<code>defaults</code>	ein Tupel, das die Defaultwerte der Positionsparameter enthält, sofern diese vorhanden sind

Tabelle 21.2 Attribute einer `inspect.ArgSpec`-Instanz

Zur näheren Erläuterung dient folgendes Beispiel:

```
>>> def funktion(a, b, c=1, d=2, *args):
...     pass
...
>>> inspect.getargspec(funktion)
ArgSpec(args=['a', 'b', 'c', 'd'], varargs='args',
        keywords=None, defaults=(1, 2))
```

In diesem Fall besteht die Funktionsschnittstelle also aus den Parametern `a`, `b`, `c` und `d` sowie aus einer Möglichkeit, beliebige weitere Positionsparameter zu übergeben. Auf diese zusätzlichen Parameter wird funktionsintern über den Namen `args` zugegriffen. Eine Möglichkeit, beliebige Schlüsselwortparameter zu übergeben, gibt es nicht; das entsprechende Element des Tupels ist `None`. Schlussendlich

können dem Tupel noch die vordefinierten Werte für die letzten beiden Parameter `c` und `d` entnommen werden.

inspect.getargvalues(frame)

Gibt Informationen über die einer Funktion tatsächlich übergebenen Werte in Form einer `inspect.ArgInfo`-Instanz zurück. Diese verfügt über die in folgender Tabelle aufgeführten Attribute. Auch hier ist es möglich, die Attribute über einen Index anzusprechen:

Index	Attribut	Beschreibung
0	args	eine Liste der für die Positionsparameter übergebenen Werte
1	varargs	der Name, über den zusätzliche Positionsparameter funktionsintern angesprochen werden
2	keywords	der Name des Parameters, über den zusätzliche Schlüsselwortparameter funktionsintern angesprochen werden
3	locals	ein Dictionary, das die Namen und Werte aller lokalen Variablen des übergebenen Frame-Objekts <i>frame</i> enthält

Tabelle 21.3 Attribute einer `inspect.ArgInfo`-Instanz

Das sogenannte *Frame-Objekt* repräsentiert einen Codeblock und kann mithilfe der Funktion `inspect.currentframe` erzeugt werden:

```
>>> def funktion(a, b, c=1, d=2):
...     print(inspect.getargvalues(inspect.currentframe()))
...
>>> funktion("Hallo", "Welt")
ArgInfo(args=['a', 'b', 'c', 'd'], varargs=None, keywords=None,
        locals={'a': 'Hallo', 'c': 1, 'b': 'Welt', 'd': 2})
```

Beachten Sie, dass die beiden `None`-Elemente des Tupels nur Bedeutung haben, wenn die Funktionsschnittstelle das Übergeben beliebiger Positions- oder Schlüsselwortparameter ermöglicht, was hier nicht der Fall ist.

21.3 Formatierte Ausgabe von Instanzen – pprint

Das Modul `pprint` (für *pretty print*) gibt eine formatierte Repräsentation eines Python-Datentyps auf dem Bildschirm aus. Damit macht das Modul insbesondere die Ausgabe großer Listen oder Dictionaries besser lesbar und bietet sich somit förmlich an, in einer interaktiven Debug-Sitzung zur Ausgabe verschiedener Werte verwendet zu werden.

Im Modul `pprint` sind die folgenden Funktionen enthalten. Um die Beispiele ausführen zu können, muss zuvor das Modul `pprint` eingebunden werden:

```
>>> import pprint
```

`pprint.pprint(object[, stream[, indent[, width[, depth]]])`

Gibt die Instanz *object* formatiert auf dem Stream *stream* aus. Wenn Sie den Parameter *stream* nicht übergeben, wird nach `sys.stdout` geschrieben. Über die Parameter *indent* und *depth* lässt sich die Formatierung der Ausgabe steuern. Dabei kann für *indent* die Anzahl der Leerzeichen übergeben werden, die für eine Einrückung verwendet werden soll. Der Parameter *indent* ist mit 1 vorbelegt.

Über den optionalen Parameter *width* kann die maximale Anzahl an Zeichen angegeben werden, die die Ausgabe breit sein darf. Dieser Parameter ist mit 80 Zeichen vorbelegt.

Der Parameter *depth* ist ebenfalls eine ganze Zahl und bestimmt, bis zu welcher Tiefe Unterinstanzen, beispielsweise also verschachtelte Listen, ausgegeben werden sollen. Im folgenden Beispiel wird die Funktion `pprint` dazu verwendet, die Liste `sys.path` formatiert und damit lesbar auszugeben:

```
>>> import sys
>>> pprint.pprint(sys.path, indent=4)
[
    'C:\\Python30\\python30.zip',
    'C:\\Python30\\DLLs',
    'C:\\Python30\\lib',
    'C:\\Python30\\lib\\plat-win',
    'C:\\Python30',
    'C:\\Python30\\lib\\site-packages']
```

Zum Vergleich geben wir `sys.path` noch einmal unformatiert mit `print` aus:

```
>>> print(sys.path)
['', 'C:\\Python30\\python30.zip', 'C:\\Python30\\DLLs',
'C:\\Python30\\lib', 'C:\\Python30\\lib\\plat-win',
'C:\\Python30', 'C:\\Python30\\lib\\site-packages']
```

Die Ausgabe eines Dictionarys sieht folgendermaßen aus:

```
>>> d = {}
>>> d["path"] = sys.path
>>> d["hallo"] = "welt"
>>> d["aaa"] = "Sortierte Ausgabe"
>>> pprint.pprint(d)
```



```
{'aaa': 'Sortierte Ausgabe',
  'hallo': 'welt',
  'path': ['',
            'C:\\Python30\\python30.zip',
            'C:\\Python30\\DLLs',
            'C:\\Python30\\lib',
            'C:\\Python30\\lib\\plat-win',
            'C:\\Python30',
            'C:\\Python30\\lib\\site-packages']}
```

Es wurde ein Dictionary angelegt, das unter anderem die Liste `sys.path` als Wert enthält. Damit ist dies nicht nur ein Beispiel für die formatierte Ausgabe eines Dictionarys, sondern zugleich dafür, wie verschachtelte Instanzen formatiert werden. Beachten Sie, dass die Schlüssel-Wert-Paare des Dictionarys bei der Ausgabe nach den Schlüsseln sortiert werden.

pprint.pformat(object[, indent[, width[, depth]]])

Die Funktion `pformat` hat eine ähnliche Bedeutung wie `pprint`, aber mit dem Unterschied, dass `pformat` die formatierte Ausgabe in Form eines Strings zurückgibt, anstatt sie in einen Stream zu schreiben. Die Schnittstelle von `pformat` ist bis auf den hier überflüssigen *stream*-Parameter mit der von `pprint` identisch.

pprint.isreadable(object)

Gibt `True` zurück, wenn die formatierte Ausgabe der Instanz *object* lesbar ist. Lesbar bedeutet in diesem Zusammenhang, dass die Ausgabe als Python-Code gelesen werden kann und mithilfe der Built-in Function `eval` aus der Ausgabe wieder die zugrundeliegende Python-Instanz rekonstruiert werden könnte.

Wenn *object* eine Instanz eines Basisdatentyps ist, gibt die Funktion in den meisten Fällen `True` zurück:

```
>>> pprint.isreadable(sys.path)
True
>>> a = []
>>> a.append(a)
>>> pprint.isreadable(a)
False
```

Im zweiten Teil des Beispiels wurde `False` zurückgegeben, weil es sich um eine rekursive Liste handelt, die sich selbst als Element enthält. Eine solche Liste kann aus naheliegenden Gründen nicht vollständig ausgegeben werden. Auch Instanzen selbsterstellter Klassen können nicht lesbar repräsentiert werden. Letzteres liegt daran, dass die Ausgabe dann den Code zum Erstellen der Klasse sowie der Instanz enthalten müsste, was keinen Sinn ergibt.

pprint.isrecursive(object)

Gibt `True` zurück, wenn die Instanz *object* rekursiv ist, also sich selbst als Element enthält.

```
>>> a = []
>>> a.append(a)
>>> pprint.isrecursive(a)
True
```

Damit sind die grundlegendsten Möglichkeiten, die das Modul `pprint` bietet, erläutert, und wir werden uns im Folgenden mit dem Modul `logging` beschäftigen.

21.4 Logdateien – logging

Das Modul `logging` stellt ein flexibles Interface zum Protokollieren eines Programmlaufs bereit. Protokolliert wird der Programmablauf, indem an unterschiedlichen Stellen im Programm Meldungen an das `logging`-Modul abgesetzt werden. Diese Meldungen können unterschiedliche Dringlichkeitsstufen haben. So gibt es beispielsweise Fehlermeldungen, Warnungen oder Informationen. Das Modul `logging` kann diese Meldungen auf vielfältige Weise verarbeiten. Üblich ist es, die Meldung mit einem Timestamp zu versehen und entweder auf dem Bildschirm auszugeben oder in eine Datei zu schreiben.

In diesem Abschnitt wird die Verwendung des Moduls `logging` anhand mehrerer Beispiele im interaktiven Modus gezeigt. Um die Beispielprogramme korrekt ausführen zu können, muss zuvor das Modul `logging` eingebunden sein:

```
>>> import logging
```

Bevor Meldungen an den sogenannten *Logger* geschickt werden können, muss dieser durch Aufruf der Funktion `basicConfig` initialisiert werden. Die Funktion `basicConfig` bekommt verschiedene Schlüsselwortparameter übergeben. Im folgenden Beispiel wird ein `Logger` eingerichtet, der alle eingehenden Meldungen in die Logdatei `programm.log` schreibt:

```
>>> logging.basicConfig(
...     filename = "programm.log",
...     filemode = "a")
```

Über den Schlüsselwortparameter `filemode` kann der Modus angegeben werden, in dem die Logdatei geöffnet werden soll. Eine Liste aller verfügbaren Modi und ihrer Bedeutung finden Sie in Abschnitt 9.4, »Verwendung des Dateiobjekts«. Der Parameter ist mit `"a"` für »append« vorbelegt und könnte hier auch weggelassen werden.

Jetzt können mithilfe der im Modul enthaltenen Funktion `log` Meldungen an den Logger übergeben werden. Die Funktion `log` bekommt dabei die Dringlichkeitsstufe der Meldung als ersten und die Meldung selbst in Form eines Strings als zweiten Parameter übergeben:

```
>>> logging.log(logging.ERROR, "Ein Fehler ist aufgetreten")
>>> logging.log(logging.INFO, "Dies ist eine Information")
```

Durch das Aufrufen der Funktion `shutdown` wird der Logger korrekt deinitialisiert, und eventuell noch anstehende Schreiboperationen werden durchgeführt:

```
>>> logging.shutdown()
```

Natürlich sind nicht nur die Dringlichkeitsstufen `ERROR` und `INFO` verfügbar. Die folgende Tabelle listet alle vordefinierten Stufen auf, aus denen Sie wählen können. Die Tabelle ist dabei nach Dringlichkeit geordnet, wobei die dringendste Stufe zuletzt aufgeführt wird.

Level	Beschreibung
<code>logging.NOTSET</code>	keine Dringlichkeitsstufe
<code>logging.DEBUG</code>	eine Meldung, die nur für den Programmierer zur Fehlersuche interessant ist
<code>logging.INFO</code>	eine Informationsmeldung über den Programmstatus
<code>logging.WARNING</code>	eine Warnmeldung, die auf einen möglichen Fehler hinweist
<code>logging.ERROR</code>	eine Fehlermeldung, nach der das Programm weiterarbeiten kann
<code>logging.CRITICAL</code>	eine Meldung über einen kritischen Fehler, der das sofortige Beenden des Programms oder der aktuell durchgeführten Operation zur Folge hat

Tabelle 21.4 Vordefinierte Dringlichkeitsstufen

Beachten Sie, dass aus Gründen des Komforts zu jeder Dringlichkeitsstufe eine eigene Funktion existiert. So sind die beiden Funktionsaufrufe von `logging.log` aus dem letzten Beispiel äquivalent zu:

```
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
```

Wenn Sie sich die Logdatei nach dem Aufruf dieser beiden Funktionen ansehen, werden Sie feststellen, dass es lediglich einen einzigen Eintrag gibt:

```
ERROR:root:Ein Fehler ist aufgetreten
```

Das liegt daran, dass der Logger in seiner Basiskonfiguration nur Meldungen loggt, deren Dringlichkeit größer oder gleich der einer Warnung ist. Um auch De-

bug- und Info-Meldungen mitzuloggen, müssen Sie beim Aufruf der Funktion `basicConfig` im Schlüsselwortparameter `level` einen geeigneten Wert übergeben:

```
logging.basicConfig(
    filename="programm.log",
    level = logging.DEBUG)
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
```

In diesem Beispiel wurde die Mindestdringlichkeit auf `DEBUG` gesetzt. Das bedeutet, dass alle Meldungen, die mindestens eine Dringlichkeit von `DEBUG` haben, geloggt werden. Folglich erscheinen auch beide Meldungen in der Logdatei:

```
ERROR:root:Ein Fehler ist aufgetreten
INFO:root:Dies ist eine Information
```

21.4.1 Das Meldungsformat anpassen

Zu Beginn von Abschnitt 21.4 wurde erwähnt, dass man Einträge in einer Logdatei gerne mit der Uhrzeit versieht, zu der die Meldung eingegangen ist. Eine solche Uhrzeit wird aber, wie in den vorangegangenen Beispielen zu sehen war, standardmäßig nicht angezeigt. Es gibt allerdings eine Möglichkeit, das Format der geloggtten Meldung anzupassen. Dazu übergeben Sie beim Funktionsaufruf von `basicConfig` der Schlüsselwortparameter `format`:

```
logging.basicConfig(
    filename="programm.log",
    level = logging.DEBUG,
    format = "%(asctime)s [%(levelname)-8s] %(message)s")
logging.error("Ein Fehler ist aufgetreten")
logging.info("Dies ist eine Information")
logging.error("Und schon wieder ein Fehler")
```

Sie sehen, dass ein Format-String übergeben wurde, der die Vorlage für eine Meldung enthält, wie sie später in der Logdatei stehen soll. Beachten Sie, dass es sich dabei um die bekannte String-Formatierung handelt, bei der die Bezeichner `asctime` für den Timestamp, `levelname` für die Dringlichkeitsstufe und `message` für die Meldung verwendet wurden. Die von diesem Beispiel generierten Meldungen sehen folgendermaßen aus:

```
2007-09-27 03:28:55,811 [ERROR   ] Ein Fehler ist aufgetreten
2007-09-27 03:29:00,690 [INFO    ] Dies ist eine Information
2007-09-27 03:29:12,686 [ERROR   ] Und schon wieder ein Fehler
```

Die folgende Tabelle listet die wichtigsten Bezeichner auf, die innerhalb des `format`-Formatstrings verwendet werden dürfen. Beachten Sie, dass einige der Bezeichner je nach Kontext, in dem die Meldung erzeugt wird, keine Bedeutung haben.

Level	Beschreibung
<code>%(asctime)s</code>	Zeitpunkt der Meldung. Beachten Sie, dass das Datums- und Zeitformat beim Funktionsaufruf von <code>basicConfig</code> über den Parameter <code>datefmt</code> angegeben werden kann. Näheres dazu folgt im Anschluss an diese Tabelle.
<code>%(filename)s</code>	der Dateiname der Programmdatei, in der die Meldung abgesetzt wurde
<code>%(funcName)s</code>	der Name der Funktion, in der die Meldung abgesetzt wurde
<code>%(levelname)s</code>	die Dringlichkeitsstufe der Meldung
<code>%(lineno)s</code>	die Quellcodezeile, in der die Meldung abgesetzt wurde
<code>%(message)s</code>	der Text der Meldung
<code>%(module)s</code>	Der Name des Moduls, in dem die Meldung abgesetzt wurde. Der Modulname entspricht dem Dateinamen ohne Dateiendung.
<code>%(pathname)s</code>	der Pfad zur Programmdatei, in der die Meldung abgesetzt wurde
<code>%(process)s</code>	die ID des Prozesses, in dem die Meldung abgesetzt wurde
<code>%(thread)s</code>	die ID des Threads, in dem die Meldung abgesetzt wurde

Tabelle 21.5 Bezeichner im Formatstring

Was uns an diesen Meldungen vielleicht noch stört, ist das Format des Timestamps. Zum einen wird das amerikanische Datumsformat verwendet, und zum anderen ist eine Auflösung bis auf die Millisekunde genau für unsere Zwecke ein bisschen zu fein. Das Format des Timestamps kann beim Aufruf von `basicConfig` über den Schlüsselwortparameter `datefmt` angegeben werden:

```
logging.basicConfig(
    filename="programm.log",
    level = logging.DEBUG,
    format = "%(asctime)s %(levelname)s: %(message)s",
    datefmt = "%d.%m.%Y %H:%M:%S")
logging.error("Ein Fehler ist aufgetreten")
```

Die in der Vorlage für das Datumsformat verwendeten Platzhalter wurden in Abschnitt 16.1, »Elementare Zeitfunktionen – time«, eingeführt. Die von diesem Beispiel erzeugte Meldung sieht folgendermaßen aus:

```
27.09.2007 03:38:49 ERROR: Ein Fehler ist aufgetreten
```

21.4.2 Logging Handler

Bisher wurde ausschließlich besprochen, wie das Modul `logging` dazu verwendet werden kann, alle eingehenden Meldungen in eine Datei zu schreiben. Tatsächlich ist das Modul in dieser Beziehung sehr flexibel und erlaubt es, nicht nur in Dateien, sondern beispielsweise auch in Streams zu schreiben oder die Meldungen über eine Netzwerkverbindung zu schicken. Dafür werden sogenannte *Logging Handler* verwendet. Um genau zu sein, haben wir in den vorherigen Abschnitten bereits einen impliziten Handler verwendet, ohne uns darüber im Klaren zu sein.

Um einen speziellen Handler einzurichten, muss eine Instanz der Handler-Klasse erzeugt werden. Diese kann dann vom Logger verwendet werden. Im folgenden Beispiel sollen alle Meldungen auf einen Stream, nämlich `sys.stdout`, geschrieben werden; dazu wird die Handler-Klasse `logging.StreamHandler` verwendet:

```
import logging
import sys

handler = logging.StreamHandler(sys.stdout)
frm = logging.Formatter("%(asctime)s %(levelname)s: %(message)s",
                        "%d.%m.%Y %H:%M:%S")
handler.setFormatter(frm)

logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.critical("Ein wirklich kritischer Fehler")
logger.warning("Und eine Warnung hinterher")
logger.info("Dies hingegen ist nur eine Info")
```

Zunächst wird der Handler, in diesem Fall ein `StreamHandler`, instantiiert. Im nächsten Schritt wird eine Instanz der Klasse `Formatter` erzeugt. Diese Klasse kapselt die Formatierungsanweisungen, die wir in den vorherigen Beispielen beim Aufruf der Funktion `basicConfig` übergeben haben. Mithilfe der Methode `setFormatter` werden dem Handler die Formatierungsanweisungen bekanntgegeben.

Um den Handler beim Logger zu registrieren, benötigen wir Zugriff auf die bisher implizit verwendete Logger-Instanz. Diesen Zugriff erlangen wir über die Funktion `logging.getLogger`. Danach wird über `addHandler` der Handler hinzugefügt und über `setLevel` die gewünschte Dringlichkeitsstufe eingestellt.

Beachten Sie, dass die Meldungen im Folgenden nicht über Funktionen des Moduls `logging`, sondern über die Methoden `critical`, `warning` und `info` der

Logger-Instanz `logger` abgesetzt werden. Das Beispielprogramm gibt folgenden Text auf dem Bildschirm aus:

```
27.09.2007 17:21:46 CRITICAL: Ein wirklich kritischer Fehler
27.09.2007 17:21:46 WARNING: Und eine Warnung hinterher
27.09.2007 17:21:46 INFO: Dies hingegen ist nur eine Info
```

Im Folgenden sollen die wichtigsten zusätzlichen Handler-Klassen beschrieben werden, die im Modul `logging` enthalten sind.

`logging.FileHandler(filename[, mode[, encoding[, delay]])`

Dieser Handler schreibt die Logeinträge in die Datei *filename*. Dabei wird die Datei im Modus *mode* geöffnet. Beachten Sie, dass der Handler `FileHandler` auch implizit durch Angabe der Schlüsselwortparameter `filename` und `filemode` beim Aufruf der Funktion `basicConfig` verwendet werden kann.

Der Parameter *encoding* kann dazu verwendet werden, das zum Schreiben der Datei verwendete Encoding festzulegen. Wenn Sie für den letzten Parameter `True` übergeben, wird mit dem Öffnen der Datei so lange gewartet, bis tatsächlich Daten geschrieben werden sollen.

`logging.StreamHandler([stream])`

Dieser Handler schreibt die Logeinträge in den Stream *stream*. Beachten Sie, dass der Handler `StreamHandler` auch implizit durch Angabe des Schlüsselwortparameters `stream` beim Aufruf der Funktion `basicConfig` verwendet werden kann.

`logging.handlers.SocketHandler(host, port)`

Dieser Handler sendet die Logeinträge über eine TCP-Netzwerkschnittstelle an den Rechner mit dem Hostnamen *host* unter Verwendung des Ports *port*. Beachten Sie, dass die Meldungen komprimiert werden und der Rechner, der die Logeinträge entgegennimmt, diese speziell entpacken muss. Aus Platzgründen werden wir darauf hier nicht eingehen, Sie finden aber ein ausführliches Beispielprogramm in der Dokumentation des Moduls `logging` unter dem Stichwort »Sending and receiving events across a network«.

`logging.handlers.DatagramHandler(host, port)`

Dieser Handler sendet die Logeinträge über eine UDP-Netzwerkschnittstelle und wird ähnlich verwendet wie der Handler `SocketHandler`.

`logging.handlers.SMTPHandler(mailhost, from, to, subject[, credentials])`

Dieser Handler sendet die Logeinträge als E-Mail an die Adresse *to*. Dabei wird *subject* als Betreff und *from* als Absenderadresse eingetragen. Über den Parameter

mailhost geben Sie den zu verwendenden SMTP-Server an. Sollte dieser Server eine Authentifizierung verlangen, können Sie ein Tupel, das Benutzernamen und Passwort enthält, für den optionalen letzten Parameter *credentials* übergeben.

21.5 Automatisiertes Testen

Pythons Standardbibliothek stellt zwei Module zur testgetriebenen Entwicklung (engl. *test-driven development*) bereit. Unter *testgetriebener Entwicklung* versteht man eine Art der Programmierung, bei der viele kleine Abschnitte des Programms, sogenannte *Units*, durch automatisierte Testdurchläufe auf Fehler geprüft werden. Bei der testgetriebenen Entwicklung wird das Programm nach kleineren, in sich geschlossenen Arbeitsschritten so lange verbessert, bis es wieder alle bisherigen und alle hinzugekommenen Tests besteht. Auf diese Weise können sich durch das Hinzufügen von neuem Code keine Fehler in alten, bereits getesteten Code einschleichen.

In Python ist das möglicherweise bekannte Konzept der Unit Tests im Modul `unittest` implementiert. Das Modul `doctest` ermöglicht es, Testfälle innerhalb eines Docstrings, beispielsweise einer Funktion, unterzubringen. Im Folgenden werden wir uns zunächst mit dem Modul `doctest` beschäftigen, um danach zum Modul `unittest` voranzuschreiten.

21.5.1 Testfälle in Docstrings – doctest

Das Modul `doctest` erlaubt es, Testfälle innerhalb des Docstrings einer Funktion, Methode, Klasse oder eines Moduls zu erstellen, die beim Aufruf der im Modul `doctest` enthaltenen Funktion `testmod` getestet werden. Die Testfälle innerhalb eines Docstrings werden dabei nicht in einer komplizierten neuen Definitionssprache verfasst, sondern können direkt aus einer Sitzung im interaktiven Modus in den Docstring kopiert werden.

Beachten Sie bei der Verwendung von Doctests, dass Docstrings auch bzw. hauptsächlich für die Dokumentation beispielsweise einer Funktion gedacht sind. Aus diesem Grund sollten Sie die Testfälle im Docstring möglichst einfach und lehrreich halten, so dass der resultierende Docstring auch in Dokumentationen Ihres Programms verwendet werden kann.

Das folgende Beispiel erläutert die Verwendung des Moduls `doctest` anhand der Funktion `fak`, die die Fakultät einer ganzen Zahl berechnen und zurückgeben soll.

```
import doctest
```



```
def fak(n):
    """
        Berechnet die Fakultät einer ganzen Zahl.

        >>> fak(5)
        120
        >>> fak(10)
        3628800
        >>> fak(20)
        2432902008176640000

        Es muss eine positive ganze Zahl uebergeben werden.

        >>> fak(-1)
        Traceback (most recent call last):
            ...
        ValueError: Keine negativen Zahlen!
    """
    res = 1
    for i in range(2, n+1):
        res *= i
    return res

if __name__ == "__main__":
    doctest.testmod()
```

Im Docstring der Funktion `fak` steht zunächst ein erklärender Text. Dann folgt, durch eine leere Zeile davon abgetrennt, ein Auszug aus Pythons interaktivem Modus, in dem Funktionsaufrufe von `fak` mit ihren Rückgabewerten stehen. Diese Testfälle werden beim Ausführen des Tests nachvollzogen und entweder für wahr oder für falsch befunden.

Auf diese einfachen Fälle folgen, jeweils durch eine Leerzeile eingeleitet, ein weiterer erklärender Text sowie ein Ausnahmefall, in dem eine negative Zahl übergeben wurde. Beachten Sie, dass Sie den Stacktrace eines auftretenden Tracebacks im Docstring weglassen können. Auch die im Beispiel stattdessen geschriebenen Auslassungszeichen sind optional.

Der letzte Testfall wurde in der Funktion noch nicht berücksichtigt, so dass dieser im Test fehlschlagen wird. Um den Test zu starten, muss die Funktion `testmod` des Moduls `doctest` aufgerufen werden. Aufgrund der `if`-Abfrage

```
if __name__ == "__main__":
    doctest.testmod()
```

wird diese Funktion immer dann aufgerufen, wenn die Programmdatei direkt ausgeführt wird. Der Test wird hingegen nicht durchgeführt, wenn die Programmdatei von einem anderen Python-Programm als Modul eingebunden wird. Im provozierten Fehlerfall lautet das Testresultat folgendermaßen:

```
*****
File "fak.py", line 17, in __main__.fak
Failed example:
    fak(-1)
Expected:
    Traceback (most recent call last):
    ...
    ValueError: Keine negativen Zahlen!
Got:
    1
*****
1 items had failures:
  1 of  4 in __main__.fak
***Test Failed*** 1 failures.
```

Jetzt erweitern wir die Funktion `fak` dahingehend, dass sie im Falle eines negativen Parameters die gewünschte Exception wirft:

```
def fak(n):
    """
    [...]
    """
    if n < 0:
        raise ValueError("Keine negativen Zahlen!")

    res = 1
    for i in range(2, n+1):
        res *= i
    return res
```

Durch diese Änderung werden bei erneutem Durchführen des Tests keine Fehler mehr angezeigt. Um genau zu sein: Es wird überhaupt nichts angezeigt. Das liegt daran, dass generell nur fehlgeschlagene Testfälle auf dem Bildschirm ausgegeben werden. Sollten Sie auch auf die Ausgabe geglückter Testfälle bestehen, starten Sie die Programmdatei mit der Option `-v` (für *verbose*).

Bei der Verwendung von Doctests sollten Sie unbedingt beachten, dass die in den Docstrings geschriebenen Vorgaben Zeichen für Zeichen mit den Ausgaben der intern ausgeführten Testfälle verglichen werden. Dabei sollten Sie stets im Hinterkopf behalten, dass die Ausgaben bestimmter Datentypen nicht immer gleich sind. So stehen beispielsweise die Schlüssel-Wert-Paare eines Dictionarys in kei-

ner garantierten Reihenfolge, so dass Sie innerhalb eines Doctests nie ein Dictionary als Ergebnis ausgeben sollten. Des Weiteren gibt es Informationen, die vom Interpreter oder anderen Gegebenheiten abhängen; beispielsweise entspricht die Identität einer Instanz intern ihrer Speicheradresse und wird sich deswegen natürlich beim Neustart des Programms ändern.

Eine weitere Besonderheit, auf die Sie achten müssen, ist, dass eine Leerzeile in der erwarteten Ausgabe einer Funktion durch den String `<BLANKLINE>` gekennzeichnet werden muss, da eine Leerzeile als Trennung zwischen Testfällen und Dokumentation fungiert.

Flags

Um einen Testfall genau an Ihre Bedürfnisse anzupassen, können Sie sogenannte *Flags* vergeben. Das sind Einstellungen, die Sie aktivieren oder deaktivieren können. Ein Flag wird in Form eines Kommentars hinter den Testfall im Docstring geschrieben. Wird das Flag von einem Plus (+) eingeleitet, wird es aktiviert, bei einem Minus (-) deaktiviert. Bevor wir zu einem konkreten Beispiel kommen, sollen die drei wichtigsten Flags eingeführt werden.

Flag	Bedeutung
ELLIPSIS	Wenn dieses Flag gesetzt ist, kann die Angabe ... für eine beliebige Ausgabe einer Funktion verwendet werden. So können veränderliche Angaben wie Speicheradressen oder Ähnliches in größeren Ausgaben überlesen werden.
NORMALIZE_WHITESPACES	Wenn dieses Flag gesetzt ist, werden Whitespace-Zeichen nicht in den Ergebnisvergleich einbezogen. Das ist besonders dann interessant, wenn Sie ein langes Ergebnis auf mehrere Zeilen umbrechen möchten.
SKIP	Dieses Flag veranlasst das Überspringen des Tests. Das ist beispielsweise dann nützlich, wenn Sie im Docstring zu Dokumentationszwecken eine Reihe von Beispielen liefern, aber nur wenige davon bei einem Testlauf berücksichtigt werden sollen.

Tabelle 21.6 Doctest-Flags

In einem einfachen Beispiel möchten wir den Doctest der bereits bekannten Fakultätsfunktion um die Berechnung der Fakultät einer relativ großen Zahl erweitern. Da es müßig wäre, alle Stellen des Ergebnisses im Doctest anzugeben, soll die Zahl mithilfe des Flags `ELLIPSIS` gekürzt angegeben werden.

```

import doctest

def fak(n):
    """
        Berechnet die Fakultät einer ganzen Zahl.

        >>> fak(1000) # doctest: +ELLIPSIS
        402387260077093773543702...000
        >>> fak("Bla") # doctest: +SKIP
        'BlubbBlubb'
    """
    res = 1
    for i in range(2, n+1):
        res *= i
    return res

if __name__ == "__main__":
    doctest.testmod()

```

Das Setzen der Flags wurde fett hervorgehoben. Wie Sie sehen, umfasst das Beispiel einen zweiten – offensichtlich fehlschlagenden – Test, bei dem aber das SKIP-Flag gesetzt wurde. Aufgrund dessen wird ein Testlauf hier keinen Fehler feststellen.

Bleibt noch zu sagen, dass insbesondere die Funktion `testmod` eine Fülle von Möglichkeiten bietet, die Testergebnisse im Programm zu verwenden oder den Prozess des Testens an Ihre Bedürfnisse anzupassen. Sollten Sie daran interessiert sein, bietet sich die Python-Dokumentation an, in der die Funktion besprochen wird.

21.5.2 Unit Tests – unittest

Das zweite Modul zur testgetriebenen Entwicklung heißt `unittest` und ist ebenfalls in der Standardbibliothek enthalten. Das Modul `unittest` implementiert die Funktionalität des aus Java bekannten Moduls `jUnit`, das den De-facto-Standard zur testgetriebenen Entwicklung in Java darstellt.

Der Unterschied zum Modul `doctest` besteht darin, dass die Testfälle bei `unittest` außerhalb des eigentlichen Programmcodes in einer eigenen Programmdatei in Form von regulärem Python-Code definiert werden. Das vereinfacht die Ausführung der Tests und hält die Programmdokumentation sauber. Umgekehrt ist mit dem Erstellen der Testfälle allerdings mehr Aufwand verbunden.

Um einen neuen Testfall bei `unittest` zu erstellen, müssen Sie eine von der Basisklasse `unittest.TestCase` abgeleitete Klasse erstellen, in der einzelne Testfälle

als Methoden implementiert sind. Die folgende Klasse implementiert die gleichen Testfälle, die wir im vorherigen Abschnitt mit dem Modul `doctest` durchgeführt haben. Dabei muss die zu testende Funktion `fak` in der Programmdatei `fak.py` implementiert sein, die von unserer Test-Programmdatei als Modul eingebunden wird.

```
import unittest
import fak

class MeinTest(unittest.TestCase):

    def testberechnung(self):
        self.failUnlessEqual(fak.fak(5), 120)
        self.failUnlessEqual(fak.fak(10), 3628800)
        self.failUnlessEqual(fak.fak(20), 2432902008176640000)

    def testausnahmen(self):
        self.failUnlessRaises(ValueError, fak.fak, -1)

if __name__ == "__main__":
    unittest.main()
```

Es wurde eine Klasse namens `MeinTest` erzeugt, die von der Basisklasse `unittest.TestCase` erbt. In der Klasse `MeinTest` wurden zwei Testmethoden namens `testberechnung` und `testausnahmen` implementiert. Beachten Sie, dass der Name solcher Testmethoden mit `test` beginnen muss, damit sie später auch tatsächlich zum Testen gefunden und ausgeführt werden.

Innerhalb der Testmethoden werden die Methoden `failUnlessEqual` bzw. `failUnlessRaises` verwendet, die den Test fehlschlagen lassen, wenn die beiden angegebenen Werte nicht gleich sind bzw. wenn die angegebene Exception nicht geworfen wurde.

Um den Testlauf zu starten, wird die Funktion `unittest.main` aufgerufen. Die Fallunterscheidung

```
if __name__ == "__main__":
    unittest.main()
```

bewirkt, dass der Unit Test nur durchgeführt wird, wenn die Programmdatei direkt ausgeführt wird, und ausdrücklich nicht, wenn die Programmdatei als Modul in ein anderes Python-Programm importiert wurde. Die aufgerufene Funktion `unittest.main` erzeugt, um den Test durchzuführen, Instanzen aller Klassen, die im aktuellen Namensraum existieren und von `unittest.TestCase` erben. Dann werden alle Methoden dieser Instanzen aufgerufen, deren Namen mit `test` beginnen.

Die Ausgabe des Beispiels lautet im Erfolgsfall:

```
..
```

```
-----
Ran 2 tests in 0.000s
```

```
OK
```

Dabei stehen die beiden Punkte zu Beginn für zwei erfolgreich durchgeführte Tests. Ein fehlgeschlagener Test würde durch ein `F` gekennzeichnet.

Im Fehlerfall wird die genaue Bedingung angegeben, die zum Fehler geführt hat:

```
.F
```

```
=====
FAIL: testberechnung (__main__.MeinTest)
-----
```

```
Traceback (most recent call last):
  File "testen.py", line 7, in testberechnung
    self.failUnlessEqual(fak.fak(5), 12)
AssertionError: 120 != 12
```

```
-----
Ran 2 tests in 0.001s
```

```
FAILED (failures=1)
```

Die Klasse `TestCase`

An dieser Stelle werden die wichtigsten Methoden der Klasse `TestCase` des Moduls `unittest` besprochen. Die hier vorgestellten Methoden können entweder in einer von `TestCase` abgeleiteten Klasse implementiert oder für die Tests selbst aufgerufen werden.

Im Folgenden sei `tc` eine Instanz der Klasse `TestCase`.

`tc.setUp()`

Die Methode `setUp` wird vor jedem Aufruf einer der implementierten Testmethoden aufgerufen und kann somit für den Test benötigten Initialisierungscode enthalten. Eine in der Methode `setUp` geworfene Exception wird als Error in den Testbericht eingetragen, und der Test wird abgebrochen.

`tc.tearDown()`

Die Methode `tearDown` wird nach jedem Aufruf einer der implementierten Testmethoden aufgerufen und kann somit abschließenden Code enthalten. Eine in der Methode `tearDown` geworfene Exception wird als Error in den Testbericht eingetragen.

Die nun folgenden Methoden können innerhalb der Testmethoden verwendet werden, um die Testbedingungen festzulegen. Beachten Sie, dass es jeweils zwei äquivalente Methoden gibt, eine `assert`- und eine `fail`-Methode. Der Unterschied liegt allein in der Namensgebung, in der die Bedeutung der Methode für den Testfall entweder positiv (`assert`-Methoden) oder negativ (`fail`-Methoden) formuliert ist.

`tc.assertTrue(expr[, msg])`,

`tc.failUnless(expr[, msg])`

Der Test schlägt fehl, wenn `expr` `False` ergibt. Der optionale Parameter `msg` kennzeichnet die Beschreibung, die für diesen Fehler angezeigt werden soll. Jede der kommenden Methoden verfügt über diesen Parameter und verwendet ihn mit der gleichen Bedeutung. Wir werden daher im Folgenden nicht mehr auf den Parameter `msg` eingehen.

`tc.assertEqual(first, second[, msg])`,

`tc.failUnlessEqual(first, second[, msg])`

Der Test schlägt fehl, wenn die Parameter `first` und `second` nicht gleich sind. Beachten Sie, dass auf Gleichheit und nicht auf Identität geprüft wird.

`tc.assertNotEqual(first, second[, msg])`,

`tc.failIfEqual(first, second[, msg])`

Der Test schlägt fehl, wenn die Parameter `first` und `second` gleich sind. Beachten Sie, dass auf Gleichheit und nicht auf Identität geprüft wird.

`tc.assertAlmostEqual(first, second[, places[, msg]])`

`tc.failUnlessAlmostEqual(first, second[, places[, msg]])`

Rundet `first` und `second` auf `places` Stellen genau und vergleicht die beiden Werte. Der Test schlägt fehl, wenn die gerundeten Werte nicht gleich sind.

`tc.assertNotAlmostEqual(first, second[, places[, msg]])`,

`tc.failIfAlmostEqual(first, second[, places[, msg]])`

Rundet `first` und `second` auf `places` Stellen genau und vergleicht die beiden Werte. Der Test schlägt fehl, wenn die gerundeten Werte gleich sind.

`tc.assertRaises(exception, callable, ...)`,

`tc.failUnlessRaises(exception, callable, ...)`

Der Test schlägt fehl, wenn die Funktion `callable` nicht die Exception `exception` wirft, falls sie mit den Parametern aufgerufen wird, die anstelle der Auslassungszeichen (...) stehen.

tc.assertFalse(expr[, msg])

tc.failIf(expr[, msg])

Der Test schlägt fehl, wenn *expr* *True* ergibt.

tc.fail([msg])

Der Test schlägt bedingungslos fehl.

21.6 Traceback-Objekte – traceback

Das Modul `traceback` existiert zum Umgang mit dem sogenannten *Traceback-Objekt*. Ein *Traceback-Objekt* hält den Kontext fest, aus dem eine Exception geworfen wurde, und liefert damit die Informationen, die bei einem *Traceback* auf dem Bildschirm angezeigt werden. Zu diesen Informationen gehört vor allem die Funktionshierarchie, der sogenannte *Callstack*.

Ein *Traceback-Objekt* wird beim Werfen einer Exception automatisch erzeugt. Generell können Sie auf das *Traceback-Objekt* einer gerade abgefangenen Exception über die Funktion `sys.exc_info` des Moduls `sys` zugreifen. Im Folgenden werden die im Modul `traceback` enthaltenen Funktionen besprochen. Dabei werden alle Beispiele im folgenden Kontext ausgeführt:

```
>>> import traceback
>>> import sys
>>> def f1():
...     raise TypeError
...
>>> def f2():
...     f1()
...
>>> try:
...     f2()
... except TypeError:
...     tb = sys.exc_info()[2]
...
>>> tb
<traceback object at 0xb7c49414>
```

Es existiert also ein *Traceback-Objekt* `tb`, das den *Callstack* einer *TypeError-Exception* beschreibt, die zuvor aus einer verschachtelten Funktionshierarchie heraus geworfen wurde.

traceback.print_tb(traceback[, limit[, file]])

Gibt den Stacktrace des Traceback-Objekts formatiert auf dem Bildschirm aus. Über den optionalen Parameter *limit* geben Sie an, wie viele Einträge des Stacktraces maximal ausgegeben werden sollen. Für den dritten, optionalen Parameter kann ein geöffnetes Dateiobjekt übergeben werden, in das der Stacktrace geschrieben wird. Standardmäßig wird in den Stream `sys.stderr` geschrieben.

```
>>> traceback.print_tb(tb)
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in f2
  File "<stdin>", line 2, in f1
```

traceback.print_exception(type, value, traceback[, limit[, file[, chain]]])

Diese Funktion gibt einen vollständigen Traceback auf dem Bildschirm aus. Die Ausgabe ist genauso formatiert wie die einer normalen, nicht abgefangenen Exception. Für die beiden zusätzlichen Parameter *type* und *value* müssen Exception-Typ und Exception-Wert übergeben werden. Die restlichen Parameter haben dieselbe Bedeutung wie bei `print_tb`. So könnte beispielsweise der Funktionsaufruf von `print_exception` im interaktiven Modus folgendermaßen aussehen:

```
>>> traceback.print_exception(TypeError,
... TypeError("Hallo Welt"), tb)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in f2
  File "<stdin>", line 2, in f1
TypeError: Hallo Welt
```

Wenn Sie für den letzten, optionalen Parameter *chain* `False` übergeben, werden die Attribute `__cause__` und `__context__` einer Exception nicht mit in die Traceback-Ausgabe aufgenommen. Der Parameter *chain* ist mit `True` vorbelegt.

traceback.print_exc([, limit[, file[, chain]]])

Wie `print_exception`, jedoch immer für die aktuell abgefangene Exception. Die Parameter *limit*, *file* und *chain* haben genau dieselbe Bedeutung wie bei `print_exception`. Beachten Sie, dass diese Funktion nur innerhalb eines `except`-Zweiges aufgerufen werden kann.

```
>>> try:
...     raise TypeError
... except TypeError:
...     traceback.print_exc()
...
Traceback (most recent call last):
```

```
File "<stdin>", line 2, in <module>
TypeError
```

traceback.format_exc([limit[, chain]])

Diese Funktion arbeitet wie `print_exc`, mit dem Unterschied, dass der formatierte Traceback nicht auf dem Bildschirm bzw. in eine Datei ausgegeben, sondern als String zurückgegeben wird.

traceback.print_last([, limit[, file[, chain]])

Diese Funktion arbeitet wie `print_exception`, jedoch immer für die zuletzt abgefangene Exception. Diese Funktion kann auch außerhalb eines `except`-Zweiges aufgerufen werden.

```
>>> traceback.print_last()
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in f2
  File "<stdin>", line 2, in f1
TypeError
```

traceback.extract_tb(traceback[, limit])

Gibt eine Liste von aufbereiteten Stacktrace-Einträgen des Traceback-Objekts *traceback* zurück. Ein aufbereiteter Stacktrace-Eintrag ist ein Tupel der folgenden Form:

```
(Dateiname, Zeilennummer, Funktionsname, Text)
```

Der optionale Parameter *limit* hat die gleiche Bedeutung wie bei `print_tb`.

```
>>> traceback.extract_tb(tb)
[('<stdin>', 2, '<module>', None),
 ('<stdin>', 2, 'f2', None),
 ('<stdin>', 2, 'f1', None)]
```

traceback.format_list(lst)

Die Funktion `format_list` bekommt eine Liste von Tupeln übergeben, wie sie beispielsweise von der Funktion `extract_tb` zurückgegeben wird. Aus diesen Informationen erzeugt `format_list` eine Liste von aufbereiteten Strings der folgenden Form:

```
>>> traceback.format_list(traceback.extract_tb(tb))
[' File "<stdin>", line 2, in <module>\n',
 ' File "<stdin>", line 2, in f2\n',
 ' File "<stdin>", line 2, in f1\n']
```

traceback.format_exception_only(type, value)

Formatiert eine Exception mit dem Typ *type* und dem Wert *value* zu einer Liste von Strings. Jeder der in dieser Liste enthaltenen Strings repräsentiert eine Zeile der Ausgabe.

```
>>> traceback.format_exception_only(TypeError,
... TypeError("Hallo Welt"))
['TypeError: Hallo Welt\n']
```

traceback.format_exception(type, value, tb[, limit[, chain]])

Formatiert eine Exception mit dem Typ *type*, dem Wert *value* und dem Stacktrace *tb* zu einer Liste von Strings. Jeder String dieser Liste repräsentiert eine Zeile der Ausgabe. Die Parameter *limit* und *chain* haben die gleiche Bedeutung wie bei der Funktion `print_exception`.

```
>>> traceback.format_exception(TypeError,
... TypeError("Hallo Welt"), tb)
['Traceback (most recent call last):\n',
'  File "<stdin>", line 2, in <module>\n',
'  File "<stdin>", line 2, in f2\n',
'  File "<stdin>", line 2, in f1\n',
'TypeError: Hallo Welt\n']
```

traceback.format_tb(tb[, limit])

Der Aufruf dieser Funktion ist äquivalent zu `format_list(extract_tb(tb, limit))`.

21.7 Analyse des Laufzeitverhaltens

Die Optimierung eines Programms ist ein wichtiger Teilbereich der Programmierung und kann viel Zeit in Anspruch nehmen. In der Regel wird zunächst ein lauffähiges Programm erstellt, das alle gewünschten Anforderungen erfüllt, bei dem jedoch noch nicht unbedingt Wert auf die Optimierung der Algorithmik gelegt wird. Das liegt vor allem daran, dass man oftmals erst beim fertigen Programm die tatsächlichen Engpässe erkennt und im frühen Stadium somit eventuell viel Zeit in die Optimierung völlig unkritischer Bereiche investieren würde.

Um das Laufzeitverhalten eines Python-Programms möglichst genau zu erfassen, existieren die drei Module `timeit`, `profile` und `cProfile` in der Standardbibliothek von Python. Diese Module sollen das Thema der nächsten Abschnitte sein.

21.7.1 Laufzeitmessung – `timeit`

Das Modul `timeit` der Standardbibliothek ermöglicht es, genau zu messen, wie lange ein Python-Programm zur Ausführung braucht. Üblicherweise wird `timeit` dazu verwendet, die Laufzeit zweier verschiedener Algorithmen für dasselbe Problem zu vergleichen.

Sie erinnern sich sicherlich noch, dass im Kapitel über Funktionen ein rekursiver Algorithmus zur Berechnung der Fakultät angegeben wurde. Es wurde gesagt, dass ein laufzeitoptimierter iterativer Algorithmus im Vergleich zu seinem rekursiven Pendant stets effizienter ist. Das wollen wir in diesem Abschnitt anhand des `timeit`-Moduls überprüfen und zusätzlich testen, um wie viel Prozent die iterative Variante tatsächlich schneller ausgeführt werden kann.

Um die Laufzeit eines Python-Codes zu testen, muss die im Modul `timeit` enthaltene Klasse `Timer` instantiiert werden. Der Konstruktor der Klasse `Timer` hat folgende Schnittstelle:

`timeit.Timer([stmt[, setup[, timer]])`

Erzeugt eine Instanz der Klasse `Timer`. Der zu analysierende Python-Code kann dem Konstruktor in Form des Parameters `stmt` als String übergeben werden. Für den zweiten Parameter `setup` kann ebenfalls ein String übergeben werden, der den Python-Code enthält, der zur Initialisierung von `stmt` benötigt wird. Demzufolge wird `setup` auch vor `stmt` ausgeführt. Beide Parameter sind optional und mit dem String "pass" vorbelegt.

Als dritter optionaler Parameter `timer` kann eine Zeitgeberfunktion übergeben werden. Dies sollte eine der Funktionen `time.time` oder `time.clock` des Moduls `time` sein. Standardmäßig wird diejenige dieser beiden Funktionen verwendet, die auf dem aktuellen System die höchste Auflösung bietet. Das ist `time.time` unter Windows und `time.clock` unter Unix-artigen Betriebssystemen. Es ist normalerweise nicht notwendig, diesen Parameter anzugeben.

Die Klasse `Timer`

Nachdem eine Instanz der Klasse `Timer` erzeugt worden ist, besitzt sie drei Methoden, die im Folgenden besprochen werden sollen. Dabei sei `t` eine Instanz der Klasse `Timer`.

`t.timeit([number])`

Diese Methode führt zunächst den `setup`-Code einmalig aus und wiederholt danach den beim Konstruktor für `stmt` übergebenen Code `number`-mal. Wenn der optionale Parameter `number` nicht angegeben wurde, wird der zu messende Code 1.000.000-mal ausgeführt.

Die Funktion gibt die Zeit zurück, die das Ausführen des gesamten Codes (also inklusive aller Wiederholungen, jedoch exklusive des Setup-Codes) in Anspruch genommen hat. Der Wert wird in Sekunden als Gleitkommazahl zurückgegeben.

Hinweis

Um das Ergebnis von äußeren Faktoren so unabhängig wie möglich zu machen, wird für die Dauer der Messung die Garbage Collection des Python-Interpreters deaktiviert. Sollte die Garbage Collection ein wichtiger mitzumessender Teil Ihres Codes sein, so lässt sie sich mit einem Setup-Code von `"gc.enable()"` wieder aktivieren.

t.repeat([repeat[, number]])

Ruft die Methode `timeit.repeat`-mal auf und gibt die Ergebnisse in Form einer Liste von Gleitkommazahlen zurück. Der Parameter *number* wird dabei der Methode `timeit` bei jedem Aufruf übergeben.

Hinweis

Es ist normalerweise keine gute Idee, den Mittelwert aller von `repeat` zurückgegebenen Werte zu bilden und diesen als durchschnittliche Laufzeit auszugeben. Andere Prozesse, die auf Ihrem System laufen, verfälschen die Ergebnisse aller Messungen. Vielmehr sollten Sie den kleinsten Wert der zurückgegebenen Liste als minimale Laufzeit annehmen, da dies die Messung mit der geringsten Systemaktivität war.

t.print_exc([file])

Sollte im zu analysierenden Code eine Exception geworfen werden, wird die Analyse sofort abgebrochen und ein Traceback ausgegeben. Der Stacktrace dieses Tracebacks ist jedoch nicht immer optimal, da er sich nicht auf den tatsächlich ausgeführten Quellcode bezieht.

Um einen aussagekräftigeren Stacktrace auszugeben, können Sie eine geworfene Exception abfangen und die Methode `print_exc` aufrufen. Diese Methode gibt einen Traceback auf dem Bildschirm aus, der sich direkt auf den zu analysierenden Code bezieht und damit die Fehlersuche erleichtert.

Durch Angabe des optionalen Parameters *file* leiten Sie die Ausgabe in eine beliebige Datei um.

Beispiel

Eingangs wurde erwähnt, dass wir das Modul `timeit` dazu verwenden werden, zu prüfen, um wie viel Prozent die iterative Fakultätsberechnung schneller ist als die rekursive.

Dazu binden wir zunächst das Modul `timeit` ein und implementieren die beiden Berechnungsfunktionen:

```
import timeit

def fak1(n):
    res = 1
    for i in range(2, n+1):
        res *= i
    return res

def fak2(n):
    if n > 0:
        return fak2(n-1)*n
    else:
        return 1
```

Danach erzeugen wir für beide Funktionen jeweils eine Instanz der Klasse `Timer`:

```
t1 = timeit.Timer("fak1(50)", "from __main__ import fak1")
t2 = timeit.Timer("fak2(50)", "from __main__ import fak2")
```

Beachten Sie, dass wir im Setup-Code zunächst die gewünschte Berechnungsfunktion aus dem Namensraum des Hauptprogramms `__main__` in den Namensraum des zu testenden Programms importieren müssen. Im eigentlich zu analysierenden Code wird nur noch die Berechnung der Fakultät von 50 unter Verwendung der jeweiligen Berechnungsfunktion angestoßen.

Schlussendlich wird die Laufzeitmessung mit 1.000.000 Wiederholungen gestartet und das jeweilige Ergebnis ausgegeben:

```
print("Iterativ: ", t1.timeit())
print("Rekursiv: ", t2.timeit())
```

Die Ausgabe des Programms lautet:

```
Iterativ: 12.8919649124
Rekursiv: 28.9529950619
```

Das würde bedeuten, dass der iterative Algorithmus etwa doppelt so schnell ist wie der rekursive. Doch diese Daten sind noch nicht wirklich repräsentativ, denn es könnte sein, dass der Test der rekursiven Funktion durch einen im System laufenden Prozess ausgebremst wurde. Aus diesem Grund starten wir einen erneuten Test:

```
print("Iterativ: ", min(t1.repeat(100, 10000)))
print("Rekursiv: ", min(t2.repeat(100, 10000)))
```

Dieses Mal führen wir eine Testreihe durch, die einen Test mit 10.000 Einzelwiederholungen 100-mal wiederholt und das kleinste der Ergebnisse ausgibt. Die Ergebnisse sind, relativ gesehen, annäherungsweise deckungsgleich mit denen der vorherigen Tests:

```
Iterativ:  0.162111997604
Rekursiv: 0.272562026978
```

Beachten Sie, dass die absoluten Zahlenwerte sehr stark vom verwendeten System abhängen. Auf einem schnelleren Computer sind sie dementsprechend kleiner.

21.7.2 Profiling – cProfile

Um eine Laufzeitanalyse eines vollständigen Python-Programms anzufertigen, wird ein sogenannter *Profiler* verwendet. Ein Profiler überwacht einen kompletten Programmdurchlauf und listet nach Beenden des Programms detailliert auf, wie viel Prozent der Laufzeit beispielsweise in welcher Funktion verbraucht wurden. Auf diese Weise kann der Programmierer die laufzeittechnischen Engpässe des Programms erkennen und an sinnvollen Stellen mit der Optimierung des Programms beginnen.

Grundsätzlich gilt: Je mehr Prozent der Laufzeit in einer bestimmten Funktion verbraucht werden, desto mehr Zeit sollten Sie investieren, um diese Funktion zu optimieren. Dagegen wäre es Zeitverschwendung, stundenlang eine Funktion zu optimieren, die vielleicht nur einmal zur Initialisierung des Programms aufgerufen wird.

Seit Python Version 2.5 ist in der Standardbibliothek ein neuer Profiler namens `cProfile` enthalten. Dieser bildet die Schnittstelle des alten Profilers `profile` ab, ist jedoch im Gegensatz zu diesem in C statt in Python geschrieben. Aus diesem Grund ist der Overhead von `cProfile` kleiner, und die Zeitmessungen sind somit besser. Wir werden hier den Profiler `cProfile` besprechen. Da dieser jedoch über die gleiche Schnittstelle wie `profile` verfügt, gilt die Beschreibung genauso für den alten Profiler.

Beachten Sie, dass der Profiler `cProfile` möglicherweise nicht für alle Python-Interpreter verfügbar ist. Das reine Python-Pendant `profile` hingegen kann überall verwendet werden.

Verwendung des Moduls

Im Modul `cProfile` sind im Wesentlichen zwei wichtige Funktionen enthalten, die im Folgenden besprochen werden sollen.

cProfile.run(command[, filename])

Diese Funktion führt den als *command* übergebenen String mithilfe einer `exec`-Anweisung aus und führt während der Ausführung eine detaillierte Laufzeitanalyse durch. Üblicherweise wird für *command* ein Funktionsaufruf der Hauptfunktion eines größeren Programms übergeben.

Über den zweiten, optionalen Parameter *filename* kann eine Datei angegeben werden, in die das Ergebnis der Laufzeitanalyse geschrieben wird. Wenn dieser Parameter nicht angegeben wurde, wird das Ergebnis auf dem Bildschirm ausgegeben. Bei diesem Ergebnis der Analyse handelt es sich um eine tabellarische Auflistung aller Funktionsaufrufe. Wie diese Tabelle aussieht und wie sie zu lesen ist, wird anhand des nächsten Beispiels erklärt.

cProfile.runctx(command, globals, locals[, filename])

Diese Funktion verhält sich wie `run`, mit dem Unterschied, dass über die Parameter *globals* und *locals* der globale und lokale Kontext festgelegt werden kann, in dem *command* ausgeführt wird. Für die Parameter *globals* und *locals* kann ein Dictionary übergeben werden, wie es von den Built-in Functions `globals` und `locals` zurückgegeben wird.

Beispiel

Im Folgenden soll eine Laufzeitanalyse für ein kleines Beispielprogramm erstellt werden. Dazu betrachten wir zunächst den Quelltext des Programms:

```
import math

def calc1(n):
    return n**2

def calc2(n):
    return math.sqrt(n)

def calc3(n):
    return math.log(n+1)

def programm():
    for i in range(100):
        calc1(i)
    for j in range(100):
        calc2(j)
    for k in range(100):
        calc3(k)

programm()
```


Im Programm existieren drei kleine Funktionen namens `calc1`, `calc2` und `calc3`, die jeweils eine ganze Zahl als Parameter übergeben bekommen, dann eine mathematische Operation auf dieser Zahl anwenden und das Ergebnis zurückgeben. In der Hauptfunktion `programm` befinden sich drei ineinander verschachtelte Schleifen, die jeweils über alle ganzen Zahlen von 0 bis 99 iterieren und eine der drei Berechnungsfunktionen aufrufen. Die Frage, die wir mithilfe des Profilers lösen möchten, lautet, an welcher Stelle sich eine Optimierung des Programms besonders lohnen würde und wo sie überflüssig wäre.

Der Profiler wird folgendermaßen in das Programm eingebunden:

```
import cProfile
[...]
cProfile.run("programm()")
```

Dabei stehen die Auslassungszeichen für den Code des Beispielprogramms. Beachten Sie, dass die Codezeile `programm()` des Beispielprogramms jetzt überflüssig ist. Das Ausführen der Laufzeitanalyse gibt folgendes Ergebnis aus:

```
2020106 function calls in 6.408 CPU seconds

Ordered by: standard name
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	6.408	6.408	<string>:1(<module>)
1	0.000	0.000	0.000	0.000	iso8859_15.py:14(decode)
1	1.690	1.690	6.408	6.408	test.py:10(programm)
100	0.000	0.000	0.000	0.000	test.py:4(calc1)
10000	0.029	0.000	0.045	0.000	test.py:6(calc2)
1000000	3.183	0.000	4.673	0.000	test.py:8(calc3)
1	0.000	0.000	0.000	0.000	{built-in method charmap_decode}
1	0.000	0.000	6.408	6.408	{built-in method exec}
1000000	1.490	0.000	1.490	0.000	{built-in method log}
10000	0.016	0.000	0.016	0.000	{built-in method sqrt}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}

Jede Zeile dieser Tabelle bezieht sich auf eine Funktion des Beispielprogramms. Die Spaltenbeschriftungen der Tabelle sind vielleicht nicht ganz klar, weswegen sie kurz erläutert werden sollen:

- `ncalls` steht für die Anzahl von Funktionsaufrufen der Funktion.
- `tottime` steht für die Gesamtzeit in Sekunden, die in der Funktion verbracht wurde. Dabei werden Aufrufe von Unterfunktionen nicht einbezogen.
- `percall` steht für den Quotienten von `tottime` und `ncalls`.

- ▶ `cumtime` steht für die Gesamtzeit in Sekunden, die in der Funktion verbracht wurde. Dabei werden Aufrufe von Unterfunktionen einbezogen.
- ▶ `percall` steht für den Quotienten von `cumtime` und `ncalls`.
- ▶ `filename:lineno(function)` steht für den Funktionsnamen, inklusive Angabe der Programmdatei und der Zeile, an der die Funktion im Quellcode steht.

Die vom Profiler angezeigte Tabelle gibt einen guten Überblick darüber, wo die zeitkritischen Funktionen des Programms liegen. In diesem Fall sticht die Funktion `calc3` hervor, die insgesamt 1.000.000-mal aufgerufen wird und in der sich satte 73 % der Laufzeit abspielen. Die 10.000-mal aufgerufene Funktion `calc2` macht hingegen nur 0,7 % der Gesamtlaufzeit aus. Die restliche Laufzeit wird, abgesehen von einem verschwindend geringen Prozentsatz in `calc1`, in der Hauptfunktion `programm` verbracht.

Zugegebenermaßen hätte man dieses Ergebnis auch anhand des Programms abschätzen können. Jede Schleife iteriert über 100 Zahlen und ruft in jedem Iterationsschritt »ihre« Funktion auf. Damit wird die innerste Funktion $100^3 = 1.000.000$ -mal aufgerufen. Auch die prozentuale Laufzeit der Funktionen `calc3` und `calc2` liegt in etwa um Faktor 100 auseinander. Etwaige Schwankungen rühren daher, dass unterschiedliche Berechnungen durchgeführt werden (Logarithmusfunktion gegen Wurzelfunktion).

Auch wenn dieses Beispiel etwas künstlich wirkt, lässt sich die Vorgehensweise auf ein größeres, zeitkritisches Projekt übertragen. Im Falle unseres Beispiels wäre man gut damit beraten, alle Ressourcen in die Optimierung der Funktion `calc3` zu stecken, da diese mit 1.000.000 Aufrufen und 73 % Laufzeitanteil doch stark dominiert.

21.7.3 Tracing – trace

Im letzten Abschnitt haben wir besprochen, welche Möglichkeiten Python bietet, ein Programm mithilfe eines Profilers zu untersuchen. Dies funktioniert im besprochenen Beispiel sehr gut, hat aber auch einen großen Nachteil: Der Profiler arbeitet auf der Funktionsebene. Das bedeutet, dass immer nur die Laufzeit ganzer Funktionen gemessen wird. Häufig ist es aber so, dass es auch innerhalb einer größeren Funktion Teile gibt, die laufzeittechnisch gesehen bedeutungslos sind, und Teile, die sehr laufzeitintensiv sind. In einem solchen Fall greift man zu einem anderen Hilfsmittel, dem sogenannten *Tracer*.

Ein Tracer, in Python über das Modul `trace` verfügbar, überwacht einen Programmablauf und registriert dabei, wie oft jede einzelne Codezeile des Programms

ausgeführt wurde. Eine solche *Überdeckungsanalyse* wird im Wesentlichen aus zwei Gründen durchgeführt:

- ▶ Mithilfe einer Überdeckungsanalyse lassen sich Codezeilen ausfindig machen, die besonders häufig aufgerufen werden und daher möglicherweise besonders laufzeitintensiv sind. Diese Zeilen könnten Sie dann gezielt optimieren. Beachten Sie aber, dass ein Tracer nicht die tatsächliche Laufzeit einer Codezeile misst, sondern nur, wie oft diese Zeile im Programmfluss ausgeführt wurde.
- ▶ Häufig muss bei sicherheitsrelevanten Programmen eine Überdeckungsanalyse vorgelegt werden, um zu beweisen, dass bei einem Test jede Codezeile mindestens einmal ausgeführt wurde. Auf diese Weise versucht man zu vermeiden, dass beispielsweise der Autopilot eines Flugzeugs ausfällt, weil eine Codezeile ausgeführt wurde, an die man beim Testen der Software nicht gedacht hat.

In diesem Abschnitt möchten wir die Überdeckungsanalyse durchführen, um laufzeitkritische Stellen in einem Programm zu identifizieren. Dazu erstellen wir eine leicht modifizierte Version des Beispielprogramms aus dem vorangegangenen Abschnitt. »Modifiziert« bedeutet, dass der Code ohne Unterfunktionen geschrieben wurde.

```
import math

def programm():
    for i in range(100):
        i**2
        for j in range(100):
            math.sqrt(j)
            for k in range(100):
                math.log(k+1)
```

Die Überdeckungsanalyse wird mithilfe des Moduls `trace` durchgeführt. Dazu ist folgender zusätzlicher Code nötig:

```
import trace
import sys

tracer = trace.Trace(
    ignoredirs = [sys.prefix, sys.exec_prefix],
    trace = 0)
tracer.run("programm()")

r = tracer.results()
r.write_results(show_missing=True, coverdir="ergebnis")
```

Zunächst wird eine Instanz der Klasse `Tracer` erzeugt. Diese bekommt zwei Schlüsselwortparameter übergeben. Über den Parameter `ignoredirs` wird eine Liste von Verzeichnissen übergeben, deren enthaltene Module nicht in die Überdeckungsanalyse einbezogen werden sollen. In diesem Fall möchten wir keine Module der Standardbibliothek übergeben und fügen deshalb die entsprechenden Verzeichnisse `sys.prefix` und `sys.exec_prefix` an. Den zweiten Parameter, `trace`, setzen wir auf 0, da sonst jede während des Programmlaufs ausgeführte Zeile auf dem Bildschirm ausgegeben wird.

Danach führen wir, analog zum Profiler, die Methode `run` der `Trace`-Instanz aus und übergeben dabei den auszuführenden Python-Code. Nachdem der `Tracer` durchgelaufen ist, können die Ergebnisse über die Methode `results` der `Trace`-Instanz abgeholt werden. Wir möchten die Ergebnisse in diesem Fall nicht weiterverarbeiten und speichern sie deshalb mithilfe der Methode `write_results` auf der Festplatte. Dabei geben wir über den Parameter `coverdir` das Unterverzeichnis an, in dem die Ergebnisse gespeichert werden sollen. Wenn für den Parameter `show_missing` `True` übergeben wird, werden Codezeilen, die während des Programmlaufs niemals ausgeführt wurden, mit einem Pfeil gekennzeichnet.

Das Ergebnis wird im Unterordner *ergebnis* als Textdatei mit dem Dateinamen *modulname.cover* abgespeichert, wobei *modulname* durch den Namen Ihres getesteten Moduls ersetzt wird. In unserem Beispiel sieht das Ergebnis folgendermaßen aus:

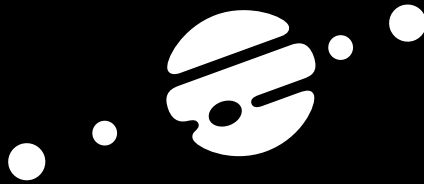
```
import trace
>>>>> import sys

>>>>> import math
>>>>> def programm():
    101:     for i in range(100):
    100:         i**2
10100:     for j in range(100):
10000:         math.sqrt(j)
1010000:     for k in range(100):
1000000:         math.log(k+1)

>>>>> tracer = trace.Trace(
>>>>>     ignoredirs = [sys.prefix, sys.exec_prefix],
>>>>>     trace = 0)
>>>>> tracer.run("programm()")

>>>>> r = tracer.results()
>>>>> r.write_results(show_missing=True, coverdir="ergebnis")
```

Sie sehen, dass die Ergebnisse zu einer gut lesbaren Datei aufbereitet werden. Im Prinzip ist die Datei in zwei Spalten aufgeteilt: Rechts steht der Quellcode des Programms und links die Anzahl der Aufrufe jeder Codezeile. Die Pfeile in der linken Spalte weisen auf Codezeilen hin, die während des überwachten Programmlaufs niemals ausgeführt wurden. Beachten Sie, dass diese Zeilen natürlich nur nicht ausgeführt wurden, solange die Überwachung des Programms aktiv war.



Teil IV

Weiterführende Themen

Herzlich willkommen zum vierten Teil dieses Buches. Zu diesem Zeitpunkt sollen Sie sich sowohl mit der Sprache Python selbst als auch mit der Standardbibliothek auskennen. In den nun folgenden Kapiteln möchten wir einige ausgesuchte Themen der Python-Programmierung vertiefen. Dazu zählen vor allem die Distribution von Python-Projekten, grafische Benutzeroberflächen und der Einsatz von Python als serverseitige Programmiersprache.

»The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.«

– Donald E. Knuth

22 Distribution von Python-Projekten

Es ist anzunehmen, dass Sie im Laufe dieses Buches bereits das eine oder andere eigenständige Python-Programm geschrieben haben. Vielleicht haben Sie sogar schon ein Programm oder Modul in Python geschrieben, das auch für andere Leute von Nutzen sein könnte. In diesem Moment stellt sich zwangsläufig die Frage, wie ein Python-Programm oder -Modul adäquat veröffentlicht werden kann. Idealerweise sollte es so geschehen, dass der Benutzer kein Experte sein muss, um es zu installieren.

Dazu ist in Pythons Standardbibliothek vor allem das Modul `distutils` enthalten, mit dem sich fertige Distributionen Ihres Programms oder Moduls erstellen lassen. Danach werden wir uns mit dem Drittanbietermodul `py2exe` beschäftigen, das Ihr Python-Programm zusammen mit dem gesamten Python-Interpreter zu einer einzigen ausführbaren Datei zusammenschweißt.

22.1 Erstellen von Distributionen – `distutils`

Das Paket `distutils` zielt auf die Distribution von Python-Modulen ab. Selbstverständlich könnten Sie einfach den Quellcode des Moduls ins Internet stellen und dazu eine Installationsanweisung liefern. Das scheint zwar aus Sicht des Entwicklers erst einmal sehr bequem zu sein, ist aber aus Sicht des Benutzers nicht gerade komfortabel. Der Benutzer muss dann, um das Programm zu installieren, von Hand allerlei Dateien kopieren und sich dabei immer strikt an gewisse Regeln halten. Das kann man von einem Benutzer nicht verlangen. Der Installationsprozess soll durch das Paket `distutils` automatisiert werden, indem sogenannte *Distributionen* erstellt werden.

Bei einer Distribution unterscheidet man grundsätzlich zwei Typen:

- ▶ Eine sogenannte *Quellcodedistribution* (engl. *source distribution*) ist ein Archiv, das den Quellcode Ihres Moduls enthält. Zusätzlich zu dem Quellcode existiert ein Installationsscript namens *setup.py*, das die Installation des Moduls durchführt. Der Benutzer braucht diese Art einer Distribution also nur herunterzuladen, zu entpacken und das Installationsscript zu starten. Der Vorteil einer Quellcodedistribution ist ihre Plattformunabhängigkeit. Es muss also nur eine Distribution erstellt werden, die für jedes unterstützte Betriebssystem verwendet werden kann.
- ▶ Eine sogenannte *Binärdistribution* (engl. *binary distribution*) ist eine ausführbare Datei, die die Installation Ihres Moduls automatisch durchführt. Der Benutzer braucht diese Art einer Distribution also nur herunterzuladen und auszuführen. Eine Binärdistribution ist für den Benutzer besonders komfortabel, da er nur zwei Arbeitsschritte auszuführen hat. Allerdings bedeutet eine Binärdistribution mehr Aufwand für den Entwickler, denn er muss das Installationsprogramm für verschiedene Plattformen erstellen.

Zum Erstellen einer Distribution sind mit dem `distutils`-Paket im Allgemeinen folgende Arbeitsschritte nötig:

- ▶ Schreiben Ihres Moduls oder Pakets¹
- ▶ Schreiben des Installationsscripts *setup.py*
- ▶ Erstellen einer Quellcodedistribution bzw. einer Binärdistribution

Diese Arbeitsschritte werden in den folgenden Abschnitten detailliert besprochen.

Hinweis

Grundsätzlich lassen sich mit `distutils` nicht nur Distributionen von Modulen oder Paketen erstellen, sondern auch von *Extensions* (dt. *Erweiterungen*). Solche Extensions können später wie ein Modul oder Paket eingebunden werden, sind aber im Gegensatz zu normalen Modulen oder Paketen in einer anderen Programmiersprache, üblicherweise C oder C++, geschrieben. Wir werden in Abschnitt 25.2.3, »Erzeugen der Extension«, auch auf die Verwendung von `distutils` im Zusammenhang mit Extensions eingehen.

22.1.1 Schreiben des Moduls

Dieser Punkt sollte so weit klar sein. Rufen Sie sich aber noch einmal ins Gedächtnis, dass es einen Unterschied zwischen einem *Modul* und einem *Paket* gibt. Wäh-

¹ Eigentlich handelt es sich dabei nicht um einen Arbeitsschritt zum Erstellen einer Distribution; dennoch ist es einleuchtenderweise eine unverzichtbare Voraussetzung. Beachten Sie, dass Sie auch mehrere Module und/oder Pakete in eine gemeinsame Distribution verpacken können. Näheres dazu erfahren Sie im Laufe dieses Abschnitts.

rend ein Modul aus nur einer Programmdatei besteht, ist ein Paket ein Ordner, der mehrere Untermodule oder -Pakete enthalten kann. Ein Paket erkennt man an der Programmdatei `__init__.py` im Paketverzeichnis. Die Unterscheidung der Begriffe »Modul« und »Paket« wird beim Erstellen des Installationsscripts noch eine Rolle spielen.

An dieser Stelle soll das Beispielmodul entwickelt werden, auf das wir uns im gesamten Kapitel beziehen werden. Dabei handelt es sich um ein sehr einfaches Modul, das die grundlegende Funktionalität von `distutils` demonstriert. Bemerkungen zu komplexeren Distributionen, die beispielsweise Pakete oder Ähnliches enthalten, finden Sie an der jeweiligen Stelle im Text.

Sinn und Zweck des Beispielmoduls ist es, einen beliebigen Text so zu verändern, dass er sich ähnlich wie dieser liest:

*Nach eienr Stidue der Cmabridge Uinverstiaet ist es eagl, in wlehcser Reiehn-
fogle die Bchustebaen in Woeretrn vokrmomen.*

*Es ist nur withcig, dsas der ertse und lettze Bchusatbe an der richthgien Stlele
snid. Der Rset knan total falcsh sein, und man knan es onhe Porbelme leesn.*

*Das ist so, wiel das mneschilche Geihrn nihct jeedn Bchustbaen liset, sodnern
das Wrot als gaznes.*

Das Modul stellt dabei eine Funktion `verwirble_text` bereit, die einen String übergeben bekommt und diesen dann so »verwirbelt« zurückgibt, dass nur der erste und letzte Buchstabe sicher auf ihrem Platz bleiben.

```
import random

def verwirble_text(text):
    liste = []
    for wort in text.split():
        w = list(wort[1:-1])
        random.shuffle(w)
        liste.append(wort[0] + "".join(w) + wort[-1])
    return " ".join(liste)
```

Die Funktion iteriert in einer Schleife über alle im übergebenen String enthaltenen Wörter. Bei jedem Schleifendurchlauf wird aus dem jeweiligen Wort der Teilstring extrahiert, der verwirbelt werden soll. Dabei wird sichergestellt, dass der erste und der letzte Buchstabe nicht in diesen Teilstring mit aufgenommen werden. Zum Verwirbeln der Buchstaben wird die Funktion `shuffle` des Moduls `random` verwendet. Schlussendlich werden der verwirbelte String, der Anfangsbuchstabe und der Endbuchstabe zusammengefügt und an den resultierenden Text angehängt.

Beachten Sie, dass die Funktion der Einfachheit halber von einem absolut gutartigen String ausgeht. Das bedeutet insbesondere, dass der String keine Satzzeichen enthalten sollte.

Im Folgenden soll nun eine Distribution dieses Moduls `verwirbeln` erstellt werden, damit auch andere Python-Programmierer möglichst komfortabel in den Genuss dieses überaus mächtigen Werkzeugs gelangen können.

22.1.2 Das Installationsscript

Der erste Schritt zur Distribution des eigenen Moduls ist das Erstellen eines *Installationsscripts*. Dies ist eine Python-Programmdatei namens `setup.py`, über die später das Erstellen der Distribution abläuft. Auch die Installation einer Quellcodedistribution aufseiten des Benutzers geschieht durch Aufruf dieser Programmdatei.

In unserem Beispiel muss im Installationsscript nur die Funktion `setup` des Moduls `distutils.core` aufgerufen werden.

```
from distutils.core import setup

setup(
    name = "verwirbeln",
    version = "1.0",
    author = "Micky Maus",
    author_email = "micky@maus.de",
    py_modules = ["verwirbeln"]
)
```

Dieser Funktion übergeben wir verschiedene Keyword Arguments, die Informationen über das Modul enthalten. Zusätzlich bekommt die Funktion über den Parameter `py_modules` alle Programmdateien übergeben, die der Distribution angehören sollen. Auf diese Weise ist es auch möglich, mehrere selbstgeschriebene Module in einer Distribution anzubieten.

Das ist tatsächlich schon alles. Diese Programmdatei kann jetzt dazu verwendet werden, das Modul auf einem beliebigen Rechner mit Python-Installation zu installieren oder eine Distribution des Moduls zu erstellen. Wie das im Einzelnen funktioniert, klären wir später, zunächst betrachten wir einmal die Funktion `setup`.

`distutils.core.setup(arguments)`

Die Funktion `setup` des Moduls `distutils.core` muss in der Programmdatei `setup.py` aufgerufen werden und stößt den jeweils gewünschten Installationsprozess an. Dazu müssen Sie der Funktion verschiedene Keyword Arguments übergeben, die Informationen über das Modul bzw. Paket bereitstellen. Tabelle 22.1 listet die wichtigsten möglichen Argumente auf und klärt kurz ihre Bedeutung.

Wenn nichts anderes angegeben ist, handelt es sich bei den jeweiligen Parametern um Strings.

Parametername	Beschreibung
name	der Name der Distribution
version	die Versionsnummer der Distribution
description	eine kurze Beschreibung der Distribution
long_description	eine ausführliche Beschreibung der Distribution
author	der Name des Autors
author_email	die E-Mail-Adresse des Autors
maintainer	der Name des Paketverwalters (Maintainer), sofern dies nicht der Autor selbst ist
maintainer_email	die E-Mail-Adresse des Paketverwalters
url	die URL einer Homepage mit weiteren Informationen zur Distribution
download_url	die URL, unter der die Distribution direkt heruntergeladen werden kann
packages	eine Liste von Strings, die die Namen aller Pakete enthält, die in der Distribution enthalten sein sollen
package_dir	Ein Dictionary, über das Pakete in Unterverzeichnissen in die Distribution aufgenommen werden können. Näheres zur Verwendung von <code>package_dir</code> folgt weiter unten.
package_data	Ein Dictionary, über das Dateien, die zu einem Paket gehören, mit in die Distribution aufgenommen werden können. Näheres zur Verwendung von <code>package_data</code> finden Sie weiter unten.
py_modules	eine Liste von Strings, die die Namen aller Python-Module enthält, die in der Distribution enthalten sein sollen
scripts	eine Liste von Strings, die die Namen aller Scriptdateien enthält, die in der Distribution enthalten sein sollen
data_files	Eine Liste von Tupeln, über die zusätzliche Dateien in die Distribution mit aufgenommen werden können. Näheres zur Verwendung von <code>data_files</code> finden Sie weiter unten.
ext_modules	Eine Liste von <code>distutils.core.Extension</code> -Instanzen, die die Namen aller Python-Erweiterungen enthält, die kompiliert werden und in der Distribution enthalten sein sollen. Näheres zu diesem Thema erfahren Sie in Abschnitt 25.2, »Schreiben von Extensions«.
script_name	Der Name des Installationsscripts, das in der Distribution verwendet werden soll. Dieser Parameter ist mit <code>sys.argv[0]</code> , also dem Namen des Scripts vorbelegt, das gerade ausgeführt wird.
license	Ein String, der die Lizenz angibt, unter der die Distribution veröffentlicht wird.

Tabelle 22.1 Mögliche Schlüsselwortparameter für setup

Distribution von Paketen

Wenn Ihr Projekt statt aus einzelnen Modulen aus einem oder mehreren Paketen besteht, müssen Sie die Namen aller Pakete, die in die Distribution aufgenommen werden sollen, über den Schlüsselwortparameter `packages` angeben:

```
from distutils.core import setup

setup(
    [...]
    packages = ["paket1", "paket2", "paket1.unterpaket1"]
)
```

In diesem Fall werden die Pakete `paket1` und `paket2`, die sich im Hauptverzeichnis befinden müssen, in die Distribution aufgenommen. Zusätzlich wird noch das Paket `unterpaket1` aufgenommen, das sich innerhalb des Pakets `paket1` befindet. Beachten Sie, dass Sie durchaus sowohl Pakete über `packages` als auch einzelne Module über `py_modules` in die Distribution aufnehmen können.

Natürlich möchte Sie niemand dazu zwingen, das Installationsscript im Hauptordner abzulegen, in dem möglicherweise bereits relativ viele Programmdateien liegen. Oftmals existiert neben dem Installationsscript ein Ordner *src* oder *source*, in dem sich dann die Module oder Pakete der Distribution befinden. Um solch einen Unterordner im Installationsscript bekannt zu machen, übergeben Sie den Schlüsselwortparameter `package_dir` beim Aufruf von `setup`:

```
from distutils.core import setup

setup(
    [...]
    package_dir = {"": "src"},
    packages = ["paket1", "paket2", "paket1.unterpaket1"]
)
```

Damit wird das Programmverzeichnis ("") auf das Verzeichnis `src` gelegt. Beachten Sie, dass diese Angabe allgemein für Pakete getätigt werden kann. So können Sie beispielsweise über einen weiteren Eintrag in diesem Dictionary mit dem Schlüssel `"paket3"` und dem Wert `"pfad/zu/meinem/paket/paket3"` ein drittes Paket einbinden, das sich in einem anderen Verzeichnis befindet als die beiden Pakete `paket1` und `paket2`. Danach kann `paket3` einfach über die Liste `packages` in die Distribution aufgenommen werden. Auch Unterpakete von `paket3` brauchen dann nicht mehr über den vollständigen Pfad angesprochen zu werden.

Distribution zusätzlicher Dateien

Neben Modulen und Paketen gehören möglicherweise weitere Dateien zu Ihrem Projekt und sollten damit auch Platz in der Distribution finden. Dazu zählen zunächst einfache Scriptdateien. Diese implementieren beispielsweise ein Tool, das

im Zusammenhang mit Ihrem Paket steht. Der Unterschied zwischen einem Modul und einer Scriptdatei ist, dass das Modul selbst keinen Python-Code ausführt, sondern nur Funktionen oder Klassen bereitstellt, während eine Scriptdatei ein lauffähiges Programm enthält. Solche Scriptdateien können beim Aufruf von `setup` durch den Schlüsselwortparameter `scripts` übergeben werden. Dabei muss für `scripts`, wie für andere Parameter auch, eine Liste von Strings übergeben werden, die jeweils einen Dateinamen enthalten.

Ein kleiner Service, den das Paket `distutils` in Bezug auf Scriptdateien durchführt, ist das automatische Anpassen der Shebang-Zeile an das Betriebssystem, auf dem die Distribution installiert wird.

Die nächste Kategorie zusätzlicher Dateien sind Ressourcen, die von bestimmten Paketen benötigt werden und in diesen enthalten sind. Beispielsweise erfordert das Paket `paket1` die beiden Dateien `hallo.txt` und `welt.txt`. In einem solchen Fall können diese Dateien über den Schlüsselwortparameter `package_data` in Form eines Dictionarys übergeben werden:

```
setup(
    [...]
    packages = ["paket1", "paket2", "paket1.unterpaket1"],
    package_data = {"paket1" : ["hallo.txt", "welt.txt"]}
)
```

Anstatt jede Datei einzeln anzugeben, können auch Wildcards verwendet werden. So würde der Wert `["*.txt"]` alle Textdateien einbinden, die sich im Verzeichnis des Paketes `paket1` befinden.

Zu guter Letzt ist es möglich, sonstige Dateien mit in die Distribution aufzunehmen. Dazu zählen alle Dateien, die in keine der vorherigen Kategorien passen, beispielsweise Konfigurationsdateien, Hilfeseiten oder Ähnliches. Diese Dateien können über den Schlüsselwortparameter `data_files` beim Funktionsaufruf von `setup` als Liste von Tupeln übergeben werden:

```
setup(
    [...]
    data_files = [("grafiken", ["test1.bmp", "test2.bmp"])
                  ("config", ["programm.cfg"])]
)
```

In diesem Fall werden die Dateien `test1.bmp` und `test2.bmp` aus dem Verzeichnis `grafiken` sowie die Datei `programm.cfg` aus dem Verzeichnis `config` in die Distribution übernommen. Die Verzeichnisse verstehen sich relativ zum Pfad des Installationsscripts. Beachten Sie, dass Sie hier durchaus auch absolute Pfade, beispielsweise für eine systemweite Konfigurationsdatei, angeben können.

Hinweis

Beachten Sie allgemein, dass Sie Ordner innerhalb eines Pfades immer durch einen einfachen Slash (/) voneinander trennen sollten. Das Paket `distutils` kümmert sich dann um die korrekte »Übersetzung« des Pfades in das Format des jeweiligen Betriebssystems.

22.1.3 Erstellen einer Quellcodedistribution

Nachdem Sie das Installationsscript geschrieben haben, können Sie mit dessen Hilfe eine Quellcodedistribution Ihres Pakets oder Moduls erstellen. Dazu wechseln Sie in das Verzeichnis, in dem das Installationsscript liegt, und führen es mit dem Argument `sdist` aus:

```
setup.py sdist
```

Dieser Befehl erzeugt die Quellcodedistribution im Unterordner `dist` nach dem Namensschema *Projektname-Version.Format*. Dabei können Sie das Format des Archivs über die Option `--formats` angeben. Es ist zudem möglich, eine Distribution in mehreren Archivformaten zu erstellen:

```
setup.py sdist --formats=zip,gzip
```

Mögliche Werte sind dabei `zip` für ein zip-Archiv (**.zip*), `gzip` für ein gz-komprimiertes tar-Archiv (**.tar.gz*), `bzip2` für ein bz2-komprimiertes tar-Archiv (**.tar.bz2*), `zzip` für ein Z-komprimiertes tar-Archiv (**.tar.Z*) sowie `tar` für ein unkomprimiertes tar-Archiv. Wenn die Option `--formats` nicht angegeben wurde, wird unter Windows ein zip-Archiv und unter Unix-Systemen ein gz-komprimiertes tar-Archiv erstellt.

In das Archiv werden alle Dateien aufgenommen, die im Installationsscript eingetragen wurden. Zusätzlich wird eine Datei namens *README* oder *README.txt* automatisch in das Archiv mit aufgenommen, sofern eine solche im selben Ordner wie das Installationsscript existiert.

Das resultierende Archiv, die Quellcodedistribution, kann jetzt veröffentlicht und verbreitet werden. Der Benutzer, der diese Distribution herunterlädt, kann Ihr Modul bzw. Ihr Paket so installieren, wie in Abschnitt 22.1.5, »Beispiel für die Verwendung einer Distribution«, beschrieben wird.

Hinweis

Beim Erstellen einer Distribution wird eine Datei namens *MANIFEST* erzeugt. Diese Textdatei enthält die Pfade zu allen Dateien, die in die Distribution mit aufgenommen werden. Beim erneuten Erstellen der Distribution werden diese Pfade aus der *MANIFEST*-Datei wieder ausgelesen, sofern die Datei existiert.

Wenn das Installationsscript aktueller ist als die *MANIFEST*-Datei, wird die *MANIFEST*-Datei beim nächsten Erstellvorgang aktualisiert. Trotzdem ist es gelegentlich notwendig, dieses Aktualisieren explizit zu erzwingen:

```
setup.py sdist --force-manifest
setup.py sdist --manifest-only
```

Mit diesen Aufrufen von *setup.py* wird das Aktualisieren der *MANIFEST*-Datei vor dem Erstellen der Distribution erzwungen bzw. ausschließlich die *MANIFEST*-Datei aktualisiert.

22.1.4 Erstellen einer Binärdistribution

Neben einer Quellcodedistribution ist das Erstellen einer Binärdistribution von besonderem Interesse, da diese den wenigsten Installationsaufwand hat. Umgekehrt bedeutet es allerdings mehr Arbeit für Sie, da für verschiedene Betriebssysteme ganz unterschiedliche Formate für Binärdistributionen erstellt werden müssen. Das prominenteste dieser Formate ist ein Windows Installer, aber auch RPM-Pakete für RPM-basierende Linux-Distributionen können erstellt werden.

Beachten Sie, dass Sie neben einer Binärdistribution stets auch eine Quellcodedistribution Ihres Projekts veröffentlichen sollten, da es Betriebssysteme gibt, die weder mit einem RPM-Paket noch mit einem Windows Installer etwas anfangen können.

Zum Erzeugen einer Binärdistribution wird das Installationsscript mit den folgenden Argumenten aufgerufen werden:

Argument	Bedeutung
bdist_rpm	Erzeugt ein RPM-Paket für RPM-basierende Linux-Distributionen wie beispielsweise Fedora Core oder openSUSE.
bdist_wininst	Erzeugt einen Windows Installer, der dazu da ist, ein Modul bzw. Paket auf einem Windows-System zu installieren. Es wird die ausführbare Datei <i>Projektname-Version-win32.exe</i> im Unterordner <i>dist</i> erzeugt.

Tabelle 22.2 Mögliche Schlüsselwortparameter für setup

Da alle Informationen, die zum Erstellen der Binärdistribution benötigt werden, bereits im Installationsscript angegeben wurden, ist das Erzeugen einer Binärdistribution tatsächlich mit den folgenden Aufrufen von *setup.py* erledigt:

```
setup.py bdist_wininst
setup.py bdist_rpm
```


Beachten Sie allgemein, dass Sie eine Binärdistribution unabhängig von einer Quellcodedistribution erzeugen können. Es ist aber sinnvoll, ein Projekt sowohl in Form einer Binärdistribution für Windows als auch in Form einer Quellcodedistribution für andere Betriebssysteme anzubieten.

Hinweis

Solange Ihr Projekt aus reinen Python-Modulen besteht, also weder Pakete noch Extensions umfasst, kann die Installationsdatei für Windows auch unter anderen Betriebssystemen, beispielsweise unter Linux, erzeugt werden. Sobald aber Pakete oder Erweiterungen enthalten sind, muss dafür ein Windows-System verwendet werden.

22.2 Distributionen installieren

Nachdem Sie jetzt das grundlegende Handwerkszeug zum Erstellen einer Binär- und Quellcodedistribution erlernt haben, sollen hier noch ein paar Worte zur Verwendung der Distributionen selbst folgen.

Zu einer Binärdistribution brauchen wir dabei nicht besonders viel zu sagen, denn die Installationsprozedur entspricht dem auf dem jeweiligen Betriebssystem üblichen Vorgehen.

Wie eine Quellcodedistribution installiert wird, ist hingegen nicht ganz intuitiv und sollte bei Ihren eigenen Distributionen unbedingt in einer *Readme*-Datei erklärt werden. Die Installation einer Quellcodedistribution verläuft nach dem folgenden Schema:

- ▶ Herunterladen der Distribution
- ▶ Entpacken des Archivs
- ▶ Ausführen der Programmdatei *setup.py* mit dem Argument `install`

Sie sehen, dass auch für die Installation einer Distribution die Programmdatei *setup.py* verantwortlich ist:

```
setup.py install
```

Wenn die Programmdatei *setup.py* mit dem Argument `install` ausgeführt wird, installiert sie die Distribution in die Python-Umgebung, die auf dem System installiert ist. Beachten Sie, dass dafür, je nach System, Administrator- oder Root-Rechte erforderlich sind.

*»Premature optimization is the root of all evil«
– Donald E. Knuth*

23 Optimierung

Als Programmierer sollten Sie mit der Zeit einen gewissen Sinn für die Ästhetik und Eleganz eines Programms entwickeln, der Ihnen sagt, wann Ihr Programm »schön« ist. Zwei wichtige Punkte eines eleganten Programms sind die Einfachheit des Ansatzes und die Laufzeiteffizienz. Diese beiden Grundsätze stehen sich in gewissem Maße gegenüber, denn häufig ist der effizienteste Ansatz nicht gerade von Klarheit geprägt.

Aus diesem Grund ist es bei einem nicht-zeitkritischen Programm absolut legitim, Effizienz und Einfachheit gegeneinander abzuwägen und zu einem gesunden Kompromiss zu gelangen. Bei einem zeitkritischen Programm ist die beste Lösung hingegen immer die effizienteste. Wir werden uns in diesem Kapitel ausschließlich mit der Optimierung der Laufzeit eines Python-Programms beschäftigen. Abgesehen von der Laufzeit können Programme jedoch noch in Hinblick auf andere Bereiche optimiert werden. So ist es beispielsweise durchaus üblich, eine Speicherplatzoptimierung durchzuführen, und letztendlich kann ein Programm auch in Hinblick auf die Einfachheit und Klarheit des Quelltextes hin optimiert werden. Der Begriff »Optimierung« wird im Folgenden ausschließlich im Sinne von »Laufzeitoptimierung« verstanden.

Beachten Sie, dass wir uns dabei rein auf Python-spezifische Optimierungsstrategien konzentrieren. Den höchsten Laufzeitgewinn erzielen Sie jedoch mit der Optimierung der Algorithmik selbst. Doch das Optimieren von Algorithmen ist ein Thema für sich und soll hier keine Berücksichtigung finden. Beachten Sie zudem, dass wir häufig kleinere Python-Codes einander gegenüberstellen werden, die den gleichen Effekt haben, sich jedoch teils gravierend in ihrer Laufzeit unterscheiden. Die in diesem Zusammenhang als »falsch« dargestellte Alternative ist natürlich nicht tatsächlich falsch, sondern im Vergleich zur anderen Variante nur ineffizient. Dennoch führen beide Alternativen zum gesteckten Ziel, so dass Sie frei entscheiden können, welcher der Varianten Sie den Vorzug gewähren möchten.

Bei einigen der hier gezeigten Optimierungsmöglichkeiten, beispielsweise dem »Inlinen« von Funktionsaufrufen oder dem Umgehen von Lookups, ist der absolute Laufzeitgewinn gering und der Verlust an Übersichtlichkeit im Quellcode deutlich. Deswegen sollten Sie solche Optimierungsstrategien nur einsetzen, wenn es unbedingt nötig ist.

23.1 Die Optimize-Option

Grundsätzlich können Sie das Laufzeitverhalten eines Python-Programms beeinflussen, indem Sie es mit der Kommandozeilenoption `-O` ausführen. Diese Option veranlasst den Interpreter dazu, den resultierenden Byte-Code zu optimieren. Das bedeutet, dass `assert`-Anweisungen und Konstrukte wie

```
if __debug__:
    mache_etwas()
```

nicht ins Kompilat aufgenommen werden und somit keinen Einfluss auf das Laufzeitverhalten des optimierten Programms mehr haben. Der optimierte Byte-Code wird in Dateien mit der Dateiendung `.pyo` gespeichert.

Durch die Kommandozeilenoption `-OO` ist es möglich, das Programm über das normale Maß hinaus zu optimieren. Wenn dies gewünscht ist, werden alle im Quelltext enthaltenen Docstrings ignoriert und nicht mit in das Kompilat aufgenommen. Auch damit erreichen Sie ein wenig mehr Laufzeiteffizienz, wenngleich sich der Gewinn in Grenzen halten sollte. Beachten Sie dann aber, dass Sie keine Docstrings mehr auslesen können, weil keine vorhanden sind. Es ist dann beispielsweise für die Built-in Function `help` nicht mehr möglich, eine Hilfeseite zu Elementen Ihres Moduls zu generieren.

23.2 Mutable vs. immutable

Bei der Einführung der Basisdatentypen zu Beginn dieses Buches haben wir sehr genau unterschieden zwischen Datentypen, die mutable (also veränderlich), und solchen, die immutable (also unveränderlich) sind. Dass diese Unterscheidung auch performancetechnische Relevanz hat, sehen wir in diesem Abschnitt.

In folgendem Beispiel soll ein Tupel mit den Zahlen von 0 bis 1.000 gefüllt werden. Selbstverständlich kennen wir dafür effiziente Möglichkeiten, doch versuchen wir es einmal mit der naiven Herangehensweise:

```
tup = ()
for i in range(1000):
    tup += (i,)
```

Zunächst wird das Tupel angelegt und dann in einer Schleife über alle Zahlen von 0 bis 1.000 die jeweils aktuelle Zahl an das Tupel angehängt. Leider haben wir beim Schreiben dieses Codes nicht berücksichtigt, dass es sich bei `tuple` um einen unveränderlichen Datentyp handelt. Das bedeutet, dass beim Hinzufügen eines Werts zu unserem Tupel `tup` jedes Mal eine neue `tuple`-Instanz erzeugt wird und die Einträge der alten Instanz in diese neue umkopiert werden müssen. Das kostet Zeit.

Als Alternative zu obigem Beispiel verwenden wir nun das veränderliche Pendant des Tupels, die Liste:

```
lst = []
for i in range(1000):
    lst += [i]
```

Im Gegensatz zur vorherigen Version muss hier beim Anhängen einer neuen Zahl keine neue `list`-Instanz erzeugt werden, was sich beim Vergleich der Laufzeit der beiden Beispiele deutlich niederschlägt: Die zweite Variante kann bis zu 90 % schneller ausgeführt werden als die erste.

Interessant ist in diesem Beispiel auch die Entwicklung des Laufzeitunterschieds in Abhängigkeit von der Anzahl der einzutragenden Zahlen. Je mehr Elemente ein Tupel enthält, desto aufwendiger ist es, seine Elemente in eine neue `tuple`-Instanz zu überführen. Hätten wir die obigen Beispiele also mit Zahlen zwischen 0 und 10.000 durchgeführt, hätte sich ein noch um einiges eindrucksvollerer Laufzeitunterschied zwischen den beiden Varianten ergeben.

Beachten Sie, dass das hier gezeigte Beispiel in älteren Python-Versionen (älter als Python 2.5) auch für Strings funktioniert. Inzwischen greifen hier Optimierungen des Python Interpreters, die verhindern, dass in jedem Schleifendurchlauf ein neuer String erzeugt wird.

23.3 Funktionsaufrufe

Eine weitere laufzeittechnisch gesehen teure Angelegenheit sind Funktionsaufrufe, weswegen Sie Funktionsaufrufe in häufig durchlaufenen Schleifen auf ihre Notwendigkeit hin überprüfen sollten. Bei besonders simplen Funktionen kann

es sinnvoll sein, die Funktion zu »inlinen«¹, den Funktionsinhalt also direkt in die Schleife zu schreiben.

Der folgende, zugegebenermaßen etwas künstliche Code:

```
def f(s):
    return s.upper()
ergebnis = f("Hallo Welt")
```

läuft um etwa 30 % langsamer als diese Variante:

```
ergebnis = "Hallo Welt".upper()
```

23.4 Schleifen

Betrachten wir noch einmal das zweite Beispiel aus dem ersten Abschnitt dieses Kapitels:

```
lst = []
for i in range(1000):
    lst += [i]
```

Wenn eine Schleife zum Erzeugen einer Liste, eines Dictionarys oder eines Sets verwendet wird, sollten Sie sich stets überlegen, ob Sie dasselbe Ergebnis nicht auch mit einer List bzw. Dict oder Set Comprehension erreichen können. Diese können schneller ausgeführt werden als eine analoge `for`-Schleife. Die folgende List Comprehension, die die gleiche Liste wie die obige Schleife erzeugt, kann um ca. 60 % schneller ausgeführt werden:

```
lst = [i for i in range(1000)]
```

Ein Aufruf der Built-in Function `map` ist ähnlich effizient wie eine List Comprehension.

23.5 C

Grundsätzlich gilt, dass ein in Python geschriebener Programmteil, auch wenn er noch so optimiert wurde, in puncto Geschwindigkeit niemals ein vergleichbares C-Programm schlagen kann. Aus diesem Grund sollten Sie an einer laufzeitkritischen Stelle so oft wie möglich auf Algorithmen zurückgreifen, die in C imple-

1 Das Wort »inlinen« kommt vom Schlüsselwort `inline` aus C++, das Funktionsaufrufe durch den Programmcode der jeweiligen Funktion ersetzt.

mentiert wurden. So lohnt es sich beispielsweise immer, eine Built-in Function einzusetzen, anstatt den entsprechenden Algorithmus selbst zu implementieren.

In Kapitel 25, »Anbindung an andere Programmiersprachen«, wird behandelt, wie Sie eigene Module oder Programmteile in C schreiben.

23.6 Lookup

Wenn über einen Modulnamen auf eine Funktion zugegriffen wird, die in diesem Modul enthalten ist, muss bei jedem Funktionsaufruf ein sogenannter *Lookup* durchgeführt werden. Dieser Lookup muss nicht durchgeführt werden, wenn eine direkte Referenz auf das Funktionsobjekt besteht. Stellen Sie sich einmal vor, Sie wollten die Quadratwurzeln aller natürlichen Zahlen zwischen 0 und 100 bestimmen. Dazu kommt einem zunächst folgender Ansatz in den Sinn:

```
import math
for i in range(100):
    math.sqrt(i)
```

Wesentlich effizienter ist es jedoch, die Funktion `sqrt` des Moduls `math` direkt zu referenzieren und über diese Referenz anzusprechen:

```
import math
s = math.sqrt
for i in range(100):
    s(i)
```

Die Schleife der zweiten Variante kann um ca. 15 % schneller ausgeführt werden als die Schleife der ersten Variante.

23.7 Exceptions

Stellen Sie sich einmal vor, Sie müssten in einer sehr frequentierten Schleife mit einem sich ständig ändernden Index `i` auf eine Liste `liste` zugreifen, können aber nicht sicher sein, ob ein Element `liste[i]` tatsächlich existiert. Wenn ein Element mit dem Index `i` existiert, soll dieses zurückgegeben werden, andernfalls 0. In einem solchen Fall ist es in der Regel ineffizient, vor dem Zugriff zu prüfen, ob ein `i`-tes Element existiert:

```
def f(liste, i):
    if i in liste:
        return liste[i]
```

```

else:
    return 0

```

In der Regel ist es wesentlich effizienter, einfach auf das *i*-te Element zuzugreifen und im Falle einer geworfenen `IndexError`-Exception den Wert 0 zurückzugeben:

```

def f(liste, i):
    try:
        return liste[i]
    except IndexError:
        return 0

```

23.8 Keyword Arguments

Das Übergeben von Positionsparametern beim Funktions- oder Methodenaufruf ist im Vergleich zu Schlüsselwortparametern grundsätzlich effizienter. Dazu soll folgende Funktion betrachtet werden, die vier Parameter erwartet:

```

def f(a, b, c, d):
    return "{0} {1} {2} {3}".format(a,b,c,d)

```

Der Funktionsaufruf

```
f("Hallo", "du", "schoene", "Welt")
```

kann um ca. 15 % schneller ausgeführt werden als der Funktionsaufruf

```
f(a="Hallo", b="du", c="schoene", d="Welt")
```

Beachten Sie, dass dies kein Grund sein sollte, allgemein auf Schlüsselwortparameter zu verzichten, denn die absolute Laufzeiterparnis liegt auch noch bei relativ vielen Funktionsaufrufen in einem nicht wahrnehmbaren Bereich.

»What you see is all you get.«
– Brian Kernighan

24 Grafische Benutzeroberflächen

Nachdem wir uns bisher ausschließlich mit schlichten Konsolenanwendungen beschäftigt haben, kann das Kribbeln in den Fingern und damit der Schritt zur grafischen Benutzeroberfläche nicht mehr länger unterdrückt werden. Im Gegensatz zur textorientierten Oberfläche von Konsolenanwendungen sind Programme mit grafischer Oberfläche intuitiver zu bedienen, grafisch ansprechender und werden im Allgemeinen als moderner empfunden. Die grafische Benutzeroberfläche eines Programms, auch *GUI* (*Graphical User Interface*) genannt, besteht zunächst aus sogenannten *Fenstern* (engl. *windows*). Innerhalb dieser Fenster lassen sich beliebige sogenannte *Steuerelemente*, häufig auch *Widgets* oder *Controls* genannt, platzieren. Unter Steuerelementen versteht man einzelne Bedieneinheiten, aus denen sich die grafische Benutzeroberfläche als Ganzes zusammensetzt. So ist beispielsweise ein Button (dt. *Schaltfläche*) oder ein Textfeld ein Steuerelement.

Sowohl die Terminologie als auch die Implementierung einer grafischen Oberfläche hängen sehr stark davon ab, welche Bibliothek, auch *Toolkit* genannt, verwendet wird. Aus diesem Grund werden wir zunächst allgemein verschiedene Toolkits auflisten, die mit Python verwendet werden können, und erst im zweiten Abschnitt zur eigentlichen Programmierung einer grafischen Benutzeroberfläche kommen.

24.1 Toolkits

Unter einem Toolkit versteht man, wie bereits gesagt, eine Bibliothek, mit deren Hilfe sich Programme mit grafischer Benutzeroberfläche erstellen lassen. Neben einigen plattformabhängigen Toolkits, beispielsweise den MFC (*Microsoft Foundation Classes*) für Windows, sind gerade im Zusammenhang mit Python plattformunabhängige Toolkits wie Qt, Gtk oder wxWidgets interessant. Diese Toolkits sind zumeist für C (Gtk) oder C++ (Qt, wxWidgets) geschrieben, lassen sich jedoch durch sogenannte *Bindings* auch mit Python ansprechen. Im Folgenden sol-

len die wichtigsten Python-Bindings für GUI-Toolkits aufgelistet und kurz erläutert werden.

tkinter

Website: <http://wiki.python.org/moin/TkInter>

Toolkit: Tk

Das Toolkit Tk wurde ursprünglich für die Sprache Tcl (*Tool Command Language*) entwickelt und ist das einzige Toolkit, das in der Standardbibliothek Pythons enthalten ist. Das Modul `tkinter` (*Tk interface*) erlaubt es, Tk-Anwendungen zu schreiben, und bietet damit eine interessante Möglichkeit, kleinere Anwendungen mit einer grafischen Benutzeroberfläche zu versehen, für die der Benutzer später keine zusätzlichen Bibliotheken installieren muss. Ein Beispiel für ein Tk-Programm ist die Entwicklungsumgebung IDLE, die jeder Python-Version beiliegt.

Im Anschluss an diese Vorstellung der im Zusammenhang mit Python gängigsten Toolkits werden Sie eine ausführliche Einführung in die Programmierung grafischer Benutzeroberflächen mit `tkinter` finden.

PyGtk

Website: <http://www.pygtk.org>

Toolkit: Gtk

Das Toolkit Gtk (GIMP Toolkit) wurde ursprünglich für das Grafikprogramm GIMP entwickelt und zählt heute neben Qt zu den meistverbreiteten plattformübergreifenden Toolkits. Sowohl das Toolkit selbst als auch die Python-Bindings PyGtk stehen unter der *GNU Lesser General Public License* und können frei heruntergeladen und verwendet werden.

Das Gtk-Toolkit ist die Grundlage der freien Desktop-Umgebung GNOME und erfreut sich daher, gerade unter Linux-Anwendern, einer hohen Beliebtheit. Obwohl es eigentlich für C geschrieben wurde, ist Gtk von Grund auf objektorientiert und kann somit gut mit Python verwendet werden.

Beachten Sie, dass ebenfalls Python-Bindings für die Bibliotheken der Desktop-Umgebung Gnome existieren. Sie heißen *PyGnome*. Mit diesen Bindings können Sie spezielle Features von Gnome nutzen und Ihre Anwendung somit perfekt in die Gnome-Umgebung integrieren.

Zum Zeitpunkt der Drucklegung dieses Buches wurde noch keine Version von PyGtk veröffentlicht, die mit Python 3 kompatibel ist.

PyQt

Website: <http://www.riverbankcomputing.co.uk>

Toolkit: Qt

Bei Qt handelt es sich um ein umfassendes Framework, das von der norwegischen Nokia-Tochter *Qt Software* (ehemals *Trolltech*) entwickelt wird und sowohl ein GUI-Toolkit als auch viel GUI-fremde Funktionalität enthält. Das durch und durch objektorientierte C++-Framework ist die Basis der freien Desktop-Umgebung KDE (*K Desktop Environment*) und aus diesem Grund ähnlich verbreitet und beliebt wie das Gtk-Toolkit. Sowohl Qt selbst als auch die Python-Bindings sind unter einem dualen Lizenzsystem erhältlich. Solange Sie Ihre Software unter einer freien Lizenz, beispielsweise der GPLv2, veröffentlichen, stehen Qt und PyQt unter der GPL¹. Für den kommerziellen Gebrauch muss eine Lizenzgebühr entrichtet werden. Qt wird für viele kommerzielle Projekte verwendet, darunter zum Beispiel Google Earth, Skype und Opera.

Zum Zeitpunkt der Drucklegung dieses Buches ist auch PyQt noch nicht in einer mit Python 3 kompatiblen Version erschienen.

Wenn Sie sich dennoch für die GUI-Programmierung mit PyQt interessieren, – beispielsweise, weil Sie noch Python 2.x einsetzen oder inzwischen eine zu Python 3 kompatible Version von PyQt herausgekommen ist –, finden Sie ein ausführliches Kapitel zu diesem Thema in der ersten Auflage dieses Buches, die auf der Buch-CD enthalten ist.

wxPython

Website: <http://www.wxpython.org/>

Toolkit: wxWidgets

Bei wxWidgets, ehemals wxWindows, handelt es sich um ein freies GUI-Toolkit, dessen erste Version bereits 1992 erschien. Das Toolkit ist dementsprechend weit entwickelt und für alle gängigen Plattformen verfügbar. Gerade in Bezug auf Python ist wxWidgets beliebt und gut dokumentiert. Eines der Ziele des wxWidgets-Projektes ist es, das Look & Feel der verschiedenen Plattformen, auf denen das Toolkit lauffähig ist, bestmöglich abzubilden. Dies wird insbesondere dadurch erreicht, dass wxWidgets die Steuerelemente nicht selbst zeichnet, sondern dafür auf die Routinen der jeweiligen Plattform zurückgreift. Sowohl wxWidgets als auch wxPython befinden sich ständig in aktiver Entwicklung.

1 Mit Qt 4.5 ist die LGPL als dritte Lizenz dazugekommen. Damit kann Qt auch in Projekten verwendet werden, deren Quellcode nicht veröffentlicht wird, ohne dass dafür Lizenzkosten anfallen.

24.2 Einführung in tkinter

Nachdem wir die verschiedenen GUI-Toolkits vorgestellt haben, für die Python-Bindings existieren, möchten wir uns der Programmierung grafischer Benutzeroberflächen widmen. Dazu enthält die Standardbibliothek das Modul `tkinter`, über das sich grafische Oberflächen mit dem Tk-Toolkit programmieren lassen. Das Modul `tkinter` ist die einzige Möglichkeit, ohne die Installation von Drittanbieterbibliotheken eine grafische Benutzeroberfläche in Python zu schreiben. Aus diesem Grund lohnt es sich, einen Blick auf `tkinter` und seine Möglichkeiten zu werfen.

24.2.1 Ein einfaches Beispiel

Zum Einstieg in die Verwendung von `tkinter` möchten wir ein einfaches Beispielprogramm präsentieren und anschließend besprechen. Das Programm bringt einen Dialog auf den Bildschirm, der den Benutzer dazu auffordert, seinen Namen einzugeben. Durch einen Klick auf einen Button wird der Name umgedreht, die Buchstaben erscheinen also in umgekehrter Reihenfolge. Ein weiterer Button beendet den Dialog. Die folgende Grafik zeigt, wie die Oberfläche später aussehen wird. Links sehen Sie die GUI vor Betätigen des »Umdrehen«-Buttons und rechts danach.



Abbildung 24.1 Die erste grafische Oberfläche

Jetzt möchten wir den Quelltext besprechen, der dem Programm zugrunde liegt. Wir werden dabei recht oberflächlich bleiben. In späteren Abschnitten sollen alle Teile des Programms zusammen mit den anderen Komponenten von Tk ausführlich erklärt werden. Doch zunächst zum Quelltext:

```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        tkinter.Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

    def createWidgets(self):
        self.nameEntry = tkinter.Entry(self)
        self.nameEntry.pack()
        self.name = tkinter.StringVar()
```

```

self.name.set("Ihr Name...")
self.nameEntry["textvariable"] = self.name

self.ok = tkinter.Button(self)
self.ok["text"] = "Ok"
self.ok["command"] = self.quit
self.ok.pack(side="right")

self.rev = tkinter.Button(self)
self.rev["text"] = "Umdrehen"
self.rev["command"] = self.onReverse
self.rev.pack(side="right")

def onReverse(self):
    self.name.set(self.name.get()[::-1])

```

Zunächst wird das Modul `tkinter` eingebunden und eine Klasse erstellt, die von `tkinter.Frame` erbt. Die Basisklasse `Frame` repräsentiert ein Rahmen-Widget. Das ist ein Steuerelement, das standardmäßig unsichtbar und ohne nennenswerte Funktionalität ist; es stellt also den idealen Ausgangspunkt für ein eigenes Widget bzw. einen eigenen Dialog dar. Wir benutzen das `Frame`-Widget praktisch als Container für die Steuerelemente unseres Dialogs.

Im Konstruktor von `MyApp` wird der Konstruktor der Basisklasse und die Methode `pack` gerufen. Diese Methode meldet das Widget, für das die Methode aufgerufen wurde, beim sogenannten *Packer* an. Der Packer ordnet die angemeldeten Steuerelemente dann gemäß den Wünschen des Programmierers auf der Oberfläche an. Insbesondere werden die Positionen der Steuerelemente in der Regel nicht durch feste Koordinaten vorgegeben, sondern durch den Packer dynamisch ermittelt.

Die Methode `createWidget` wird als letzte im Konstruktor aufgerufen und hat die Aufgabe, die für unseren Dialog benötigten Steuerelemente zu initialisieren. Dazu wird zunächst eine Instanz der Klasse `tkinter.Entry` erzeugt, die ein Eingabefeld repräsentiert. Dieses wird dann mittels `pack` auf die Oberfläche gebracht. Danach wird eine Instanz der Klasse `tkinter.StringVar` unter dem Namen `self.name` erzeugt, über die wir später den im Eingabefeld `self.nameEntry` eingegebenen Text auslesen oder verändern können. Initial setzen wir den Text auf `Ihr Name...` und melden die Steuerelementvariable beim `Entry`-Widget an. Dies geschieht, indem man genau wie bei einem Dictionary den Schlüssel `textvariable` der `Entry`-Instanz beschreibt.

Auf analoge Art und Weise erzeugen wir im weiteren Verlauf der Funktion noch die beiden Buttons, die im Dialog zur Verfügung stehen sollen. Bei einem Button können auf ähnliche Weise wie bei einem `Entry`-Widget die Optionen `text` für die Beschriftung des Buttons und `command` beschrieben werden. Über die Option

`command` wird eine Funktion festgelegt, die gerufen wird, wenn der Benutzer auf den Button geklickt hat. Im Falle des OK-Buttons setzen wir das `command`-Attribut auf die `quit`-Methode. Diese Methode stammt von der Basisklasse `Frame` und beendet den Dialog.

Beim Aufruf der `pack`-Methoden der beiden Buttons sehen Sie einen Weg, dem Packer einen Positionswunsch für ein Steuerelement mitzuteilen. Über den Schlüsselwortparameter `side` können Sie also beispielsweise festlegen, ob das Steuerelement links- oder rechtsbündig angeordnet werden soll. Dies ist natürlich nicht die einzige mögliche Angabe. Weitere Möglichkeiten lernen Sie im Laufe dieses Kapitels kennen.

Zum Schluss folgt die Implementation der Funktion `onReverse`, die gerufen wird, wenn der Benutzer auf den Button `self.rev` geklickt hat. Hier wird über die Steuerelementvariable `self.name` der im Eingabefeld eingegebene Text ausgelesen, umgedreht und wieder in das Eingabefeld geschrieben.

Wir haben jetzt eine Klasse erstellt, die unseren Dialog repräsentiert. Was noch fehlt, ist der Code, der die Klasse instantiiert und den Dialog anzeigt. Das geht folgendermaßen:

```
root = tkinter.Tk()
app = MyApp(root)
app.mainloop()
```

Zunächst wird eine Instanz der Klasse `tkinter.Tk` erzeugt, die den Dialograhmen repräsentiert. Danach instantiiieren wir unsere Applikationsklasse `MyApp` und übergeben dabei die `Tk`-Instanz als Vaterinstanz. Durch den Aufruf von `mainloop` in der dritten Zeile wird der Dialog angezeigt. Die Methode blockiert so lange, bis der Dialog beendet wird.

So viel zum ersten einführenden Tk-Programm. Im Folgenden werden wir uns mit wichtigen Aspekten von Tk befassen, beispielsweise dem Konzept der Steuerelementvariablen und dem Packer. Anschließend besprechen wir die wichtigsten Steuerelemente.

24.2.2 Steuerelementvariablen

Im einführenden Beispiel wurden sogenannte *Steuerelementvariablen* verwendet, um den Datenaustausch zwischen Programm und Oberfläche zu realisieren. Eine Steuerelementvariable ist an eine bestimmte Information eines Steuerelements gebunden, beispielsweise an den Text eines Eingabefeldes, und enthält stets den Wert, den das Steuerelement momentan anzeigt. Umgekehrt lässt sich über die

Steuerelementvariable die entsprechende Information des Steuerelements aus dem Programm heraus verändern.

Es ist klar, dass Steuerelementvariablen nicht zwangsläufig Strings sein müssen – denken Sie an ein Steuerelement zur Eingabe einer ganzen Zahl. Unglücklicherweise können die Basisdatentypen von Python nicht als Datentypen für Steuerelementvariablen verwendet werden. Wir müssen für Steuerelementvariablen auf spezielle Datentypen zurückgreifen, die im Modul `tkinter` enthalten sind. Die folgende Tabelle listet alle verfügbaren Typen auf:

Datentyp	Entsprechender Python-Typ
<code>tkinter.BooleanVar</code>	<code>bool</code>
<code>tkinter.DoubleVar</code>	<code>float</code>
<code>tkinter.IntVar</code>	<code>int</code>
<code>tkinter.StringVar</code>	<code>str</code>

Tabelle 24.1 Datentypen für Steuerelementvariablen

Alle diese Datentypen erben von der Basisklasse `tkinter.Variable`, deren Schnittstelle wir nun in einem Auszug vorstellen werden. Dazu sei *v* eine Instanz eines der obigen Datentypen:

`Variable([master[, value[, name]])`

Erzeugt und initialisiert eine Steuerelementvariable. Über das Argument *master* kann ein Master-Widget angegeben werden. Dies ist insbesondere dann sinnvoll, wenn Sie mehrere Tk-Instanzen gleichzeitig verwenden. Sollte nur eine Tk-Instanz existieren, wird diese automatisch als Master-Widget verwendet, und der Parameter *master* braucht nicht angegeben zu werden.

Für den Parameter *value* kann ein Wert angegeben werden, den die erzeugte Instanz speichern soll. Der hier übergebene Wert wird in den Datentyp der Steuerelementvariable konvertiert. Beachten Sie, dass eine gescheiterte Konvertierung erst beim Aufruf der Methode `get` auffällt.

Über den dritten Parameter *name* kann ein Variablenname angegeben werden, unter dem die Variable intern (auf Tcl-Ebene) angesprochen wird. Standardmäßig wird der Name `"PY_VARn"` verwendet, wobei *n* eine laufende Nummer ist. Dieser Parameter ist sehr speziell und wird in der Regel nicht benötigt.

`v.get()`

Die Methode `get` einer Steuerelementvariable erlaubt es, auf den in ihr gespeicherten Wert zuzugreifen. Beachten Sie, dass erst beim Aufruf von `get` eine Ex-

ception geworfen wird, wenn die Variable einen ungültigen Wert speichert. Insbesondere ein Aufruf der Methode `set` mit einem ungültigen Wert wird stillschweigend akzeptiert.

```
>>> t = tkinter.Tk()
>>> v = tkinter.StringVar(value="Hallo Welt")
>>> v.get()
'Hallo Welt'
```

v.set(value)

Die Methode `set` ist das Gegenstück zu `get` und erlaubt es, den in der Steuerelementvariablen gespeicherten Wert mit *value* zu überschreiben:

```
>>> t = tkinter.Tk()
>>> v = tkinter.StringVar(value="Hallo Welt")
>>> v.set("Bla Blubb")
>>> v.get()
'Bla Blubb'
```

Dies waren die wichtigsten Methoden einer Steuerelementvariablen. Im Folgenden möchten wir eine Einführung in die Verwendung des Packers bringen.

24.2.3 Der Packer

Eine grafische Benutzeroberfläche besteht aus einer Reihe von Steuerelementen, die in einer sinnvollen Struktur im Dialog angeordnet sind. Theoretisch ist es möglich, die Position eines jeden Steuerelements »per Hand« festzulegen. Dieser Ansatz ist problematisch, denn der Programmierer interessiert sich in der Regel nicht für die genauen Koordinaten der Steuerelemente, sondern möchte vielmehr die Struktur vorgeben, in der sie angeordnet werden sollen. Zudem müsste der Programmierer das Vergrößern und Verkleinern des Dialogs selbst behandeln und die Steuerelemente gegebenenfalls verschieben oder in ihrer Größe anpassen.

Bei Tk gibt es den sogenannten *Packer*, der die Aufgabe übernimmt, die Steuerelemente im Dialog anzuordnen. Der Programmierer braucht nur in Form von Anweisungen zur Ausrichtung eines Steuerelements eine Struktur für den Dialog vorzugeben.

Die Steuerelemente eines Dialogs sind in Tk, ähnlich wie bei anderen Toolkits, hierarchisch angeordnet. Das bedeutet, dass jedes Steuerelement über ein übergeordnetes Väterelement verfügt. Außerdem darf jedes Steuerelement beliebig viele Kindelemente enthalten. In der Tk-Terminologie wird ein Väterelement *Master* und ein Kindelement *Slave* genannt. Die Hierarchie der Steuerelemente in unserem Beispielprogramm zu Beginn dieses Kapitels veranschaulicht Abbildung 24.2.

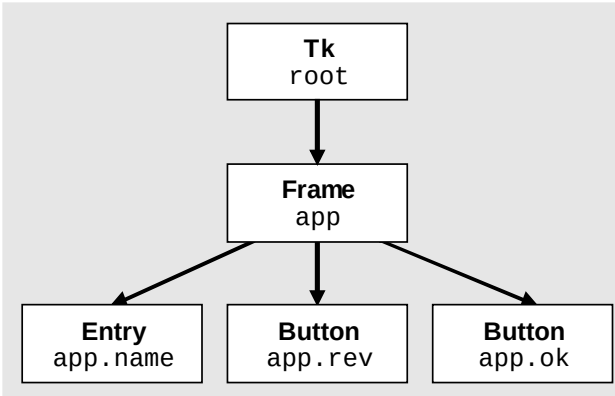


Abbildung 24.2 Steuerelementhierarchie im Beispielprogramm

Die Hierarchie der Steuerelemente ist wichtig, um die Arbeitsweise des Packers zu verstehen. Dieser ordnet nämlich die Kindelemente innerhalb ihres Vaterelements an und dann das Vaterelement mitsamt Kindelemente in dessen Vaterelement. Es ist also sinnvoll, Steuerelemente zu kapseln, damit sie gemeinsam vom Packer angeordnet werden können. Zu diesem Zweck verwendet man häufig das `Frame`-Widget.

Wie bereits gesagt, kann dem Packer in gewisser Weise ein Layout vorgegeben werden, nach dem er die Widgets anzuordnen hat. Im Beispielprogramm haben wir das erreicht, indem wir beim Aufruf der Methode `pack` der beiden Buttons den Schlüsselwortparameter `side` mit dem Wert `"right"` übergeben haben. Der Packer platziert Widgets mit dieser Layoutanweisung rechtsbündig an der Oberfläche. Im Folgenden möchten wir exemplarisch anhand des inzwischen bekannten Programms demonstrieren, wie der Packer arbeitet.

Der Packer arbeitet stets auf einem rechteckigen Teilbereich des Fensters, in den folgenden Bildern schwarz umrandet. Zu Beginn, wenn noch kein Widget »gepackt« wurde, ist dies das gesamte Fenster:



Abbildung 24.3 Arbeitsweise des Packers, Schritt 1

Als Erstes wird das `Entry`-Widget ohne nähere Layout-Angabe gepackt. In einem solchen Fall ordnet der Packer die Widgets vertikal an. Das `Entry`-Widget wird also ganz nach oben und über die volle Breite des Dialogs gelegt:



Abbildung 24.4 Arbeitsweise des Packers, Schritt 2

Interessant ist, was mit dem Arbeitsbereich passiert: Dieser wird verkleinert und umfasst das soeben gepackte Widget nicht mehr. Ein einmal platziertes Widget wird also nicht mehr aufgrund von Layoutangaben späterer Widgets hin und her geschoben.

Als Nächstes werden die Buttons in den verbleibenden Arbeitsbereich eingefügt. Sie haben die Layoutanweisung »rechtsbündig« und werden deshalb horizontal vom rechten Dialogrand aus platziert. Sie beanspruchen die vollständige Höhe des Arbeitsbereichs, da sie horizontal platziert werden, und lassen links einen Restarbeitsbereich übrig, in dem der Packer eventuelle weitere Widgets platziert:



Abbildung 24.5 Arbeitsweise des Packers, Schritt 3

So viel zur allgemeinen Arbeitsweise des Packers. Im Folgenden möchten wir die Layoutanweisungen, die Sie dem Packer in Form von Schlüsselwortparametern beim Aufruf der `pack`-Methode erteilen können, genauer unter die Lupe nehmen.

24.2.4 Ausrichtung

Der Parameter `side` bestimmt, an welcher Seite des Arbeitsbereichs ein Widget platziert wird. Mögliche Werte für `side` sind "left", "right", "top", "bottom". Beachten Sie, dass ein auf der linken bzw. rechten Seite platziertes Widget die gesamte Höhe und ein oben bzw. unten platziertes Widget die gesamte Breite des Arbeitsbereichs beansprucht. Differenziertere Layouts erreichen Sie mithilfe eines als Container missbrauchten `Frame`-Widgets.

Der Parameter `side` wurde bereits ausführlich besprochen und bedarf daher keiner weiteren Beispiele.

24.2.5 Padding

Sicherlich ist Ihnen schon aufgefallen, dass der Dialog, den wir in unserer Beispielanwendung erzeugt haben, recht seltsam aussieht. Das liegt daran, dass die

Widgets ohne Zwischenraum direkt aneinandergelegt werden. Mithilfe des sogenannten *Padding*s (dt. *Füllmaterial*) kann man die Größe dieses Zwischenraums festlegen und so ein weniger gedrungenes Dialogbild erreichen.

Grundsätzlich unterscheidet man zwei Arten von Padding: äußeres und inneres. Das *äußere Padding* beschreibt den Abstand, mit dem ein Widget neben ein anderes platziert wird. Dieser Wert wird mithilfe der Schlüsselwortparameter `padx` und `pady` der `pack`-Methode übergeben. Als Wert muss hier eine ganze Zahl angegeben werden, die dem gewünschten Abstand in Pixel entspricht. Abbildung 24.6 zeigt unseren Dialog, in dem jedes Steuerelement ein äußeres Padding von 5 Pixeln bekommen hat:



Abbildung 24.6 Padding im Einsatz

Beim *inneren Padding* handelt es sich um eine Abstandsangabe innerhalb eines Widgets. In diesem Widget enthaltene Slave-Widgets respektieren dieses Padding und werden entsprechend weiter innen angeordnet. Diese Form des Paddings ist besonders für `Frame`-Widgets interessant. Das innere Padding wird analog zum äußeren Padding über die Schlüsselwortparameter `ipadx` und `ipady` festgelegt.

24.2.6 Übersicht

Abgesehen von den bisher besprochenen Layoutanweisungen existieren einige weitere. Die folgende Tabelle gibt eine vollständige Übersicht über alle Layoutanweisungen.

Parameter	Mögliche Werte	Bedeutung
after	Widget	Das Steuerelement soll nach dem angegebenen Widget gepackt werden.
anchor	"n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	Wenn der dem Widget zugeteilte Bereich größer ist als das Widget, kann über <code>anchor</code> die Ausrichtung des Widgets innerhalb dieses Bereichs festgelegt werden. Bei den möglichen Werten handelt es sich um die Himmelsrichtungen sowie »zentriert«.

Tabelle 24.2 Layoutanweisungen

Parameter	Mögliche Werte	Bedeutung
before	Widget	Das Steuerelement soll vor dem angegebenen Widget gepackt werden.
expand	bool	Legt fest, ob die Position des Widgets bei Vergrößerung des Master-Widgets angepasst werden soll. Im Anschluss an diese Tabelle finden Sie ein Beispiel.
fill	"x", "y", "both", "none"	Die Größe des Widgets wird bei Vergrößerung des Master-Widgets angepasst. Die Größe kann dabei horizontal, vertikal, vollständig oder gar nicht angepasst werden. Im Anschluss an diese Tabelle finden Sie ein Beispiel.
in	Widget	Fügt das Steuerelement in das angegebene Master-Widget ein.
ipadx	int	Pixel-Größe für das horizontale innere Padding
ipady	int	Pixel-Größe für das vertikale innere Padding
padx	int	Pixel-Größe für das horizontale äußere Padding
pady	int	Pixel-Größe für das vertikale äußere Padding
side	"left", "right", "top", "bottom"	die Seite des Arbeitsbereichs, an der das Widget eingefügt wird

Tabelle 24.2 Layoutanweisungen (Forts.)

Abbildung 24.7 demonstriert die Auswirkungen der Layoutanweisungen `fill` und `expand` konkret an einem Beispiel:



Abbildung 24.7 Die Layoutanweisungen `fill` und `expand`

Das obere linke Bild zeigt das Verhalten, das der Dialog unseres Beispielprogramms ohne weitere Layoutanweisungen beim Vergrößern des Fensters an den Tag legt. Links daneben wurde die Layoutanweisung `expand=True` verwendet, unten links `fill="both"` und unten rechts schließlich die Kombination aus `expand` und `fill`.

24.2.7 Events

Beim Schreiben einer Tk-Anwendung wird nach dem Erstellen und Instantiieren der Applikationsklasse der Kontrollfluss durch Aufruf der Methode `mainloop` an das Tk-Framework abgegeben. Es stellt sich die Frage, auf welchem Wege wir beispielsweise auf Eingaben des Benutzers reagieren können, wenn wir gar keine wirkliche Kontrolle über das Programm und die grafische Oberfläche haben. Aus diesem Grund ist in Tk eine Reihe sogenannter *Events* definiert. Ein Event ist beispielsweise ein Tastendruck oder Mausklick des Benutzers. Mithilfe der Methode `bind` eines Widgets können wir eine selbstdefinierte Methode an ein Event binden. Eine an ein Event gebundene Methode wird vom Tk-Framework immer dann gerufen, wenn das entsprechende Event eintritt, der Benutzer also beispielsweise eine spezielle Taste gedrückt hat.

Nachfolgend werden wir zunächst die bereits angesprochene Methode `bind` besprechen, die jedes Widget besitzt, und anschließend ein Beispielprogramm entwickeln.

`bind(event, func[, add])`

Die Methode `bind` bindet die Funktion *func* an das Event *event*. Dabei muss für *func* das Funktionsobjekt einer Funktion übergeben werden, die genau einen Parameter erwartet, das sogenannte *Event-Objekt*. Diese Funktion wird *Event Handler* genannt. Wenn für den optionalen Parameter *add* der Wert `True` übergeben wird und es bereits andere Funktionen gibt, die an das Event *event* gebunden sind, werden diese nicht gelöscht, sondern *func* nur in die Liste dieser Funktionen eingereiht. Standardmäßig werden vorherige Bindungen überschrieben.

Für den wichtigsten Parameter, *event*, muss ein String übergeben werden, der das Event spezifiziert, an das die Funktion *func* gebunden werden soll. Eine solche Event-Spezifikation hat die folgende Form:

"<Modifier-Modifier-Type-Detail>"

Die beiden *Modifier*-Einträge in der Event-Spezifikation sind optional und erlauben es beispielsweise, einen Mausklick bei gedrückter -Taste und einen normalen Mausklick gesondert zu betrachten. Der *Type*-Eintrag kennzeichnet den Event-Typ, und über den *Detail*-Eintrag kann eine nähere Spezifikation er-

folgen, beispielsweise geben Sie hier an, welche Maustaste gedrückt werden muss, um das Event auszulösen.

So löst das von "<Shift-ButtonPress-1>" spezifizierte Event beispielsweise genau dann aus, wenn der Benutzer bei gedrückter -Taste mit der linken Maustaste in das Widget klickt.

Beginnen wir mit den Werten, die für die `Modifier`-Einträge in der Event-Spezifikation verwendet werden können. Hierbei handelt es sich, wie bereits anhand eines Beispiels demonstriert, um Zusatzbedingungen, unter denen das `Type`-Event auftreten muss, beispielsweise also die Eingabe eines »a« bei gedrückter -Taste. Die wichtigsten Modifier sind in der folgenden Tabelle aufgelistet und erläutert:

Attribut	Bedeutung
Alt	Die  -Taste muss beim Auftreten des Events gedrückt sein.
Control	Die  -Taste muss beim Auftreten des Events gedrückt sein.
Lock	Die Feststelltaste muss beim Auftreten des Events aktiviert sein.
Shift	Die  -Taste muss beim Auftreten des Events gedrückt sein.
Buttonx	Die Maustaste x muss beim Auftreten des Events gedrückt sein. Die Zahl x spezifiziert dabei, welche Taste gemeint ist: 1 steht für die linke, 2 für die mittlere und 3 für die rechte Maustaste. Weitere Maustasten werden entsprechend fortlaufend nummeriert. Alternative Schreibweise: Bx
Double	Das Event muss zweimal kurz hintereinander auftreten. Beispielsweise lässt sich mithilfe von <code>Double</code> ein Event für einen Doppelklick realisieren.
Triple	Das Event muss dreimal kurz hintereinander auftreten.

Tabelle 24.3 Modifier

Nachdem wir uns um die Modifier gekümmert haben, werden wir nun den `Type`-Eintrag der Event-Spezifikation besprechen. Die möglichen `Type`-Einträge werden in der folgenden Tabelle aufgelistet und erläutert. Sollte ein `Detail`-Argument erforderlich sein, wird dies ebenfalls erwähnt.

Die Spalte »Typ« gibt eine ganze Zahl an, die jedes Event eindeutig identifiziert. Diese Zahl kann später beim Empfangen eines Events ausgelesen werden, um den Event-Typ zu bestimmen.

Event	Typ	Bedeutung
KeyPress	2	Der Benutzer hat eine Taste gedrückt. Über den Detail-Eintrag in der Event-Spezifikation kann das Event auf eine bestimmte Taste beschränkt werden. Hier sind die folgenden, meist selbsterklärenden Werte möglich: <code>Alt_L</code>, <code>Alt_R</code>, <code>BackSpace</code>, <code>Cancel</code>, <code>Caps_Lock</code>, <code>Control_L</code>, <code>Control_R</code>, <code>Delete</code>, <code>End</code>, <code>Escape</code>, <code>F1-F12</code>, <code>Home</code> (»Pos1«), <code>Insert</code>, <code>Left</code>, <code>Up</code>, <code>Right</code>, <code>Down</code>, <code>Next</code> (»Page down«), <code>Num_Lock</code>, <code>Pause</code>, <code>Print</code>, <code>Prior</code> (»Page up«), <code>Return</code>, <code>Scroll_Lock</code>, <code>Shift_L</code>, <code>Shift_R</code>, <code>Tab</code>. Außerdem kann ein einzelnes Zeichen angegeben werden. Beachten Sie, dass das Widget den Eingabefokus haben muss, um dieses Event empfangen zu können. Alternative Schreibweisen: <code>Key</code> statt <code>KeyPress</code> Der Type-Eintrag <code>Key</code> oder <code>KeyPress</code> kann auch weggelassen werden. Die Eventspezifikation <code><Alt_L></code> ist beispielsweise äquivalent zu <code><KeyPress-Alt_L></code>. Wenn in der Event-Spezifikation die spitzen Klammern fehlen, wird der enthaltene Text als Eingabe von der Tastatur erwartet, damit das Event auslöst. Die Event-Spezifikation <code>"abc"</code> löst also genau dann ein <code>KeyPress</code>-Event aus, wenn der Benutzer die Zeichenfolge »abc« eingetippt hat.
KeyRelease	3	Der Benutzer hat eine Taste losgelassen. Hier können die gleichen Detail-Angaben verwendet werden wie bei <code>KeyPress</code>. Beachten Sie aber, dass die alternativen Schreibweisen, die bei <code>KeyPress</code> existieren, für <code>KeyRelease</code> nicht gültig sind.
ButtonPress	4	Der Benutzer hat eine Maustaste über dem Widget gedrückt.² Im Detail-Eintrag der Event-Spezifikation kann die Nummer einer Maustaste angegeben werden, um sich auf diese zu beschränken. Dabei entspricht 1 der linken, 2 der mittleren und 3 der rechten Maustaste. Wird der Detail-Eintrag weggelassen, so ist es egal, mit welcher Maustaste der Benutzer geklickt hat. Alternative Schreibweisen: <code>Button</code> oder schlicht <code><n></code>³ (wobei <code>n</code> die Nummer der Maustaste ist).

Tabelle 24.4 Events

- 2 Beachten Sie, dass das Widget, über dem die Maustaste gedrückt wurde, sozusagen ein »Abonnement« auf Mausevents bekommt. Konkret bedeutet das, dass die `Release`- und `Motion`-Events auch an dieses Widget gesendet werden, wenn sich die Maus inzwischen über einem anderen Widget befindet. Erst das Loslassen der Taste hebt dieses »Abonnement« auf.
- 3 In diesem Fall lautet die vollständige Event-Spezifikation `<n>`.

Event	Typ	Bedeutung
ButtonRelease	5	Der Benutzer hat eine Maustaste losgelassen. Auch hier kann über den <code>Detail</code> -Eintrag in der Event-Spezifikation angegeben werden, welche Maustaste gemeint ist.
Motion	6	Der Benutzer hat die Maus über dem Widget bewegt. Mithilfe des <code>Buttonx</code> -Modifiers kann dabei festgelegt werden, dass eine bestimmte Maustaste gedrückt sein soll, während der Benutzer die Maus bewegt.
Enter	7	Der Benutzer hat den Mauszeiger in den Bereich des Widgets bewegt.
Leave	8	Der Benutzer hat mit dem Mauszeiger den Bereich des Widgets verlassen.
FocusIn	9	Das Widget hat den Eingabefokus bekommen.
FocusOut	10	Das Widget hat den Eingabefokus verloren.
Expose	12	Das Widget war vollständig oder zum Teil von anderen Fenstern verdeckt und wurde vom Benutzer wieder hervorgeholt. In diesem Fall müssen möglicherweise Teile des Widgets neu gezeichnet werden.
Destroy	17	Das Widget wurde zerstört.
Configure	22	Das Widget hat seine Größe oder Position verändert.
MouseWheel	38	Der Benutzer hat das Mausrad bewegt. Beachten Sie, dass dieses Event unabhängig von der Position des Mauszeigers immer das Widget empfängt, das den Eingabefokus besitzt.

Tabelle 24.4 Events (Forts.)

Nachdem ein Event ausgelöst wurde, wird der damit verbundene Event Handler aufgerufen. Dieser Handler bekommt dann ein *Event-Objekt*, genauer gesagt eine `tkinter.Event`-Instanz, übergeben, das nähere Informationen über das aufgetretene Event enthält. Die folgende Tabelle listet die wichtigsten Attribute des Event-Objekts auf und erklärt sie. Beachten Sie dabei, dass `tkinter.Event` eine sehr allgemeine Klasse ist und abhängig vom ausgelösten Event nur ein Teil der Attribute sinnvoll ist.

Attribut	Bedeutung
<code>char</code>	Enthält das eingegebene Zeichen als String. Ist dieses Zeichen nicht lesbar, referenziert <code>char</code> einen leeren String. Gültig für: <code>KeyPress</code> , <code>KeyRelease</code>

Tabelle 24.5 Attribute des Event-Objekts

Attribut	Bedeutung
delta	Gibt an, wie weit der Benutzer das Mausrad gedreht hat. Dabei hat <code>delta</code> bei einer Bewegung nach vorn ein positives und bei einer Bewegung nach hinten ein negatives Vorzeichen. Gültig für: <code>MouseWheel</code>
focus	Gibt an, ob das Widget, über dem sich die Maus befindet, den Eingabefokus hat oder nicht. Gültig für: <code>Enter</code> , <code>Leave</code>
height	Enthält die neue Höhe des Widgets in Pixel. Gültig für: <code>Configure</code> , <code>Expose</code>
keycode	Enthält den Keycode der gedrückten Taste. Dies ist ein eindeutiger, ganzzahliger Identifier einer Taste. Gültig für: <code>KeyPress</code> , <code>KeyRelease</code>
keysym	Enthält den symbolischen Namen der gedrückten Taste. Gültig für: <code>KeyPress</code> , <code>KeyRelease</code>
time	Enthält einen ganzzahligen Timestamp, der den Zeitpunkt des Events beschreibt. Es handelt sich um die seit dem Systemstart vergangene Zeit in Millisekunden. Gültig bei allen Event-Typen.
type	Enthält den ganzzahligen Identifier des Event-Typen. Gültig für alle Event-Typen.
widget	Referenziert das Widget, in dem das Event aufgetreten ist. Gültig für alle Event-Typen.
width	Enthält die neue Breite des Widgets in Pixel. Gültig bei: <code>Configure</code> , <code>Expose</code>
x	Enthält die X-Koordinate des Mauszeigers in Pixel. Diese Angabe versteht sich relativ zur linken oberen Ecke des Widgets. Gültig für alle Event-Typen.
x_root	Enthält die X-Koordinate des Mauszeigers, diesmal allerdings relativ zur linken oberen Ecke des Bildschirms. Gültig für: <code>ButtonPress</code> , <code>ButtonRelease</code> , <code>KeyPress</code> , <code>KeyRelease</code> , <code>Motion</code>
y	Enthält die Y-Koordinate des Mauszeigers. Diese Angabe versteht sich relativ zur linken oberen Ecke des Widgets. Gültig für alle Event-Typen.
y_root	Enthält die Y-Koordinate des Mauszeigers, relativ zur linken oberen Ecke des Bildschirms. Gültig für: <code>ButtonPress</code> , <code>ButtonRelease</code> , <code>KeyPress</code> , <code>KeyRelease</code> , <code>Motion</code>

Tabelle 24.5 Attribute des Event-Objekts (Forts.)

Damit wäre die Einführung in die wichtigsten Event-Typen abgehandelt. An dieser Stelle möchten wir noch ein Beispielprogramm bringen, das die Verwendung von Events noch einmal veranschaulicht. Die grafische Oberfläche des Programms enthält ein Label, ein Widget, das nur einen Text anzeigt, ein Eingabefeld sowie einen Button, über den sich der Dialog beenden lässt. Das Programm soll bestimmte Events, die an das Eingabefeld gesendet werden, empfangen und den Eingang des Events durch eine entsprechende Nachricht im Label bestätigen:

```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        tkinter.Frame.__init__(self, master)
        self.pack()
        self.createWidgets()
        self.createBindings()

    def createWidgets(self):
        self.label = tkinter.Label(self)
        self.label.pack()
        self.label["text"] = "Bitte sende ein Event"

        self.entry = tkinter.Entry(self)
        self.entry.pack()

        self.ok = tkinter.Button(self)
        self.ok.pack()
        self.ok["text"] = "Beenden"
        self.ok["command"] = self.quit

    def createBindings(self):
        self.entry.bind("Entenhausen", self.eventEntenhausen)
        self.entry.bind("<ButtonPress-1>", self.eventMouseClicked)
        self.entry.bind("<MouseWheel>", self.eventMouseWheel)

    def eventEntenhausen(self, event):
        self.label["text"] = "Sie kennen das geheime Passwort!"

    def eventMouseClicked(self, event):
        self.label["text"] = "Mausklick an Position " \
                               "{0},{1}".format(event.x, event.y)

    def eventMouseWheel(self, event):
        if event.delta < 0:
            self.label["text"] = "Bitte bewegen Sie das Mausrad\
                                   " in die richtige Richtung."
        else:
            self.label["text"] = "Vielen Dank."
```

Zunächst werden wie gehabt die drei Widgets der Anwendung in der Methode `createWidgets` initialisiert, die vom Konstruktor aufgerufen wird. Danach wird die Methode `createBindings` aufgerufen, die dafür zuständig ist, die Event Handler zu registrieren. In dieser Methode werden drei Events berücksichtigt: Das Eingeben der Zeichenfolge »Entenhausen«, das Klicken mit der linken Maustaste und das Bewegen des Mausekzes. Beachten Sie, dass sich alle drei Events auf das Eingabefeld, das Entry-Widget, beziehen. Es werden also beispielsweise nur Mausklicks über diesem Eingabefeld als Event registriert.

Danach folgen die Event-Handler-Methoden `eventEntenhausen`, `eventMouseClick` und `eventMouseWheel`. In der Methode `eventMouseWheel` wird ausgenutzt, dass das Vorzeichen von `event.delta` angibt, in welche Richtung das Mausekz bewegt wurde.

Wir haben die Verwendung des Label-Widgets noch nicht im Detail besprochen, doch es ist ersichtlich, dass Sie über den Schlüssel `text` den Text verändern können, den es anzeigt. Eine ausführlichere Besprechung dieses Widgets finden Sie im nächsten Abschnitt.

Schlussendlich fehlt nur noch der Code, der den Dialog an die Oberfläche bringt. Dieser hat sich in Bezug auf das vorherige Beispielprogramm nicht verändert:

```
root = tkinter.Tk()
app = MyApp(master=root)
app.mainloop()
```

Das Beispielprogramm sieht folgendermaßen aus:

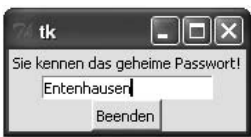


Abbildung 24.8 Beispielprogramm mit Events

Damit ist die Einführung in die Welt der Events beendet. Im Folgenden möchten wir die verschiedenen Steuerelementtypen besprechen, die Sie in `tkinter`-Anwendungen verwenden können.

24.2.8 Die Steuerelemente

Nachdem wir uns mit den Grundlagen von Tk beschäftigt haben, werden wir uns endlich wissen, welche Widgets uns zur Verfügung stehen und wie wir sie benutzen können. Eine solche Übersicht liefert dieser Abschnitt. Die folgende Tabelle

listet alle Tk-Widgets auf, die im Folgenden besprochen werden, und gibt die Seite an, auf der die jeweiligen Erklärungen zu finden sind.

Widget	Erklärung
Button	S. 673
Canvas	S. 696
Checkbutton	S. 676
Entry	S. 679
Label	S. 680
LabelFrame	S. 681
Listbox	S. 682
Menu	S. 686
Menubutton	S. 689
OptionMenu	S. 690
Radiobutton	S. 678
Scrollbar	S. 691
Spinbox	S. 692
Text	S. 694
Widget	S. 670

Tabelle 24.6 Erklärte Widgets

Widget

Die Klasse `tkinter.Widget` ist die Basisklasse aller Widgets und soll hier deswegen als erste besprochen werden. Sie stellt eine Grundfunktionalität bereit, die für jedes Widget verfügbar ist. Dazu zählen zunächst einmal die sogenannten `winfo`-Methoden (für »Widget Info«), mit deren Hilfe Sie Informationen über ein Widget, beispielsweise seine Position, herausfinden. In der folgenden Tabelle werden die wichtigsten `winfo`-Methoden erläutert; dabei sei `w` ein Widget.

Schnittstelle	Bedeutung
<code>w.winfo_children()</code>	Gibt eine Liste zurück, die alle Unterwidgets von <code>w</code> enthält. Auch Unterwidgets von Unterwidgets von <code>w</code> sind in der Liste enthalten.
<code>w.winfo_class()</code>	Gibt den Namen der Widget-Klasse von <code>w</code> als String zurück. Beispielsweise wird "Frame" zurückgegeben, wenn es sich bei <code>w</code> um ein Frame-Widget handelt.

Tabelle 24.7 `winfo`-Methoden

Schnittstelle	Bedeutung
<code>w.winfo_geometry()</code>	Gibt die Position und die Dimension des Widgets <code>w</code> in Form eines Strings zurück. Beachten Sie, dass die Position relativ zum übergeordneten Widget zu sehen ist. Der zurückgegebene String hat die Form " <code>BxH+X+Y</code> ", wobei <code>B</code> die Breite, <code>H</code> die Höhe, <code>X</code> die X-Position und <code>Y</code> die Y-Position des Widgets ist.
<code>w.winfo_height()</code>	Gibt die Höhe des Widgets <code>w</code> in Pixel zurück.
<code>w.winfo_pointerx()</code>	Gibt die X-Koordinate des Mauszeigers relativ zur oberen linken Ecke des Bildschirms zurück.
<code>w.winfo_pointery()</code>	Gibt die Y-Koordinate des Mauszeigers relativ zur oberen linken Ecke des Bildschirms zurück.
<code>w.winfo_reqheight()</code>	Gibt die für das Widget <code>w</code> angeforderte Höhe zurück. Diese Methode ist beispielsweise direkt nach dem Instantiieren eines Widgets sinnvoll, da das Widget zu diesem Zeitpunkt möglicherweise noch nicht gepackt wurde und somit keine realen Dimensionen hat.
<code>w.winfo_reqwidth()</code>	Gibt die für das Widget <code>w</code> angeforderte Breite zurück.
<code>w.winfo_rootx()</code>	Gibt die X-Position des Widgets <code>w</code> relativ zur oberen linken Ecke des Bildschirms zurück.
<code>w.winfo_rooty()</code>	Gibt die Y-Position des Widgets <code>w</code> relativ zur oberen linken Ecke des Bildschirms zurück.
<code>w.winfo_screenheight()</code>	Gibt die Höhe des Bildschirms in Pixel zurück.
<code>w.winfo_screenwidth()</code>	Gibt die Breite des Bildschirms in Pixel zurück.
<code>w.winfo_width()</code>	Gibt die Breite des Widgets <code>w</code> in Pixel zurück.
<code>w.winfo_x()</code>	Gibt die X-Koordinate des Widgets <code>w</code> relativ zum übergeordneten Widget zurück.
<code>w.winfo_y()</code>	Gibt die Y-Koordinate des Widgets <code>w</code> relativ zum übergeordneten Widget zurück.

Tabelle 24.7 `winfo`-Methoden (Forts.)

Abgesehen von den soeben besprochenen `winfo`-Methoden bietet ein Widget allerlei weitere Methoden, von denen wir an dieser Stelle nur die wichtigsten kurz erläutern möchten. Dabei sei `w` stets eine `tkinter.Widget`-Instanz.

Schnittstelle	Bedeutung
w.bind(event, func[, add])	Besprochen auf Seite 663.
w.bind_all(event, func[, add])	Wie bind, mit dem Unterschied aber, dass das Event event in allen Widgets der Applikation mit der Handler-Funktion func verbunden wird.
w.unbind(event[, funcid])	Löst alle existierenden Bindungen an das Event event auf. Über den Parameter funcid kann eine bestimmte Verbindung gezielt aufgelöst werden. Jede mit einem Event verbundene Handler-Funktion bekommt dazu eine ID zugeteilt, die beim Aufruf der Methode bind zurückgegeben wird.
w.unbind_all(event)	wie unbind, jedoch stets für alle Verbindungen an das Event event und alle Widgets der Applikation
w.cget(key)	Gibt den Wert der Option key von w zurück. Die für das Widget w gültigen Optionen lassen sich über die Methode keys herausfinden.
w.configure([cnf][, **kw])	Setzt eine oder mehrere Optionen des Widgets w. Die Optionen können dabei als Dictionary über den Parameter cnf oder in Form von Schlüsselwortparametern übergeben werden. Näheres zu Optionen finden Sie im Laufe dieses Abschnitts. Alternative Schreibweise: config.
w.destroy()	Zerstört das Widget w und alle ihm untergeordneten Widgets.
w.focus_set()	Gibt w den Eingabefokus. Wenn der Eingabefokus gerade nicht bei der Applikation liegt, wird er an w gegeben, sobald die Applikation ihn bekommt. Alternative Schreibweise: w.focus()
w.info()	Gibt ein Dictionary zurück, das die Ausrichtungsanweisungen an den Packer für w enthält.
w.keys()	Gibt eine Liste aller für w gültigen Optionen zurück.
w.mainloop()	Startet die Hauptschleife des Widgets w. Von nun an übernimmt das Tk-Toolkit den Programmfluss. Die Methode mainloop blockiert so lange, bis die Anwendung beendet wurde.

Tabelle 24.8 Methoden eines Widgets

Schnittstelle	Bedeutung
<code>w.pack([cnf][, **kw])</code>	Instruiert den Packer des übergeordneten Widgets, das Widget <code>w</code> zu packen. Dabei können die eingangs erwähnten Layoutanweisungen sowohl als Schlüsselwortparameter als auch in Form eines Dictionarys für den Parameter <code>cnf</code> übergeben werden.
<code>w.quit()</code>	Zerstört alle Widgets und beendet die Applikation.
<code>w.slaves()</code>	Gibt ein iterierbares Objekt über alle <code>w</code> untergeordneten Widgets zurück.

Tabelle 24.8 Methoden eines Widgets (Forts.)

Nach dieser etwas trockenen Vorstellung der Basisfunktionalität eines Widgets möchten wir etwas konkreter werden und uns den Widgets widmen, die später in der grafischen Oberfläche tatsächlich eingesetzt werden.

Button

Eine Instanz der Klasse `tkinter.Button` stellt ein *Button-Widget* dar; es entspricht einer Schaltfläche an der grafischen Oberfläche. Sie haben bereits im Beispielprogramm gesehen, wie man einen Button verwendet. Wichtig ist dabei, nach dem Instanzieren der `Button`-Klasse die Optionen `text` und `command` zu setzen, über die die Beschriftung des Buttons und die Handler-Funktion festgelegt werden. Die Handler-Funktion wird vom Tk-Framework gerufen, wenn der Benutzer auf den Button geklickt hat. Üblicherweise sind die folgenden vier Codezeilen ausreichend, um einen Button zu erzeugen und anzuzeigen:

```
b = tkinter.Button(master)
b["text"] = "Beschriftung"
b["command"] = handlerfunktion
b.pack()
```

Dabei muss für `master` das dem Button übergeordnete Widget angegeben werden.

Im Folgenden möchten wir die wichtigsten Optionen auflisten, die Sie im Zusammenhang mit einem Button verwenden können. Beachten Sie, dass die `Button`-Klasse viele dieser Optionen mit anderen Steuerelementklassen teilt. In solchen Fällen werden wir auf eine weitere Erläuterung bei den anderen Steuerelementen verzichten und nur einen Hinweis darauf geben, wo Sie eine Erklärung finden.

Option	Wertebereich	Bedeutung
activebackground	"#RRGGBB" ⁴	Legt die Hintergrundfarbe des Buttons im aktiven Zustand fest, also wenn sich der Mauszeiger über dem Button befindet. Alternativ sind Farbangaben wie "red", "blue" usw. möglich.
activeforeground	"#RRGGBB"	Legt die Vordergrundfarbe des Buttons im aktiven Zustand fest. Bei einem Button entspricht die Vordergrundfarbe der Schriftfarbe. Alternativ sind Farbangaben wie "red", "blue" usw. möglich.
anchor	"n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	Legt die Position der Beschriftung fest. Bei einem Wert von "sw" (»Southwest«) befindet sich die Beschriftung beispielsweise in der unteren linken Ecke des Buttons.
background	"#RRGGBB"	Legt die Hintergrundfarbe des Buttons im nicht aktiven Zustand fest. Alternative Schreibweise: bg
borderwidth	int	Legt die Dicke des Steuerelementrands fest. Die Dicke wird dabei in Pixeln angegeben. Alternative Schreibweise: bd
command	Funktionsobjekt	Setzt die Handler-Funktion, die gerufen wird, wenn der Benutzer den Button gedrückt hat. Die Handler-Funktion darf keine Parameter erwarten.
foreground	"#RRGGBB"	Legt die Vordergrundfarbe des Buttons im inaktiven Zustand fest. Alternative Schreibweise: fg

Tabelle 24.9 Optionen eines Buttons

4 Dabei handelt es sich um einen *RGB-Farbwert*. Im String steht hinter der Raute zunächst der Rot-, dann der Grün- und schließlich der Blauanteil der Farbe, jeweils als zweistellige Hexadezimalzahl geschrieben. Die einzelnen Farbangaben gehen von 00 (keine Anteil dieser Farbe) bis FF (voller Anteil dieser Farbe). Die RGB-Farbwerte #000000 und #FFFFFF entsprechen Schwarz und Weiß.

Option	Wertebereich	Bedeutung
height	int	Legt die Höhe des Buttons fest. Die Höhe wird dabei in Textzeilen gemessen.
justify	"left", "center", "right"	Legt bei einem Button mit mehrzeiliger Beschriftung die Ausrichtung der Textzeilen zueinander fest.
overrelief	"raised", "sunken", "flat", "ridge", "solid", "groove"	Legt die Darstellungsart des Buttons fest, wenn sich der Mauszeiger über ihm befindet. Die verschiedenen Darstellungsarten eines Buttons können Sie im Anschluss an diese Tabelle betrachten.
padx	int	Legt das Padding in X-Richtung fest.
pady	int	Legt das Padding in Y-Richtung fest.
relief	"raised", "sunken", "flat", "ridge", "solid", "groove"	Legt die Darstellungsart des Buttons fest. Die verschiedenen Darstellungsarten eines Buttons können Sie im Anschluss an diese Tabelle betrachten.
state	"normal", "active", "enabled", "disabled"	Setzt den Status des Buttons. Die Status <code>enabled</code> und <code>normal</code> sind synonym und bezeichnen den Normalzustand eines Buttons. Im <code>disabled</code> -Zustand ist der Button für den Benutzer nicht bedienbar. An der Benutzeroberfläche ist der Button dann grau hinterlegt. Der <code>active</code> -Zustand ist bei einem Button ebenfalls gleich dem Normalzustand. Bei einigen Steuerelementen ist dies jedoch der Zustand, in dem sich der Mauszeiger über dem Steuerelement befindet.
takefocus	bool	Legt fest, ob der Button den Fokus bekommen soll, wenn der Benutzer mit der  -Taste die Widgets des Dialogs durchschaltet.
text	str	Setzt die Beschriftung des Buttons.

Tabelle 24.9 Optionen eines Buttons (Forts.)

Option	Wertebereich	Bedeutung
textvariable	StringVar	Setzt eine Steuerelementvariable für die Beschriftung des Buttons. Die Verwendung dieser Variablen funktioniert analog zum Entry-Widget, für das sie bereits im einführenden Beispielprogramm demonstriert wurde.
width	int	Legt die Breite des Buttons fest. Die Breite wird dabei in Zeichen angegeben.

Tabelle 24.9 Optionen eines Buttons (Forts.)

Beachten Sie, dass die obige Liste unvollständig ist. Neben den aufgeführten Optionen existieren einige weitere, die allerdings zu speziell sind und den Rahmen dieses Buches sprengen würden. Wenn Sie sich für eine vollständige Auflistung aller Optionen interessieren, finden Sie diese im Internet in einer Tk-Dokumentation.

Abschließend möchten wir noch anhand eines Screenshots zeigen, wie das Aussehen eines Buttons mithilfe der `relief`-Option verändert werden kann:



Abbildung 24.9 Darstellungsarten eines Buttons

Damit ist der Button, das erste und eines der elementarsten Widgets, besprochen, und wir wenden uns weiteren Widgets zu.

Checkbutton

Der *Checkbutton* (auch Checkbox genannt) ist ein Steuerelement, das dem Button sehr ähnlich ist. Ein Checkbutton besteht aus einer kleinen Box und einer Beschriftung und kann zwei Status annehmen: aktiviert und deaktiviert. Der Benutzer kann durch einen Klick auf die kleine Box den Status ändern. Der aktivierte Zustand eines Checkbuttons wird durch einen Haken oder ein Kreuz in der Box angezeigt.



Abbildung 24.10 Zwei Checkbuttons im Einsatz

Die Erstellung und Verwendung eines Checkbuttons sind denkbar einfach und funktionieren ähnlich wie bei einem Button. Zusätzlich muss aber eine Steuerelementvariable für den aktuellen Status des Checkbuttons angelegt werden:

```
check = tkinter.Checkbutton(master)
check["text"] = "Hallo Welt"

checked = tkinter.BooleanVar()
checked.set(True)
check["variable"] = self.checked

check.pack()
```

Nach der Instantiierung des Checkbuttons kann über die Option `text` eine Beschriftung festgelegt werden. Dann wird mithilfe der Option `variable` eine Steuerelementvariable für den Status des Checkbuttons zugewiesen. Schlussendlich wird der Checkbutton durch Aufruf der Methode `pack` an die grafische Oberfläche gebracht.

Analog zu einem Button kann über die Option `command` eine Handler-Funktion angegeben werden, die vom Tk-Framework gerufen wird, wenn der Benutzer den Status des Checkbuttons wechselt. Innerhalb dieser Handler-Funktion kann dann über die Steuerelementvariable `checked` der aktuelle Status des Checkbuttons erfragt werden.

In der folgenden Tabelle werden die wichtigsten für Checkbuttons spezifischen Optionen aufgelistet und kurz erläutert:

Option	Wertebereich	Bedeutung
<code>indicatoron</code>	<code>bool</code>	Gibt an, ob die kleine Box vor der Beschriftung angezeigt werden soll oder nicht. Wenn diese Option auf <code>False</code> gesetzt ist, verhält sich der Checkbutton wie ein Button, der bei einem Klick dauerhaft eingedrückt bleibt und erst durch einen weiteren Klick wieder hervorgeholt werden kann.
<code>offvalue</code>	beliebig	Legt den Wert fest, den die Steuerelementvariable annehmen soll, wenn der Checkbutton deaktiviert ist. Beachten Sie, dass die Steuerelementvariable einen entsprechenden Typ haben muss, um diesen Wert speichern zu können.
<code>onvalue</code>	beliebig	Legt den Wert fest, den die Steuerelementvariable annehmen soll, wenn der Checkbutton aktiviert ist.

Tabelle 24.10 Spezifische Optionen eines Checkbuttons

Option	Wertebereich	Bedeutung
variable	BooleanVar (üblicherweise)	Setzt eine Steuerelementvariable, über die der Status des Checkbuttons gelesen und verändert werden kann. Beachten Sie, dass der Typ der Steuerelementvariablen zu dem für die Werte von <code>onvalue</code> und <code>offvalue</code> verwendeten Datentyp passen muss.

Tabelle 24.10 Spezifische Optionen eines Checkbuttons (Forts.)

Die Optionen `activebackground`, `activeforeground`, `anchor`, `background`, `bd`, `bg`, `borderwidth`, `command`, `fg`, `foreground`, `height`, `justify`, `overrelief`, `padx`, `pady`, `relief`, `state`, `takefocus`, `text`, `textvariable` und `width` verhalten sich wie bei einem Button.

Radiobutton

Ein *Radiobutton* ist wie ein Checkbutton ein Steuerelement, das durch einen Klick des Benutzers aktiviert oder deaktiviert werden kann. Das Besondere am Radiobutton ist, dass Sie eine Gruppe von Radiobuttons definieren können, innerhalb derer immer genau einer der Radiobuttons aktiviert ist. Im folgenden Beispielcode darf sich der Benutzer die Hauptstadt eines G8-Staates aussuchen:

```
auswahl = ["Berlin", "London", "Moskau", "Ottawa",  
           "Paris", "Rom", "Tokio", "Washington DC"]  
  
var = tkinter.StringVar()  
var.set("Moskau")  
  
for a in auswahl:  
    b = tkinter.Radiobutton(master, text=a, value=a,  
                           variable=var, command=handler)  
    b.pack(anchor="w")
```

Zunächst werden eine Liste aller Städte, die zur Auswahl stehen sollen, und eine Steuerelementvariable angelegt. Diese Variable enthält später stets den Namen der Stadt, die aktuell vom Benutzer ausgewählt ist. Zu Beginn wird Moskau ausgewählt.

Dann wird in einer Schleife über alle Städte iteriert und für jede Stadt ein Radiobutton angelegt. Hier sehen Sie eine zweite Möglichkeit, Optionen zu setzen: Anstatt die *Radiobutton*-Instanz nach dem Instantiieren wie ein Dictionary zu verwenden, können Sie die Optionen auch dem Konstruktor als Schlüsselwortparameter übergeben. In diesem Fall werden die Optionen `text` für die Beschriftung, `value` für den Wert, den die Steuerelementvariable annimmt, wenn der Radiobutton ausge-

wählt wird, `variable` für die Steuerelementvariable und `command` für die Handler-Funktion gesetzt. Analog zum Checkbutton wird diese Handler-Funktion immer dann gerufen, wenn der Benutzer seine Auswahl verändert.

Es ist offensichtlich, dass alle Radiobuttons, die dieselbe Steuerelementvariable besitzen, eine Gruppe bilden.

Die grafische Oberfläche des obigen Beispielprogramms sieht aus wie in Abbildung 24.11:

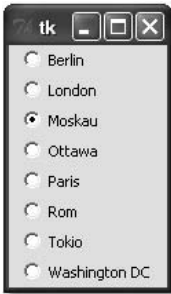


Abbildung 24.11 Eine Menge Radiobuttons

Die Optionen `activebackground`, `activeforeground`, `anchor`, `background`, `bd`, `bg`, `borderwidth`, `command`, `fg`, `foreground`, `height`, `justify`, `overrelief`, `padx`, `pady`, `relief`, `state`, `takefocus`, `text`, `textvariable` und `width` verhalten sich wie bei einem Button. Die Optionen `indicatoron` und `variable` verhalten sich wie bei einem Checkbutton.

Entry

Bei einem *Entry-Widget* handelt es sich um ein einzeliges Eingabefeld, in das der Benutzer beliebigen Text schreiben kann. Der folgende Beispielcode erzeugt ein Entry-Widget und schreibt einen Text hinein:

```
entry = tkinter.Entry(master)
entryVar = tkinter.StringVar()
entryVar.set("Hallo Welt")
entry["textvariable"] = entryVar
entry.pack()
```

Der Code ist denkbar einfach: Es werden ein Entry-Widget und eine Steuerelementvariable instantiiert. Dann wird die Steuerelementvariable auf einen Wert gesetzt und mit dem Eingabefeld verbunden. Schlussendlich wird das Eingabefeld dem Packer übergeben. Abbildung 24.12 zeigt die grafische Oberfläche dieses Beispielprogramms:



Abbildung 24.12 Ein Entry-Widget

Die Optionen `background`, `bd`, `bg`, `borderwidth`, `fg`, `foreground`, `justify`, `relief`, `takefocus` und `width` verhalten sich wie bei einem `Button`. Die wichtigsten für ein `Entry-Widget` spezifischen Optionen werden in der folgenden Tabelle besprochen:

Option	Wertebereich	Bedeutung
<code>show</code>	<code>str</code>	Legt ein Zeichen fest, das bei einer Eingabe des Benutzers statt des tatsächlich eingegebenen Textes angezeigt werden soll. Diese Option ist beispielsweise für Passworтеingaben interessant.
<code>textvariable</code>	<code>Variable</code>	Setzt eine Steuerelementvariable, über die auf den im Eingabefeld angezeigten Text lesend und schreibend zugegriffen werden kann.

Tabelle 24.11 Spezifische Optionen eines `Entry-Widgets`

Label

Ein *Label-Widget* ist ein sehr einfaches Widget, dessen einzige Aufgabe es ist, einen Text an der grafischen Oberfläche anzuzeigen. Das folgende Beispielprogramm zeigt die Verwendung des `Label-Widgets`:

```
label = tkinter.Label(master)
label["text"] = "Hallo Welt"
label.pack()
```

Nach dem Instantiieren der Klasse `tkinter.Label` wird über die Option `text` die Beschriftung des Labels verändert und das Label schlussendlich dem Packer übergeben. Statt die Option `text` zu verwenden, wäre es an dieser Stelle auch möglich gewesen, über die Option `textvariable` eine Steuerelementvariable zu definieren und diese mit dem gewünschten Text zu beschreiben.

Die grafische Oberfläche dieses Beispielprogramms ist wenig spektakulär und in Abbildung 24.13 zu sehen:



Abbildung 24.13 Ein Label im Einsatz

Da ein Label ein Widget darstellt, das wenig Interaktion mit dem Benutzer ermöglicht, gibt es auch keine für ein Label spezifischen Optionen. Die Optionen `activebackground`, `activeforeground`, `anchor`, `background`, `bd`, `bg`, `borderwidth`, `command`, `fg`, `foreground`, `height`, `justify`, `overrelief`, `padx`, `pady`, `relief`, `state`, `takefocus`, `text`, `textvariable` und `width` verhalten sich aber wie bei einem Button.

LabelFrame

Ein *LabelFrame-Widget* ist eine spezielle Form des Frame-Widgets und dient zur Gruppierung von Steuerelementen. Das LabelFrame-Widget zeichnet einen beschrifteten Rahmen um die ihm untergeordneten Widgets. Dies demonstriert der folgende Code:

```
group = tkinter.LabelFrame(master, text="Entenhausen")
c1 = tkinter.Checkbutton(group, text="Donald Duck")
c1.pack(anchor="w")
c2 = tkinter.Checkbutton(group, text="Dagobert Duck")
c2.pack(anchor="w")
c3 = tkinter.Checkbutton(group, text="Gustav Gans")
c3.pack(anchor="w")
group.pack()
```

Die Verwendung des LabelFrame-Widgets beschränkt sich offenbar auf die Instanzierung der `tkinter.LabelFrame`-Klasse, die Angabe der `text`-Option für die Beschriftung des Rahmens und die Erzeugung der untergeordneten Widgets. Die in diesem Beispiel erzeugte grafische Oberfläche sieht aus wie in Abbildung 24.14:



Abbildung 24.14 Ein LabelFrame-Widget

Neben der bereits angesprochenen Option `text` für die Beschriftung des Rahmens bietet ein LabelFrame-Widget zwei weitere wichtige Optionen an, die in der folgenden Tabelle zusammengefasst werden:

Option	Wertebereich	Bedeutung
labelanchor	"n", "ne", "e", "se", "s", "sw", "w", "nw"	Legt die Position und die Ausrichtung der Beschriftung auf dem Rahmen fest.
labelwidget	Widget	Statt einer schlichten Beschriftung kann über diese Option auch ein beliebiges Widget auf dem Rahmen positioniert werden.

Tabelle 24.12 Spezifische Optionen eines LabelFrame-Widgets

Die Optionen `bd`, `borderwidth`, `fg`, `foreground`, `relief`, `text`, `background`, `bg`, `height`, `padx`, `pady`, `takefocus` und `width` verhalten sich wie bei einem `Button`.

Listbox

Bei einer *Listbox* handelt es sich ganz allgemein um ein Steuerelement, das eine Liste von Einträgen darstellt. Je nach Anwendung darf der Benutzer einen oder mehrere Einträge auswählen oder modifizieren. Die Verwendung einer *Listbox* in `tkinter` zeigt das folgende Beispielprogramm:

```
eintraege = ["Berlin", "London", "Moskau", "Ottawa",  
            "Paris", "Rom", "Tokio", "Washington DC"]  
  
l = tkinter.Listbox(master)  
l.pack()  
  
for stadt in eintraege:  
    l.insert("end", stadt)
```

Zunächst legen wir eine Liste an, die die Einträge enthält, die wir später in die *Listbox* schreiben möchten. Dann erzeugen und packen wir eine Instanz der Klasse `tkinter.Listbox`. Schlussendlich rufen wir für jeden gewünschten Eintrag die Methode `insert` der *Listbox* auf, die den jeweiligen Eintrag ans Ende der *Listbox* anhängt. Die von diesem Quellcode erzeugte *Listbox* sieht folgendermaßen aus:



Abbildung 24.15 Die fertige Listbox

Diese Listbox an sich ist noch wenig spektakulär. Was fehlt, sind zum einen Möglichkeiten, die Listbox zu modifizieren, beispielsweise einen Eintrag zu löschen, und zum anderen Wege, um die Auswahl des Benutzers einzubinden. Dazu betrachten wir zunächst die wichtigsten Methoden, die eine `tkinter.Listbox`-Instanz mitbringt:

Methode	Bedeutung
<code>curselection()</code>	Gibt eine Liste zurück, die die Indizes der aktuell ausgewählten Einträge enthält.
<code>delete(first[, last])</code>	Löscht den Eintrag mit dem Index <code>first</code> bzw. alle Einträge mit einem Index zwischen <code>first</code> und <code>last</code> .
<code>get(first[, last])</code>	Gibt den Eintrag mit dem Index <code>first</code> bzw. eine Liste mit den Einträgen, deren Indizes zwischen <code>first</code> und <code>last</code> liegen, zurück.
<code>insert(index, *elements)</code>	Fügt die Elemente <code>elements</code> an der Position <code>index</code> in die Listbox ein. Die bestehenden Einträge der Listbox werden dabei nicht überschrieben.
<code>itemcget(index, option)</code>	Gibt den Wert der Option <code>option</code> des Eintrags mit dem Index <code>index</code> zurück.
<code>itemconfig(index, **option)</code>	Ermöglicht das Setzen von Optionen des Eintrags mit dem Index <code>index</code> . Die Optionen müssen als Schlüsselwortparameter angegeben werden.
<code>selection_clear(first[, last])</code>	Hebt eine eventuelle Auswahl des Eintrags mit dem Index <code>first</code> bzw. der Einträge mit einem Index zwischen <code>first</code> und <code>last</code> auf. Alternative Schreibweise: <code>select_clear</code>
<code>selection_includes(index)</code>	Gibt <code>True</code> zurück, wenn der Eintrag mit dem Index <code>index</code> in der Listbox ausgewählt ist.
<code>selection_set(first[, last])</code>	Fügt das Element mit dem Index <code>first</code> bzw. die Elemente mit Indizes zwischen <code>first</code> und <code>last</code> zur aktuellen Auswahl hinzu.
<code>size()</code>	Gibt die Anzahl der Einträge der Listbox zurück.

Tabelle 24.13 Methoden einer Listbox

Zu guter Letzt besprechen wir noch die wichtigsten Optionen, die im Zusammenhang mit einer Listbox verwendet werden können:

Option	Wertebereich	Bedeutung
selectmode	"single", "browse", "multiple", "extended"	Legt das Auswahlverhalten der Listbox fest. Bei den Werten "single" und "browse" kann immer nur ein Eintrag der Listbox ausgewählt werden. Mit den Werten "multiple" und "extended" können mehrere Einträge ausgewählt werden.
xscrollcommand	Funktionsobjekt	Möglichkeit zur Anbindung einer horizontalen Scrollbar. Näheres dazu erfahren Sie im Abschnitt über die Scrollbar auf Seite 691.
yscrollcommand	Funktionsobjekt	Möglichkeit zur Anbindung einer vertikalen Scrollbar. Näheres dazu erfahren Sie im Abschnitt über die Scrollbar auf Seite 691.

Tabelle 24.14 Spezifische Optionen eines Listbox-Widgets

Das vorangegangene Beispielprogramm zur Listbox war sehr statisch. Der Benutzer konnte zwar einen Eintrag der Listbox auswählen, doch daraufhin passierte nichts. Das folgende Beispielprogramm zeigt, wie Sie auf eine Änderung der Benutzerauswahl reagieren können. Dazu erstellen wir die folgende Applikationsklasse:

```
class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        tkinter.Frame.__init__(self, master)
        self.pack()

        self.eintraege = ["Berlin", "London", "Moskau", "Ottawa",
                          "Paris", "Rom", "Tokio", "Washington DC"]
        self.lb = tkinter.Listbox(self)
        self.lb.pack(fill="both", expand="true")
        self.lb["selectmode"] = "extended"
        self.lb.insert("end", *self.eintraege)
        self.lb.bind("<<ListboxSelect>>", self.selectionChanged)
        self.lb.selection_set(0)

        self.label = tkinter.Label(self)
        self.label.pack()
        self.selectionChanged(None)
```

Im Konstruktor wird zunächst eine Listbox angelegt und mit den bereits bekannten Städtenamen gefüllt. Danach verbinden wir eine Handler-Methode mit dem sogenannten *virtuellen Event* <<ListboxSelect>>. Ein virtuelles Event ist ein spe-

zieltes Event, das nur für mit einem bestimmten Widget-Typ verwendet werden kann.⁵ Das `<<ListboxSelect>>`-Event wird immer dann gerufen, wenn der Benutzer die Auswahl in der Listbox verändert hat.

Dann wird das erste Element der Listbox als einziges ausgewählt und ein Label-Widget erzeugt. Zum Schluss wird die Handler-Methode `selectionChanged` aufgerufen, um das Label-Widget mit einem sinnvollen Text zu versehen. Die `selectionChanged`-Methode sieht folgendermaßen aus:

```
def selectionChanged(self, event):
    self.label["text"] = "Wir fahren nach: " + ", ".join(
        (self.lb.get(i) for i in self.lb.curselection()))
```

Jedes Mal, wenn der Benutzer die Auswahl in der Listbox verändert hat, wird diese Methode gerufen. Hier werden die Indizes der ausgewählten Einträge ausgelesen, die dazugehörigen Städtenamen herausgefunden und durch ein Komma getrennt in das Label geschrieben.

Der Code, der die obige Applikationsklasse instantiiert und die grafische Oberfläche anzeigt, kann unverändert von vorherigen Beispielprogrammen übernommen werden und sieht so aus:

```
root = tkinter.Tk()
app = MyApp(root)
app.mainloop()
```

Die mit diesem Beispielprogramm erzeugte grafische Oberfläche ist in Abbildung 24.16 dargestellt:



Abbildung 24.16 Eine Listbox mit Benutzerinteraktion

⁵ Sie können auch Ihre eigenen virtuellen Events definieren. Das geschieht über die Methode `event_add` eines Widgets und wird in diesem Buch nicht thematisiert. Weitere Informationen dazu finden Sie in einer Tk-Dokumentation im Internet.

Menu

Bei einer komplexeren grafischen Benutzeroberfläche befindet sich direkt unter der Titelleiste eines Dialogs häufig eine *Menüleiste*, die mehrere Menüs enthält. Ein *Menü* ist eine Art Schaltfläche, über die der Benutzer eine Liste weiterer Kommandos erreichen kann. Üblicherweise findet man beispielsweise im Menü DATEI die Kommandos SPEICHERN und SPEICHERN UNTER. Das folgende Beispielprogramm zeigt, wie Sie einen tkinter-Dialog mit einem Menü ausstatten:

```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master=None):
        tkinter.Frame.__init__(self, master)

        self.menuBar = tkinter.Menu(master)
        master.config(menu=self.menuBar)
        self.fillMenuBar()

        self.pack()

    def fillMenuBar(self):
        self.menuFile = tkinter.Menu(self.menuBar, tearoff=False)
        self.menuFile.add_command(label="Öffnen",
                                   command=self.handler)
        self.menuFile.add_command(label="Speichern",
                                   command=self.handler)
        self.menuFile.add_command(label="Speichern unter",
                                   command=self.handler)
        self.menuFile.add_separator()
        self.menuFile.add_command(label="Beenden",
                                   command=self.quit)
        self.menuBar.add_cascade(label="Datei",
                                   menu=self.menuFile)

    def handler(self):
        print("Hallo Welt!")
```

Die tkinter.Tk-Instanz, die jeder tkinter-Anwendung zugrunde liegt, besitzt eine Option `menu`, über die eine Menüleiste gesetzt werden kann. Das geschieht innerhalb des Konstruktors der Klasse `MyApp`, in dem über den Parameter `master` auf die tkinter.Tk-Instanz zugegriffen werden kann. Dort wird zunächst die Menüleiste als `tkinter.Menu`-Instanz erzeugt und schließlich über die Option `menu` als Menüleiste eingetragen.

Die Methode `fillMenuBar`, die vom Konstruktor aus aufgerufen wird, hat die Aufgabe, die frisch erzeugte Menüleiste zu befüllen. Dazu wird zunächst ein

Menü erzeugt, das fortan unter dem Namen `menuFile` verfügbar ist. Über den Parameter `tearoff` kann gesteuert werden, ob ein Ablösen des Menüs möglich sein soll. Dieses Verhalten ist bei den meisten Desktop-Umgebungen unidiomatisch und wurde deshalb nicht zugelassen. Sie können aber mit dieser Einstellung experimentieren.

Danach werden dem Menü über die Methode `add_command` Menüpunkte hinzugefügt. Diese erhalten eine Beschriftung (`label`) und eine Handler-Funktion (`command`), die analog zur Handler-Funktion eines Buttons aufgerufen wird, wenn der Benutzer den jeweiligen Menüpunkt angewählt hat. In diesem Beispiel wird dann die Methode `handler` aufgerufen, die durch Ausgabe eines Textes demonstriert, dass das Beispielprogramm funktioniert. Einzig beim Menüpunkt `BEENDEN` wird `self.quit` als Handler-Methode eingetragen, um die Anwendung zu beenden.

Über die Methode `add_separator` kann eine Trennlinie ins Menü eingefügt werden, um thematisch zusammengehörende Menüpunkte auch optisch zu gruppieren.

Schlussendlich wird über die Methode `add_cascade` der Menüleiste das neue Menü unter dem Titel `DATEI` hinzugefügt.

Nun fehlt nur noch der Code, der die Klasse `MyApp` instantiiert und anzeigt. Dieser hat sich im Vergleich zu den vorangegangenen Beispielprogrammen nicht verändert und sieht so aus:

```
root = tkinter.Tk()
app = MyApp(root)
app.mainloop()
```

Die grafische Oberfläche des Beispielprogramms sieht aus wie in Abbildung 24.17:

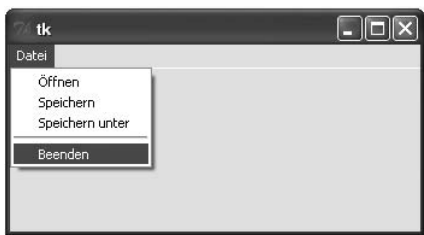


Abbildung 24.17 Eine Menüleiste

Dieses simple Beispiel lässt sich noch etwas erweitern, denn neben der Methode `add_command` zum Hinzufügen eines einfachen Menüpunkts und der Methode `add_separator` zum Hinzufügen einer Trennlinie verfügt eine `Menu`-Instanz über

die Methoden `add_checkbutton` und `add_radiobutton`. Diese beiden Methoden erlauben es, Radiobuttons und Checkbuttons in einem Menü zu verwenden. Die Optionen, die die Radio- bzw. Checkbuttons näher spezifizieren, werden den Methoden als Schlüsselwortparameter übergeben.

Eine grafische Oberfläche, die von diesen Möglichkeiten Gebrauch macht, könnte aussehen wie in Abbildung 24.18:



Abbildung 24.18 Eine komplexere Menüleiste

Tabelle 24.15 listet noch einmal die wichtigsten Methoden der `tkinter.Menu`-Klasse auf.

Methode	Bedeutung
<code>add_cascade([cnf][, **kw])</code>	Fügt ein Untermenü hinzu. Die <code>tkinter.Menu</code> -Instanz des Untermenüs muss für den Schlüsselwortparameter <code>menu</code> angegeben werden. Über den Schlüsselwortparameter <code>label</code> kann die Beschriftung des Untermenüs bestimmt werden.
<code>add_checkbutton([cnf][, **kw])</code>	Fügt einen Checkbutton als Menüeintrag hinzu. Optionen für den Checkbutton, insbesondere <code>label</code> und <code>command</code> , können als Schlüsselwortparameter übergeben werden.
<code>add_command([cnf][, **kw])</code>	Fügt einen einfachen Menüeintrag hinzu. Wichtig sind hier die Optionen <code>label</code> für die Beschriftung und <code>command</code> für die Handler-Funktion.
<code>add_radiobutton([cnf][, **kw])</code>	Fügt einen Radiobutton als Menüeintrag hinzu. Optionen für den Radiobutton, insbesondere <code>label</code> und <code>command</code> , können als Schlüsselwortparameter übergeben werden.
<code>add_separator([cnf][, **kw])</code>	Fügt eine Trennlinie ins Menü ein.

Tabelle 24.15 Methoden einer `tkinter.Menu`-Instanz

Methode	Bedeutung
<code>delete(index1[, index2])</code>	Löscht den Menüeintrag an der Stelle <code>index1</code> bzw. alle Menüeinträge zwischen den Stellen <code>index1</code> und <code>index2</code> . Die Menüeinträge sind beginnend bei 0 aufsteigend durchnummeriert.
<code>entrycget(index, option)</code>	Gibt den Wert der Option <code>option</code> des Menüeintrags an der Stelle <code>index</code> zurück.
<code>entryconfig(index[, cnf][, **kw])</code>	Ruft die <code>configure</code> -Methode des Menüeintrags an der Stelle <code>index</code> mit den übergebenen Schlüsselwortparametern auf. Alternative Schreibweise: <code>entryconfig</code>
<code>post(x, y)</code>	Zeigt das Menü an der Position an, die durch <code>x</code> und <code>y</code> bestimmt wird. Auf diese Weise lässt sich zum Beispiel ein Kontextmenü realisieren. Die Koordinaten <code>x</code> und <code>y</code> verstehen sich relativ zur oberen linken Ecke des Bildschirms.
<code>unpost()</code>	Versteckt das Menü wieder.

Tabelle 24.15 Methoden einer `tkinter.Menu`-Instanz (Forts.)

Menubutton

Bei einem *Menubutton* handelt es sich um eine Schaltfläche, die ein Menü anzeigt, wenn der Benutzer sie betätigt. Wie ein solches Steuerelement verwendet werden kann, zeigt der folgende Beispielcode:

```
mb = tkinter.Menubutton(master, text="Hallo Welt")

menu = tkinter.Menu(mb, tearoff=False)
menu.add_checkbutton(label="Donald Duck")
menu.add_checkbutton(label="Onkel Dagobert")
menu.add_checkbutton(label="Tick, Trick und Track")

mb["menu"] = menu
mb.pack()
```

Zunächst erzeugen wir eine Instanz der Klasse `tkinter.Menubutton`. Die Beschriftung der *Menubutton*-Schaltfläche legen wir dabei über die Option `text` fest. Danach erstellen wir das Menü, das beim Betätigen des *Menubuttons* angezeigt werden kann. Dabei handelt es sich um eine `tkinter.Menu`-Instanz, die ganz wie im vorangegangenen Abschnitt erzeugt und verwendet werden kann.

Schlussendlich wird das Menü über die Option `menu` des *Menubuttons* an diesen gebunden und der *Menubutton* an den Packer übergeben. Die resultierende grafische Oberfläche zeigt Abbildung 24.19.



Abbildung 24.19 Ein Menubutton im Einsatz

Beachten Sie, dass sich die Beschriftung des Menubuttons unabhängig von dem im Menü ausgewählten Eintrag nicht ändert.

Wie im Beispielpogramm gesehen, wird über die Option `menu` des Menubuttons das Menü festgelegt, das er anzeigen soll. Die Option `direction` steuert, an welcher Stelle, relativ zur Position des Menubuttons, das Menü angezeigt werden soll. Hier sind die Werte "above", "below", "left" und "right" möglich.

Die Optionen `activebackground`, `activeforeground`, `anchor`, `background`, `bd`, `bg`, `borderwidth`, `command`, `fg`, `foreground`, `height`, `justify`, `overrelief`, `padx`, `pady`, `relief`, `state`, `takefocus`, `text`, `textvariable` und `width` verhalten sich aber wie bei einem Button.

OptionMenu

Bei dem vorhin besprochenen Menubutton haben wir festgestellt, dass sich die Beschriftung des Buttons nicht automatisch ändert, wenn der Benutzer einen Menüpunkt angewählt hat. Genau dieses Verhalten ist jedoch häufig gewünscht. Ein Steuerelement, das sich so verhält, wird oft als *Dropdown-Liste* bezeichnet. Im Tk-Framework nennt sich dieses Steuerelement *OptionMenu*. Der folgende Code zeigt, wie sich das OptionMenu-Widget verwenden lässt:

```
lst = ["Linux", "MacOS X", "Windows"]
var = tkinter.StringVar()
op = tkinter.OptionMenu(master, var, *lst)
var.set("Linux")
op.pack()
```

Nachdem wir eine Liste mit den für die Dropdown-Liste gewünschten Einträgen und eine Steuerelementvariable angelegt haben, instantiiieren wir eine `tkinter.OptionMenu`-Instanz und übergeben dabei sowohl die Steuerelementvariable `var` als auch die Liste der Einträge. Beachten Sie, dass die Liste der Einträge ausgepackt wird, die Einträge also in Form von Positionsparametern übergeben werden müssen. Schlussendlich setzen wir die Steuerelementvariable auf

einen initialen Wert und rufen die `pack`-Methode des `OptionMenu`-Widgets auf. Die von diesem Code erzeugte GUI sieht aus wie in Abbildung 24.20.

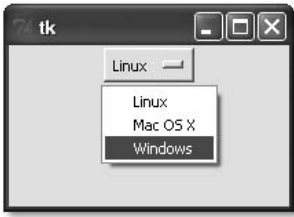


Abbildung 24.20 Ein `OptionMenu`-Widget im Einsatz

Um auf eine Auswahl des Benutzers reagieren zu können, existiert wieder die `Option command`. Über diese Option kann eine Handler-Funktion angegeben werden, die immer dann vom Tk-Framework gerufen wird, wenn der Benutzer eine Auswahl getroffen hat. Beachten Sie, dass die Handler-Funktion einen Parameter – den Text des aktuell ausgewählten Elements – erwarten muss.

Zusätzlich können die Optionen `activebackground`, `activeforeground`, `anchor`, `background`, `bd`, `bg`, `borderwidth`, `fg`, `foreground`, `height`, `justify`, `padx`, `pady`, `relief`, `state`, `takefocus`, `text` und `width` verwendet werden wie bei einem `Button`. Die Option `indicatoron` verhält sich wie bei einem `Checkbutton`, die Option `direction` wie bei einem `Menubutton`.

Scrollbar

Es kommt häufig vor, dass der Inhalt eines Widgets, beispielsweise die Einträge einer Liste, mehr Platz benötigen, als das Widget bietet. Für solche Fälle erlauben es bestimmte Widget-Typen, eine sogenannte *Scrollbar* anzubinden. Das folgende Beispiel zeigt, wie man eine Scrollbar im Zusammenhang mit einer `Listbox` verwendet:

```
lb = tkinter.Listbox(master)
lb.pack(side="left")

sb = tkinter.Scrollbar(master)
sb.pack(fill="y", side="left")

lb.insert("end", *[i*i for i in range(200)])
lb["yscrollcommand"] = sb.set
sb["command"] = lb.yview
```

Zunächst werden eine `Listbox` und eine `Scrollbar` erzeugt und auf der Oberfläche so angeordnet, dass die `Scrollbar` rechts neben der `Listbox` steht. Danach wird die

Listbox mit den Quadraten der Zahlen zwischen 0 und 50 gefüllt. Was jetzt noch fehlt, ist die Verbindung zwischen der Scrollbar und der Listbox, denn momentan haben wir nur zwei voneinander unabhängige Widgets erzeugt.

Um die Scrollbar an die Listbox zu binden, setzen wir zunächst die Option `yscrollcommand` der Listbox auf die Methode `set` der Scrollbar. Dies erlaubt ein automatisches Anpassen der Scrollbar, wenn die Einträge der Listbox über die Pfeiltasten oder das Mausrad gescrollt werden. Danach wird die `command`-Option der Scrollbar auf die Methode `yview` der Listbox gesetzt. Nun ist auch das Scrollen der Listbox mit der Scrollbar möglich.

Die von obigem Beispiel erzeugte grafische Oberfläche ist in Abbildung 24.21 dargestellt.

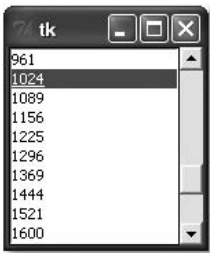


Abbildung 24.21 Eine Listbox mit angebundener Scrollbar

Statt der Option `yscrollcommand` und der Methode `yview` hätten wir auch die Option `xscrollcommand` und die Methode `xview` verwenden können, um eine horizontale Scrollbar zu realisieren.

Beachten Sie, dass die hier demonstrierte Möglichkeit, eine Scrollbar anzubinden, nicht nur bei einer Listbox funktioniert, sondern auch bei Canvas-, Entry-, Spinbox- und Text-Widgets.

Da die Methoden und Optionen der `Scrollbar`-Klasse sehr speziell sind und in den meisten Fällen nicht benötigt werden, möchten wir sie an dieser Stelle nicht besprechen, sondern es bei der beispielhaften Erklärung belassen.

Spinbox

Bei einer *Spinbox* handelt es sich um ein Widget, in das der Benutzer eine ganze Zahl eintragen kann. Zusätzlich kann der Benutzer die eingetragene Zahl über zwei Schaltflächen am Rand des Widgets nach oben oder unten korrigieren. Der folgende Codeausschnitt demonstriert die Verwendung einer Spinbox:

```
s = tkinter.Spinbox(master)
s["from"] = 0
s["to"] = 100
s.pack()
```

Die Spinbox wird instantiiert, danach werden über die Optionen `from` und `to` die Grenzen festgelegt, in denen sich die gespeicherte Zahl bewegen darf. In diesem Beispiel darf keine Zahl größer als 100 oder kleiner als 0 eingetragen werden. Beachten Sie, dass die Option `from`, wenn sie in Form eines Schlüsselwortparameters bei der Instantiierung von `tkinter.Spinbox` oder einem Aufruf der Methode `configure` übergeben wird, als `from_` geschrieben werden muss. Dies ist wichtig, da `from` in Python ein Schlüsselwort ist.

Die in dem obigen Beispiel erzeugte Spinbox ist in Abbildung 24.22 abgebildet.



Abbildung 24.22 Eine Spinbox im Einsatz

Anstatt zwei Grenzen über die Optionen `from` und `to` anzugeben, können Sie die erlaubten Werte auch konkret angeben. Dies geschieht über die Option `values`:

```
s = tkinter.Spinbox(master)
s["values"] = (2,3,5,7,11,13,19)
s.pack()
```

In diesem Fall kann der Benutzer eine der Primzahlen zwischen 2 und 19 in der Spinbox auswählen. Beachten Sie, dass die Reihenfolge, in der die Zahlen in der Spinbox erscheinen, durch die Reihenfolge der Werte im Tupel gegeben ist. Es muss kein Zusammenhang zur tatsächlichen Ordnungsrelation der Zahlen gegeben sein.

Wenn Sie die Werte, die die Spinbox annehmen kann, konkret angeben, können Sie dort auch andere Datentypen als `int` verwenden:

```
s["values"] = ("A", "B", "C")
```

In diesem Beispiel kann der Benutzer die Spinbox verwenden, um einen der drei Buchstaben auszuwählen.

Die Optionen `activebackground`, `background`, `bd`, `bg`, `borderwidth`, `command`, `fg`, `foreground`, `justify`, `relief`, `state`, `takefocus`, `textvariable` und `width` verhalten sich aber wie bei einem Button. Die folgende Tabelle listet die wichtigsten Optionen auf, die für eine Spinbox spezifisch sind:

Option	Wertebereich	Bedeutung
format	"%pad1.pad2f"	Legt die Formatierung der Ausgabe fest. Dabei bezeichnet <code>pad1</code> die Anzahl der für den Vor-kommaanteil reservierten Ziffern und <code>pad2</code> die Zahl der darzustellenden Nachkommastellen.
from	float	untere Grenze für den in der Spinbox darge-stellten Wert
increment	float	Legt den Wert fest, um den der Wert bei einem Klick auf die kleinen Buttons der Spin-box inkrementiert bzw. dekrementiert werden soll.
to	float	obere Grenze für den in der Spinbox darge-stellten Wert
values	tuple, list	Legt die möglichen Werte der Spinbox konkret fest.
xscrollcommand	Funktionsobjekt	Möglichkeit zur Anbindung eine horizontalen Scrollbar. Näheres dazu erfahren Sie im Abschnitt über die Scrollbar auf Seite 691.

Tabelle 24.16 Spezifische Optionen eines Spinbox-Widgets

Text

Bisher haben wir nur eine Möglichkeit kennengelernt, mithilfe des Entry-Wid-gets einzelne Texteingaben vom Benutzer zu verlangen. Das *Text-Widget* erlaubt es uns, einen mehrzeiligen und formatierten Text anzuzeigen oder vom Benutzer eingeben zu lassen.

Das Text-Widget ist sehr mächtig, weswegen wir es hier anhand von Beispielen erklären werden. Eine umfassende Erläuterung aller Möglichkeiten des Text-Widgets finden Sie in einer Tk-Dokumentation.

Das folgende Beispiel zeigt, wie man das Text-Widget dazu verwendet, formatier-ten Text anzuzeigen:

```
text = tkinter.Text(master)
text.pack()

text.tag_config("o", foreground="orange")
text.tag_config("u", underline=True)

text.insert("end", "Hallo Welt\n")
text.insert("end", "Dies ist ein langer, oranger Text\n", "o")
text.insert("end", "Und unterstrichen ebenfalls", "u")
```

Zunächst wird das Text-Widget instantiiert und gepackt. Danach definieren wir sogenannte Tags, die es uns später erlauben, den darzustellenden Text zu formatieren. In diesem Fall definieren wir das Tag `b` für blauen und das Tag `u` für unterstrichenen Text.

Danach fügen wir drei Textzeilen jeweils ans Ende des im Widget enthaltenen Textes an. Die erste Textzeile soll unformatiert, die zweite blau und die dritte unterstrichen angezeigt werden. Abbildung 24.23 zeigt, wie der Text dargestellt wird:

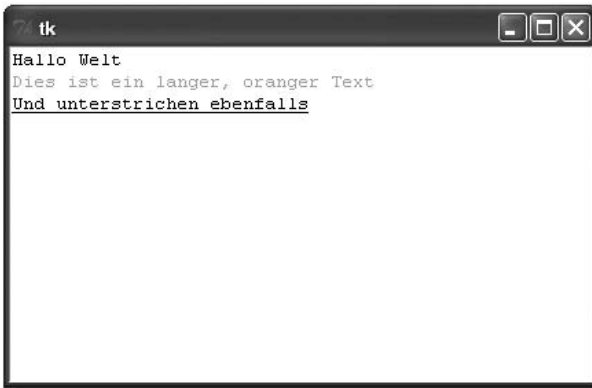


Abbildung 24.23 Das Text-Widget

Standardmäßig ist es dem Benutzer erlaubt, den im Text-Widget dargestellten Text zu verändern. Das folgende Beispiel zeigt, wie Sie auf eine Eingabe des Benutzers im Text-Widget reagieren und den eingegebenen Text auslesen:

```
text = tkinter.Text(master)
text.pack()

def textChanged(event):
    print("Text: {}".format(text.get("1.0", "end")))

text.bind("<KeyRelease>", textChanged)
```

Im Gegensatz zum Entry-Widget ist es nicht möglich, eine Steuerelementvariable für den in einem Text-Widget enthaltenen Text einzurichten. Das liegt daran, dass das Text-Widget auch Formatierungsanweisungen für den darzustellenden Text benötigt. Diese müssten in einer String-Steuerelementvariablen umständlich über spezielle Codefolgen nachgebildet werden.

Statt einer Steuerelementvariablen bietet das Text-Widget die Methode `get`, über die Sie den im Widget dargestellten Text auslesen können. Beachten Sie, dass es

sich um den reinen Text handelt; jegliche Formatierungsanweisungen gehen beim Auslesen mittels `get` verloren.

Im Beispielprogramm wurde im Wesentlichen ein Event Handler für das `KeyRelease`-Event eingerichtet. Dieses wird immer dann ausgelöst, wenn der Benutzer eine Taste loslässt, während das Text-Widget den Eingabefokus besitzt. Beachten Sie, dass wir das `KeyRelease`-Event verwenden und nicht `KeyPress`. Würden wir das `KeyPress`-Event nutzen, würde unser Event Handler aufgerufen, bevor das vom Benutzer eingegebene Zeichen ins Text-Widget eingetragen wurde.

Im Event Handler `textChanged` rufen wir die Methode `get` des Text-Widgets auf. Diese Methode bekommt zwei Indizes übergeben, die angeben, welches Teilstück des Textes ausgelesen werden soll. In diesem Fall interessieren wir uns für den gesamten im Widget enthaltenen Text und geben die Indizes `1.0` und `end` an. Der Index `1.0` liest sich als »erste Zeile, nulltes Zeichen«, wobei zu beachten ist, dass die Indizierung der Zeilen bei eins und die der Spalten, also der Zeichen, bei null beginnt. Der Index `1.0` bezeichnet also das erste Zeichen des im Widget dargestellten Textes. Der Index `end` steht selbstverständlich für das letzte Zeichen des im Widget enthaltenen Textes.

Wenn Sie nicht möchten, dass der Benutzer den im Text-Widget dargestellten Text editieren kann, können Sie das Text-Widget über die `state`-Option deaktivieren. Näheres zur `state`-Option finden Sie bei der Erklärung des Button-Widgets auf Seite 673.

Außerdem ist es möglich, eine horizontale oder vertikale Scrollbar mit einem Text-Widget zu verbinden. Dies geschieht analog zum Listbox-Widget über die Optionen `xscrollcommand` und `yscrollcommand`. Ein Beispiel dazu finden Sie im Abschnitt über die Scrollbar auf Seite 691.

Da das Text-Widget sehr mächtig ist, möchten wir es bei dieser Einführung belassen. Eine vollständige Erläuterung des Widgets mit all seiner Funktionalität würde den Rahmen dieses Kapitels sprengen. Um ein Beispiel der Vielseitigkeit des Text-Widgets zu geben, sei gesagt, dass es möglich ist, Grafiken in den Text einzubetten. Nähere Informationen dazu finden Sie in einer Tk-Dokumentation im Internet.

Canvas

Das sogenannte *Canvas-Widget* (dt. *Leinwand*) ist ein sehr flexibles Widget, in dem beliebige Grafiken dargestellt werden können. Sie können das Canvas-Widget beispielsweise benutzen, um ein Diagramm zu zeichnen oder um ein Bild darzustellen. Im folgenden Beispielprogramm wird das Canvas-Widget verwendet, um einen Kreis und zwei Linien zu zeichnen:

```

cv = tkinter.Canvas(master, width=200, height=200)
cv.pack()

cv.create_oval(50, 50, 150, 150, fill="orange", width=3)
cv.create_line(50, 150, 150, 50, width=3)
cv.create_line(50, 50, 150, 150, width=3)

```

Zunächst wird ein quadratisches Canvas-Widget mit einer Seitenlänge von 200 Pixeln erzeugt. In dieser Zeichenfläche können wir nun über die `create`-Methoden des Canvas-Widgets grundlegende geometrische Formen zeichnen. In diesem Fall verwenden wir die Methoden `create_oval` und `create_line`, um den Kreis bzw. die beiden Linien zu zeichnen.

Die `create`-Methoden bekommen jeweils zwei Koordinatenpaare als erste Parameter übergeben. Diese spezifizieren die Position, an der die Form gezeichnet werden soll. Die Koordinatenangaben beziehen sich auf das lokale Koordinatensystem im Canvas-Widget, dessen Ursprung in der oberen linken Ecke des Widgets liegt. Beachten Sie, dass die positive Y-Achse dieses Koordinatensystems nach unten zeigt. Das Koordinatenpaar (50, 100) bezeichnet also den Punkt, der 50 Pixel rechts und 100 Pixel unterhalb der oberen linken Ecke des Canvas-Widgets liegt.

Die Methode `create_oval` bekommt die obere linke und die untere rechte Ecke des die Ellipse umgebenden Rechtecks übergeben. Dadurch sind Position und Form der Ellipse vollständig beschrieben. Die Methode `create_line` bekommt Start- und Zielpunkt der Linie übergeben.

Zusätzlich können Sie den `create`-Methoden Optionen in Form von Schlüsselwortparametern übergeben, die das Aussehen der gezeichneten Form spezifizieren. In diesem Fall setzen wir die Optionen `fill` für die Füllfarbe und `width` für die Stiftstärke.

Die von diesem Code erzeugte Zeichnung sieht aus Abbildung 24.24:

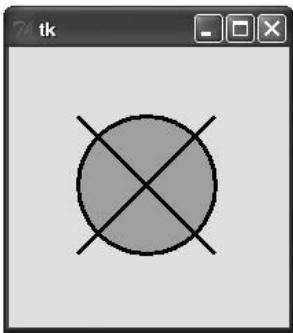


Abbildung 24.24 Eine Zeichnung in einem Canvas-Widget

Im Folgenden möchten wir die `create`-Methoden, die das Canvas-Widget bereitstellt, besprechen und gegebenenfalls ein Beispiel präsentieren.

`create_arc(x1, y1, x2, y2, **kw)`

Die Methode `create_arc` fügt ein Teilstück der durch die Koordinaten `x1, y1, x2` und `y2` festgelegten Ellipse zum Canvas-Widget hinzu. In Form von Schlüsselwortparametern können Optionen übergeben werden. Zwei wichtige Informationen werden durch diese Optionen festgelegt: welches Teilstück gezeichnet werden soll und welche Form es haben soll.

Die Option `style` darf auf einen der Strings `"pieslice"`, `"chord"` und `"arc"` gesetzt werden und legt fest, wie der Ellipsenausschnitt aussehen soll. Die Grafik Abbildung 24.25 zeigt die Resultate dreier Aufrufe von `create_arc`, bei denen lediglich die `style`-Option verändert wurde.



Abbildung 24.25 Von links nach rechts: `pieslice`, `chord` und `arc`

Um festzulegen, welches Teilstück der Ellipse gezeichnet werden soll, sind die Optionen `start` und `extent` notwendig. Die `start`-Option bestimmt den Winkel, ab dem das zu zeichnende Teilstück beginnt. Die `extent`-Option ist ebenfalls ein Winkel und bezeichnet die Ausdehnung des Teilstücks. Beachten Sie, dass die Winkel in Grad gemessen und gegen den Uhrzeigersinn abgetragen werden. Die Optionen `start` und `extent` sind mit `0.0` bzw. `90.0` vorbelegt.

Die folgende Tabelle listet die wichtigsten Optionen auf, die Sie der `create_arc` Methode übergeben können:

Option	Wertebereich	Bedeutung
<code>dash</code>	<code>list</code> , <code>tuple</code>	Legt ein Muster für die Umrandung des Teilstücks fest. Ein Wert von <code>(4,1,6,2)</code> bedeutet beispielsweise, dass nach 4 Pixeln Linie eine Lücke von einem Pixel und nach 6 darauffolgenden Pixeln eine Lücke von 2 Pixeln folgt. Danach beginnt das Muster von vorn.
<code>extent</code>	<code>float</code>	Der Ausdehnungswinkel des Teilstücks. Die Option ist mit <code>90.0</code> vorbelegt.

Tabelle 24.17 Optionen der `create_arc`-Methode

Option	Wertebereich	Bedeutung
fill	"#RRGGBB"	die Füllfarbe des Teilstücks
outline	"#RRGGBB"	die Farbe der Umrandung
start	float	der Winkel, ab dem das Teilstück beginnt
style	"pieslice", "chord", "arc"	Der Typ des Teilstücks. In Abbildung 24.25 sind die drei möglichen Typen dargestellt.
width	float	die Breite der Umrandung

Tabelle 24.17 Optionen der `create_arc`-Methode (Forts.)

Neben den hier vorgestellten Optionen existieren die Optionen `activedash`, `activefill`, `activeoutline`, `activewidth`, `disableddash`, `disabledfill`, `disabledoutline` und `disabledwidth`.

Der Unterschied zu den in der Tabelle vorgestellten Optionen besteht darin, dass diese zusätzlichen Optionen nur unter einer bestimmten Bedingung beachtet werden. So wird die Figur beispielsweise in der für `activefill` angegebenen Farbe gefüllt, wenn sich der Mauszeiger über ihr befindet. Die `disabled`-Optionen werden dann berücksichtigt, wenn das Widget über die `side`-Option des Canvas-Widgets deaktiviert wurde.

`create_image(x, y, **kw)`

Die Methode `create_image` erlaubt es, ein Bild in einem Canvas-Widget darzustellen. Wie das funktioniert, erklärt das folgende Beispielprogramm:

```
cv = tkinter.Canvas(master, width=200, height=200)
cv.pack()

img = tkinter.PhotoImage(file="cover.gif")
cv.create_image(0, 0, image=img, anchor="nw")
```

Zunächst wird mithilfe der `tkinter.PhotoImage`-Klasse das Bild *cover.gif* geladen. Beachten Sie, dass die `PhotoImage`-Klasse ausschließlich die Grafikformate GIF und PPM/PGM unterstützt.⁶ Danach wird die geladene Grafik durch Aufruf der Methode `create_image` an die Stelle (0,0) ins Canvas-Widget gezeichnet. Über die `image`-Option teilen wir beim Aufruf von `create_image` mit, welche Grafik wir zeichnen möchten.

⁶ Wenn Sie weitere Grafikformate benötigen, sollten Sie sich mit der *Python Imaging Library* (PIL) auseinandersetzen, die eine mit `PhotoImage` kompatible Klasse bereitstellt. Die Python Imaging Library ist eine Drittanbieterbibliothek zur Grafikverarbeitung und ist im Internet unter <http://www.pythonware.com> zu finden.

Die zweite angegebene Option `anchor` gibt an, in welcher Beziehung das übergebene Koordinatenpaar zur gezeichneten Grafik stehen soll. Der im Beispiel übergebene Wert `"nw"` bedeutet, dass an der Position (x,y) im lokalen Koordinatensystem des Canvas-Widgets die obere linke Ecke des Bildes liegen soll. Die `anchor`-Option ist mit `"center"` vorbelegt; hätten wir sie also nicht angegeben, läge die Mitte des Bildes im Punkt (x,y) .

Die von obigem Beispielprogramm erzeugte grafische Oberfläche sieht aus wie in Abbildung 24.26.

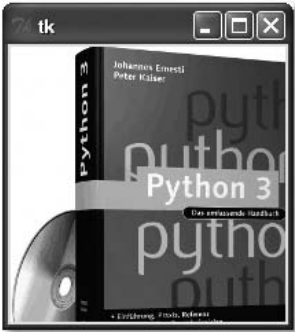


Abbildung 24.26 Eine Grafik im Canvas-Widget

`create_line(*coords, **kw)`

Die Methode `create_line` zeichnet einen Linienzug ins Canvas-Widget. Dazu wird ihr eine beliebige Anzahl Koordinatenpaare als Positionsparameter übergeben. Die Optionen, über die die Eigenschaften des Linienzugs definiert werden können, werden als Schlüsselwortparameter übergeben. Im folgenden Beispiel zeichnen wir das bekannte »Haus vom Nikolaus« mithilfe eines einzigen Aufrufs der `create_line`-Methode:

```
cv = tkinter.Canvas(master, width=100, height=150)
cv.pack()

punkte = (10, 140,    90, 60,    10, 60,
          50, 10,     90, 60,    90, 140,
          10, 140,    10, 60,    90, 140)
cv.create_line(*punkte, width=3)
```

Das Resultat dieses Beispielprogramms zeigt Abbildung 24.27.

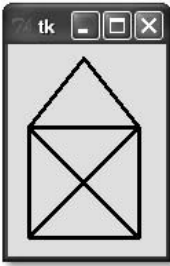


Abbildung 24.27 Das Haus vom Nikolaus

Beachten Sie, dass der mit der Methode `create_line` gezeichnete Linienzug nicht zwangsläufig geschlossen ist, auch wenn das im Beispiel der Fall ist. Dies ist ein Unterscheidungsmerkmal zu der sonst ähnlichen Methode `create_polygon`.

Mit Ausnahme von `extent`, `start` und `style` lassen sich die Optionen von `create_arc` auch bei `create_line` verwenden.

`create_oval(x1, y1, x2, y2, **kw)`

Die Methode `create_oval` zeichnet eine Ellipse. Position und Form der Ellipse werden über die beiden Punkte $(x1, y1)$ und $(x2, y2)$ festgelegt, die den unteren linken und den unteren rechten Eckpunkt des die Ellipse umfassenden Rechtecks angeben. Abbildung 24.28 zeigt eine mit `create_oval` gezeichnete Ellipse samt umfassendem Rechteck und Lage der beiden Eckpunkte:

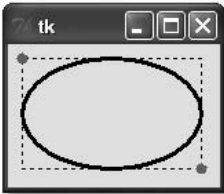


Abbildung 24.28 Die beiden Eckpunkte beschreiben die Ellipse.

Der Methode `create_oval` können Sie dieselben Optionen übergeben wie der Methode `create_line`.

`create_polygon(*coords, **kw)`

Die Methode `create_polygon` zeichnet ein Polygon in das Canvas-Widget. Das Polygon ist durch seine Eckpunkte gegeben, die in Form von beliebig vielen Koordinatenpaaren übergeben werden. Im folgenden Beispielprogramm verwenden wir die Methode `create_polygon` dazu, ein orange gefülltes Dreieck zu zeichnen:

```

cv = tkinter.Canvas(master, width=100, height=100)
cv.pack()

punkte = (10, 10,
          90, 50,
          10, 90)
cv.create_polygon(*punkte, width=3, fill="orange",
                 outline="black")

```

Das resultierende Polygon sieht aus wie in Abbildung 24.29:

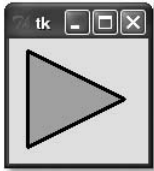


Abbildung 24.29 Ein Dreieck mit `create_polygon`

Sie sehen, dass die übergebenen Punkte miteinander verbunden werden. Zusätzlich wird die Fläche durch eine Verbindung zwischen dem ersten und dem letzten Punkt geschlossen und kann gefüllt werden. Die Methode `create_polygon` ist auch dazu in der Lage, konkave Polygone⁷ zu zeichnen und zu füllen, wie Abbildung 24.30 zeigt:

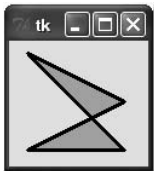


Abbildung 24.30 Ein konkaves Polygon

In diesem Fall wurde die Liste der Punkte aus dem vorangegangenen Beispiel um den Punkt (90,90) erweitert.

`create_rectangle(x1, y1, x2, y2, **kw)`

Die Methode `create_rectangle` zeichnet ein Rechteck in das Canvas-Widget. Das Rechteck wird durch die obere linke Ecke ($x1, y1$) und die untere rechte Ecke ($x2, y2$) festgelegt.

⁷ Ein konkaves Polygon ist ein Polygon, das zwei Punkte enthält, deren Verbindungsline nicht vollständig innerhalb des Polygons verläuft.

Die Verwendung der Methode ist denkbar einfach; der einzige Unterschied zu `create_ellipse` besteht darin, dass nicht die von dem beschriebenen Rechteck eingefasste Ellipse, sondern das Rechteck selbst gezeichnet wird. Auch die möglichen Optionen entsprechen denen der Methode `create_ellipse`. Es sind daher keine weiteren Beispiele vonnöten.

`create_text(x, y, **kw)`

Die Methode `create_text` ermöglicht es, beliebigen Text in ein Canvas-Widget zu schreiben. Standardmäßig wird der Text zentriert an die durch `x` und `y` bestimmte Position geschrieben. Dieses Verhalten lässt sich über die Option `anchor` verändern.

Dies soll anhand des folgenden Beispielprogramms demonstriert werden:

```
cv = tkinter.Canvas(master, width=110, height=100)
cv.pack()

font1 = ("Arial", 12, "italic")
font2 = ("Courier New", 12, "bold italic")
font3 = ("Comic Sans MS", 12, "bold")

cv.create_text(55, 30, text="Hallo Welt", font=font1)
cv.create_text(55, 50, text="Hallo Welt", font=font2)
cv.create_text(55, 70, text="Hallo Welt", font=font3)
```

Nach dem Erzeugen des Canvas-Widgets werden drei Tupel angelegt, die jeweils einen Schrifttyp definieren. Näheres dazu, wie man die auf dem aktuellen System verfügbaren Schriftarten herausfindet, erfahren Sie in Abschnitt 24.2.10 unter »tkinter.font«. An dieser Stelle möchten wir uns damit begnügen, dass das erste Element des Tupels die Schriftart enthält, das zweite die Schriftgröße und im dritten Element weitere Angaben wie Fettdruck (`bold`) oder Kursivschrift (`italic`) stehen können.

Nachdem die Schriftarten spezifiziert wurden, rufen wir die Methode `create_text` drei Mal auf, jeweils mit einer anderen Schriftart. Den zu schreibenden Text geben wir über die Option `text` an und den Schrifttyp über die Option `font`. Das Beispielprogramm erzeugt die grafische Oberfläche aus Abbildung 24.31.



Abbildung 24.31 Text in einem Canvas-Widget

In der folgenden Tabelle möchten wir eine Übersicht über die wichtigsten Optionen geben, die Sie im Zusammenhang mit der `create_text`-Methode verwenden können.

Option	Wertebereich	Bedeutung
anchor	"n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	Gibt an, in welcher Relation die bei <code>create_text</code> angegebene Position zum zu schreibenden Text stehen soll. Standardmäßig ist diese Option mit "center" vorbelegt, das heißt, die Mitte des Textes liegt an der angegebenen Position.
fill	"#RRGGBB"	die Schriftfarbe
font	tuple	die Schriftart
justify	"left", "right", "center"	Gibt bei mehrzeiligen Texten an, wie die einzelnen Zeilen zueinander ausgerichtet werden sollen. Standardmäßig werden sie linksbündig (left) ausgerichtet.
text	str	der darzustellende Text
width	int	Gibt an, wie breit eine Textzeile maximal sein darf. Längere Zeilen werden umbrochen. Dieser Wert wird in Pixeln angegeben.

Tabelle 24.18 Optionen der `create_text`-Methode

`create_window(x, y, **kw)`

Die Methode `create_window` ermöglicht es, Steuerelemente innerhalb eines Canvas-Widgets zu platzieren. Die Verwendung von `create_window` ist denkbar simpel: Das Steuerelement wird an die durch `x` und `y` festgelegte Position gezeichnet, wobei ein eventuell für die Option `anchor` übergebener Wert berücksichtigt wird. Das Steuerelement selbst ist vorher zu instantiieren und wird über die Option `window` angegeben.

Das folgende Beispielprogramm kombiniert eine kleine Zeichnung und ein Steuerelement zu einer bedrohlichen grafischen Oberfläche:

```
cv = tkinter.Canvas(master, width=200, height=100)
cv.pack()

cv.create_oval(10, 10, 190, 90, fill="red", width=3)
b = tkinter.Button(None, text="Selbstzerstörung",
                    background="red", activebackground="red",
                    foreground="white", activeforeground="white")
cv.create_window(100, 50, window=b)
```

Der Quellcode sollte selbsterklärend sein und erzeugt die GUI aus Abbildung 24.32.



Abbildung 24.32 Ein in ein Canvas-Widget eingebetteter Button

Nun ja, Sie müssen den Button ja nicht betätigen.

Zum Abschluss möchten wir noch einige Bemerkungen dazu machen, wie Sie eine einmal angefertigte Zeichnung wieder modifizieren können. Dazu ist zu sagen, dass jede `create`-Methode einen ganzzahligen Index zurückliefert, über den Sie nachträglich auf das erstellte Element zugreifen können.

Es gibt einige wesentliche Methoden, mit deren Hilfe sich ein einmal gezeichnetes Element modifizieren lässt:

Methode	Bedeutung
<code>delete(index)</code>	Löscht das Element mit dem Index <code>index</code> aus dem Canvas-Widget.
<code>itemconfig(index, **options)</code>	Ermöglicht das Anpassen der Optionen des Elements mit dem Index <code>index</code> .
<code>move(index, dx, dy)</code>	Verschiebt das Element mit dem Index <code>index</code> um <code>dx</code> Pixel nach rechts und um <code>dy</code> Pixel nach unten.

Tabelle 24.19 Methoden zur Manipulation von Elementen einer Zeichnung

Damit beenden wir die Einführung in die von `tkinter` bereitgestellten Steuerelemente. Die hier vorgestellten Auflistungen, seien es Optionen oder Methoden, sind nicht vollständig, denn eine vollständige Vorstellung der Steuerelemente würde den Rahmen dieser Einführung sprengen. Dennoch haben Sie in diesem Abschnitt einen umfassenden Überblick über die Möglichkeiten bekommen, die die Steuerelemente des Tk-Frameworks bieten.

24.2.9 Die Klasse Tk

Die Klasse `Tk`, die in jeder `tkinter`-Anwendung instantiiert wird, repräsentiert ein sogenanntes Top-Level-Fenster, also ein Widget, das kein übergeordnetes

Widget besitzt. Aus diesem Grund kommt der Tk-Instanz eine gewisse Bedeutung zu, die in diesem Abschnitt besprochen werden soll.

Die Tk-Instanz stellt einige Methoden zur Kommunikation mit dem *Window-Manager* des Betriebssystems bereit. Diese Methoden erlauben es beispielsweise, den Namen der Anwendung in der Titelleiste oder die Fensterposition zu ändern. In der folgenden Tabelle werden die wichtigsten dieser *wm*-Methoden aufgelistet und erklärt.

Methode	Bedeutung
<code>wm_geometry([geom])</code>	Setzt die Position und die Größe des Fensters. Der Parameter <code>geom</code> muss ein String der Form "widthxheight+x+y" sein, wobei <code>width</code> und <code>height</code> der gewünschten Breite und Höhe entsprechen. Die Angaben <code>x</code> und <code>y</code> bestimmen die Position des Fensters relativ zur linken oberen Ecke des Bildschirms. Wenn <code>geom</code> nicht übergeben wird, werden die aktuelle Dimension und Position des Fensters in Form des oben beschriebenen Strings zurückgegeben.
<code>wm_iconify()</code>	Minimiert das Fenster in die Taskleiste des Betriebssystems.
<code>wm_deiconify()</code>	Holt das minimierte Fenster wieder hervor. Wenn das Fenster nicht minimiert war, passiert nichts.
<code>wm_resizable([width[, height]])</code>	Legt fest, ob der Benutzer die Fenstergröße in horizontaler (<code>width</code>) oder vertikaler (<code>height</code>) Richtung verändern darf. Wenn die Parameter <code>width</code> und <code>height</code> nicht angegeben werden, wird das aktuelle Verhalten in Form eines zweielementigen Tupels zurückgegeben.
<code>wm_title([title])</code>	Setzt den Fenstertitel auf <code>title</code> . Wenn <code>title</code> nicht angegeben wurde, wird der aktuelle Fenstertitel zurückgegeben.

Tabelle 24.20 Methoden zur Kommunikation mit dem Window-Manager

Interessant ist auch, dass die Tk- und die Widget-Klasse eine gemeinsame Basis-klassse haben, weswegen die in Abschnitt 24.2.8, in den Erläuterungen zur Widget-Klasse, eingeführten *winfo*-Methoden auch bei jeder Tk-Instanz verfügbar sind.

24.2.10 Weitere Module

Zum Schluss möchten wir noch einen Überblick über die im `tkinter`-Paket enthaltenen Module geben, da diese einige durchaus interessante Bereiche abdecken und in mancher Hinsicht die Arbeit mit dem Tk-Framework erleichtern.

`tkinter.scrolledtext`

Im Abschnitt über das Text-Widget haben wir gesagt, dass es über die Option `yscrollcommand` möglich ist, ein Text-Widget mit einer vertikalen Scrollbar auszustatten. Da aber eine solche vertikale Scrollbar bei einem Text-Widget in der Regel erwünscht ist, wäre es umständlich, jedes Mal den Code zum Instantiieren und Anbinden der Scrollbar schreiben zu müssen.

Für diesen Zweck existiert das Modul `scrolledtext` im Paket `tkinter`, das das Widget `ScrolledText` bereitstellt. Dieses Widget verhält sich wie ein Text-Widget, ist aber standardmäßig mit einer vertikalen Scrollbar ausgestattet, so dass sich der Programmierer um diese nicht mehr zu kümmern braucht.

Das folgende Beispielprogramm zeigt die Verwendung des `ScrolledText`-Widgets:

```
import tkinter, tkinter.scrolledtext
master = tkinter.Tk()

st = tkinter.scrolledtext.ScrolledText(master)
st.pack()

master.mainloop()
```

Erwartungsgemäß sieht die grafische Oberfläche, die dieses Programm anzeigt, aus wie in Abbildung 24.33:

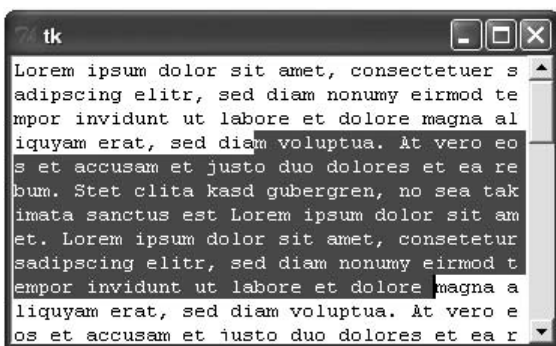


Abbildung 24.33 Das `ScrolledText`-Widget

Intern besteht das `ScrolledText`-Widget aus einem `Text`-Widget, einer Scrollbar und einem `Frame`-Widget, das die beiden anderen Widgets umfasst. Über die Attribute `vbar` und `frame` des `ScrolledText`-Widgets können Sie auf die Scrollbar und das `Frame`-Widget zugreifen.

Beachten Sie, dass das `ScrolledText`-Widget standardmäßig keine horizontale Scrollbar mitbringt. Diese müssen Sie genau wie beim `Text`-Widget selbst erzeugen und anbinden.

tkinter.colorchooser

Stellen Sie sich vor, Sie haben ein Programm geschrieben, in dem Sie ein Diagramm zeichnen, und möchten das Programm konfigurierbar gestalten. Der Benutzer soll auswählen können, in welchen Farben das Diagramm dargestellt wird. Jetzt kommt das Problem auf Sie zu, in welcher Form der Benutzer die gewünschte Farbe eingeben soll. Ideal wäre ein Dialog, in dem der Benutzer die darstellbaren Farben sieht und seine favorisierte Farbe einfach anklickt.

Für solche und ähnliche Probleme gibt es sogenannte *Standarddialoge*, die für bestimmte Fragen an den Benutzer gemacht sind, die immer wieder gestellt werden. Solche Standarddialoge haben für den Programmierer den Vorteil, dass er keine eigenen kreieren muss. Für den Benutzer liegt der Vorteil darin, dass er sich nicht ständig mit verschiedenen grafischen Oberflächen für dieselbe Fragestellung konfrontiert sieht, sondern immer denselben vertrauten Dialog vorfindet.

Auch im Tk-Framework ist es möglich, die Standarddialoge des Betriebssystems bzw. der Desktop-Umgebung zu nutzen. Module dafür werden wir in diesem und den folgenden Abschnitten besprechen.

Das Modul `colorchooser` des Pakets `tkinter` bietet einen Standarddialog an, der den Benutzer zur Eingabe einer Farbe auffordert. Dazu enthält das Modul im Wesentlichen die Funktion `askcolor`, die einen Dialog zur Farbauswahl anzeigt und die vom Benutzer eingestellte Farbe zurückgibt.

Das folgende Beispielprogramm demonstriert die Verwendung der Funktion `askcolor`:

```
import tkinter, tkinter.colorchooser

master = tkinter.Tk()
cv = tkinter.Canvas(master, width=100, height=40)
cv.pack()

color = "#FF0000"
colorFrame = cv.create_rectangle(0, 0, 100, 40,
                                fill=color, width=0)
```

```
def changeColor():
    global color, colorFrame
    res = tkinter.colorchooser.askcolor(color)
    if res[1]:
        color = res[1]
        cv.itemconfig(colorFrame, fill=color)

b = tkinter.Button(master, text="Farbe ändern",
                    command=changeColor)

b.pack()
master.mainloop()
```

Zunächst wird ein Canvas-Widget erstellt und die globale Variable `color` für die aktuell ausgewählte Farbe angelegt, die wir initial auf Rot setzen. Dann zeichnen wir ein Rechteck in das Canvas-Widget und merken uns den von der Methode `create_rectangle` zurückgegebenen Index.

Danach folgt die Definition der Funktion `changeColor`, die als Handler-Funktion des am Ende des Programms erzeugten Buttons dient. Wenn der Benutzer den Button FARBE ÄNDERN betätigt, wird die Funktion `changeColor` vom Tk-Framework gerufen.

In der Funktion `changeColor` rufen wir die angesprochene Funktion `askcolor` des `colorchooser`-Moduls auf, um den Standarddialog zur Auswahl einer Farbe auf den Bildschirm zu bringen. Dabei übergeben wir die bisher gewählte Farbe, die dann im Auswahldialog voreingestellt ist.

Das Resultat des Funktionsaufrufs ist ein Tupel mit zwei Elementen. Das erste Element ist wieder ein Tupel, das die Zahlenwerte der einzelnen Farbkomponenten enthält. Das zweite und für uns wichtigere Element ist ein String der Form `"#RRGGBB"`, wie wir ihn bereits des Öfteren angesprochen haben. Einen solchen String können wir direkt weiterverarbeiten. In diesem Fall interessieren wir uns also nur für das zweite Element des resultierenden Tupels.

Wenn der Benutzer den Auswahldialog mit ABBRECHEN beendet hat, wird das Tupel `(None, None)` zurückgegeben. Diesen Fehlerfall behandeln wir nach dem Aufruf von `askcolor`. Wenn der Benutzer eine gültige Farbe ausgewählt hat, passen wir die globale Referenz `color` an und ändern die Farbe des Rechtecks im Canvas-Widget. Dies ist uns möglich, da wir uns den von der Funktion `create_rectangle` zurückgegebenen Index gemerkt haben.

Die grafische Oberfläche des Beispielsprogramms inklusive des Standarddialogs ist in der Abbildung 24.34 zu sehen.

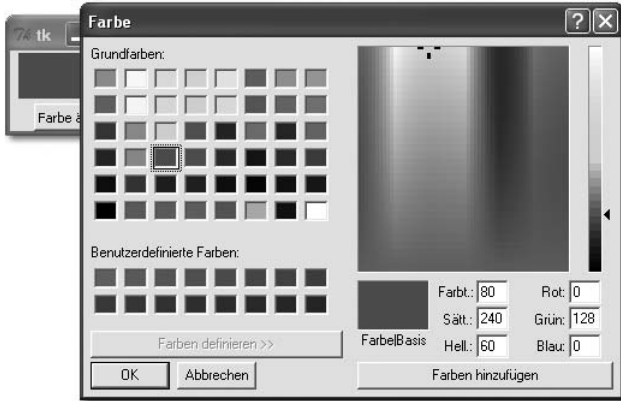


Abbildung 24.34 Der Standarddialog zur Farbauswahl

Beachten Sie, dass andere Betriebssysteme auch ein anderes Layout für diesen Standarddialog verwenden.

tkinter.filedialog

Eine wichtige Klasse von Standarddialogen sind Dateidialoge, die den Benutzer dazu auffordern, Dateien oder Ordner von der Festplatte auszuwählen. Sei es, um sie in das Programm zu laden oder gewisse Inhalte zu speichern. Dateidialoge werden ständig benötigt.

Das Modul `filedialog` des Pakets `tkinter` stellt vorgefertigte Dateidialoge für alle Lebenslagen bereit. In der Regel genügt ein Funktionsaufruf, um den Dialog in die eigene Anwendung zu integrieren. Im Folgenden besprechen wir die vom Modul `filedialog` bereitgestellten Funktionen.

askdirectory(**options)

Die Funktion `askdirectory` zeigt einen Standarddialog an, der den Benutzer dazu auffordert, ein Verzeichnis auszuwählen. Der Pfad des vom Benutzer gewählten Verzeichnisses wird als String zurückgegeben.

In Form von Schlüsselwortparametern können Sie einige Optionen übergeben, die das Verhalten des Dialogs steuern. Die wichtigsten dieser Optionen möchten wir in der folgenden Tabelle vorstellen:

Option	Wertebereich	Bedeutung
<code>initialdir</code>	<code>str</code>	Das Verzeichnis, das im Dialog voreingestellt ist. Standardmäßig ist das Arbeitsverzeichnis des Programms vorgegeben.
<code>mustexist</code>	<code>bool</code>	Gibt an, ob das vom Benutzer ausgewählte Verzeichnis existieren muss oder nicht.
<code>title</code>	<code>str</code>	der Dialogtitel

Tabelle 24.21 Optionen der `askdirectory`-Funktion

askopenfilename(options)**

Öffnet einen Dialog, der den Benutzer dazu auffordert, eine Datei auszuwählen. Der Pfad zu dieser Datei wird in Form eines Strings zurückgegeben.

Die folgende Tabelle listet die wichtigsten Optionen auf, die Sie der Funktion `askopenfilename` in Form von Schlüsselwortparametern übergeben können:

Option	Wertebereich	Bedeutung
<code>defaulttextextension</code>	<code>str</code>	ein String, der an den eingegebenen Dateinamen angehängt wird, wenn er keine Dateierweiterung hat.
<code>filetypes</code>	<code>list, tuple</code>	Ein Tupel der Form <code>(("Textdateien", ".txt"), ("Alle Dateien", "*"))</code> . Der Benutzer kann die hier angegebenen Dateierweiterungen aus einer Liste im Dialog auswählen. Es werden dann nur noch die Dateien des aktuellen Verzeichnisses angezeigt, die dem ausgewählten Typ entsprechen.
<code>initialdir</code>	<code>str</code>	wie bei <code>askdirectory</code>
<code>initialfile</code>	<code>str</code>	Gibt an, welche Datei im Dialog voreingestellt sein soll. Der Pfad zu dieser Datei muss entweder relativ zu <code>initialdir</code> oder absolut sein.
<code>multiple</code>	<code>bool</code>	Gibt an, ob es dem Benutzer erlaubt sein soll, mehrere Dateien auszuwählen. In diesem Fall wird von <code>askopenfilename</code> ein Tupel mit den Pfaden aller ausgewählten Dateien zurückgegeben.
<code>title</code>	<code>str</code>	Gibt den Dialogtitel an.

Tabelle 24.22 Optionen der `askopenfilename`-Funktion

Analog zur Funktion `askopenfilename` existiert die Funktion `askopenfile` im Modul `filedialog`. Diese Funktion verhält sich wie `askopenfile`, mit dem Unterschied, dass die ausgewählte Datei geöffnet und das geöffnete Dateiojekt zurückgegeben wird. Dazu übergeben Sie der Funktion `askopenfile` optional einen Positionsparameter, der den Modus angibt, in dem die Datei geöffnet werden soll. Standardmäßig wird die Datei zum Lesen geöffnet.

Außerdem existieren die Funktionen `askopenfilenames` und `askopenfiles`, die sich wie die Funktionen `askopenfilename` bzw. `askopenfile` verhalten, wenn die Option `multiple` auf `True` gesetzt wurde.

`asksaveasfilename(**options)`

Die Funktion `asksaveasfilename` zeigt einen Dialog an der grafischen Oberfläche an, der den Benutzer dazu auffordert, eine Datei zum Speichern eines bestimmten Inhalts auszuwählen. Im Gegensatz zur Funktion `askopenfilename` muss die ausgewählte Datei nicht unbedingt existieren.

Die Optionen `defaultextension`, `filetypes`, `initialdir`, `initialfile` und `title` können auch bei `asksaveasfilename` verwendet werden und haben dieselbe Bedeutung wie bei `askopenfilename`. Zudem existiert die Funktion `asksaveasfile`, die die vom Benutzer ausgewählte Datei öffnet und das resultierende Dateiojekt zurückgibt. Der Modus, in dem die Datei geöffnet werden soll, kann der Funktion `asksaveasfile` als einziger Positionsparameter übergeben werden. Standardmäßig wird die Datei zum Schreiben geöffnet.

`tkinter.font`

In Abschnitt 24.2.8, in den Erläuterungen zum Canvas-Widget, haben Sie gesehen, wie Sie einen beliebigen Text in das Canvas-Widget zeichnen können. Dabei konnten Sie über eine Option eine Schriftart angeben, die zum Schreiben des Textes verwendet werden soll. Zu diesem Zeitpunkt haben wir uns einiger Standardschriftarten bedient, die auf jedem System verfügbar sind. Es fehlte eine Möglichkeit, herauszufinden, auf welche Schriftarten wir tatsächlich zurückgreifen können.

Das Modul `font` des Pakets `tkinter` bietet in Form der Funktion `families` eine Lösung für dieses Problem. Diese Funktion gibt ein Tupel zurück, das die Namen aller Schriftarten enthält, die Sie beispielsweise im Zusammenhang mit der `font`-Option bei einem Aufruf der Methode `create_text` des Canvas-Widgets verwenden können. Der Funktion `tkinter.font.families` brauchen keine Parameter übergeben zu werden.

Abgesehen von dieser Funktion enthält das `font`-Modul noch eine Klasse `Font`, deren Beschreibung außerhalb des Fokus dieser Einführung liegt. Nähere Informationen dazu finden Sie beispielsweise in der interaktiven Hilfe zu `tkinter.font`.

tkinter.messagebox

Das Modul `messagebox` des Pakets `tkinter` ermöglicht es, durch einen einfachen Funktionsaufruf sogenannte Message Boxes anzuzeigen. Eine *Message Box* ist ein kleines Fenster mit einer Botschaft an den Benutzer. Eine Message Box kann dazu verwendet werden, den Benutzer über einen Fehler zu informieren oder ihm eine Frage zu stellen. Es gibt mehrere Typen von Message Boxes, beispielsweise einen, der zusätzlich zur Botschaft ein entsprechendes Icon für eine Fehlermeldung anzeigt, oder einen, der die beiden Buttons »Ja« und »Nein« anbietet, über die der Benutzer eine in der Nachricht gestellte Frage beantworten kann.

Im Folgenden werden wie die im Modul `messagebox` enthaltenen Funktionen besprochen. Alle besprochenen Funktionen verfügen über dieselbe Schnittstelle, weswegen wir nur bei der ersten Funktion darauf eingehen werden.

askokcancel([title[, message]][, **options])

Die Funktion `askokcancel` zeigt eine Message Box mit dem Fenstertitel *title* und der Botschaft *message* an. Die Message Box kann über die Buttons »OK« und »Abbrechen« geschlossen werden. Welchen der Buttons der Benutzer betätigt hat, erfahren Sie über den Rückgabewert der Funktion. Die Funktion gibt `True` zurück, wenn der Benutzer die Message Box über den Button »OK« geschlossen hat, und `False` bei »Abbrechen«.



Abbildung 24.35 Eine mit `askokcancel` erzeugte Message Box

Der Funktion `askokcancel` können Sie einige Optionen in Form von Schlüsselwortparametern übergeben. Die folgende Tabelle listet die wichtigsten dieser Optionen auf:

Option	Wertebereich	Bedeutung
default	str	Legt den sogenannten Default-Button der Message Box fest. Dieser Button wird grafisch hervorgehoben und kann über die Leertaste betätigt werden. Im Falle der askokcancel-Funktion sind die Werte "ok" und "cancel" für die default-Option möglich.
detail	str	Für die detail-Option kann ein Text angegeben werden, der nähere Angaben zur Botschaft der Message Box enthält. Dieser Text wird unter die Botschaft geschrieben. Auf einigen Systemen wird die Botschaft grafisch vom detail-Text getrennt.
icon	"error", "info", "question", "warning"	Legt das Icon fest, das auf der linken Seite der Message Box angezeigt wird.

Tabelle 24.23 Optionen der askokcancel-Funktion

Diese Optionen sind auch für die folgenden Funktionen gültig.

askquestion([title[, message]][, **options])

Die Funktion askquestion zeigt eine Message Box an, in der dem Benutzer eine Frage gestellt wird. Diese kann der Benutzer über die Buttons »Ja« und »Nein« beantworten. Je nach Antwort des Benutzers gibt die Funktion askquestion "yes" bzw. "no" zurück.

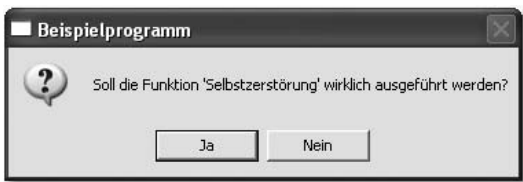


Abbildung 24.36 Eine mit askquestion erzeugte Message Box

askretrycancel([title[, message]][, **options])

Die Funktion askretrycancel erzeugt eine Message Box, die den Benutzer über das Fehlschlagen einer Operation informiert. Die Message Box gibt dem Benutzer die Möglichkeit, über die Buttons »Wiederholen« und »Abbrechen« auf diese Situation zu reagieren. Je nach Wahl des Benutzers gibt die Funktion True (»Wiederholen«) oder False (»Abbrechen«) zurück.

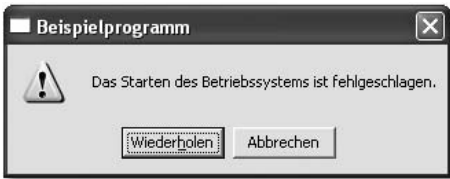


Abbildung 24.37 Eine mit askretrycancel erzeugte Message Box

askyesno([title[, message]][, **options])

Die Funktion `askyesno` verhält sich wie `askquestion`, aber mit dem Unterschied, dass sie `True` zurückgibt, wenn der Benutzer die Message Box mit »Ja« schließt, und im anderen Fall `False`. Die Funktion `askquestion` gibt die Strings "yes" bzw. "no" zurück.

askyesnocancel([title[, message]][, **options])

Die Funktion `askyesnocancel` zeigt eine Message Box an, die über die Buttons »Ja«, »Nein« und »Abbrechen« verfügt. Die Funktion gibt `True` zurück, wenn der Benutzer die Message Box über »Ja« geschlossen hat, `False`, wenn der Benutzer »Nein« verwendete, und `None` im Falle von »Abbrechen«.

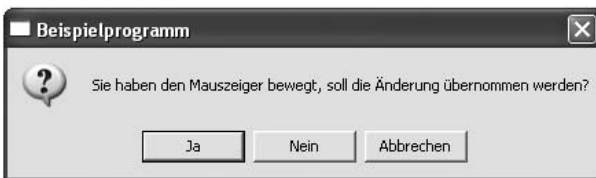


Abbildung 24.38 Eine von askyesnocancel erzeugte Message Box

showerror([title[, message]][, **options])

Die Funktion `showerror` zeigt eine Message Box an, die den Benutzer über einen Fehler informiert. Die Message Box hat ein entsprechendes Symbol auf der linken Seite und kann nur über den Button »OK« geschlossen werden.

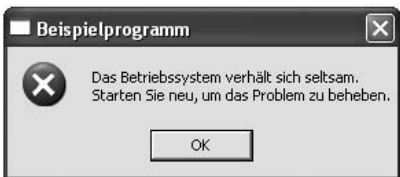


Abbildung 24.39 Eine mit showerror erzeugte Message Box

showinfo([title[, message]][, **options])

Die Funktion `showinfo` erzeugt eine Message Box zur Information des Benutzers. Die Message Box hat ein entsprechendes Symbol auf der linken Seite und kann nur über den Button »OK« geschlossen werden.



Abbildung 24.40 Eine mit `showinfo` erzeugte Message Box

showwarning([title[, message]][, **options])

Die Funktion `showwarning` erzeugt eine Message Box zur Warnung des Benutzers. Die Message Box hat ein entsprechendes Symbol auf der linken Seite und kann nur über den Button »OK« geschlossen werden.

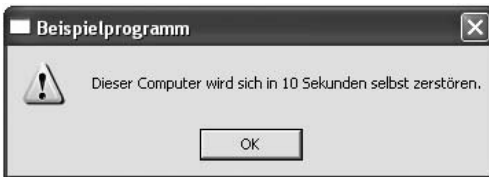


Abbildung 24.41 Eine mit `showwarning` erzeugte Message Box

tkinter.simpledialog

Das Modul `simpledialog` des Pakets `tkinter` stellt weitere Standarddialoge bereit, mit deren Hilfe sich Werte vom Benutzer einlesen lassen. Dazu enthält das Modul im Wesentlichen drei Funktionen, die wir im Folgenden besprechen werden.

askfloat(title, prompt, **kw)

Die Funktion `askfloat` des Moduls `simpledialog` erzeugt einen Dialog, der den Benutzer zur Eingabe einer Gleitkommazahl auffordert. Dazu müssen Sie der Funktion den gewünschten Dialogtitel (*title*) und einen erklärenden Text für den Benutzer (*prompt*) angeben.

Zusätzlich können Sie einige Optionen in Form von Schlüsselwortparametern übergeben. Die wichtigsten dieser Optionen möchten wir in der folgenden Tabelle vorstellen.

Option	Wertebereich	Bedeutung
initialvalue	float	Gibt den Wert an, der im Dialog voreingestellt sein soll.
maxvalue	float	Gibt den größten Wert an, den der Benutzer eingeben darf.
minvalue	float	Gibt den kleinsten Wert an, den der Benutzer eingeben darf.

Tabelle 24.24 Optionen der askfloat-Funktion

Der von der askfloat-Funktion erzeugte Dialog ist in Abbildung 24.42 zu sehen.

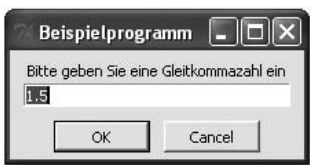


Abbildung 24.42 Ein mit askfloat erzeugter Dialog

Wenn der Benutzer den Dialog mit »OK« schließt, gibt die Funktion askfloat die eingestellte Zahl zurück. Schließt der Benutzer den Dialog über »Abbrechen«, gibt askfloat None zurück.

askinteger(title, prompt, **kw)

Die Funktion askinteger zeigt einen Dialog an, der den Benutzer zur Eingabe einer ganzen Zahl auffordert. Die Schnittstelle ist mit der von askfloat bis auf den Datentyp einiger Optionen und den Rückgabebetyp identisch.

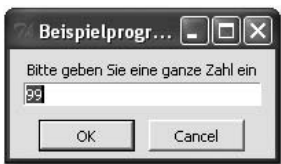


Abbildung 24.43 Ein mit askinteger erzeugter Dialog

askstring(title, prompt, **kw)

Die Funktion askstring erzeugt einen Dialog, der den Benutzer zur Eingabe eines Strings auffordert.

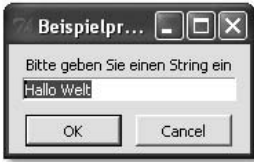


Abbildung 24.44 Ein mit `askstring` erzeugter Dialog

Die Optionen `minvalue` und `maxvalue` verstehen sich wie bei `askinteger` und `askfloat`. Die Strings werden dann nach der in Python geltenden Ordnungsrelation für Strings verglichen.

*»Wenn die Sprache nicht stimmt,
ist das, was gesagt wird,
nicht das, was gemeint ist.«
– Konfuzius*

25 Anbindung an andere Programmiersprachen

Dieser Abschnitt beschäftigt sich mit der Interoperabilität zwischen Python und anderen Programmiersprachen, hier ausschließlich C. Grundsätzlich gibt es dabei zwei Strategien, ein Programm zu schreiben, das Python mit einer anderen Sprache kombiniert:

1. In einem Python-Programm soll C-Code ausgeführt werden.
2. In einem C-Programm soll ein Python-Script ausgeführt werden.

Zu 1.: In einem größeren Projekt kann Python als sehr komfortable und gut zu wartende Sprache beispielsweise für die Programmlogik eingesetzt werden, während man einige wenige zeitkritische Algorithmen des Projekts aus Effizienzgründen in einer nicht interpretierten Sprache wie C oder C++ schreibt. Zu diesem Ansatz besprechen wir im ersten Abschnitt, wie Sie mit dem Modul `ctypes` der Standardbibliothek auf dynamische Bibliotheken, beispielsweise Windows-DLLs, zugreifen können. Der zweite Abschnitt zeigt Möglichkeiten auf, C- oder C++-Code direkt in den Python-Quelltext einzubetten.

Zu 2.: Häufig möchte man auch den umgekehrten Weg beschreiten und in einem größeren C/C++-Projekt Python als eingebettete Scriptsprache für dynamische Elemente des Programms verwenden. In einem Computerspiel könnte beispielsweise C/C++ für die hauptsächlich laufzeitkritische Hauptanwendung und die gesamte Algorithmik verwendet werden, während Python für die dynamischen Elemente des Spiels, beispielsweise Ereignisse in bestimmten Levels oder das Verhalten verschiedener Spielfiguren, genutzt wird. Dieser Ansatz soll in Abschnitt 25.3, »Python als eingebettete Scriptsprache«, verfolgt werden.

Grundsätzlich benötigen Sie zum Verständnis der folgenden Kapitel zumindest rudimentäre Kenntnisse der Programmiersprache C.

25.1 Dynamisch ladbare Bibliotheken – ctypes

Mit dem Modul `ctypes` ist es möglich, Funktionen einer sogenannten *dynamisch ladbaren Bibliothek*, im Folgenden *dynamische Bibliothek* genannt, aufzurufen. Zu solchen dynamischen Bibliotheken zählen beispielsweise DLL-Dateien (*Dynamic Link Library*) unter Windows oder SO-Dateien (Shared Object) unter Linux bzw. Unix.

Das Aufrufen von Funktionen einer dynamischen Bibliothek ist besonders dann sinnvoll, wenn bestimmte lauffzeitkritische Teile eines Python-Programms in einer hardwarenäheren und damit effizienteren Programmiersprache geschrieben werden sollen oder wenn Sie schlicht eine in C oder C++ geschriebene Bibliothek in Python nutzen möchten.

Beachten Sie grundsätzlich, dass das Erstellen einer dynamischen Bibliothek keine Eigenschaft der Programmiersprache C ist. Im Gegenteil: Eine dynamische Bibliothek kann als eine sprachunabhängige Schnittstelle zwischen verschiedenen Programmen betrachtet werden. Es ist absolut möglich, ein Python-Programm zu schreiben, das auf eine in C geschriebene dynamische Bibliothek zugreift, die ihrerseits auf eine dynamische Bibliothek zugreift, die in Pascal geschrieben wurde. Dies gilt allerdings nur für Sprachen, die sich zu einer dynamischen Bibliothek kompilieren lassen – PHP bliebe beispielsweise außen vor.

25.1.1 Ein einfaches Beispiel

Zum Einstieg in das Modul `ctypes` dient ein einfaches Beispiel. Im Beispielprogramm soll die dynamische Bibliothek der *C Runtime Library* eingebunden und die darin enthaltene Funktion `printf` dazu genutzt werden, den Text »Hallo Welt« auszugeben. Die C Runtime Library ist unter Windows unter dem Namen *msvcrt.dll* und unter Unix-ähnlichen Systemen unter dem Namen *libc.so.6* zu finden. Dazu betrachten wir zunächst den Quellcode des Beispielprogramms:

```
import ctypes
bibliothek = ctypes.CDLL("MSVCRT")
bibliothek.printf(b"Hallo Welt\n")
```

Zunächst wird das Modul `ctypes` eingebunden und dann eine Instanz der Klasse `CDLL`¹ erzeugt. Eine Instanz dieser Klasse repräsentiert eine geladene dynamische Bibliothek. Beachten Sie, dass die Dateierdung *.dll* unter Windows weggelassen

1 Die Klasse `CDLL` ist nur für Bibliotheken zuständig, die der `cdecl`-Aufrufkonvention genügen. Gerade Bibliotheken der Windows API verwenden jedoch die `stdcall`-Konvention. Zum Laden solcher Bibliotheken existiert unter Windows die Klasse `WinDLL` im `ctypes`-Modul.

werden muss, wenn eine System-Bibliothek geladen werden soll. Unter Linux sähe die Instanziierung der Klasse `CDLL` folgendermaßen aus:

```
bibliothek = ctypes.CDLL("libc.so.6")
```

Nachdem die dynamische Bibliothek eingebunden worden ist, kann die Funktion `printf` wie eine Methode der `CDLL`-Instanz aufgerufen werden.²

Beachten Sie, dass die normale `printf`-Funktion einen `bytes`-String erwartet. Um mit richtigen Strings zu arbeiten, kann ihr Pendant `wprintf` gerufen werden:

```
bibliothek.wprintf("Hallo Welt\n")
```

Behalten Sie bei der Arbeit mit `ctypes` aber immer im Hinterkopf, dass es teilweise große Unterschiede zwischen den Datentypen von C/C++ und denen von Python gibt. Es können also nicht alle Funktionen so einfach verwendet werden. Im Laufe dieses Abschnitts werden wir noch eingehend darauf zurückkommen.

Achtung

Häufig verlangen C-Funktionen einen Pointer auf einen String als Parameter, über den der String dann manipuliert wird. Beachten Sie unbedingt, dass Sie dann keine Instanzen des Python-Datentypen `str` oder `bytes` übergeben dürfen. Das liegt daran, dass diese zu den unveränderlichen Datentypen gehören und auch von einer C-Funktion nicht neu beschrieben werden können.

Um einer solchen Funktion dennoch einen beschreibbaren String zur Verfügung zu stellen, dient die Funktion `create_string_buffer`, die wir zusammen mit den anderen Funktionen des Moduls `ctypes` gegen Ende dieses Abschnitts besprechen werden.

25.1.2 Die eigene Bibliothek

An dieser Stelle soll eine dynamische Bibliothek erstellt werden, auf die wir dann in den folgenden Abschnitten zugreifen werden. Die Bibliothek ist in C geschrieben und enthält drei Funktionen mit unterschiedlich komplexer Schnittstelle. Wir werden hier nur den Quelltext der drei Funktionen zeigen, auf die wir uns in den folgenden Beispielen beziehen. Es wird keine Anleitung geben, wie Sie den C-Code zu einer dynamischen Bibliothek kompilieren können. Das ist nicht Gegenstand dieses Buches und würde sich zudem bei den verschiedensten Betriebssystemen und Entwicklungsumgebungen zum Teil stark unterscheiden. Sie finden dazu nach einer kurzen Recherche im Internet genügend Anleitungen.

² Beachten Sie, dass die Funktion `printf` nicht nach `sys.stdout`, sondern in den tatsächlichen `stdout`-Stream des Betriebssystems schreibt. Das bedeutet für Sie, dass Sie im obigen Beispiel nur dann eine Ausgabe sehen, wenn Sie das Programm in einer Python-Shell ausführen. Keine Ausgabe erscheint dagegen beispielsweise in IDLE.

```

// Berechnet die Fakultät einer ganzen Zahl
int fakultaet(int n)
{
    int i;
    int ret = 1;
    for(i = 1; i <= n; i++)
        ret *= i;
    return ret;
}

// Berechnet die Länge eines Vektors im R3
double veclen(double x, double y, double z)
{
    return sqrt(x*x + y*y + z*z);
}

// Bubblesort
void sortiere(int *array, int len)
{
    int i, j, tmp;
    for(i = 0; i < len; i++)
    {
        for(j = 0; j < i; j++)
        {
            if(array[j] > array[i])
            {
                tmp = array[j];
                array[j] = array[i];
                array[i] = tmp;
            }
        }
    }
}

```

Die erste Funktion, `fakultaet`, berechnet die Fakultät einer ganzen Zahl und gibt das Ergebnis ebenfalls in Form einer ganzen Zahl zurück. Die zweite Funktion, `veclen`, errechnet die Länge eines dreidimensionalen Vektors. Sie bekommt dazu die drei Koordinaten des Vektors in Form von drei Gleitkommazahlen übergeben und gibt die Länge des Vektors ebenfalls in Form einer Gleitkommazahl zurück.

Die dritte, etwas komplexere Funktion, `sortiere`, implementiert den sogenannten *Bubblesort*-Algorithmus, um ein Array von beliebig vielen ganzen Zahlen aufsteigend zu sortieren. Dazu übergeben wir der Funktion einen Pointer auf das erste Element sowie die Anzahl der Elemente des Arrays.

Im Folgenden gehen wir davon aus, dass der oben stehende Quellcode zu einer dynamischen Bibliothek kompiliert wurde und unter dem Namen *bibliothek.dll* im jeweiligen Programmverzeichnis der kommenden Beispielpprogramme zu finden ist.

Datentypen

An dieser Stelle haben wir eine fertige dynamische Bibliothek mit drei Funktionen, die wir jetzt mit `ctypes` aus einem Python-Programm heraus aufrufen können. Der praktischen Umsetzung dieses Vorhabens stehen jedoch die teilweise inkompatiblen Datentypen von C und Python im Wege. Solange Instanzen der Datentypen `int`, `str`, `bytes` oder `NoneType`³ übergeben werden, funktioniert der Funktionsaufruf einwandfrei, denn diese Instanzen können eins zu eins nach C übertragen werden. So ist beispielsweise der Aufruf der Funktion `fakultaet` mit keinerlei Problemen behaftet:

```
from ctypes import CDLL
bib = CDLL("bibliothek.dll")
print(bib.fakultaet(5))
```

Doch bereits das Übergeben einer `float`-Instanz scheitert. Für diesen und andere Datentypen von C implementiert das Modul `ctypes` entsprechende Gegenstücke in Python, deren Instanzen über die Schnittstelle geschickt werden können. Die folgende Tabelle listet alle in `ctypes` enthaltenen Datentypen sowie ihre Entsprechungen in C und Python auf.

Datentyp (ctypes)	Datentyp (C)	Datentyp (Python)
c_char	char	bytes (ein Zeichen)
c_wchar	wchar_t	str (ein Zeichen)
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int

Tabelle 25.1 Datentypen des Moduls ctypes

3 Wenn die Instanz `None` an eine C-Funktion übergeben wird, kommt sie dort als `NULL`-Pointer an. Umgekehrt wird ein von einer C-Funktion zurückgegebener `NULL`-Pointer in Python zu `None`.

Datentyp (ctypes)	Datentyp (C)	Datentyp (Python)
c_ulong	unsigned long	int
c_longlong	__int64, long long	int
c_ulonglong	unsigned __int64, unsigned long long	int
c_float	float	int
c_double	double	int
c_char_p	char *	bytes, None
c_wchar_p	wchar_t *	str, None
c_void_p	void *	int, None

Tabelle 25.1 Datentypen des Moduls ctypes (Forts.)

All diese ctypes-Datentypen können durch Aufruf ihres Konstruktors instantiiert und mit einer Instanz des angegebenen Python-Datentyps initialisiert werden. Über das Attribut `value` lässt sich der jeweilige Wert eines ctypes-Datentyps verändern.

```
>>> import ctypes
>>> f = ctypes.c_float(1.337)
>>> f
c_float(1.3370000123977661)
>>> d = ctypes.c_double(1.337)
>>> d
c_double(1.337)
>>> s = ctypes.c_char_p("Hallo Welt")
>>> s.value
'Hallo Welt'
>>> null = ctypes.c_void_p(None)
>>> null
c_void_p(None)
```

Um ein Array eines bestimmten Datentyps anzulegen, wird der zugrundeliegende Datentyp mit der Anzahl der Elemente, die er aufnehmen soll, multipliziert. Das Ergebnis ist ein Datentyp, der das gewünschte Array speichert. Im konkreten Beispiel sieht das so aus:

```
>>> arraytyp = ctypes.c_int * 5
>>> a = arraytyp(1, 5, 2, 1, 9)
>>> a
<__main__.c_long_Array_5 object at 0xb7af82fc>
```

Einen solchen Array-Typ können wir beispielsweise der Funktion `sortiere` übergeben, die ein Array von ganzen Zahlen sortiert:

```
from ctypes import CDLL, c_int
bib = CDLL("bibliothek.dll")
arraytyp = c_int * 10
a = arraytyp(0,2,5,2,8,1,4,7,3,8)
print("Vorher: ", [i for i in a])
bib.sortiere(a, 10)
print("Nachher: ", [i for i in a])
```

Die Ausgabe dieses Beispielprogramms lautet:

```
Vorher:  [0, 2, 5, 2, 8, 1, 4, 7, 3, 8]
Nachher:  [0, 1, 2, 2, 3, 4, 5, 7, 8, 8]
```

25.1.3 Schnittstellenbeschreibung

Im folgenden Beispiel sollen die Parameter der Funktion `veclen`, wie von der Funktion verlangt, als Gleitkommazahlen übergeben werden.

```
from ctypes import CDLL, c_double
bib = CDLL("bibliothek.dll")
print(bib.veclen(c_double(1.5), c_double(2.7), c_double(3.9)))
```

Wird dieser Code ausgeführt, so erhalten wir

```
1077412479
```

als Ergebnis, was nun wirklich nicht der gesuchten Vektorlänge entspricht. Wie es zu diesem Fehler kommen konnte und wie er sich vermeiden lässt, soll Thema dieses Abschnitts sein.

Der Rückgabewert und die Parameter einer Funktion, also ihre Schnittstelle, sind in C anders als in Python an bestimmte Datentypen gebunden. Der oben beschriebene Problemfall resultiert daraus, dass nach dem Laden einer dynamischen Bibliothek von ctypes angenommen wird, dass jede enthaltene C-Funktion eine ganze Zahl zurückgibt, was natürlich in vielen Fällen falsch ist. Die eigentliche Gleitkommazahl wurde also aus Unwissenheit als ganze Zahl interpretiert und entsprechend ausgegeben. Damit dies in Zukunft nicht mehr passiert, können wir über das Attribut `restype` eines Funktionsobjekts der Datentyp des Rückgabewertes explizit angeben:

```
from ctypes import CDLL, c_double
bib = CDLL("bibliothek.dll")
bib.veclen.restype = c_double
print(bib.veclen(c_double(1.5), c_double(2.7), c_double(3.9)))
```

Bei diesem Beispielprogramm wird der Rückgabewert der C-Funktion korrekt interpretiert, wie die Ausgabe zeigt:

```
4.97493718553
```

Es wurde angesprochen, dass auch die Parameter einer Funktion in C an einen bestimmten Datentyp gebunden sind. Übergäben Sie beispielsweise im obigen Programm statt des Datentyps `c_double` Instanzen des Datentyps `c_float`, so würde bereits bei der Parameterübergabe ein Fehler in der Interpretation der Daten passieren, der letztlich in einem falschen Rückgabewert mündet.

Python bietet Ihnen an, über das Attribut `argtypes` die Datentypen der Parameter festzulegen. Wenn das gemacht wird, werden übergebene Instanzen eines falschen Datentyps in den korrekten Datentyp konvertiert oder es wird, wenn dies nicht möglich ist, eine `ctypes.ArgumentError`-Exception geworfen. Im folgenden Programm wird die vollständige Schnittstelle der Funktion `veclen` vorgegeben:

```
from ctypes import CDLL, c_double
bib = CDLL("bibliothek.dll")
bib.veclen.restype = c_double
bib.veclen.argtypes = [c_double, c_double, c_double]
print(bib.veclen(c_double(1.5), c_double(2.7), c_double(3.9)))
```

Zwar werden unter Verwendung des Moduls `ctypes` in vielen Fehlerfällen Exceptions geworfen, beispielsweise wenn zu viele, zu wenige oder die falschen Parameter übergeben werden, doch Sie sollten sich immer vergegenwärtigen, dass Sie mit `ctypes` viele Schutzmechanismen von Python umgehen und möglicherweise direkt im Speicher arbeiten. Es ist also durchaus möglich, unter Verwendung von `ctypes` den Python-Interpreter zum Absturz zu bringen. Und mit »Absturz« ist keine Exception im bisherigen Sinne gemeint, sondern ein tatsächlicher Absturz beispielsweise durch einen Speicherzugriffsfehler.

25.1.4 Verwendung des Moduls

An dieser Stelle möchten wir noch einen kurzen Überblick über die wichtigsten im Modul `ctypes` enthaltenen Funktionen bieten, die Ihnen die Arbeit mit C-Funktionen teils erheblich erleichtern.

`ctypes.addressof(obj)`

Gibt die Speicheradresse der Instanz *obj* zurück. Für den Parameter *obj* muss dabei eine Instanz eines `ctypes`-Datentyps übergeben werden.

ctypes.byref(obj)

Erzeugt einen Pointer auf die Instanz *obj* eines ctypes-Datentyps. Der zurückgegebene Pointer kann einer C-Funktion übergeben werden.

ctypes.cast(obj, type)

Die Funktion `cast` bildet den Cast-Operator von C in Python ab. Die Funktion gibt eine neue Instanz des ctypes-Datentyps *type* zurück, die auf die gleiche Speicherstelle verweist wie *obj*.

ctypes.create_string_buffer(init_or_size[, size])

Diese Funktion erzeugt einen veränderlichen String-Buffer, in den aus einer C-Funktion heraus geschrieben werden kann. Zurückgegeben wird ein Array von `c_char`-Instanzen. Für den ersten Parameter *init_or_size* können Sie entweder eine ganze Zahl übergeben, die die Länge des zu erzeugenden Buffers enthält, oder einen String, mit dem der Buffer initialisiert werden soll. Beachten Sie im Falle eines Strings, dass der Buffer ein Zeichen größer gemacht wird, als der String lang ist. In dieses letzte Zeichen wird der Terminator `\0` geschrieben.

Wenn Sie für *init_or_size* einen String übergeben haben, können Sie über den Parameter *size* die Größe des Buffers festlegen, sofern nicht die Länge des Strings genommen werden soll.

ctypes.create_unicode_buffer(init_or_size[, size])

Diese Funktion verhält sich wie `create_string_buffer`, mit dem Unterschied, dass ein Array von veränderlichen `c_wchar`-Instanzen, also ein Unicode-Buffer, erzeugt und zurückgegeben wird.

ctypes.sizeof(obj_or_type)

Die Funktion `sizeof` bildet den `sizeof`-Operator von C auf Python ab. Zurückgegeben wird die Größe der übergebenen Instanz bzw. des übergebenen ctypes-Datentyps in Byte.

ctypes.string_at(address[, size])

Gibt den String zurück, der an der Speicheradresse *address* steht. Sollte der String im Speicher nicht null-terminiert sein, so kann über den Parameter *size* die genaue Länge des Strings übergeben werden.

Für *address* muss eine ganze Zahl übergeben werden, die sinnvollerweise mit `addressof` geholt und verändert wurde.

ctypes.wstring_at(address[, size])

Die Funktion `wstring_at` funktioniert wie `string_at`, nur für Unicode-Strings.

In diesem Abschnitt konnte Ihnen nur ein Einblick in die Funktionalität von `ctypes` gegeben werden. So enthält das Modul `ctypes` noch weitere Konzepte zur Verwendung von Pointern, Strukturen und Unions beispielsweise. Sollte Ihr Interesse am Modul `ctypes` geweckt worden sein und möchten Sie mehr darüber herausfinden, sei Ihnen die ausführliche Python-Dokumentation zu diesem Thema ans Herz gelegt.

25.2 Schreiben von Extensions

Im letzten Abschnitt haben wir uns mit einem Ansatz beschäftigt, bestimmte Teile eines Programms in C zu implementieren, in eine dynamische Bibliothek auszulagern und dann aus dem Python-Programm heraus anzusprechen. Das ist, wie Sie sich sicherlich vorstellen können, für eine Anwendung eine tolle Sache, doch gerade beim Schreiben eines Moduls wäre der `ctypes`-Ansatz aufgrund seiner Einschränkungen an der Schnittstelle nicht besonders praktikabel. Stattdessen ist es möglich und auch durchaus üblich, ein Python-Modul in reinem C zu schreiben und dann ohne `ctypes` zu verwenden. Solche Module werden *Extensions* (dt. *Erweiterungen*) genannt.

Solche Extensions werden mithilfe der sogenannten *Python API* geschrieben. Mit dieser API ist es möglich, in C beispielsweise Instanzen eines Python-Datentyps zu erzeugen und zu verarbeiten. Dies ist unerlässlich, wenn man bedenkt, dass die in C geschriebene Extension später als vollwertiges Modul auftreten soll und dass auf keinen Fall Probleme an der Schnittstelle, wie sie beispielsweise mit `ctypes` auftreten, vorkommen dürfen. Auch das Werfen von Python-Exceptions aus einer C-Extension heraus wird mithilfe der Python API möglich.

Sie werden in diesem Abschnitt nur eine knappe Referenz zur Python API finden, da wir hier primär einen praxisorientierten Einstieg in das Schreiben von Extensions bieten möchten. Eine umfassende Referenz finden Sie in der Python-Dokumentation unter dem Stichwort »Python/C API«. Die Dokumentation enthält außerdem ausführliche Informationen zum Schreiben von Erweiterungen bzw. zum Einbetten des Python-Interpreters in ein C-Programm unter dem Stichwort »Extending and Embedding the Python Interpreter«.

25.2.1 Ein einfaches Beispiel

In diesem Abschnitt soll eine einfache Beispiel-Extension in C geschrieben werden. Wir nehmen uns vor, eine Extension namens `chiffre` zu erstellen, die verschiedene

Funktionen zur Verschlüsselung eines Strings bereitstellt. Da es sich um ein Beispiel handelt, werden wir uns auf eine einzelne Funktion namens `caesar` beschränken, die eine Verschiebeciffre, auch »Cäsar-Verschlüsselung« genannt, durchführen soll. Die Funktion soll später folgendermaßen verwendet werden können:

```
>>> chiffre.caesar("HALLOWELT", 13)
'UNYYBJRYG'
```

Dabei entspricht der zweite Parameter der Anzahl Stellen, um die jeder Buchstabe des ersten Parameters verschoben wird.

Im Folgenden werden wir eine Extension schreiben, die das Modul `chiffre` inklusive der Funktion `caesar` für ein Python-Programm bereitstellt. Die Quelldatei der Erweiterung lautet *chiffre.c*. Der Quelltext der Erweiterung wird nun Schritt für Schritt besprochen.

```
#include <Python.h>
```

```
static PyObject *chiffre_caesar(PyObject *self, PyObject *args);
```

Zunächst wird der Header der Python API eingebunden. Sie finden die Header-Datei *Python.h* im Unterordner *include* Ihrer Python-Installation. Außerdem schreiben wir zu Beginn der Quelldatei den Prototyp der Funktion `chiffre_caesar`, die später der Funktion `caesar` des Moduls `chiffre` entsprechen soll.

Beachten Sie, dass in der Header-Datei *Python.h* einige Präprozessor-Anweisungen enthalten sind, die sich auf andere Header-Dateien auswirken. Aus diesem Grund sollte *Python.h* immer vor den Standard-Headern eingebunden werden.

Dann wird die sogenannte *Method Table* erstellt, die alle Funktionen der Extension auflistet:

```
static PyMethodDef ChiffreMethods[] =
{
    {"caesar", chiffre_caesar, METH_VARARGS,
     "Perform Caesar cipher encryption."},
    {NULL, NULL, 0, NULL}
};
```

Jeder Eintrag der Method Table enthält zunächst den Namen, den die Funktion oder Methode in Python tragen soll, dann den Namen der assoziierten C-Funktion, dann die Kennzeichnung der Art der Parameterübergabe und schlussendlich eine Beschreibung der Funktion als String. Das Makro `METH_VARARGS` besagt, dass alle Parameter, die der Funktion `caesar` in Python übergeben werden, in der C-Funktion `chiffre_caesar` in Form eines Tupels ankommen. Dies ist die bevorzugte Art der Parameterübergabe. Ein mit Nullen gefüllter Eintrag beendet die Method Table.

Nach der Method Table muss noch eine zweite Struktur gefüllt werden, die Informationen über unser Modul `chiffre` enthalten wird:

```
static PyModuleDef ChiffreModule =
{
    PyModuleDef_HEAD_INIT,
    "chiffre",
    "Performing insane encryption operations",
    -1,
    ChiffreMethods
};
```

Die in diesem Beispiel wichtigen Angaben sind der Modulname und der Docstring des Moduls an zweiter bzw. dritter sowie die zuvor erstellte Method Table an letzter Stelle.

Es folgt die Initialisierungsfunktion der Erweiterung namens `PyInit_chiffre`:

```
PyMODINIT_FUNC PyInit_chiffre(void)
{
    return PyModule_Create(&ChiffreModule);
}
```

Sie wird vom Interpreter aufgerufen, wenn das Modul `chiffre` zum ersten Mal eingebunden wird, und hat die Aufgabe, das Modul einzurichten und dem Interpreter die Moduldefinition (`ChiffreModule`) zu übergeben. Dazu wird die Funktion `PyModule_Create` aufgerufen.

Beachten Sie, dass die Funktion `PyInit_chiffre` genannt werden muss, wobei `chiffre` natürlich bei einem anderen Modulnamen entsprechend angepasst werden muss.

Jetzt folgt die Funktion `chiffre_caesar`. Das ist die C-Funktion, die bei einem Aufruf der Python-Funktion `chiffre.caesar` aufgerufen wird.

```
static PyObject *chiffre_caesar(PyObject *self, PyObject *args)
{
    char *text, *encrypted, *c, *e;
    PyObject *result = NULL;
    int cipher, length;

    if (!PyArg_ParseTuple(args, "si", &text, &cipher))
        return NULL;

    length = strlen(text);
    encrypted = (char *)malloc(length+1);
    encrypted[length] = '\0';

    for(c = text, e = encrypted; *c; c++, e++)
        *e = ((*c - 'A' + cipher) % 26) + 'A';
```

```

result = Py_BuildValue("s", encrypted);
free(encrypted);

return result;
}

```

Dabei werden als Parameter immer zwei Pointer auf eine `PyObject`-Struktur übergeben. Eine solche `PyObject`-Struktur entspricht ganz allgemein einer Referenz auf ein Python-Objekt. In C werden also alle Instanzen aller Datentypen auf `PyObject`-Strukturen abgebildet. Durch Funktionen der Python API lassen sich dann datentyp-spezifische Eigenschaften der Instanzen auslesen. Doch kommen wir nun zur Bedeutung der übergebenen Parameter im Einzelnen.

Der erste Parameter, `self`, würde nur dann benötigt, wenn die Funktion `chiffre_caesar` eine Python-Methode implementieren würde, und ist in diesem Beispiel immer `NULL`. Der zweite Parameter, `args`, ist ein Tupel und enthält alle der Python-Funktion übergebenen Parameter. Auf die Parameter kann über die API-Funktion `PyArg_ParseTuple` zugegriffen werden.

Die Funktion `PyArg_ParseTuple` bekommt zunächst den Parameter `args` selbst übergeben und danach einen String, der die Datentypen der enthaltenen Parameter kennzeichnet. `s` steht dabei für einen String und `i` für eine ganze Zahl. Im Folgenden zeigt `text` auf den übergebenen String, während `cipher` den zweiten übergebenen Parameter, die ganze Zahl, enthält. Beachten Sie, dass `text` auf den Inhalt des übergebenen Python-Strings zeigt und aus diesem Grund nicht verändert werden darf. Bedenken Sie außerdem, dass die Funktion `PyArg_ParseTuple` eine `TypeError`-Exception wirft, wenn eine Instanz eines unpassenden Typs übergeben wurde, und eine `UnicodeError`-Exception, wenn sich ein übergebener String nicht mittels des Standardencodings in einen C-String kodieren ließ. Näheres zu Exceptions erfahren Sie im nächsten Abschnitt.

Nachdem der zu verschlüsselnde Text in `text` und die zu verwendende Anzahl Stellen in `cipher` stehen, kann die tatsächliche Verschlüsselung durchgeführt werden. Dazu wird zunächst ein neuer C-String mit der Länge des Strings `text` erzeugt, der später den verschlüsselten Text enthalten soll. Zum Verschlüsseln wird dann in einer Schleife über alle Buchstaben des übergebenen Strings iteriert und jeder Buchstabe um die angegebene Anzahl Stellen verschoben. Der auf diese Weise veränderte Buchstabe wird dann in den neu erstellten String `encrypted` geschrieben.

Beachten Sie, dass dieser Algorithmus weder für Kleinbuchstaben noch für Sonderzeichen, sondern ausschließlich für Großbuchstaben funktioniert. Diese Einschränkung erhöht nicht nur die Übersicht, sondern erlaubt es uns später, einen – zugegebenermaßen künstlichen – Fehlerfall zu erzeugen.

25.2.2 Exceptions

Wir haben bereits gesagt, dass der in der Funktion `chiffre_caesar` verwendete Algorithmus nur mit Strings arbeiten kann, die allein aus Großbuchstaben bestehen. Es wäre natürlich ein Leichtes, die Funktion `chiffre_caesar` dahingehend anzupassen, dass auch Kleinbuchstaben verschlüsselt und Sonderzeichen übersprungen werden. Doch zu Demonstrationszwecken soll in diesem Beispiel stattdessen eine `ValueError`-Exception geworfen werden, wenn der übergebene String nicht ausschließlich aus Großbuchstaben besteht.

Eine eingebaute Exception kann mithilfe der Funktion `PyErr_SetString` geworfen werden, wobei der Funktion der Exception-Typ, in diesem Fall ist es `PyExc_ValueError`, und die Fehlerbeschreibung übergeben werden. Die Funktion `chiffre_caesar` sieht inklusive Fehlerbehandlung so aus:

```
static PyObject *chiffre_caesar(PyObject *self, PyObject *args)
{
    char *text, *encrypted, *c, *e;
    PyObject *result = NULL;
    int cipher, length;

    if(!PyArg_ParseTuple(args, "si", &text, &cipher))
        return NULL;

    length = strlen(text);
    encrypted = (char *)malloc(length+1);
    encrypted[length] = '\\0';

    for(c = text, e = encrypted; *c; c++, e++)
    {
        if(((*c < 'A') || (*c > 'Z'))
        {
            PyErr_SetString(PyExc_ValueError,
                            "Character out of range");

            return NULL;
        }
        *e = ((*c - 'A' + cipher) % 26) + 'A';
    }

    result = Py_BuildValue("s", encrypted);
    free(encrypted);

    return result;
}
```

Nachdem die Extension kompiliert, gelinkt und installiert wurde (Näheres zu diesen Vorgängen erfahren Sie im nächsten Abschnitt), können wir sie tatsächlich im interaktiven Modus ausprobieren:

```
>>> import chiffre
>>> chiffre.caesar("HALLOWELT", 13)
'UNYYBJRYG'
>>> chiffre.caesar("UNYYBJRYG", 13)
'HALLOWELT'
>>> chiffre.caesar("Hallo Welt", 13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Character out of range
```

Bislang haben wir die Cäsar-Verschlüsselung immer mit dem Verschiebungsparameter 13 durchgeführt. Das entspricht dem *ROT13*-Algorithmus, der eine relativ große Bekanntheit genießt (was aber nicht heißt, dass er besonders sicher wäre). Der Vorteil von ROT13 ist, dass der verschlüsselte String durch nochmaliges Verschlüsseln mit ROT13 entschlüsselt wird, wie auch im obigen Beispiel zu sehen ist. Selbstverständlich sind aber auch andere Verschiebungsparameter möglich.

25.2.3 Erzeugen der Extension

Da die soeben geschriebene Quelldatei *chiffre.c* in einer ganz bestimmten Art kompiliert und gelinkt werden muss, kommt es uns nur zupass, dass das Paket *distutils* auch das Kompilieren und Linken von Erweiterungen automatisiert.

Beachten Sie, dass zum Kompilieren der Extension ein C-Compiler auf Ihrem System installiert sein muss. Unter Linux wird vom Paket *distutils* der GCC- und unter Windows der MSVC-Compiler von Visual Studio 2008 (sofern es installiert ist) verwendet. Visual Studio ist inzwischen in einer eingeschränkten Version kostenlos von Microsoft zu beziehen. Sollten Sie ein Windows-System einsetzen und Visual Studio 2008 nicht installiert haben, bietet Ihnen das Paket *distutils* an, stattdessen eine MinGW-Installation zu verwenden.

Das Installationsscript *setup.py* sieht in Bezug auf unsere einfache Beispielextension folgendermaßen aus:

```
from distutils.core import setup, Extension

modul = Extension("chiffre", sources=["chiffre.c"])
setup(
    name = "PyChiffre",
    version = "1.0",
    description = "Module for encryption techniques.",
    ext_modules = [modul]
)
```

Zunächst wird eine Instanz der Klasse `Extension` erzeugt und ihrem Konstruktor der Name der Extension und eine Liste der zugrundeliegenden Quelldateien übergeben. Beim Aufruf der Funktion `setup` wird, abgesehen von den üblichen Parametern, der Schlüsselwortparameter `ext_modules` übergeben. Dort muss eine Liste von `Extension`-Instanzen übergeben werden, die mit dem Installations-script kompiliert, gelinkt und in die Distribution aufgenommen werden sollen.

Jetzt kann das Installationsscript wie gewohnt ausgeführt werden und kompiliert bzw. installiert die Erweiterung automatisch.

Neben dem Schlüsselwortparameter `sources` können bei der Instantiierung der Klasse `Extension` noch weitere Parameter übergeben werden, die in der folgenden Tabelle kurz erläutert werden.

Parameter	Bedeutung
<code>include_dirs</code>	eine Liste von Verzeichnissen, die für das Kompilieren der Erweiterung benötigte Header-Dateien enthalten
<code>define_macros</code>	Eine Liste von Tupeln, über die beim Kompilieren der Erweiterung bestimmte Makros mit bestimmten Werten definiert werden können. Das Tupel muss folgende Struktur haben: <code>("MAKRONAME", "Wert")</code> .
<code>undef_macros</code>	eine Liste von Makronamen, die beim Kompilieren nicht definiert sein sollen
<code>libraries</code>	eine Liste von Bibliotheksnamen, gegen die die Erweiterung gelinkt werden soll
<code>library_dirs</code>	eine Liste von Verzeichnissen, in denen nach den bei <code>libraries</code> angegebenen Bibliotheken gesucht wird

Tabelle 25.2 Schlüsselwortparameter des Extension-Konstruktors

Nachdem die Extension mit `distutils` kompiliert und installiert wurde, kann sie in einer Python-Shell verwendet werden:

```
>>> import chiffre
>>> chiffre.caesar("HALLOWELT", 13)
'UNYYBJRYG'
>>> chiffre.caesar("UNYYBJRYG", 13)
'HALLOWELT'
```

25.2.4 Reference Counting

Wie Sie wissen, basiert die Speicherverwaltung Pythons auf einem sogenannten *Reference-Counting*-Algorithmus. Das bedeutet, dass Instanzen zur Entsorgung freigegeben werden, sobald keine Referenzen mehr auf sie bestehen. Das hat den

Vorteil, dass sich der Programmierer nicht um das Freigeben von allokiertem Speicher zu kümmern braucht.

Vermutlich wissen Sie ebenfalls, dass es so etwas wie Reference Counting in C nicht gibt, sondern dass dort der Programmierer für die Speicherverwaltung selbst verantwortlich ist. Wie verhält es sich damit also, wenn man Python-Instanzen in einem C-Programm verwendet?

Grundsätzlich sollten Sie sich von dem C-Idiom verabschieden, im Besitz einer bestimmten Instanz bzw. in diesem Fall einer `PyObject`-Struktur zu sein. Vielmehr können Sie allenfalls im Besitz einer Referenz auf eine Instanz bzw. eines Pointers auf eine `PyObject`-Struktur sein. Damit implementiert die Python API im Grunde das Speichermodell von Python in C. Im Gegensatz zum Speichermodell von Python erhöht bzw. verringert sich der Referenzzähler einer Instanz jedoch nicht automatisch, sondern muss in einer C-Extension explizit mitgeführt werden. Dazu können die Makros `Py_INCREF` und `Py_DECREF` der Python API folgendermaßen verwendet werden:

```
PyObject *string = PyBytes_FromString("Hallo Welt");
Py_INCREF(string);
Py_DECREF(string);
Py_DECREF(string);
```

Zunächst wird mithilfe der Funktion `PyBytes_FromString` eine Instanz des Python-Datentyps `bytes` erzeugt. In diesem Moment besitzen Sie eine Referenz auf diese Instanz. Der Reference Count ist damit gleich 1. Im Folgenden wird der Reference Count durch die Makros `Py_INCREF` und `Py_DECREF` einmal erhöht und zweimal verringert. Am Ende des Beispiels erreicht der Reference Count 0, und der erzeugte String wird der Garbage Collection zum Fraß vorgeworfen.

Das hier besprochene Beispiel ist zugegebenermaßen nicht gerade sinnvoll. Im nächsten Abschnitt wird jedoch ein Beispielprogramm erarbeitet, in dem die Anwendung des Reference Countings in der Praxis zu sehen ist.

Jetzt bliebe nur noch die Frage zu klären, wann Sie den Referenzzähler erhöhen bzw. verringern müssen. Immer dann, wenn Sie in Ihrem Programm eine Instanz eines Python-Datentyps erzeugen und eine oder mehrere Referenzen auf diese Instanz halten, müssen Sie diese Referenzen freigeben, wenn sie nicht mehr benötigt werden. Sollten Sie die Referenzen nicht freigeben, verweilt die Instanz im Speicher, obwohl sie eigentlich nicht mehr benötigt wird. Es handelt sich dann um ein sogenanntes *Memory Leak*, und Memory Leaks sind kein besonders erstrebenswerter Umstand in einem Programm.

Die zweite Möglichkeit sind sogenannte *geliehene Referenzen* (engl. *borrowed references*). Solche Referenzen sind beispielsweise über Funktionsparameter an Ihr Programm weitergereicht worden, gehören Ihnen aber nicht. Das bedeutet auch, dass Sie solche Referenzen nicht freizugeben brauchen. Ein Beispiel für geliehene Referenzen sind Funktionsparameter, die Sie grundsätzlich nicht freizugeben brauchen. Wenn Sie eine geliehene Referenz zu einer eigenen Referenz aufwerten möchten, müssen Sie den Referenzzähler der dahinterliegenden Instanz mittels `Py_INCREF` erhöhen. Das Freigeben von geliehenen Referenzen führt dazu, dass danach möglicherweise auf Speicherbereiche zugegriffen wird, die bereits freigegeben worden sind. Das kann in einigen Fällen gutgehen, führt aber häufig zu einem *Speicherzugriffsfehler*. Ähnlich wie Memory Leaks sollten Sie Speicherzugriffsfehler nach Möglichkeit vermeiden.

Als letzte Möglichkeit geben Sie eine vollwertige Referenz in Form eines Rückgabewertes an eine andere Funktion ab. Sie brauchen sich also nicht um die Freigabe Ihrer zurückgegebenen Instanzen zu kümmern.

25.3 Python als eingebettete Scriptsprache

In den vorangegangenen Abschnitten haben Sie Möglichkeiten kennengelernt, in C geschriebene Programmteile von einem Python-Programm aus aufzurufen und somit beispielsweise lauffzeitkritische Teile in ein C-Programm auszulagern.

In diesem Abschnitt soll der entgegengesetzte Weg beschritten werden: Wir möchten Python-Programme aus einem C-Programm heraus ausführen, Python also als eine Art eingebettete Scriptsprache (engl. *embedded script language*) verwenden.⁴ Auf diese Weise können wir bestimmte Teile des Programms in Python schreiben, für die Python aufgrund seiner Flexibilität einfach besser geeignet ist.

25.3.1 Ein einfaches Beispiel

Zum Einstieg soll ein C-Programm erstellt werden, das ein möglichst simples Python-Programm ausführt. Dieses Python-Programm soll lediglich ein bisschen Text und eine Zufallszahl auf dem Bildschirm ausgeben.

Das folgende C-Programm führt ein solches Python-Script aus:

4 Ein Beispiel für eine solche eingebettete Scriptsprache ist *Lua*, die besonders häufig als Scriptsprache für Computerspiele eingesetzt wird.

```
#include <Python.h>

const char *programm =
"import random\n"
"print('Guten Tag, die Zahl ist:', random.randint(0, 100))\n"
"print('Das war ... Python')\n";

int main(int argc, char *argv[])
{
    Py_Initialize();
    PyRun_SimpleString(programm);
    Py_Finalize();
}
```

Zunächst wird die Header-Datei der Python API eingebunden. Sie sehen, dass sowohl zum Erweitern als auch zum Einbetten von Python dieselbe API verwendet wird. Danach wird der String `programm` angelegt, der den Python-Code enthält, der später ausgeführt werden soll.

In der Hauptfunktion `main` wird der Python-Interpreter zuerst durch Aufruf von `Py_Initialize` initialisiert. Danach wird das zuvor im String `programm` abgelegte Python-Script durch Aufruf der Funktion `PyRun_SimpleString` ausgeführt und der Interpreter schlussendlich durch die Funktion `Py_Finalize` wieder beendet.

Statt der Funktion `PyRun_SimpleString` hätten wir auch die Funktion `PyRun_SimpleFile` aufrufen können, um den Python-Code aus einer Datei zu lesen.

Wichtig ist, dass dem Compiler das Verzeichnis bekannt ist, in dem die Header-Datei *Python.h* liegt. Außerdem muss das Programm gegen die Python API gelinkt werden. Diese ist als dynamische Bibliothek *python30.lib* im Unterordner *lib* Ihrer Python-Installation zu finden.⁵ Wenn sowohl das Kompilieren als auch das Linken ohne Probleme vonstattengegangen sind, werden Sie feststellen, dass das Programm tatsächlich funktioniert:

```
Guten Tag, die Zufallszahl ist: 64
Das war ... Python
```

Sie sehen, dass wir uns für dieses Beispielprogramm einer sehr abstrakten Schnittstelle bedient haben, denn es ist beispielsweise nicht möglich, mit dem Python-Script zu interagieren. Das Python-Script läuft bislang völlig autonom, und es können keine Werte zwischen ihm und dem C-Programm ausgetauscht wer-

5 Diese Angabe bezieht sich auf Windows. Unter Linux wird die Bibliothek in den meisten Fällen in das Verzeichnis für systemweite Bibliotheken, */usr/lib*, installiert. Beim Linken Ihres Programms mit dem GCC können Sie die Python 3.0-Bibliothek mit der Option `-lpython3.0` einbinden.

den. Aber gerade die Interaktion mit dem Hauptprogramm macht die Qualität einer eingebetteten Scriptsprache aus.

25.3.2 Ein komplexeres Beispiel

Sie haben sicherlich erkannt, dass das vorangegangene Beispielprogramm noch nicht der Stein der Weisen war. In diesem Abschnitt wollen wir ein besseres, komplexeres Programm entwickeln, das Funktionen eines Python-Scripts direkt aufruft und Werte über die Funktionsschnittstelle schickt bzw. entgegennimmt. Außerdem soll das C-Programm dazu in der Lage sein, eigene Funktionen zu definieren, die aus dem Python-Script heraus aufgerufen werden können. Sie werden feststellen, dass auch dies dem Schreiben von Erweiterungen stark ähnelt.

Das folgende C-Programm lädt ein Python-Script, das eine Funktion `entscheide` implementiert. Diese Funktion soll sich für einen von zwei übergebenen Strings entscheiden. Die Funktion könnte beispielsweise deshalb in ein Python-Script ausgelagert worden sein, weil der Programmierer es dem Anwender ermöglichen will, die Funktion selbst zu implementieren und das Programm somit an die eigenen Bedürfnisse anzupassen.

Der Quellcode des Beispielprogramms sieht folgendermaßen aus:

```
#include <Python.h>

int main(int argc, char *argv[])
{
    char *ergebnis;
    PyObject *modul, *funk, *prm, *ret;

    Py_Initialize();
    PySys_SetPath(L".");
    modul = PyImport_ImportModule("script");

    if(modul)
    {
        funk = PyObject_GetAttrString(modul, "entscheide");
        prm = Py_BuildValue("(yy)", "Hallo", "Welt");
        ret = PyObject_CallObject(funk, prm);

        ergebnis = PyBytes_AsString(ret);
        printf("Das Script hat sich fuer '%s' entschieden\n",
                                                       ergebnis);

        Py_DECREF(prm);
        Py_DECREF(ret);
        Py_DECREF(funk);
        Py_DECREF(modul);
    }
}
```

```

    }
else
    printf("Fehler: Modul nicht gefunden\n");
Py_Finalize();
}

```

In der Hauptfunktion `main` des C-Programms wird zunächst der Python-Interpreter durch Aufruf von `Py_Initialize` initialisiert. Danach wird durch die Funktion `PySys_SetPath` das lokale Programmverzeichnis als einziger Ordner festgelegt, aus dem Module importiert werden können. Beachten Sie, dass dieser Funktionsaufruf sowohl dem C- als auch dem Python-Programm verbietet, globale Module wie beispielsweise `math` einzubinden. Wenn Sie solche Module benötigen, dürfen Sie die Import-Pfade nicht, wie es in diesem Beispiel geschehen ist, überschreiben, sondern Sie sollten sich den Pfad mit `Py_GetPath` holen, ihn um das Verzeichnis `.` erweitern und mit `PySys_SetPath` setzen. Beachten Sie, dass das lokale Programmverzeichnis standardmäßig nicht als `import`-Pfad eingetragen ist.

Durch Aufruf der Funktion `PyImport_ImportModule` wird ein Modul eingebunden und als `PyObject`-Pointer zurückgegeben. Wenn die entsprechenden Pfade festgelegt wurden, können sowohl lokale als auch globale Module mit dieser Funktion eingebunden werden. Nachfolgend prüfen wir, ob das Modul erfolgreich geladen wurde. Bei einem Misserfolg gibt die Funktion `PyImport_ImportModule` wie die meisten anderen Funktionen, die einen `PyObject`-Pointer zurückgeben, `NULL` zurück. Beachten Sie, dass es immer ratsam ist, die zurückgegebenen `PyObject`-Pointer auf `NULL` zu testen. Im Beispielprogramm wurde dies nur exemplarisch bei `modul` gemacht.

Nachfolgend erstellen wir durch Aufruf der Funktion `PyObject_GetAttrString` einen Pointer auf die Funktion `entscheide` des Moduls `script`. Um die Funktion aufrufen zu können, müssen wir die Funktionsparameter in Form eines Tupels übergeben. Dazu erzeugen wir mittels `Py_BuildValue` ein neues Tupel, das die beiden `bytes`-Strings `"Hallo"` und `"Welt"` enthält, von denen die Funktion `entscheide` einen auswählen soll.

Durch Aufruf der Funktion `PyObject_CallObject` wird die Funktion `funk` schlussendlich aufgerufen und ihr Rückgabewert ebenfalls in Form eines Pointers auf `PyObject` zurückgegeben. Da es sich bei dem Rückgabewert um einen `bytes`-String handelt, können wir diesen mit `PyBytes_AsString` zu einem C-String konvertieren und dann mit `printf` ausgeben.

Die in diesem Beispiel aufgerufene Python-Funktion `entscheide` sieht folgendermaßen aus und befindet sich in der Programmdatei `script.py`:


```
def entscheide(a, b):
    return (a if min(a) < min(b) else b)
```

Die Funktion bekommt zwei Instanzen *a* und *b* übergeben und gibt eine der beiden zurück. Die Entscheidung, welche der beiden Instanzen zurückgegeben wird, hängt davon ab, welche in der jeweiligen Ordnungsrelation dieser Instanzen die kleinere ist. Bei zwei Strings bzw. *bytes*-Strings wird beispielsweise derjenige zurückgegeben, der den alphabetisch kleinsten Buchstaben enthält.

Im nächsten Beispielprogramm soll es dem Python-Script ermöglicht werden, bestimmte Funktionen des C-Programms aufzurufen. Es soll dem Script also gewissermaßen eine API zur Verfügung gestellt werden, die es verwenden kann. Diese Idee liegt nicht nur gedanklich sehr nah an den in Abschnitt 25.2, »Schreiben von Extensions«, besprochenen Extensions, sondern wird auch ganz ähnlich umgesetzt. Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
#include <Python.h>

static PyObject *testfunktion(PyObject *self, PyObject *args)
{
    int a, b;
    if(!PyArg_ParseTuple(args, "ii", &a, &b))
        return NULL;
    return Py_BuildValue("i", a + b);
}

static PyMethodDef MethodTable[] =
{
    {"testfunktion", testfunktion, METH_VARARGS, "Testfunktion"},
    {NULL, NULL, 0, NULL}
};

static PyModuleDef APIModule =
{
    PyModuleDef_HEAD_INIT,
    "api", NULL, -1, MethodTable
};

static PyObject *PyInit_api()
{
    return PyModule_Create(&APIModule);
}

int main(int argc, char *argv[])
{
    FILE *f;
```

```

PyImport_AppendInittab("api", &PyInit_api);
Py_Initialize();
f = fopen("script.py", "r");
PyRun_SimpleFile(f, "script.py");
fclose(f);

Py_Finalize();
}

```

Zunächst wird die Funktion `testfunktion` definiert, die später dem Python-Script über ein Modul namens `api` zur Verfügung gestellt werden soll. Im Beispiel berechnet die Funktion schlicht die Summe zweier ganzer Zahlen, die ihr als Parameter übergeben werden. Danach werden die `MethodTable` und die Moduldefinition erstellt, ganz als würden wir eine Erweiterung schreiben.

Ebenfalls analog zu den Erweiterungen wird eine Initialisierungsfunktion für das Modul namens `PyInit_api` benötigt. Durch Aufruf der Funktion `PyImport_AppendInittab` wird unser Modul `api` in die Liste der verfügbaren Module eingetragen.

Schlussendlich brauchen wir nur noch die Funktion `PyRun_SimpleFile` aufzurufen, um das Python-Script `script.py` zu interpretieren. Der Funktion `PyRun_SimpleFile` müssen wir dabei ein geöffnetes Dateiobjekt übergeben.

Das Python-Script, das von diesem C-Programm aufgerufen wird, könnte beispielsweise folgendermaßen aussehen:

```

import api
print("Zwei plus zwei ist:", api.testfunktion(2, 2))

```

25.3.3 Python-API-Referenz

Nachdem die Python API in den Themenbereichen »Erweitern und Einbetten von Python« bereits verwendet wurde, soll an dieser Stelle eine kleine Referenz dieser API stehen. Beachten Sie dabei, dass die Python API sehr umfangreich ist und in diesem Abschnitt keinesfalls vollständig behandelt werden kann. Aus diesem Grund beschränken wir uns auf die Beschreibung der Funktionen der Python API, die in den Beispielprogrammen der vorherigen Abschnitte vorgekommen sind.

Die Funktionen sind in alphabetischer Reihenfolge aufgeführt.

PyObject *Py_BuildValue(const char *format, ...)

Erzeugt eine Instanz eines Python-Datentyps mit einem bestimmten Wert. Der String *format* spezifiziert dabei den Datentyp. Die folgende Tabelle listet die wichtigsten Werte für *format* mit ihrer jeweiligen Bedeutung auf.

Formatstring	Beschreibung
"s"	Erzeugt eine Instanz des Python-Datentyps <code>str</code> aus einem C-String (<code>char *</code>).
"y"	Erzeugt eine Instanz des Python-Datentyps <code>bytes</code> aus einem C-String (<code>char *</code>).
"u"	Erzeugt eine Instanz des Python-Datentyps <code>str</code> aus einem C-Buffer mit Unicode-Daten (<code>Py_UNICODE *</code>).
"i"	Erzeugt eine Instanz des Python-Datentyps <code>int</code> aus einem C-Integer (<code>int</code>).
"c"	Erzeugt eine Instanz des Python-Datentyps <code>str</code> aus einem C-Zeichen (<code>char</code>). Der resultierende String hat die Länge 1.
"d"	Erzeugt eine Instanz des Python-Datentyps <code>float</code> aus einer C-Gleitkommazahl (<code>double</code>). Analog dazu existiert "d" für den C-Datentyp <code>float</code> .
"(...)"	Erzeugt eine Instanz des Python-Datentyps <code>tuple</code> . Anstelle der Auslassungszeichen müssen die Datentypen der Elemente des Tupels angegeben werden. "(iii)" würde beispielsweise ein Tupel mit drei ganzzahligen Elementen erzeugen.
"[...]"	Erzeugt eine Instanz des Python-Datentyps <code>list</code> .
"{...}"	Erzeugt eine Instanz des Python-Datentyps <code>dict</code> .

Tabelle 25.3 Mögliche Angaben im Formatstring

Beachten Sie, dass ein C-String, der an die Funktion `Py_BuildValue` übergeben wird, um einen Python-String zu erzeugen, stets kopiert wird. Das bedeutet insbesondere, dass Sie jeden dynamisch allokierten String wieder freigeben müssen, selbst wenn er an die Funktion `Py_BuildValue` übergeben wurde.

void Py_Finalize()

Die Funktion `Py_Finalize` deinitialisiert den Python-Interpreter und gibt den vom Interpreter belegten Speicher frei. Diese Funktion sollte beim Embedding des Python-Interpreters aufgerufen werden, wenn der Interpreter nicht mehr benötigt wird.

void Py_INCREF(PyObject *o),

void Py_DECREF(PyObject *o)

Die Makros `Py_INCREF` und `Py_DECREF` erhöhen (inkrementieren) bzw. verringern (dekrementieren) den Reference Count der Instanz `o` um 1.

void Py_Initialize()

Die Funktion `Py_Initialize` initialisiert den Python-Interpreter und sollte aufgerufen werden, bevor der Interpreter beim Embedding eingesetzt wird.

void Py_XINCREF(PyObject *o),**void Py_XDECREF(PyObject *o)**

Die Makros `Py_XINCREF` und `Py_XDECREF` erhöhen bzw. verringern den Reference Count der Instanz `o` um 1. Dabei prüfen die Makros vorher, ob für `o` NULL übergeben wurde.

int PyArg_ParseTuple(PyObject *args, const char *format, ...)

Die Funktion `PyArg_ParseTuple` liest die einer Funktion übergebenen Argumente aus und speichert sie in lokalen Variablen. Als erster Parameter muss ein Tupel übergeben werden, das die Parameter enthält. Ein solches Tupel bekommt jede Python-Funktion in C übergeben. Beachten Sie, dass mit `PyArg_ParseTuple` nur Positionsargumente ausgelesen werden können.

Der zweite Parameter *format* ist ein String, ähnlich dem Formatstring von `Py_BuildValue`, und legt fest, wie viele Parameter ausgelesen werden und welche Datentypen diese haben sollen. Zum Schluss akzeptiert die Funktion `PyArg_ParseTuple` beliebig viele Pointer auf Variablen, die mit den ausgelesenen Werten gefüllt werden sollen.

Im Erfolgsfall gibt die Funktion `True` zurück. Bei einem Fehler wirft sie eine entsprechende Exception und gibt `False` zurück.

void PyErr_SetString(PyObject *type, const char *message)

Die Funktion `PyErr_SetString` wirft eine Python-Exception. Sie bekommt den Typ der auszulösenden Exception als ersten Parameter übergeben. Das kann eine der vordefinierten Standard-Exceptions, beispielsweise `PyExc_NameError` oder `PyExc_ValueError`, sein. Als zweiter Parameter wird eine Fehlermeldung als Wert der Exception angegeben. Um eine beliebige Instanz als Wert der Exception eintragen zu lassen, kann die Funktion `PyErr_SetObject` verwendet werden.

Beachten Sie, dass Sie den Reference Count einer Standard-Exception nicht erhöhen müssen, wenn Sie sie an `PyErr_SetString` übergeben.

PyObject *PyImport_ImportModuleEx(char *name, PyObject *globals, PyObject *locals, PyObject *fromlist)

Diese C-Funktion lädt ein Python-Modul und ist äquivalent zum Aufruf der Built-in Function `__import__` aus Python heraus. Die Funktion `PyImport_ImportModule`

dient demselben Zweck wie `PyImport_ImportModuleEx`, hat aber eine vereinfachte Schnittstelle. Bei ihr braucht nur der Name des zu importierenden Moduls übergeben zu werden.

Beachten Sie, dass Sie das lokale Programmverzeichnis zuerst mit `PySys_SetPath` einrichten müssen, bevor lokale Module eingebunden werden können.

void PyMem_Free(void *p)

Die Funktion `PyMem_Free` gibt den allokierten Speicherblock frei, auf den *p* zeigt.

void *PyMem_Malloc(size_t n)

Die Funktion `PyMem_Malloc` allokiert einen Speicherbereich der Größe *n* und gibt einen Pointer auf den allokierten Speicher zurück.

PyObject *PyObject_CallObject(PyObject *callable_object, PyObject *args)

Die Funktion `PyObject_CallObject` ruft das aufrufbare Objekt *callable_object*, beispielsweise also ein Funktions- oder Methodenobjekt, auf und übergibt dabei die Parameter *args*. Wenn keine Parameter übergeben werden sollen, kann für *args* `NULL` übergeben werden.

Das Ergebnis des Objektaufrufs wird als Rückgabewert zurückgegeben. Bei einem Fehler wird `NULL` zurückgegeben.

PyObject *PyObject_GetAttrString(PyObject *o, const char *attr_name)

Gibt eine Referenz auf das Attribut mit dem Namen *attr_name* der Instanz *o* zurück. Im Fehlerfall wird `NULL` zurückgegeben.

int PyRun_SimpleFile(FILE *fp, const char *filename)

Die Funktion `PyRun_SimpleFile` führt eine Python-Programmdatei aus. Dabei wird der Inhalt der Programmdatei in Form eines geöffneten Datei-Pointers übergeben. Zusätzlich sollte der Dateiname der Programmdatei als zweiter Parameter übergeben werden.

Die Funktion gibt 0 zurück, wenn der Code erfolgreich ausgeführt wurde, und -1, wenn ein Fehler aufgetreten ist.

int PyRun_SimpleString(const char *command)

Die Funktion `PyRun_SimpleString` verhält sich ähnlich wie `PyRun_SimpleFile` mit dem Unterschied, dass der auszuführende Python-Code aus dem String *command* statt aus einer Datei gelesen wird.

char *PyBytes_AsString(PyObject *o)

Gibt den internen C-Buffer des Python-bytes-Strings *o* zurück. Beachten Sie, dass Sie nur dann in den zurückgegebenen Buffer schreiben dürfen, wenn Sie ihn zuvor beispielsweise mit der Funktion `PyBytes_FromStringAndSize` selbst erstellt haben.

PyObject *PyBytes_FromStringAndSize(const char *v, Py_ssize_t len)

Erzeugt eine Instanz des Python-Datentyps `str` mit der Länge *len* und dem Inhalt des C-Strings *v*. Für *v* kann `NULL` übergeben werden. Die Funktion gibt eine Referenz auf die erzeugte Instanz zurück.

void PySys_SetPath(wchar_t *path)

Über die Funktion `PySys_SetPath` können die Verzeichnisse festgelegt werden, in denen nach Modulen gesucht wird. Die Funktion entspricht damit dem Schreiben des Strings `sys.path` in Python. Umgekehrt können die gesetzten Verzeichnisse über die Funktion `PySys_GetPath` ausgelesen werden.

»Warum glauben einem Leute sofort, wenn man ihnen sagt, dass es am Himmel 400 Billionen Sterne gibt, aber wenn man ihnen sagt, dass die Bank frisch gestrichen ist, müssen sie draufpatschen?«
– Unbekannter Autor

26 Insiderwissen

In diesem Kapitel sollen kleinere Module vorgestellt werden, die im Programmieralltag von Nutzen sein können. Diese Module implementieren keine neuen Konzepte oder irgendetwas Weltbewegendes, sie sind einfach nur praktisch. Es lohnt sich also allemal, einen Blick auf die folgenden Abschnitte zu werfen.

26.1 URLs im Standardbrowser öffnen – `webbrowser`

Das Modul `webbrowser` dient dazu, eine Website im Standardbrowser des gerade verwendeten Systems zu öffnen. Im Folgenden werden wir die drei wichtigsten Funktionen des Moduls `webbrowser` erklären. Die Beispiele verstehen sich dabei in folgendem Kontext:

```
>>> import webbrowser
```

`webbrowser.open(url[, new[, autoraise]])`

Öffnet die URL *url* im Standardbrowser des Systems.

Für den Parameter *new* kann eine ganze Zahl zwischen 0 und 2 übergeben werden. Dabei bedeutet ein Wert von 0, dass die URL nach Möglichkeit in einem bestehenden Browserfenster geöffnet wird, 1, dass die URL in einem neuen Browserfenster geöffnet werden soll, und 2, dass die URL nach Möglichkeit in einem neuen Tab eines bestehenden Browserfensters geöffnet werden soll. Der Parameter ist mit 0 vorbelegt.

Wenn für den Parameter *autoraise* `True` übergeben wird, wird versucht, das Browserfenster mit der geöffneten URL in den Vordergrund zu holen. Beachten Sie, dass dies bei vielen Systemen automatisch geschieht.

```
>>> webbrowser.open("http://www.galileo-press.de", 2)
True
```


webbrowser.open_new(url)

Öffnet die URL *url* in einem neuen Fenster des Standardbrowsers.

```
>>> webbrowser.open_new("http://www.galileo-press.de")
True
```

webbrowser.open_new_tab(url)

Öffnet die URL *url* in einem neuen Tab eines bestehenden Fensters des Standardbrowsers. Wenn noch keine Instanz des Browsers gestartet ist, wird die URL in einem neuen Fenster geöffnet.

```
>>> webbrowser.open_new_tab("http://www.galileo-press.de")
True
```

26.2 Funktionsschnittstellen vereinfachen – functools

Im Modul `functools` ist die Funktion `partial` enthalten, mit der sich Funktionschnittstellen vereinfachen lassen. Betrachten Sie dazu die folgende Funktion:

```
def f(a, b, c, d):
    print("{0} {1} {2} {3}".format(a,b,c,d))
```

Die Funktion `f` erwartet viele Parameter, vier an der Zahl. Stellen Sie sich nun vor, wir müssten die Funktion `f` sehr häufig im Programm aufrufen und übergäben dabei für die Parameter `b`, `c` und `d` immer die gleichen Werte. Es drängt sich die Frage auf, ob sich solch ein Funktionsaufruf nicht so vereinfachen lässt, dass nur der eigentlich interessante Parameter `a` übergeben werden muss. Und genau hier setzt die Funktion `partial` des Moduls `functools` ein.

functools.partial(func[, *args][, **kwargs])

Die Funktion `partial` bekommt ein Funktionsobjekt übergeben, dessen Schnittstelle vereinfacht werden soll. Zusätzlich werden die feststehenden Positions- und Schlüsselwortparameter der Funktion *func* übergeben.

Die Funktion `partial` gibt jetzt ein neues Funktionsobjekt zurück, dessen Schnittstelle der von *func* entspricht, die jedoch um die in *args* (Arguments) und *kwargs* (Keyword Arguments) angegebenen Parameter erleichtert wurde. Bei einem Aufruf des zurückgegebenen Funktionsobjekts werden diese feststehenden Parameter wieder in die Parameterliste geschrieben.

Dies soll das folgende Beispiel anhand der oben definierten Funktion `f` demonstrieren:

```
>>> def f(a, b, c, d):
...     print("{0} {1} {2} {3}".format(a,b,c,d))
...
>>> import functools
>>> f_neu = functools.partial(f, b="du", c="schöne", d="Welt")
>>> f_neu("Hallo")
Hallo du schöne Welt
>>> f_neu("Tschüss")
Tschüss du schöne Welt
```

Zunächst wird die Funktion `f` definiert, die vier Parameter akzeptiert und diese hintereinander auf dem Bildschirm ausgibt. Da die letzten drei Parameter dieser Schnittstelle in unserem Programm immer gleich sind, möchten wir sie nicht immer wiederholen und vereinfachen die Schnittstelle mittels `partial`.

Dazu rufen wir die Funktion `partial` auf und übergeben das Funktionsobjekt von `f` als ersten Parameter. Danach folgen die drei feststehenden Werte für die Parameter `b`, `c` und `d` in Form von Schlüsselwortparametern. Die Funktion `partial` gibt ein Funktionsobjekt zurück, das der Funktion `f` mit vereinfachter Schnittstelle entspricht. Dieses Funktionsobjekt kann, wie im Beispiel zu sehen, mit einem einzigen Parameter aufgerufen werden, während dieser Parameter daraufhin offensichtlich zusammen mit seinen drei vorgegebenen Parametern beim Funktionsaufruf von `f` ankommt.

Abgesehen von Schlüsselwortparametern können Sie der Funktion `partial` auch Positionsparameter übergeben, die dann ebenfalls als solche an die zu vereinfachende Funktion weitergegeben werden. Dabei ist zu beachten, dass die feststehenden Parameter dann am Anfang der Funktionsschnittstelle stehen müssen. Dazu folgendes Beispiel:

```
>>> def f(a, b, c, d):
...     print("{0} {1} {2} {3}".format(a,b,c,d))
...
>>> f_neu = functools.partial(f, "Hallo", "du", "schöne")
>>> f_neu("Welt")
Hallo du schöne Welt
>>> f_neu("Frau")
Hallo du schöne Frau
```

Die ersten drei Parameter der Funktion `f` sind immer gleich und sollen mithilfe der Funktion `partial` als Positionsparameter festgelegt werden. Das resultierende Funktionsobjekt `f_neu` kann mit einem Parameter aufgerufen werden, der beim daraus resultierenden Funktionsaufruf von `f` neben den drei festen Parametern als vierter übergeben wird.

26.3 Versteckte Passworteingaben – `getpass`

Das Modul `getpass` ist sehr überschaubar und ermöglicht vor allem das komfortable Einlesen eines Passworts über die Tastatur. Im Folgenden werden die im Modul `getpass` enthaltenen Funktionen erklärt. Um die Beispiele ausführen zu können, müssen Sie zuvor das Modul einbinden:

```
>>> import getpass
```

`getpass.getpass([prompt[, stream]])`

Die Funktion `getpass` liest, ähnlich wie etwa `raw_input`, eine Eingabe vom Benutzer ein und gibt diese als String zurück. Der Unterschied zu `raw_input` besteht allerdings darin, dass `getpass` zur Eingabe von Passwörtern gedacht ist. Das bedeutet, dass die Eingabe des Benutzers unter Verwendung von `getpass` verdeckt geschieht, also in der Konsole nicht angezeigt wird.

Über den optionalen Parameter *prompt* kann der Text angegeben werden, der den Benutzer zur Eingabe des Passworts auffordert. Der Parameter ist mit "Password: " vorbelegt. Für den zweiten optionalen Parameter, *stream*, kann ein datei-ähnliches Objekt übergeben werden, in das die Aufforderung *prompt* geschrieben wird. Das funktioniert nur unter Unix-ähnlichen Betriebssystemen, unter Windows wird der Parameter ignoriert.

```
>>> s = getpass.getpass("Ihr Passwort bitte: ")
Ihr Passwort bitte:
>>> print(s)
Dies ist mein Passwort
```

`getpass.getuser()`

Die Funktion `getuser` gibt den Namen zurück, mit dem sich der aktuelle Benutzer im Betriebssystem eingeloggt hat.

```
>>> getpass.getuser()
'Benutzername'
```

26.4 Kommandozeilen-Interpreter – `cmd`

Das Modul `cmd` bietet eine einfache und abstrahierte Schnittstelle zum Schreiben eines zeilenorientierten Kommandointerpreters. Unter einem zeilenorientierten Kommandointerpreter versteht man eine interaktive Konsole, in der zeilenweise Kommandos eingegeben und direkt nach Bestätigung der Eingabe interpretiert werden. Der interaktive Modus ist ein bekanntes Beispiel für solch einen Kom-

mandointerpreter. In einem eigenen Projekt ließe sich `cmd` beispielsweise für eine Administrator-Konsole verwenden.

Das Modul `cmd` enthält die Klasse `Cmd`, die als Basisklasse für eigene Kommandointerpreter verwendet werden kann und dafür ein grobes Gerüst bereitstellt. Da `Cmd` als Basisklasse gedacht ist, ergibt es keinen Sinn, die Klasse direkt zu instanziiieren. Das folgende Beispielprojekt verwendet die Klasse `Cmd`, um eine rudimentäre Konsole zu erstellen. Die Konsole soll die beiden Kommandos `date` und `time` verstehen und jeweils das aktuelle Datum bzw. die aktuelle Uhrzeit ausgeben.

```
import cmd
import time

class MeineKonsole(cmd.Cmd):

    def __init__(self):
        cmd.Cmd.__init__(self)
        self.prompt = ">>> "

    def do_date(self, prm):
        d = time.localtime()
        print("Heute ist der {0}.{1}.{2}".format(d[2],d[1],d[0]))
        return False

    def help_date(self):
        print("Gibt das aktuelle Datum aus")

    def do_time(self, prm):
        z = time.localtime()
        print("Es ist {0}:{1}:{2} Uhr".format(z[3], z[4], z[5]))
        return False

    def do_exit(self, prm):
        print("Auf Wiedersehen")
        return True
```

Im Beispiel wurde die Klasse `MeineKonsole` definiert, die von `cmd.Cmd` abgeleitet ist. Im Konstruktor der Klasse wird die Basisklasse initialisiert und das Attribut `self.prompt` gesetzt. Dieses Attribut stammt von der Basisklasse und referenziert den String, der zur Eingabe eines Kommandos auffordern soll.

Unsere Konsole soll insgesamt drei Kommandos unterstützen: `date` zum Ausgeben des aktuellen Datums, `time` zum Ausgeben der aktuellen Uhrzeit und `exit` zum Beenden der Konsole. Um ein Kommando in einer `cmd.Cmd`-Konsole zu implementieren, wird einfach eine Methode `do_kommando` angelegt, wobei `kommando` durch den Namen des jeweiligen Kommandos ersetzt werden muss. In diesem Sinne finden Sie in der Klasse `MeineKonsole` die Methoden `do_date`, `do_time`

und `do_exit` für die drei verfügbaren Kommandos. Jede dieser Methoden wird aufgerufen, wenn das Kommando vom Benutzer eingegeben wurde, und bekommt als einzigen Parameter `prm` den String übergeben, den der Benutzer hinter das Kommando geschrieben hat. Die Beispielimplementation der Methoden ist denkbar einfach und braucht in dieser Stelle nicht näher erläutert zu werden.

Wichtig ist aber, dass eine Kommandomethode anhand des Rückgabewerts angibt, ob die Konsole nach diesem Kommando noch weitere Kommandos annehmen soll. Wenn die Methode `False` zurückgibt, werden weitere Kommandos entgegengenommen. Bei einem Rückgabewert von `True` wird die Kommandoschleife beendet. Beachten Sie, dass der Rückgabewert `False` einiger Methoden im obigen Beispiel eigentlich überflüssig ist, da eine Funktion oder Methode ohne Rückgabewert implizit `None` zurückgibt und der Wahrheitswert von `None` `False` ist. Dennoch ist die entsprechende `return`-Anweisung zu Demonstrationszwecken im Quellcode enthalten.

Zusätzlich zu den Kommandomethoden existiert eine Methode `help_date` als Beispielimplementation der interaktiven Hilfe, die die Klasse `cmd.Cmd` bereitstellt. Gibt der Benutzer der Konsole ein Fragezeichen oder den Befehl `help`, gefolgt von einem der verfügbaren Kommandonamen, ein, wird die Methode `help_kommando` mit dem entsprechenden Kommandonamen aufgerufen. Diese gibt dann einen kurzen erklärenden Text zu dem jeweiligen Kommando aus.

Um den obigen Code zu einem vollwertigen Programm zu ergänzen, muss die Klasse instantiiert und die Kommandoschleife durch Aufruf der Methode `cmdloop` gestartet werden:

```
konsole = MeineKonsole()
konsole.cmdloop()
```

Nach dem Starten des Programms wird der Benutzer durch Ausgabe des Prompts `>>>` dazu aufgefordert, ein Kommando einzugeben. Eine Beispielsitzung in unserer Konsole sieht folgendermaßen aus:

```
>>> date
Heute ist der 6.1.2009
>>> time
Es ist 19:26:50 Uhr
>>> time
Es ist 19:26:54 Uhr
>>> exit
Auf Wiedersehen
```

Die Hilfetexte der Kommandos können folgendermaßen angezeigt werden:

```

>>> help date
Gibt das aktuelle Datum aus
>>> ? date
Gibt das aktuelle Datum aus
>>> help
Documented commands (type help <topic>):
=====
date

Undocumented commands:
=====
exit  help  time

```

Der Befehl `help` ohne Parameter gibt eine Liste aller dokumentierten und undokumentierten Kommandos Ihrer Konsole aus.

Das hier vorgestellte Beispielprogramm versteht sich als eine einfache Möglichkeit, die Klasse `Cmd` zu verwenden. Neben der hier gezeigten Funktionalität bietet die Klasse eine Fülle von Möglichkeiten, um das Verhalten der Konsole genau an Ihre Bedürfnisse anzupassen. Sollte Ihr Interesse an dem Modul `cmd` geweckt sein, bietet sich die Python-Dokumentation als Referenz für alle Methoden und Attribute der Klasse an.

*»Mehr als die Vergangenheit interessiert mich die Zukunft,
denn in ihr gedenke ich zu leben.«*
– Albert Einstein

27 Von Python 2.6 nach Python 3.0

Im Dezember 2008 ist Python in der Version 3.0 erschienen. Das Besondere an dieser Version ist, dass sie nicht mehr abwärtskompatibel mit älteren Python-Versionen ist. Das bedeutet, dass ein Programm, das für Python 2 geschrieben wurde, mit großer Wahrscheinlichkeit nicht unter Python 3 lauffähig ist. Der Bruch mit der Abwärtskompatibilität erlaubte es den Python-Entwicklern, lange mitgeschleppte Unschönheiten und Inkonsistenzen aus der Sprache zu entfernen und so eine rundum erneuerte Version von Python zu veröffentlichen.

Alles in allem sind die mit Python 3.0 eingeführten Änderungen nicht dramatisch, es bedarf jedoch einer gewissen Umgewöhnung, um von Python 2 auf Python 3 umzusteigen. Aus diesem Grund besprechen wir in diesem Kapitel übersichtlich die wichtigsten Unterschiede zwischen Python 2 und 3.

Bei Python 3.0 ist außerdem ein Tool namens `2to3` enthalten, das das Umstellen eines größeren Projektes von Python 2 auf 3 erleichtert, indem es den Quellcode zu großen Teilen automatisch konvertiert. Die Handhabung dieses Programms erklären wir im zweiten Abschnitt.

27.1 Die wichtigsten Unterschiede

Im Folgenden werden die wichtigsten Unterschiede zwischen den Python-Versionen 2.6 und 3.0 aufgeführt. Beachten Sie, dass mit Python 3.0 auch viele subtile Änderungen, beispielsweise an den Schnittstellen vieler Module der Standardbibliothek, durchgeführt wurden, die hier naheliegenderweise nicht alle erläutert werden können. Antworten auf solch detaillierte Fragen finden Sie in den Online-Dokumentationen zu Python 2.6 und 3.0, die beide über die Python-Website <http://www.python.org> erreichbar sind.

27.1.1 Ein-/Ausgabe

In puncto Ein-/Ausgabe gab es zwei auffällige, aber schnell erklärte Änderungen:

- Das Schlüsselwort `print` aus Python 2.6 ist einer Built-in Function gleichen Namens gewichen. In der Regel brauchen hier also nur Klammern um den auszugebenden Ausdruck ergänzt zu werden:

Python 2.6	Python 3.0
<pre>>>> print "Hallo Welt" Hallo Welt</pre>	<pre>>>> print("Hallo Welt") Hallo Welt</pre>
<pre>>>> print "ABC", "DEF", 2+2 ABC DEF 4</pre>	<pre>>>> print("ABC", "DEF", 2+2) ABC DEF 4</pre>
<pre>>>> print >> f, "Dateien"</pre>	<pre>>>> print("Dateien", file=f)</pre>
<pre>>>> for i in range(3): ... print i, ... 0 1 2</pre>	<pre>>>> for i in range(3): ... print(i, end=" ") ... 0 1 2</pre>

Tabelle 27.1 `print` ist jetzt eine Funktion

Nähere Informationen, insbesondere zu den Schlüsselwortparametern der Built-in Function `print`, finden Sie in Abschnitt 10.7 unter »`print`«.

- Die zweite auffällige Änderung bezüglich Ein-/Ausgabe betrifft die Built-in Function `input`. Die `input`-Funktion aus Python 3.0 entspricht der `raw_input`-Funktion aus Python 2.6. Eine Entsprechung für die `input`-Funktion aus Python 2.6 gibt es in Python 3.0 als Built-in Function nicht, doch ihr Verhalten kann mithilfe von `eval` nachgebildet werden:

Python 2.6	Python 3.0
<pre>>>> input("Ihr Wert: ") Ihr Wert: 2**5 32</pre>	<pre>>>> eval(input("Ihr Wert: ")) Ihr Wert: 2**5 32</pre>
<pre>>>> raw_input("Ihr Wert: ") Ihr Wert: 2**5 '2**5'</pre>	<pre>>>> input("Ihr Wert: ") Ihr Wert: 2**5 '2**5'</pre>

Tabelle 27.2 `input` vs. `raw_input`

Dies waren die beiden gewöhnungsbedürftigsten Änderungen, was die Bildschirmausgabe bzw. Tastatureingabe betrifft. Bezogen auf die `input`-Funktion wurde damit eine tückische Falle beseitigt. Gerade Python-Neulingen hat das machte unerwartete Verhalten der `input`-Funktion in Python 2 zu schaffen.

27.1.2 Iteratoren

Obwohl Python 2.6 das Iterator-Konzept bereits unterstützte, gaben viele Funktionen, die für gewöhnlich zum Iterieren über eine bestimmte Menge verwendet werden, eine Liste der Elemente dieser Menge zurück, so beispielsweise die prominente Funktion `range`. Üblicherweise wurde diese Liste nun aber nur in einer `for`-Schleife durchlaufen. Dies kann durch Verwendung von Iteratoren wesentlich eleganter und speicherschonender durchgeführt werden. Aus diesem Grund geben viele der Funktionen und Methoden, die in Python 2.6 noch eine Liste zurückgaben, in Python 3.0 einen Iterator oder ein auf die Gegebenheiten zugeschnittenes iterierbares Objekt zurück.

Um diese Objekte in eine Liste zu überführen, können sie einfach der Built-in Function `list` übergeben werden.

Damit verhält sich die `range`-Funktion aus Python 3.0 gerade so wie die `xrange`-Funktion aus Python 2.6. Ein Aufruf von `list(range(...))` in Python 3.0 ist äquivalent zur `range`-Funktion aus Python 2.6.

Abgesehen von `range` sind von dieser Änderung die Built-in Functions `map`, `filter` und `zip` betroffen, die nun einen Iterator zurückgeben. Die Methoden `keys`, `items` und `values` eines Dictionarys geben jetzt ein sogenanntes `view`-Objekt zurück, das ebenfalls iterierbar ist. Die Methoden `iterkeys`, `iteritems` und `itervalues` eines Dictionarys sowie die Built-in Function `xrange` aus Python 2.6 existieren in Python 3.0 nicht mehr.

27.1.3 Strings

Die wohl grundlegendste Änderung in Python 3.0 ist die Umdeutung des Datentyps `str`. In Python 2.6 existierten zwei Datentypen für Strings: `str` und `unicode`. Während ersterer zum Speichern beliebiger Byte-Folgen verwendet werden konnte, war letzterer für Unicode-Text zuständig.

In Python 3.0 ist der Datentyp `str` nun ausschließlich für Text zuständig und mit dem `unicode`-Datentyp aus Python 2.6 vergleichbar. Zum Speichern von Byte-Folgen gibt es in Python 3.0 die Datentypen `bytes` und `bytearray`, wobei es sich bei `bytes` um einen unveränderlichen und bei `bytearray` um einen veränderlichen Datentyp handelt. Beachten Sie, dass in Python 3.0 weder das `u`-Literal für Unicode-Strings noch der Datentyp `unicode` existiert.

Die Datentypen `bytes` und `str` sind in Python 3.0 klarer voneinander abgegrenzt worden, als es bei den Datentypen `str` und `unicode` in Python 2.6 der Fall war. So ist es beispielsweise nicht mehr möglich, einen String und einen `bytes`-String ohne explizites Kodieren bzw. Dekodieren zusammenzufügen.

Die folgende Tabelle weist die Unterschiede zwischen `str` in Python 2.6 und `str` in Python 3.0 exemplarisch auf:

Python 2.6	Python 3.0
<pre>>>> u = u"Ich bin Unicode" >>> u u'Ich bin Unicode' >>> u.encode("ascii") 'Ich bin Unicode'</pre>	<pre>>>> u = "Ich bin Unicode" >>> u 'Ich bin Unicode' >>> u.encode("ascii") b'Ich bin Unicode'</pre>
<pre>>>> a = "Ich bin ASCII" >>> a.decode() u'Ich bin ASCII'</pre>	<pre>>>> a = b"Ich bin ASCII" >>> a.decode() 'Ich bin ASCII'</pre>
<pre>>>> "abc" + u"def" u'abcdef'</pre>	<pre>>>> b"abc" + "def" Traceback (most recent call last): [...] TypeError: can't concat bytes to str</pre>

Tabelle 27.3 Strings

Beachten Sie, dass insbesondere die stärkere Abgrenzung von `str` und `bytes` in Python 3.0 Auswirkungen auf die Standardbibliothek hat. Beispielsweise können Sie zur Netzwerkkommunikation ausschließlich `bytes`-Strings verwenden. Wichtig ist auch, dass der Typ der aus einer Datei eingelesenen Daten nun vom Modus abhängt, in dem die Datei geöffnet wurde. Der Unterschied zwischen Binär- und Textmodus ist in Python 3.0 also auch unter Betriebssystemen von Interesse, die diese beiden Modi von sich aus gar nicht unterscheiden.

27.1.4 Ganze Zahlen

In Python 2.6 existierten zwei Datentypen zum Speichern von ganzen Zahlen: `int` für Zahlen im 32- bzw. 64-Bit-Zahlenbereich und `long` für Zahlen beliebiger Größe. In Python 3.0 gibt es nur noch einen namens `int`, der sich aber wie `long` aus Python 2.6 verhält. Die Unterscheidung zwischen `int` und `long` war auch in Python 2.6 für den Programmierer im Wesentlichen schon uninteressant, da die beiden Datentypen automatisch ineinander konvertiert wurden.

Eine zweite Änderung erfolgt in Bezug auf die Division ganzer Zahlen. In Python 2.6 wurde in diesem Fall eine ganzzahlige Division (Integer-Division) durchgeführt, das Ergebnis war also wieder eine ganze Zahl. In Python 3.0 ist das Ergebnis der Division zweier Ganzzahlen eine Gleitkommazahl. Für die Integer-Division existiert hier der Operator `//`:

Python 2.6	Python 3.0
>>> 10 / 4 2	>>> 10 / 4 2.5
>>> 10 // 4 2	>>> 10 // 4 2
>>> 10.0 / 4 2.5	>>> 10.0 / 4 2.5

Tabelle 27.4 Integer-Division

27.1.5 Exception Handling

Beim Werfen und Fangen von Exceptions wurden kleinere syntaktische Änderungen durchgeführt. Die alte und die neue Syntax werden in folgender Tabelle anhand eines Beispiels einander gegenübergestellt:

Python 2.6	Python 3.0
<pre>try: raise SyntaxError, "Hilfe" except SyntaxError, e: print e.args</pre>	<pre>try: raise SyntaxError("Hilfe") except SyntaxError as e: print(e.args)</pre>

Tabelle 27.5 Exception Handling

Dabei ist anzumerken, dass die in obigem Beispiel unter Python 3.0 für das Werfen der Exception verwendete Syntax unter Python 2.6 auch schon funktionierte. Die unter Python 2.6 angegebene Syntax wurde aber mit Python 3.0 aus der Sprache entfernt.

27.1.6 Standardbibliothek

Mit Python 3.0 wurde auch in der Standardbibliothek gründlich aufgeräumt. Viele Module, die kaum verwendet wurden, wurden entfernt, andere umbenannt oder mit anderen zu Paketen zusammengefasst. Die folgende Tabelle listet die Module auf, die in Python3.0 umbenannt wurden:

Python 2.6	Python 3.0
_winreg	winreg
ConfigParser	configparser
Copy_reg	copyreg

Tabelle 27.6 Umbenannte Module der Standardbibliothek

Python 2.6	Python 3.0
cPickle	_pickle
Queue	queue
SocketServer	socketserver
markupbase	_markupbase
repr	reprlib
test.test_support	test.support
thread	_thread

Tabelle 27.6 Umbenannte Module der Standardbibliothek (Forts.)

Die meisten der oben aufgeführten Module werden in diesem Buch nicht thematisiert. Nähere Informationen zu ihnen finden Sie aber in der Online-Dokumentation von Python.

Neben umbenannten Modulen wurden auch einige thematisch zusammengehörige Module zu Paketen zusammengefasst. Diese sind in der folgenden Tabelle aufgelistet:

Paket in Python 3.0	Module aus Python 2.6
dbm	anydbm, dbhash, dbm, dumbdbm, gdbm, whichdb
html	HTMLParser, htmlentitydefs
http	httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib
tkinter	abgesehen von turtle alle Module, die etwas mit Tkinter zu tun haben
urllib	urllib, urllib2, urlparse, robotparse
xmlrpc	xmlrpclib, DocXMLRPCServer, SimpleXMLRPCServer

Tabelle 27.7 Verschobene Module der Standardbibliothek

27.1.7 Neue Sprachelemente in Python 3.0

Mit Python 3.0 sind eine Reihe neuer Sprachelemente in die Sprache aufgenommen worden, die wir in der folgenden Tabelle kurz auflisten möchten. In der zweiten Spalte finden Sie einen Verweis, in welchem Abschnitt dieses Buches die Neuerung besprochen wird:

Sprachelement	Abschnitt
Dict Comprehensions und Set Comprehensions	13.2, »Comprehensions«
Function Annotations	13.9, »Function Annotations«
Literal für Set	8.7, »Mengen«
Literal für Oktal- und Binärzahlen	8.3.1, »Ganzzahlen – int«
nonlocal-Anweisung	10.5.2, »Zugriff auf übergeordnete Namensräume – nonlocal«
with-Anweisung	13.8, »Die with-Anweisung«

Tabelle 27.8 Neue Sprachelemente

27.2 Automatische Konvertierung

Um die Migration von Python 2.6 nach Python 3.0 auch bei größeren Projekten zu vereinfachen, gibt es in der Python 3.0-Distribution ein Tool namens `2to3`, das wir in diesem Abschnitt besprechen möchten. Das Tool `2to3` finden Sie im Unterverzeichnis `Tools/scripts` Ihrer Python-Distribution.

Die Verwendung von `2to3` soll exemplarisch an folgendem Python 2.6-Beispielprogramm demonstriert werden:

```
def getInput(n):
    liste = []
    for i in xrange(n):
        try:
            z = int(raw_input("Bitte eine Zahl eingeben: "))
        except ValueError, e:
            raise ValueError("Das ist keine Zahl!")
        liste.append(z)
    return liste

try:
    res = getInput(5)
    print res
except ValueError, e:
    print e.args[0]
```

Dieses Programm liest mithilfe der Funktion `getInput` fünf Zahlen vom Benutzer ein und gibt eine mit diesen Zahlen gefüllte Liste aus. Wenn der Benutzer etwas eingibt, was keine Zahl ist, beendet sich das Programm mit einer Fehlermeldung. Sie sehen sofort, dass sich dieses Programm so nicht unter Python 3.0 ausführen

lässt. Die Aufrufe von `xrange`, `raw_input` sowie die beiden `except`-Anweisungen verhindern dies.

Bereits bei den obigen 14 Quellcodezeilen ist es mühselig, den Code per Hand mit Python 3.0 kompatibel zu machen. Stellen Sie sich diese Arbeit einmal für ein größeres Projekt vor! Doch glücklicherweise gibt es `2to3`. Wir rufen `2to3` einmal mit dem Namen unseres Python-Programms als einzigem Parameter auf. Das Ergebnis sieht folgendermaßen aus:

```

--- test.py (original)
+++ test.py (refactored)
@@ -1,15 +1,15
     def getInput(n):
         liste = []
-        for i in xrange(n):
+        for i in range(n):
+            try:
-                z = int(raw_input("Bitte eine Zahl eingeben: "))
-            except ValueError, e:
+                z = int(input("Bitte eine Zahl eingeben: "))
+            except ValueError as e:
+                raise ValueError("Das ist keine Zahl!")
+                liste.append(z)
         return liste
     try:
         res = getInput(5)
-        print res
-    except ValueError, e:
-        print e.args[0]
+        print(res)
+    except ValueError as e:
+        print(e.args[0])

```

Das Konvertierungsprogramm ändert Ihre angegebenen Quellcode-dateien standardmäßig nicht, sondern produziert nur einen `diff`-Ausdruck. Das ist eine spezielle Beschreibungssprache für die Unterschiede zwischen zwei Textstücken. Diesen `diff`-Ausdruck können Sie beispielsweise mithilfe des Unix-Programms `patch` in Ihre Quelldatei einpflegen. Alternativ erlauben Sie es dem `2to3`-Script über den Kommandoschalter `-w`, die angegebene Quelldatei direkt zu modifizieren. Der ursprüngliche Python 2.6-Code wird dabei als *dateiname.py.bak* gesichert. Wenn `2to3` mit dem Schalter `-w` und unserem obigen Beispielquellcode gefüttert wird, sieht der konvertierte Code hinterher so aus:

```

def getInput(n):
    liste = []

```

```

for i in range(n):
    try:
        z = int(eval(input("Bitte eine Zahl eingeben: ")))
    except ValueError as e:
        raise ValueError("Das ist keine Zahl!")
    liste.append(z)
return liste

try:
    res = getInput(5)
    print(res)
except ValueError as e:
    print(e.args[0])

```

Sie sehen, dass die eingangs angesprochenen Stellen geändert wurden, und werden feststellen, dass der übersetzte Code problemlos unter Python 3.0 lauffähig ist.

Statt einer einzelnen Programmdatei können Sie dem `2to3`-Script auch eine ganze Liste von Dateien oder Ordnern übergeben. Wenn Sie einen Ordner übergeben haben, wird jede Quelldatei in ihm oder einem seiner Unterordner konvertiert.

Zum Schluss möchten wir noch auf die wichtigsten Kommandozeilenschalter zu sprechen kommen, mit deren Hilfe Sie das Verhalten von `2to3` an Ihre Bedürfnisse anpassen können:

Schalter	Alternativ	Beschreibung
<code>-d</code>	<code>--doctests_only</code>	Ist dieser Schalter gesetzt, so werden ausschließlich die in der angegebenen Quelldatei enthaltenen Doctests nach Python 3.0 konvertiert. Standardmäßig werden Doctests nicht angerührt. Näheres über Doctests erfahren Sie in Abschnitt 21.5.1, »Testfälle in Docstrings – doctest«.
<code>-f FIX</code>	<code>--fix=FIX</code>	Mit dieser Option geben Sie vor, welche sogenannten Fixes angewandt werden sollen. Bei einem Fix handelt es sich um eine bestimmte Ersetzungsregel, beispielsweise das Ersetzen von <code>xrange</code> durch <code>range</code> .
<code>-x NOFIX</code>	<code>--nofix=NOFIX</code>	Das Gegenstück zu <code>-f</code> . Hier bestimmen Sie, welche Fixes nicht angewandt werden dürfen.
<code>-l</code>	<code>--list-fixes</code>	Durch Setzen dieses Schalters erhalten Sie eine Liste aller verfügbaren Fixes.

Tabelle 27.9 Kommandozeilenoptionen von `2to3`

Schalter	Alternativ	Beschreibung
-p	--print-function	Wenn dieser Schalter gesetzt ist, werden <code>print</code>-Anweisungen nicht konvertiert. Das ist nützlich, wenn Sie <code>print</code> bereits in Python 2.6 wie eine Funktion geschrieben oder den entsprechenden Future Import <code>print_function</code> eingebunden haben. Das Programm <code>2to3</code> kann nicht von selbst entscheiden, ob es eine <code>print</code>-Anweisung mit Klammern versehen muss oder nicht.
-w	--write	Ist dieser Schalter gesetzt, so werden die Änderungen direkt in die untersuchte Quelldatei geschrieben. Ein Backup wird unter <i>dateiname.py.bak</i> angelegt.
-n	--nobackups	Wenn dieser Schalter gesetzt ist, wird auf das Anlegen der Backup-Datei verzichtet.

Tabelle 27.9 Kommandozeilenoptionen von 2to3 (Forts.)

*»Alles, was einen Anfang hat, hat auch ein Ende –
und meistens hat das, was ein Ende hat,
auch eine Fortsetzung.«
– Sprichwort*

A Anhang

A.1 Entwicklungsumgebungen

Wie Sie wissen, ist bei Python bereits die rudimentäre Entwicklungsumgebung IDLE enthalten, die die grundlegenden Eigenschaften einer guten IDE hat. Dazu gehören unter anderem *Syntax Highlighting*, *Code Completion* und ein *integrierter Debugger*.

Allerdings ist IDLE in vielerlei Hinsicht unkomfortabel, weswegen wir an dieser Stelle eine Übersicht über andere IDEs für Python geben möchten.

Beachten Sie, dass wir hier aus naheliegenden Gründen nur die bekanntesten IDEs besprechen werden:

- ▶ Eclipse
- ▶ Eric4
- ▶ Komodo IDE
- ▶ Wing IDE

Es existiert eine große Anzahl weiterer IDEs bzw. Texteditoren für Python. Eine umfangreiche Liste finden Sie im offiziellen Python-Wiki auf <http://wiki.python.org/moin> unter dem Stichwort »Python Editors«.

Alle hier aufgeführten Entwicklungsumgebungen werden auf der angegebenen Website als Voll- oder Testversion zum Download angeboten.

A.1.1 Eclipse

Website: <http://www.eclipse.org>

Python-Plugin PyDev: <http://pydev.sourceforge.net>

Plattformen: Windows, Linux, Mac OS X

Python 3-Unterstützung: ab Version 1.4

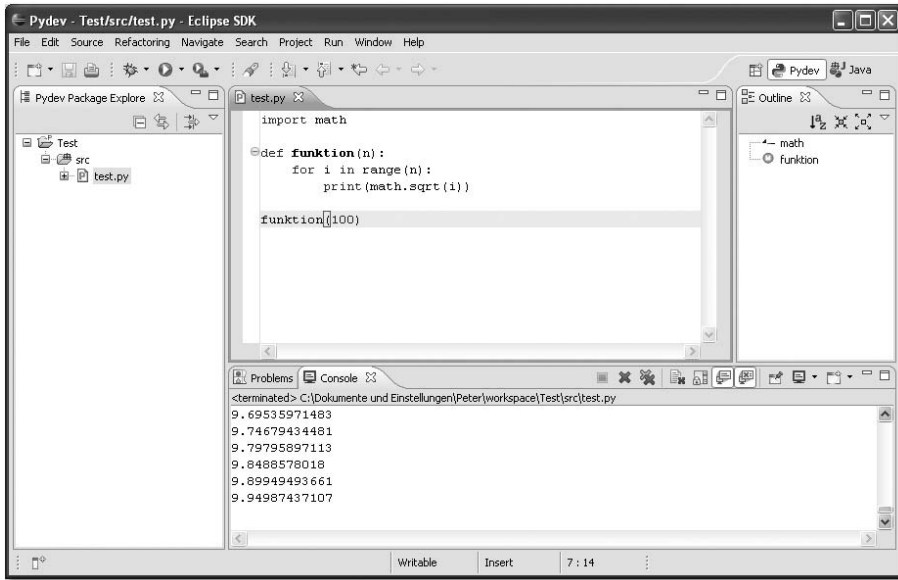


Abbildung A.1 Eclipse und PyDev im Einsatz

Die von IBM entwickelte IDE *Eclipse* war ursprünglich für Java gedacht, kann aber inzwischen mithilfe von Plugins für viele Programmiersprachen verwendet werden. Eclipse wurde 2001 von IBM unter einer Open-Source-Lizenz freigegeben. Seit 2004 ist die *Eclipse Foundation* für die Weiterentwicklung der IDE verantwortlich. Eclipse erfreut sich großer Beliebtheit und kann mithilfe des *PyDev*-Plugins auch für die Entwicklung mit Python eingesetzt werden.

Das *PyDev*-Plugin verfügt über Quellcodevervollständigung, Syntaxüberprüfung und viele weitere Annehmlichkeiten. Zudem enthält *PyDev* einen umfangreichen integrierten Python-Debugger, der auch mit Threads sinnvoll umgehen kann.

Beachten Sie, dass Eclipse ein installiertes Java SDK benötigt. Näheres dazu finden Sie auf der Eclipse-Website.

A.1.2 Eric4

Website: <http://www.die-offenbachs.de/eric/>

Plattformen: Windows, Linux, Mac OS X

Python 3-Unterstützung: nein

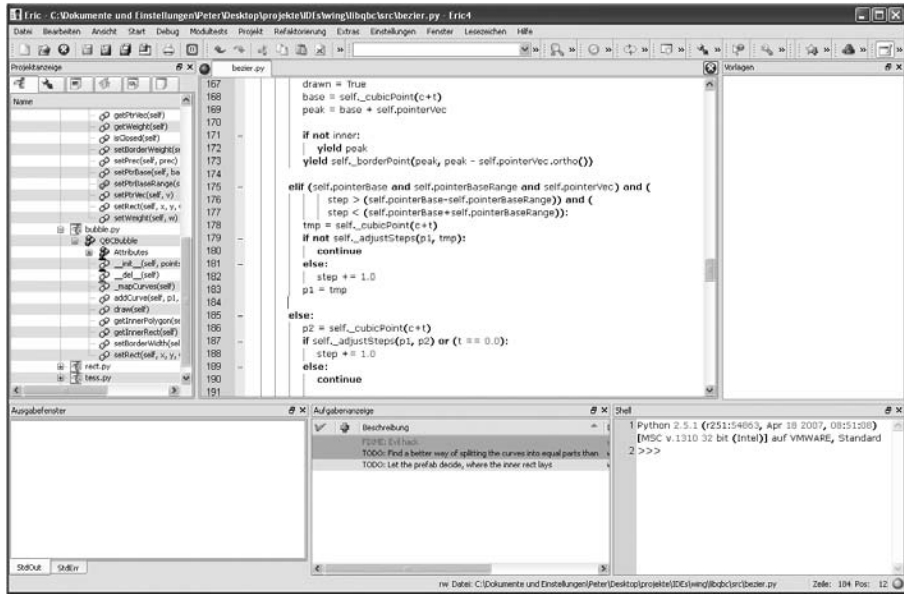


Abbildung A.2 Eric4 im Einsatz

Die freie Entwicklungsumgebung Eric4 ist selbst mit PyQt in Python geschrieben und spezialisiert sich auf PyQt4-Anwendungen.¹ Für die Entwicklung von Python-Programmen mit Qt-Oberfläche bietet die IDE viele interessante Hilfsmittel an.

Beachten Sie, dass zum Zeitpunkt der Drucklegung dieses Buchs noch keine zu Python 3 kompatible Version der Eric IDE verfügbar war.

¹ Analog dazu existiert *Eric3* für PyQt3-Anwendungen.

A.1.3 Komodo IDE

Website: <http://www.activestate.com>

Plattformen: Windows, Linux, Mac OS X

Python 3-Unterstützung: ab Version 5.1

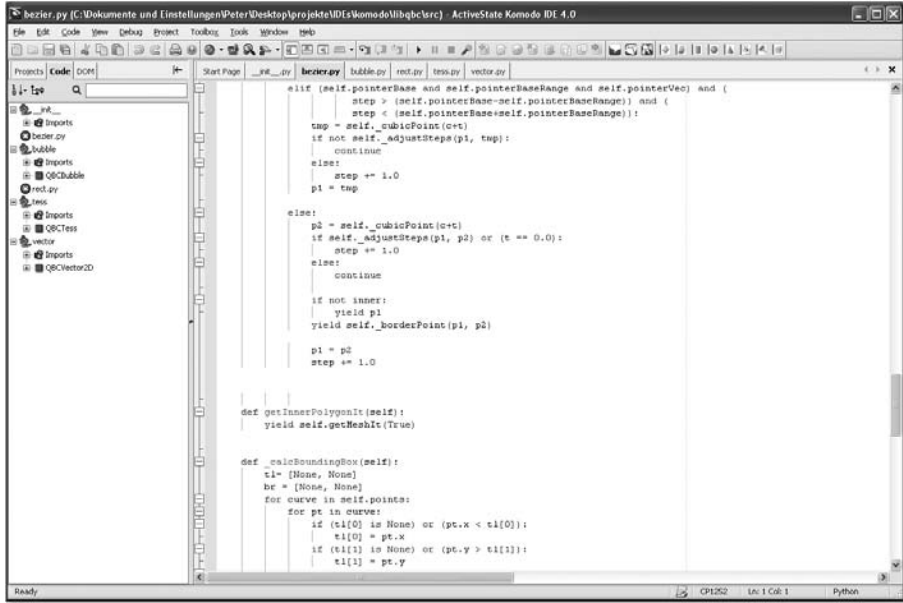


Abbildung A.3 Komodo IDE im Einsatz

Die kommerzielle Entwicklungsumgebung *Komodo IDE* wird von der kanadischen Firma *ActiveState* entwickelt und unterstützt neben Python noch viele andere Sprachen. Eine abgespeckte Version der Komodo IDE namens *Komodo Edit* kann von der Homepage der Firma kostenlos heruntergeladen werden.

Die Entwicklungsumgebung unterstützt eine *Codevervollständigung*, *Calltips*, *Syntaxüberprüfung* sowie einen komfortablen und umfassenden *grafischen Debugger*, der auch eine *Post-Mortem-Debugging*-Funktionalität beherrscht.

A.1.4 Wing IDE

Website: <http://www.wingware.com>

Plattformen: Windows, Linux, Mac OS X

Python 3-Unterstützung: geplant

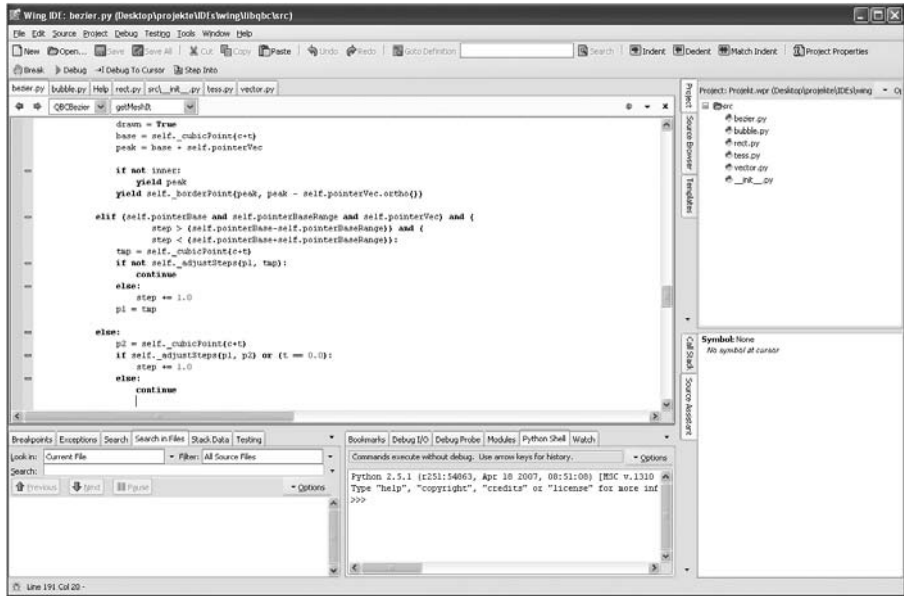


Abbildung A.4 Wing IDE im Einsatz

Die kommerzielle Entwicklungsumgebung *Wing IDE* wird von der amerikanischen Firma *Wingware* speziell für Python entwickelt und bietet komfortable Funktionen, die beim Programmieren helfen – darunter zum Beispiel einen Klassen- und Modulbrowser und die bereits von anderen IDEs bekannte *Autovervollständigung*. Zudem wartet die IDE mit einem *grafischen Debugger* auf, mit dem auch Multithread-Anwendungen auf Fehler untersucht werden können.

Grundsätzlich ist die Wing IDE in drei Versionen verfügbar, die sich in ihren Features unterscheiden. Eine stark abgespeckte Version können Sie kostenlos herunterladen und frei verwenden.

A.2 Reservierte Wörter

Die folgende Tabelle enthält Wörter, die nicht als Bezeichner verwendet werden dürfen, weil sie einem Schlüsselwort entsprechen.

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

Tabelle A.1 Liste reservierter Wörter

A.3 Operatorrangfolge

Die Operatorrangfolge legt fest, wie stark ein Operator im Vergleich zu den anderen bindet. Der zuerst aufgeführte Operator (**) bindet am stärksten, der letzte am schwächsten. Der zweiten Spalte der Tabelle können Sie die übliche Bedeutung des Operators entnehmen, das ist zumeist die Bedeutung des Operators bezogen auf ganze Zahlen.

Operator	Übliche Bedeutung
x ** y	y-te Potenz von x
~x	bitweises Komplement von x
+x, -x	positives oder negatives Vorzeichen
x * y	Produkt von x und y
x / y	Quotient von x und y
x % y	Rest bei ganzzahliger Division von x durch y
x // y	Ganzzahlige Division von x durch y
x + y	Summe von x und y
x - y	Differenz von x und y
x << n	bitweise Verschiebung um n Stellen nach links
x >> n	bitweise Verschiebung um n Stellen nach rechts

Tabelle A.2 Die Operatorrangfolge

Operator	Übliche Bedeutung
<code>x & y</code>	bitweises UND zwischen <code>x</code> und <code>y</code>
<code>x ^ y</code>	bitweises ausschließendes ODER zwischen <code>x</code> und <code>y</code>
<code>x y</code>	bitweises nicht ausschließendes ODER zwischen <code>x</code> und <code>y</code>
<code>x < y</code>	Ist <code>x</code> kleiner als <code>y</code> ?
<code>x <= y</code>	Ist <code>x</code> kleiner oder gleich <code>y</code> ?
<code>x > y</code>	Ist <code>x</code> größer als <code>y</code> ?
<code>x >= y</code>	Ist <code>x</code> größer oder gleich <code>y</code> ?
<code>x != y</code>	Ist <code>x</code> ungleich <code>y</code> ?
<code>x == y</code>	Ist <code>x</code> gleich <code>y</code> ?
<code>x is y</code>	Sind <code>x</code> und <code>y</code> identisch?
<code>x is not y</code>	Sind <code>x</code> und <code>y</code> nicht identisch?
<code>x in y</code>	Befindet sich <code>x</code> in <code>y</code> ?
<code>x not in y</code>	Befindet sich <code>x</code> nicht in <code>y</code> ?
<code>not x</code>	logische Negierung
<code>x and y</code>	logisches UND
<code>x or y</code>	logisches ODER

Tabelle A.2 Die Operatorrangfolge (Forts.)

A.4 Built-in Exceptions

In diesem Abschnitt finden Sie eine Auflistung aller in Python eingebauten Exception-Typen. Die kurze Beschreibung erläutert, in welchen Fällen der jeweilige Exception-Typ auftritt, und gibt somit eine Art Richtlinie, wann Sie einen der eingebauten Exception-Typen in Ihren Programmen verwenden sollten.

Abbildung 13.1 stellt die vollständige Vererbungshierarchie für die eingebauten Exception-Typen dar.

Beachten Sie, dass Sie eigene Exception-Typen von der Basisklasse `Exception` ableiten sollten. Selbstverständlich können Sie auch direkt von einer der eingebauten Exceptions erben, die ihrerseits von `Exception` erben.

Zunächst sehen Sie eine Tabelle mit allen Basisklassen, von denen verschiedene eingebaute Exceptions erben.

Exception	Erklärung
BaseException	Die Basisklasse aller eingebauten Exceptions. Diese Klasse stellt einen Großteil der Exception-Funktionalität bereit und wird in Abschnitt 13.1.1, »Eingebaute Exceptions«, behandelt.
Exception	Die Basisklasse aller »normalen« Exceptions. Spezielle Exceptions wie <code>SystemExit</code> oder <code>KeyboardInterrupt</code> erben nicht von <code>Exception</code> . Beachten Sie, dass alle selbstdefinierten Exception-Typen von <code>Exception</code> erben sollten.
ArithmeticError	Die Basisklasse aller Exception-Typen, die einen Fehler in einer arithmetischen Operation repräsentieren
LookupError	die Basisklasse aller Exceptions, die auftreten, wenn auf einen nicht vorhandenen Schlüssel oder Index zugegriffen wird
EnvironmentError	Die Basisklasse aller Exception-Typen, die auftreten, wenn ein Fehler außerhalb des Python-Systems passiert – beispielsweise wenn eine Dateioperation fehlschlägt. Das Attribut <code>args</code> der Exception-Instanz referenziert ein Tupel, das Fehlercode und -beschreibung enthält.

Tabelle A.3 Basisklassen für eingebaute Exceptions

Die folgende Tabelle listet alle eingebauten Exception-Typen auf, die von den vorherigen Typen erben und tatsächlich geworfen werden.

Exception	Erklärung
AssertionError	Wird geworfen, wenn eine <code>assert</code> -Anweisung fehlschlägt.
AttributeError	Wird geworfen, wenn auf ein nicht existierendes Attribut einer Instanz zugegriffen wird.
EOFError	Wird geworfen, wenn die Funktionen <code>input</code> oder <code>raw_input</code> EOF (<i>End-of-File</i>) signalisiert bekommen, ohne zuvor Daten eingelesen zu haben.
FloatingPointError	Wird geworfen, wenn eine Gleitkommaoperation fehlschlägt.
GeneratorExit	Wird geworfen, wenn die Methode <code>close</code> eines Generators gerufen wird. Diese Exception erbt direkt von <code>Exception</code> statt von <code>StandardError</code> .
IOError	Wird geworfen, wenn eine I/O-Operation, wie beispielsweise eine Bildschirmaus- oder -eingabe, fehlschlägt.

Tabelle A.4 Eingebaute Exception-Typen

Exception	Erklärung
ImportError	Wird geworfen, wenn eine <code>import</code> -Anweisung das angegebene Modul nicht finden kann oder der angegebene Name bei einer <code>from/import</code> -Anweisung nicht im Modul enthalten ist.
IndexError	Wird geworfen, wenn mit einem Index auf eine Sequenz zugegriffen wird, der außerhalb des gültigen Bereichs liegt.
KeyError	Wird geworfen, wenn mit einem Schlüssel auf ein Dictionary zugegriffen wird, der nicht in diesem vorhanden ist.
KeyboardInterrupt	Wird geworfen, wenn das Python-Programm per Tastenkombination (üblicherweise mit <code>Strg</code> + <code>C</code>) abgebrochen wird. Diese Exception erbt direkt von <code>BaseException</code> , so dass sie nicht irrtümlich abgefangen werden kann.
MemoryError	Wird geworfen, wenn nicht mehr genügend Speicher zur Ausführung einer Operation vorhanden ist.
NameError	Wird aufgerufen, wenn ein unbekannter lokaler oder globaler Bezeichner verwendet wird.
NotImplementedError	Basisklassen werfen diese Exception aus Methoden heraus, die von einer abgeleiteten Klasse überschrieben werden müssen.
OSError	Diese Exception wird vom Modul <code>os</code> geworfen und erbt von <code>EnvironmentError</code> .
OverflowError	Wird geworfen, wenn das Ergebnis einer arithmetischen Operation zu groß für den entsprechenden Datentyp ist.
ReferenceError	Wird geworfen, wenn auf ein Objekt zugegriffen wurde, das bereits von der Garbage Collection freigegeben wurde.
RuntimeError	Wird geworfen, wenn ein Fehler aufgetreten ist, der durch keinen der anderen Exception-Typen ausgedrückt wird. Das Attribut <code>args</code> enthält die genaue Fehlermeldung.
StopIteration	Wird von der Methode <code>next</code> eines Iterators geworfen, wenn kein nächstes Element existiert. Diese Exception erbt direkt von <code>Exception</code> statt von <code>StandardError</code> .
SyntaxError	Wird geworfen, wenn ein Syntaxfehler in einem Python-Programm, einer <code>exec</code> -Anweisung oder einem Aufruf von <code>eval</code> auftritt. Die Attribute <code>filename</code> , <code>lineno</code> , <code>offset</code> und <code>text</code> spezifizieren den Fehler genauer.

Tabelle A.4 Eingebaute Exception-Typen (Forts.)

Exception	Erklärung
SystemError	Wird geworfen, wenn ein interner Fehler im Python-Interpreter auftritt.
SystemExit	Wird von <code>sys.exit</code> geworfen und beendet das laufende Programm, wenn sie nicht abgefangen wird. Eine nicht abgefangene <code>SystemExit</code> -Exception verursacht keinen Traceback. Die Exception erbt direkt von <code>BaseException</code> , so dass sie nicht versehentlich abgefangen werden kann.
TypeError	Wird geworfen, wenn eine Operation auf einer Instanz durchgeführt wird, die einen dafür unpassenden Datentyp hat.
UnboundLocalError	Wird geworfen, wenn in einer Funktion oder Methode eine lokale Referenz bereits verwendet wird, bevor ihr eine Instanz zugewiesen wurde.
UnicodeError	Wird geworfen, wenn bei Encode- und Decode-Vorgängen im Zusammenhang mit Unicode Fehler auftreten.
UnicodeEncodeError	Wird geworfen, wenn ein Unicode-spezifischer Fehler bei einem Encode-Vorgang auftritt. Diese Exception erbt von <code>UnicodeError</code> .
UnicodeDecodeError	Wird geworfen, wenn ein Unicode-spezifischer Fehler bei einem Decode-Vorgang auftritt. Diese Exception erbt von <code>UnicodeError</code> .
UnicodeTranslateError	Wird geworfen, wenn ein Unicode-spezifischer Fehler bei einem Translate-Vorgang auftritt. Diese Exception erbt von <code>UnicodeError</code> .
ValueError	Wird geworfen, wenn eine Operation auf einer Instanz durchgeführt werden soll, die zwar den richtigen Typ, aber einen unpassenden Wert hat.
WindowsError	Wird geworfen, wenn ein Windows-spezifischer Fehler auftritt. Diese Exception erbt von <code>OSError</code> .
ZeroDivisionError	Wird bei einer Division durch null geworfen.

Tabelle A.4 Eingebaute Exception-Typen (Forts.)

Die folgende Tabelle listet alle eingebauten Warnungen auf. Das sind spezielle Exceptions, die beispielsweise auf die Verwendung eines veralteten Funktionsparameters hindeuten. Das Auftreten von Warnungen behindert die Ausführung des Programms nicht.

Exception	Erklärung
Warning	Basisklasse für alle Warnungen
UserWarning	Basisklasse für eigene Warnungen
DeprecationWarning	Wird bei der Verwendung eines als veraltet eingestuften Sprachfeatures geworfen.
PendingDeprecationWarning	Wird bei der Verwendung eines als zukünftig veraltet eingestuften Sprachfeatures geworfen.
SyntaxWarning	Wird bei problematischer Syntax geworfen.
RuntimeWarning	Wird bei bedenklichem Laufzeitverhalten geworfen.
FutureWarning	Wird bei der Verwendung von Sprachfeatures geworfen, die sich in Zukunft verändern werden.
ImportWarning	Wird bei problematischen import-Anweisungen geworfen.
UnicodeWarning	Wird bei Unicode-spezifischen Problemen geworfen.
BytesWarning	Wird bei bytes-spezifischen Problemen geworfen.

Tabelle A.5 Eingebaute Warnungen

A.5 Built-in Functions

Python enthält eine Menge Built-in Function, die aus didaktischen Gründen an verschiedenen Stellen des Buchs eingeführt wurden. Deshalb gibt es im Buch bislang keine Liste aller Built-in Functions. Die folgende Tabelle listet alle Built-in Functions mitsamt einer kurzen Beschreibung und einem Vermerk auf, wo die Funktion ausführlich behandelt wird. Beachten Sie, dass die Funktionen in dieser Tabelle ohne Parametersignaturen angegeben werden.

Built-in Function	Beschreibung	Erklärt auf
<code>__import__</code>	Bindet ein Modul oder Paket ein.	S. 232
<code>abs</code>	Berechnet den Betrag einer Zahl.	S. 201
<code>all</code>	Prüft, ob alle Elemente einer Sequenz <code>True</code> ergeben.	S. 201
<code>any</code>	Prüft, ob mindestens ein Element einer Sequenz <code>True</code> ergibt.	S. 201
<code>ascii</code>	Erzeugt einen druckbaren String, der das übergebene Objekt beschreibt. Dabei werden Sonderzeichen maskiert, so dass die Ausgabe nur ASCII-Zeichen enthält.	S. 202

Tabelle A.6 Built-in Functions in Python

Built-in Function	Beschreibung	Erklärt auf
<code>bin</code>	Gibt einen String zurück, der die übergebene Ganzzahl als Binärzahl darstellt.	S. 202
<code>bool</code>	Erzeugt einen booleschen Wert.	S. 202
<code>bytearray</code>	Erzeugt eine neue <code>bytearray</code> -Instanz.	S. 202
<code>bytes</code>	Erzeugt eine neue <code>bytes</code> -Instanz.	S. 203
<code>chr</code>	Gibt das Zeichen mit einem bestimmten Uni-code-Codepoint zurück.	S. 203
<code>classmethod</code>	Ähnlich wie <code>staticmethod</code> , aber die mit <code>classmethod</code> dekorierten Methoden erwarten als ersten Parameter die Klasse, mit der sie aufgerufen werden. Dieser Parameter heißt in der Regel <code>cls</code> und ist mit dem <code>self</code> von gewöhnlichen Methoden vergleichbar.	–
<code>compile</code>	Übersetzt einen übergebenen String oder eine Datei in ein ausführbares Objekt.	–
<code>complex</code>	Erzeugt eine komplexe Zahl.	S. 204
<code>delattr</code>	Löscht ein bestimmtes Attribut einer Instanz.	–
<code>dict</code>	Erzeugt ein Dictionary.	S. 204
<code>dir</code>	Gibt eine Liste aller Attribute eines Objekts zurück.	–
<code>divmod</code>	Gibt ein Tupel mit dem Ergebnis einer Ganzzahldivision und dem Rest zurück. <code>divmod(a, b)</code> ist äquivalent zu <code>(a // b, a % b)</code>	S. 205
<code>enumerate</code>	Gibt einen Aufzählungsiterator für die übergebene Sequenz zurück.	S. 205
<code>eval</code>	Wertet einen Python-Ausdruck aus.	S. 312
<code>exec</code>	Wertet einen Python-Ausdruck aus.	S. 312
<code>filter</code>	Ermöglicht es, bestimmte Elemente einer Liste herauszufiltern.	S. 206
<code>float</code>	Erzeugt eine Gleitkommazahl.	S. 206
<code>format</code>	Formatiert einen Wert mit der angegebenen Formatangabe.	S. 206
<code>frozenset</code>	Erzeugt eine unveränderliche Menge.	S. 207
<code>getattr</code>	Gibt ein bestimmtes Attribut einer Instanz zurück.	–

Tabelle A.6 Built-in Functions in Python (Forts.)

Built-in Function	Beschreibung	Erklärt auf
<code>globals</code>	Gibt ein Dictionary mit allen Referenzen des globalen Namensraums zurück.	S. 207
<code>hasattr</code>	Überprüft, ob eine Instanz über ein bestimmtes Attribut verfügt.	–
<code>hash</code>	Gibt den Hash-Wert einer Instanz zurück.	S. 207
<code>help</code>	Startet die eingebaute interaktive Hilfe von Python.	S. 207
<code>hex</code>	Gibt den Hexadezimalwert einer ganzen Zahl in Form eines Strings zurück.	S. 208
<code>id</code>	Gibt die Identität einer Instanz zurück.	S. 208
<code>input</code>	Liest einen String von der Tastatur ein.	S. 209
<code>int</code>	Erzeugt eine ganze Zahl.	S. 209
<code>isinstance</code>	Prüft, ob ein Objekt Instanz einer bestimmten Klasse ist.	–
<code>issubclass</code>	Prüft, ob eine Klasse von einer bestimmten Basis-klasse erbt.	–
<code>iter</code>	Erzeugt ein Iterator-Objekt.	S. 305, 309
<code>len</code>	Gibt die Länge einer bestimmten Instanz zurück.	S. 210
<code>list</code>	Erzeugt eine Liste.	S. 210
<code>locals</code>	Gibt ein Dictionary zurück, das alle Referenzen des lokalen Namensraums enthält.	S. 210
<code>map</code>	Wendet eine Funktion auf jedes Element einer Liste an.	S. 211
<code>max</code>	Gibt das größte Element einer Sequenz zurück.	S. 212
<code>memoryview</code>	Gibt ein <code>memoryview</code> -Objekt zurück.	–
<code>min</code>	Gibt das kleinste Element einer Sequenz zurück.	S. 213
<code>next</code>	Gibt das nächste Element des übergebenen Iterators zurück.	S. 302
<code>object</code>	Gibt die einfachste Form einer Instanz zurück, wenn sie aufgerufen wird.	–
<code>oct</code>	Gibt den Oktalwert einer ganzen Zahl in Form eines Strings zurück.	S. 213
<code>open</code>	Erzeugt ein Dateiojekt.	S. 177, 213

Tabelle A.6 Built-in Functions in Python (Forts.)

Built-in Function	Beschreibung	Erklärt auf
ord	Gibt den Unicode-Code eines bestimmten Zeichens zurück.	S. 213
pow	Führt eine Potenzoperation durch.	S. 214
print	Gibt die Übergebenen Objekte auf dem Bildschirm in anderen Ausgabegeräten aus.	S. 41, 214
property	Erzeugt ein Managed Attribute.	S. 251
range	Erzeugt einen Iterator zurück, mit dem gezählt werden kann.	S. 62, 215, 305
repr	Gibt eine String-Repräsentation einer Instanz zurück.	S. 215
reversed	Erzeugt einen Iterator, der ein iterierbares Objekt rückwärts durchläuft.	S. 216
round	Rundet eine Zahl.	S. 216
set	Erzeugt ein Set.	S. 160, 216
setattr	Setzt ein bestimmtes Attribut einer Instanz auf einen bestimmten Wert.	–
staticmethod	Erzeugt eine statische Methode.	S. 254
str	Erzeugt einen String.	S. 217
sum	Gibt die Summe aller Elemente einer Sequenz zurück.	S. 219
super	Gibt die Superklasse eines Typs zurück.	–
tuple	Erzeugt ein Tupel.	S. 219
type	Gibt den Datentyp einer Instanz zurück.	S. 70, 219
unichr	Gibt das Unicode-Zeichen mit einem bestimmten Unicode-Code zurück.	–
vars	Gibt ein Dictionary zurück, das den aktuellen Kontext enthält.	–
zip	Fasst mehrere Sequenzen zu Tupeln zusammen, um sie beispielsweise mit einer <code>for</code> -Schleife zu durchlaufen.	S. 219

Tabelle A.6 Built-in Functions in Python (Forts.)

Index

`__debug__` 322
`__future__` 313
`__import__` 232
2to3 755, 761

A

ABC 23
abs 201
Adaption 499
all 201
and 40
Anonyme Funktion 196
Anweisung 43
Anweisungskopf 43
Anweisungskörper 43
any 201
Argument 182, 426
as 224
ASCII 143
ascii 202
assert 321
Attribut 239, 245, 462
Ausdruck 39
 arithmetischer 39
 boolescher 93
 logischer 39

B

Backslash 322
Basisdatentyp 79
 bool 93
 bytes 124
 complex 98
 dict 151
 float 91
 frozenset 160, 168
 int 85
 list 111
 set 160, 167
 str 124, 217, 218, 276
 tuple 122
Basisklasse 256
Baum 463

Betaverteilung 337
Bezeichner 39
Bibliothek 223
Big Endian 418
Bildschirmausgabe 41
bin 202
Binärdistribution 636, 643
Binärsystem 88
Bindigkeit 80
Binding 651
Bit-Operation 88
 Bitverschiebung 90
 bitweises ausschließendes ODER 89
 bitweises Komplement 90
 bitweises ODER 89
 bitweises UND 88
Blockkommentar 47, 296
bool 202
boolescher Ausdruck 93
boolescher Operator 41
boolescher Wert 93
break 58
Breakpoint 593
Bubblesort 722
Buchstabe 105
Bug 591
Built-in Function 103, 201
 __import__ 232
 abs 201
 all 201
 any 201
 ascii 202
 bin 202
 bool 202
 bytearray 202
 bytes 203
 chr 203
 complex 204
 dict 204
 divmod 205
 enumerate 205
 exec 310
 filter 206
 float 206
 format 206

Built-in Function (Forts.)

frozenset 207
globals 207
hash 207
help 208
hex 208
id 208
input 209
int 209
len 210
list 210
locals 210
map 211
max 212
min 213
oct 213
open 213
ord 213
pow 214
print 214
range 215
repr 215
reversed 216
round 216
set 216
sorted 217
str 217
sum 219
tuple 219
type 219
zip 219
Busy Waiting 533
Button 673
bytearray 202
Byte-Code 24, 33
bytes 203

C

C 719
Callback-Funktion 475
Call-by-Reference 192
Call-by-Value 192
Callstack 290, 619
Canvas 696
case sensitive 39
Checkbox 676
Checkbutton 676
Children 464

chr 203
class 241
Client 518, 520
Client-Server-System 518, 521
Codepage 143
Codepoint 145
Compiler 24, 33
complex 204
Comprehension 292
 Dict 295
 List 293
 Set 296
Conditional Expression 55
continue 59
Control 651
Critical Section 445, 537
CSV 506
 Dialekt 506
Cursor 487

D

Dämon-Thread 456
Datei 172
 temporäre 511
Dateiähnliches Objekt 541
Dateiobjekt 172, 177, 541
Datenbank 483
 relationale 484
Datenstrom 171
Datentyp 70
 Mapping 151
 numerischer 82
 sequentieller 103
Deadlock 448
Debugger 591
Debugging 591
def 184
del 74, 113, 155
Deserialisieren 502
Destruktor 243
Dezimalsystem 86
dict 204
Dict Comprehension 295
Dictionary 151, 274
Differenzmenge 165
 symmetrische 165
Distribution 635
divmod 205

DLL 720
 Docstring 251, 296
 Document Object Model 463
 DOM 463
 Dropdown-Liste 690
 DST 386
 Dualsystem 86, 88
 Dynamic Link Library 720
 Dynamic Link Library → DLL
 Dynamische Bibliothek 720

E

Echte Teilmenge 163
 Eclipse 766
 Eingabeaufforderung 426
 Eingebettete Scriptsprache 736
 Einrückung 44
 Einwegkodierung 381
 ElementTree 479
 elif 53
 else 54, 57, 64, 286
 Elternelement 464
 E-Mail-Header 573
 Embedded Script Language 736
 Encoding-Deklaration 150
 Entry 679
 Entwicklungsumgebung 27, 765
 enumerate 205
 Eric 767
 Erweiterte Zuweisung 83
 Escape-Sequenz 125, 356
 ESMTP 557
 Event 457, 663
 Modifier 664
 Objekt 663
 Spezifikation 663
 Typ 664
 except 283
 Exception 280, 421, 732
 Exception Chaining 291
 Exception Handling 279
 exec 310
 Exit Code 423
 Exponent 91
 Exponentialschreibweise 91
 Exponentialverteilung 337
 Extension 636, 639, 728, 740

F

False 40, 93
 Fenster 651
 File Transfer Protocol 549
 filter 206
 finally 286
 Flag 363
 float 206
 for 61, 293
 format 206
 Formatangabe 137
 Frame-Objekt 602
 from 225
 frozenset 207
 FTP 549, 558
 Function Decorator 318
 Funktion 181
 anonyme 196
 lokale 195
 rekursive 200
 Funktionsaufruf 182
 Funktionsiterator 309
 Funktionsobjekt 187
 Funktionsschnittstelle 182

G

Gammaverteilung 338
 Ganze Zahl 36, 85
 Gauß-Verteilung 338
 Generator Expression 300
 Geplantes Sprachelement 312
 Geschwisterelement 464
 Getter-Methode 248
 Gleichverteilung 337
 Gleitkommazahl 37, 91, 343
 global 198
 Globale Referenz 197
 Globale Variable 439
 Globaler Namensraum 197
 globals 207
 GNU gettext API 376
 Grafische Benutzeroberfläche 651
 Grafische Benutzeroberfläche → GUI
 Gtk 652
 GUI 651
 Guido van Rossum 23

H

hash 207
Hash-Funktion 380
Hash-Wert 153, 380
help 208
hex 208
Hexadezimalsystem 86
Hook 422
HTTP 580

I

I18N 376
id 208
Identität 72
IDLE 27, 592, 765
if 52, 293
Imaginärteil 98
IMAP4 566
immutable 76
import 224, 231, 313
in 105, 155, 163
in place 116
Index 107
inf 92, 343
input 209
Installation 27
Installationsscript 638
Instanz 67, 241
 Wert 71
int 209
Interaktiver Modus 27, 35
Interface 246
Internationalisierung 376
Interpreter 24, 33
IP-Adresse 517
is 73
ISO-Kalender 396
Iterator 301
Iterator-Protokoll 302
Iterierbares Objekt 63
Iterieren 63
itertools 345

J

Join 493

K

Kartesisches Produkt 348
Keyword Argument 188
Kindelement 464
Klasse 240
Klassen-Member 253
Knoten 463
Kollision 381
Kombinationen 345
Kommandozeilen-Interpreter 750
Kommandozeilenparameter 425
Kommentar 47
Kommunikationssocket 518
Komodo IDE 768
Kompilat 33
Komplexe Zahl 98
Konsole 426
Konsolenanwendung 30
Konstruktor 243
Kontextmanager 314
Kontrollstruktur 51
Konvertierung 499
Koordinierte Weltzeit 385
Körperloses Tag 462

L

L10N 376
Label 680
LabelFrame 681
lambda 196
Laufzeitmessung 623
Laufzeitmodell 67
Laufzeitoptimierung 645
Laufzeitverhalten 622
Lazy Evaluation 56
Leichtgewichtprozess 439
len 210
Library 223
list 210
List Comprehensions 293
Listbox 682
Liste 111
Little Endian 418

locals 210
 Locking 443, 450
 Lock-Objekt 445, 450
 Logarithmische Normalverteilung 338
 Logdatei 605
 Logger 605
 Logging Handler 609
 Logischer Ausdruck 39
 Logischer Operator 93
 logische Negierung 94
 logisches ODER 94
 logisches UND 94
 Lokale Referenz 197
 Lokale Variable 439
 Lokaler Namensraum 197
 Lokalisierung 376
 Lokalzeit 385
 Lookup 649

M

Magic Line 32
 Magic Members 262
 *_abs_* 273
 *_add_* 271
 *_and_* 271
 *_call_* 268, 319
 *_complex_* 274
 *_contains_* 277
 *_del_* 244, 263
 *_delattr_* 265
 *_delitem_* 276
 *_dict_* 264
 *_div_* 271
 *_doc_* 297
 *_enter_* 314
 *_eq_* 268
 *_exit_* 314
 *_float_* 274
 *_floordiv_* 271
 *_ge_* 268
 *_getattr_* 264
 *_getattribute_* 264
 *_getitem_* 275, 308
 *_gt_* 268
 *_hash_* 268
 *_hex_* 274
 *_iadd_* 273
 *_iand_* 273
 Magic Members (Forts.)
 *_idiv_* 273
 *_ifloordiv_* 273
 *_ilshift_* 273
 *_imod_* 273
 *_imul_* 273
 *_index_* 274
 *_init_* 243, 263
 *_int_* 274
 *_invert_* 273
 *_ior_* 273
 *_ipow_* 273
 *_irshift_* 273
 *_isub_* 273
 *_iter_* 276, 277, 302
 *_ixor_* 273
 *_le_* 268
 *_len_* 275
 *_long_* 274
 *_lshift_* 271
 *_lt_* 268
 *_mod_* 271
 *_mul_* 271
 *_ne_* 268
 *_neg_* 273
 *_nonzero_* 268
 *_oct_* 274
 *_or_* 271
 *_pos_* 273
 *_pow_* 271
 *_radd_* 272
 *_rand_* 272
 *_rdiv_* 272
 *_repr_* 263
 *_rfloordiv_* 272
 *_rlshift_* 272
 *_rmod_* 272
 *_rmul_* 272
 *_ror_* 272
 *_rpow_* 272
 *_rrshift_* 272
 *_rshift_* 271
 *_rsub_* 272
 *_rxor_* 272
 *_setattr_* 264
 *_setitem_* 276
 *_slots_* 265
 *_str_* 263, 288
 *_sub_* 271

Magic Members (Forts.)
 __xor__ 271
Mailbox 566
Managed Attribute 251
Mantisse 91
map 211
Mapping 151
Matching 351, 373
Match-Objekt 364, 369
max 212
MD5 382
Mehrfachvererbung 258
Member 239
 privat 246
 private 247
 protected 247
 public 247
 statischer 252
Memory Leak 735
Menge 160
Menubutton 689
Menüleiste 686
Message Box 713
Method Table 729
Methode 101, 239
Methodendefinition 242
MFC 651
MIME 572
min 213
Modul 223, 636
Modularisierung 223
Modus
 interaktiver 27, 35
Multicall 587
Multiplexender Server 532
Multitasking 437
mutable 75

N

Namensraum
 globaler 197
 lokaler 197
nan 93, 343
Netzwerk-Byte-Order 531
Node 463
None 82
NoneType
 Basisdatentyp 81

nonlocal 198
Normalverteilung 338
not 40, 94
not in 105, 155, 163
Numerische Datentypen 82

O

Objekt 235, 239
 dateiähnliches 541
 iterierbares 63
Objektorientierung 235
oct 213
Oktalsystem 86
open 213
Operand 79
Operator 36, 39, 79, 93
 arithmetischer 39
 boolescher 41
 logischer 93
Operatorrangfolge 80, 770
Optimierung 645
Option 426
OptionMenu 690
or 41
ord 213
Ordnungsrelation 111
OSI-Schichtenmodell 515
Out-of-Band Data 533

P

Packer 655, 658
Padding 660
Paket 228, 636
Parameter 101, 182
 optionaler 101, 187
Parent 464
Pareto-Verteilung 338
Parser 462
pass 65
Passworteingabe 750
PDB 591
Permutationen 348
Pfad 407
Pipe 360
Plattformunabhängigkeit 24
Polygon 701
 konkaves 702

POP3 561
 Port 517
 Positional Argument 188
 Post-Mortem Debugger 594
 pow 214
 Primzahl 448
 print 41, 214
 Profiler 626
 Programm 437
 Programmdatei 30
 Programmierparadigma 24
 Protokollebene 515
 Prozess 437
 PSF-Lizenz 25
 PyDev 766
 PyGnome 652
 PyGtk 652
 PyQt 653
 Python API 25, 728
 Python Software Foundation 24
 Python-Shell 27
 Python-Website 27

Q

Qt 653
 Qt Software 653
 Quantor 354
 genügsamer 358
 Quellcodedistribution 636
 Quelltext 30
 Query 484
 Queue 454, 520

R

Radiobutton 678
 raise 282
 range 215
 Rapid Prototyping 25
 Raw-String 127
 Reader 475
 Realteil 98
 Reference Count 75, 424, 734
 Referenz 67
 globale 197
 lokale 197
 Referenzzähler 75

Regulärer Ausdruck 351
 Gruppe 359
 Syntax 352
 Regular-Expression-Objekt 364, 367
 Rekursion 200
 Rekursionstiefe 200
 Relationale Datenbank 484
 RE-Objekt 367
 repr 215
 Request Handler 536
 Reservierte Wörter 770
 return 185
 reversed 216
 ROT13 733
 round 216
 RPM 643
 Rückgabewert 103, 182

S

SAX 474
 Schleife 56
 Schleifenkörper 56
 Schleifenzähler 61
 Schlüssel-Wert-Paar 151
 Schlüsselwort 39
 Schlüsselwörter 770
 Schlüsselwortparameter 101, 188
 Schnittmenge 164
 Schnittstelle 246
 Schrittweite 109
 Scrollbar 691
 Searching 351, 372
 Seiteneffekt 192
 self 242
 Semikolon 323
 Serialisieren 502
 Serieller Server 518
 Server 518
 multiplexer 518, 532
 paralleler 518
 serieller 518
 set 216
 Set Comprehension 296
 Setter-Methode 249
 SHA-1 382
 SHA-224 382
 SHA-256 382
 SHA-384 383

- SHA-512 383
- Shared Object 720
- Shebang 32
- Shell 426
- Sibling 464
- Simple API for XML 474
- Slicing 108
 - slice-Objekt* 276
- SMTP 557
- Socket 517
 - blockierender* 525
 - nicht-blockierender* 525
- Socket API 517
- sorted 217
- Sortiervverfahren
 - stabiles* 118
- Speicherzugriffsfehler 736
- Spinbox 692
- Sprachkompilat 378
- SQL 484
- SQL Injection 490
- SQL-Statement
 - CREATE* 488
 - INSERT* 489
 - SELECT* 492
- SSH 577
- Stabiles Sortiervverfahren 118
- Standardbibliothek 24, 325
 - atexit* 435
 - cmath* 327
 - cmd* 750
 - cProfile* 626
 - csv* 506
 - ctypes* 720
 - datetime* 392
 - decimal* 339
 - distutils* 635
 - doctest* 611
 - ElementTree* 479
 - email* 572
 - ftplib* 549
 - functools* 748
 - getpass* 750
 - gettext* 376
 - gzip* 459
 - hashlib* 380
 - imaplib* 566
 - inspect* 594
 - itertools* 345
- Standardbibliothek (Forts.)
 - logging* 605
 - math* 327
 - optparse* 425
 - os* 405
 - os.path* 413
 - pickle* 513
 - platform* 425
 - poplib* 561
 - pprint* 602
 - random* 334
 - re* 351, 362
 - select* 532
 - shutil* 433
 - SMTP* 557
 - socket* 517
 - socketserver* 536
 - sqlite3* 487
 - sys* 418
 - telnetlib* 577
 - tempfile* 511
 - thread* 441
 - threading* 448
 - time* 385
 - timeit* 623
 - tkinter* 652, 654
 - trace* 629
 - traceback* 619
 - unittest* 615
 - urllib* 540
 - urllib.parse* 544
 - urllib.request* 540
 - webbrowser* 747
 - xml* 461
 - xml.dom* 465
 - xml.etree.ElementTree* 479
 - xmlrpclib* 585
- Standarddialog 708
- Steuerelement 651, 669
 - Button* 673
 - Canvas* 696
 - Checkbox* 676
 - Checkbutton* 676
 - Entry* 679
 - Label* 680
 - LabelFrame* 681
 - Listbox* 682
 - Menu* 686
 - Menubutton* 689

Steuerelement (Forts.)
 OptionMenu 690
 Radiobutton 678
 Scrollbar 691
 ScrolledText 707
 Spinbox 692
 Text 694
 Steuerelementvariable 656
 Steuerzeichen 125
 str 217
 Stream 171
 String 37, 124, 217, 218, 276
 Subklasse 256
 sum 219
 Symmetrische Differenzmenge 165
 Syntax 43
 Syntaxanalyse 462

T

Tag 461
 körperloses 462
 Tag-Name 461
 TCP 522
 Teilmenge 163
 echte 163
 Telnet 577
 Temporäre Datei 511
 Term 36
 Terminator 309
 Test
 automatisierter 611
 Text-Widget 694
 Thread 437, 439
 schlafender 438
 Tk 652
 Toolkit 651
 Toplevel-Tag 471
 Traceback 280, 423
 Traceback-Objekt 315, 421, 619
 Tracer 629
 Transmission Control Protocol 522
 Trolltech 653
 True 40, 93
 try 283
 Tupel 122
 tuple 219
 Tuple Packing 122

Tuple Unpacking 122
 type 70, 219

U

Überdeckungsanalyse 630
 Überladen 266
 UDP 520
 Unicode 145
 Unit 611
 Unit Test 615
 Unix-Epoche 385
 Unix-Timestamp 385
 URL 540, 747
 User Datagram Protocol 520
 UTC 386
 UTF 146

V

Variable 38
 globale 439
 lokale 439
 Variablentyp 46
 Verbindungssocket 518
 Vereinigungsmenge 164
 Vererbung 255
 Vergleich 40
 Vergleichsoperator 40, 84
 Virtuelle Maschine 33
 Von-Mises-Verteilung 338

W

Wahlfreier Zugriff 463
 Wahrheitswert 40
 Wallis'sches Produkt 441
 Weibull-Verteilung 339
 Wert
 boolescher 93
 while 56
 Whitespace 127, 356
 Widget 651, 670
 Wing IDE 769
 Worker-Thread 454
 Wrapper 377
 Wurzel 464

X

XML 461, 580

XML-RPC 580

Y

yield 299

Z

Zahl

ganze 36, 85

komplexe 98

Zahlensysteme 86

Zählschleife 61

Zeichen 124

Zeichenkette 37

Zeichenklasse 353, 356

Zeichenliteral 352

Zeilenkommentar 47

Zeitscheibe 438

zip 219

Zope 542

Zugriff

wahlfreier 463

Zuweisung 38, 83

erweiterte 83