



```
= urllib.request.urlopen("http://www.galileo-p
>>> d = f.info()
>>> d
<http.client.HTTPMessage object at 0xb7689e8c>
>>> d.keys()
['Date', 'Server', 'Content-Length', 'Content-
'Cache-Control', 'Expires', 'Connection']
```

Johannes Ernesti · Peter Kaiser



Python 3

Das umfassende Handbuch

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

3., aktualisierte und erweiterte Auflage

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 und ist damit nun noch einfacher und klarer strukturiert. Die Zukunft gehört Python 3!

In diesem umfassenden Handbuch werden Sie alles finden, was Sie für Ihre Arbeit mit Python 3 brauchen. Die Sprachgrundlagen werden genauso ausführlich behandelt wie professionelle Techniken, GUI-Programmierung und die Anbindung an andere Programmiersprachen.

Egal, ob Sie erst 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 Anne Scheibe

Lektorat Galileo Computing

anne.scheibe@galileo-press.de

www.galileocomputing.de

Galileo Press · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

Teil I Einstieg in Python

1	Überblick über Python	27
2	Die Arbeit mit Python	31
3	Der interaktive Modus	39
4	Grundlegendes zu Python-Programmen	51
5	Kontrollstrukturen	59
6	Das Laufzeitmodell	75
7	Basisdatentypen	87
8	Dateien	197
9	Funktionen	209

Teil II Fortgeschrittene Programmiertechniken

10	Modularisierung	257
11	Objektorientierung	269
12	Weitere Spracheigenschaften	329

Teil III Die Standardbibliothek

13	Mathematik	385
14	Strings	415
15	Datum und Zeit	451
16	Schnittstelle zum Betriebssystem	481
17	Parallele Programmierung	525
18	Datenspeicherung	549
19	Netzwerkcommunication	607
20	Debugging	691

Teil IV Weiterführende Themen

21	Distribution von Python-Projekten	739
22	Optimierung	753
23	Grafische Benutzeroberflächen	759
24	Wissenschaftliches Rechnen	881
25	Anbindung an andere Programmiersprachen	903
26	Insiderwissen	933
27	Von Python 2 nach Python 3	955

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. Legendar ist sein Ausspruch *Eppur si 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.

Lektorat Anne Scheibe

Korrektorat Monika Paff, Langenfeld

Einbandgestaltung Barbara Thoben, Köln

Titelbilder Labyrinth: ©csld-fotolia.com, Mann: Neustockimages/istockimages

Typografie und Layout Vera Brauner

Herstellung Katrin Müller

Satz III-satz, Husby

Druck und Bindung Beltz Druckpartner, Hemsbach

Gerne stehen wir Ihnen mit Rat und Tat zur Seite:

anne.scheibe@galileo-press.de

bei Fragen und Anmerkungen zum Inhalt des Buches

service@galileo-press.de

für versandkostenfreie Bestellungen und Reklamationen

britta.behrens@galileo-press.de

für Rezensionen- und Schulungsexemplare

Dieses Buch wurde gesetzt aus der TheAntiquaB (9,35/13,7 pt) in FrameMaker.

Bibliografische Information der Deutschen Nationalbibliothek

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

ISBN 978-3-8362-1925-9

© Galileo Press, Bonn 2012

3., aktualisierte und erweiterte Auflage 2012

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

Über dieses Buch	19

Teil I Einstieg in Python

1 Überblick über Python	27

1.1 Geschichte und Entstehung	27
1.2 Grundlegende Konzepte	28
1.3 Einsatzmöglichkeiten und Stärken	29
1.4 Einsatzbeispiele	30

2 Die Arbeit mit Python	31

2.1 Die Verwendung von Python	31
2.1.1 Windows	33
2.1.2 Linux	33
2.1.3 Mac OS X	33
2.2 Tippen, kompilieren, testen	34
2.2.1 Shebang	35
2.2.2 Interne Abläufe	36

3 Der interaktive Modus	39

3.1 Ganze Zahlen	40
3.2 Gleitkommazahlen	41
3.3 Zeichenketten	42
3.4 Listen	43
3.5 Variablen	43
3.6 Logische Ausdrücke	45
3.7 Funktionen und Methoden	47

3.7.1	Funktionen	47
3.7.2	Methoden	48
3.8	Bildschirm Ausgaben	49
4	Grundlegendes zu Python-Programmen	51
4.1	Grundstruktur eines Python-Programms	51
4.2	Das erste Programm	53
4.3	Kommentare	55
4.4	Der Fehlerfall	56
5	Kontrollstrukturen	59
5.1	Fallunterscheidungen	59
5.1.1	Die if-Anweisung	60
5.1.2	Conditional Expressions	63
5.2	Schleifen	64
5.2.1	Die while-Schleife	65
5.2.2	Vorzeitiger Abbruch einer Schleife	65
5.2.3	Erkennen eines Schleifenabbruchs	66
5.2.4	Vorzeitiger Abbruch eines Schleifendurchlaufs	68
5.2.5	Die for-Schleife	70
5.2.6	Die for-Schleife als Zählschleife	71
5.3	Die pass-Anweisung	73
6	Das Laufzeitmodell	75
6.1	Die Struktur von Instanzen	77
6.1.1	Datentyp	77
6.1.2	Wert	78
6.1.3	Identität	80
6.2	Referenzen und Instanzen freigeben	81
6.3	Mutable vs. immutable Datentypen	82
6.3.1	Mutable Datentypen und Seiteneffekte	84

7	Basisdatentypen	87
7.1	Operatoren	88
7.2	Das Nichts – NoneType	92
7.3	Numerische Datentypen	93
7.3.1	Arithmetische Operatoren	93
7.3.2	Vergleichende Operatoren	95
7.3.3	Konvertierung zwischen numerischen Datentypen	96
7.3.4	Ganzzahlen – int	96
7.3.5	Gleitkommazahlen – float	102
7.3.6	Boolesche Werte – bool	104
7.3.7	Komplexe Zahlen – complex	110
7.4	Parameter von Funktionen und Methoden	113
7.4.1	Wiederholung	113
7.4.2	Parameter	114
7.5	Sequentielle Datentypen	115
7.5.1	Operationen auf Instanzen sequentieller Datentypen	118
7.5.2	Listen – »list«	128
7.5.3	Unveränderliche Listen – tuple	139
7.5.4	Strings – str, bytes, bytearray	142
7.6	Mappings	173
7.6.1	Dictionary – dict	174
7.7	Mengen	185
7.7.1	Die Datentypen set und frozenset	185
7.7.2	Veränderliche Mengen – set	193
7.7.3	Unveränderliche Mengen – frozenset	195
8	Dateien	197
8.1	Datenströme	197
8.2	Daten aus einer Datei auslesen	198
8.3	Daten in eine Datei schreiben	202
8.4	Verwendung des Dateiobjekts	204

9 Funktionen 209

9.1	Schreiben einer Funktion	211
9.2	Funktionsparameter	215
9.2.1	Optionale Parameter	215
9.2.2	Schlüsselwortparameter	216
9.2.3	Beliebige Anzahl von Parametern	217
9.2.4	Reine Schlüsselwortparameter	219
9.2.5	Entpacken einer Parameterliste	221
9.2.6	Seiteneffekte	222
9.3	Namensräume	225
9.3.1	Zugriff auf globale Variablen – global	225
9.3.2	Zugriff auf den globalen Namensraum	226
9.3.3	Zugriff auf übergeordnete Namensräume – nonlocal	227
9.4	Lokale Funktionen	229
9.5	Anonyme Funktionen	230
9.6	Rekursion	231
9.7	Eingebaute Funktionen	231

Teil II Fortgeschrittene Programmiertechniken

10 Modularisierung 257

10.1	Einbinden globaler Module	257
10.2	Lokale Module	260
10.2.1	Namenskonflikte	261
10.2.2	Modulinterne Referenzen	262
10.3	Pakete	262
10.3.1	Absolute und relative Import-Anweisungen	265
10.3.2	Importieren aller Module eines Pakets	266
10.4	Built-in Functions	267

11 Objektorientierung 269

11.1	Klassen	274
11.1.1	Definieren von Methoden	276
11.1.2	Der Konstruktor und die Erzeugung von Attributen	277
11.2	Vererbung	280
11.2.1	Technische Grundlagen	281
11.2.2	Die Klasse GirokontoMitTagesumsatz	283
11.2.3	Beispiele möglicher Erweiterungen	289
11.2.4	Ausblick	293
11.2.5	Mehrfachvererbung	294
11.3	Setter und Getter und Property Attributes	295
11.4	Klassenattribute und Klassenmethoden sowie statische Methoden	298
11.5	Magic Methods und Magic Attributes	302
11.5.1	Allgemeine Magic Methods	302
11.5.2	Zugriff auf Attribute anpassen	306
11.5.3	Operatoren überladen	309
11.5.4	Datentypen emulieren	317
11.6	Built-in Functions für Objektorientierung	322
11.6.1	Funktionen für die Verwaltung der Attribute einer Instanz	323
11.6.2	Funktionen für Informationen über die Klassenhierarchie	324
11.7	Objektphilosophie	326

12 Weitere Spracheigenschaften 329

12.1	Exception Handling	329
12.1.1	Eingebaute Exceptions	330
12.1.2	Werfen einer Exception	334
12.1.3	Abfangen einer Exception	335
12.1.4	Eigene Exceptions	340
12.1.5	Erneutes Werfen einer Exception	342
12.1.6	Exception Chaining	344
12.2	Comprehensions	346
12.2.1	List Comprehensions	346
12.2.2	Dict Comprehensions	348
12.2.3	Set Comprehensions	349

12.3	Generatoren	349
12.4	Iteratoren	353
12.5	Docstrings	363
12.6	Interpreter im Interpreter	365
12.7	Die with-Anweisung	368
12.8	Function Annotations	371
12.9	Function Decorator	373
12.10	assert	377
12.11	Weitere Aspekte der Syntax	378
	12.11.1 Umbrechen langer Zeilen	378
	12.11.2 Zusammenfügen mehrerer Zeilen	379
12.12	Geplante Sprachelemente	380

Teil III Die Standardbibliothek

13	Mathematik	385
13.1	Mathematische Funktionen – math, cmath	385
	13.1.1 Zahlentheoretische Funktionen	389
	13.1.2 Exponential- und Logarithmusfunktionen	391
	13.1.3 Trigonometrische Funktionen	392
	13.1.4 Funktionen aus cmath	394
13.2	Zufallszahlengenerator – random	394
	13.2.1 Steuerungsfunktionen	396
	13.2.2 Funktionen für ganze Zahlen	397
	13.2.3 Funktionen für Sequenzen	398
	13.2.4 Alternative Generatoren	399
13.3	Präzise Dezimalzahlen – decimal	399
	13.3.1 Verwendung des Datentyps	400
	13.3.2 Nichtnumerische Werte	403
	13.3.3 Das Context-Objekt	404
13.4	Spezielle Generatoren – itertools	405

14 Strings 415

14.1	Reguläre Ausdrücke – re	415
14.1.1	Syntax regulärer Ausdrücke	416
14.1.2	Verwendung des Moduls re	427
14.1.3	Ein einfaches Beispielprogramm – Searching	438
14.1.4	Ein komplexeres Beispielprogramm – Matching	439
14.2	Lokalisierung von Programmen – gettext	442
14.2.1	Beispiel für die Verwendung von gettext	443
14.3	Hash-Funktionen – hashlib	446
14.3.1	Verwendung des Moduls	448
14.3.2	Beispiel	450

15 Datum und Zeit 451

15.1	Elementare Zeitfunktionen – time	451
15.2	Komfortable Datumsfunktionen – datetime	460
15.2.1	Attribute und Datentypen (Übersicht)	460
15.2.2	date	461
15.2.3	datetime.time	466
15.2.4	datetime.datetime	468
15.2.5	datetime.timedelta	477

16 Schnittstelle zum Betriebssystem 481

16.1	Funktionen des Betriebssystems – os	481
16.1.1	Zugriff auf den eigenen Prozess und andere Prozesse	482
16.1.2	Zugriff auf das Dateisystem	484
16.2	Umgang mit Pfaden – os.path	490
16.3	Zugriff auf das Dateisystem – shutil	495
16.3.1	Verzeichnis- und Dateioperationen	497
16.3.2	Archivoperationen	499
16.4	Zugriff auf die Laufzeitumgebung – sys	501
16.4.1	Konstanten	503
16.4.2	Exceptions	506

16.4.3	Hooks	507
16.4.4	Sonstige Funktionen	509
16.5	Informationen über das System – platform	511
16.6	Kommandozeilenparameter – argparse	512
16.6.1	Taschenrechner – ein einfaches Beispiel	513
16.6.2	Komplexeres Beispiel	518
16.7	Kopieren von Instanzen – copy	519
16.8	Das Programmende – atexit	523

17 Parallele Programmierung 525

17.1	Prozesse, Multitasking und Threads	525
17.2	Die Thread-Unterstützung in Python	528
17.3	Das Modul <code>_thread</code>	528
17.3.1	Ein Beispiel für eine rechenintensive Funktion – die Approximation von Pi	528
17.3.2	Parallele Berechnung mithilfe von Threads	529
17.3.3	Datenaustausch zwischen Threads – locking	531
17.4	Das Modul <code>threading</code>	536
17.4.1	Locking im <code>threading</code> -Modul	539
17.4.2	Worker-Threads und Queues	542
17.4.3	Ereignisse definieren – <code>threading.Event</code>	546
17.4.4	Barrieren definieren – <code>threading.Barrier</code>	546
17.4.5	Eine Funktion zeitlich versetzt ausführen – <code>threading.Timer</code>	546

18 Datenspeicherung 549

18.1	Komprimierte Dateien lesen und schreiben – <code>gzip</code>	549
18.2	XML	551
18.2.1	DOM – Document Object Model	553
18.2.2	SAX – Simple API for XML	565
18.2.3	ElementTree	570
18.3	Datenbanken	575
18.3.1	Beispieldatenbank	577
18.3.2	Pythons eingebaute Datenbank – <code>sqlite3</code>	579

18.4	Serialisierung von Instanzen – pickle	595
18.5	Das Tabellenformat CSV – csv	599
18.6	Temporäre Dateien – tempfile	604

19 Netzwerkkommunikation 607

19.1	Socket API	609
19.1.1	Client-Server-Systeme	610
19.1.2	UDP	612
19.1.3	TCP	614
19.1.4	Blockierende und nicht-blockierende Sockets	616
19.1.5	Verwendung des Moduls	618
19.1.6	Die Socket-Klasse	621
19.1.7	Netzwerk-Byte-Order	625
19.1.8	Multiplexende Server – select	627
19.1.9	socketserver	630
19.2	URLs	634
19.2.1	Zugriff auf Ressourcen im Internet – urllib.request	634
19.2.2	Verarbeiten einer URL – urllib.parse	640
19.3	FTP – ftplib	646
19.4	E-Mail	655
19.4.1	SMTP – smtplib	655
19.4.2	POP3 – poplib	660
19.4.3	IMAP4 – imaplib	665
19.4.4	Erstellen komplexer E-Mails – email	671
19.5	Telnet – telnetlib	676
19.6	XML-RPC	680
19.6.1	Der Server	680
19.6.2	Der Client	685
19.6.3	Multicall	687
19.6.4	Einschränkungen	688

20 Debugging 691

20.1	Der Debugger	691
20.2	Inspizieren von Instanzen – inspect	694

20.2.1	Datentypen, Attribute und Methoden	696
20.2.2	Quellcode	697
20.2.3	Klassen und Funktionen	699
20.3	Formatierte Ausgabe von Instanzen – pprint	703
20.4	Logdateien – logging	706
20.4.1	Das Meldungsformat anpassen	709
20.4.2	Logging Handler	711
20.5	Automatisiertes Testen	713
20.5.1	Testfälle in Docstrings – doctest	713
20.5.2	Unit Tests – unittest	718
20.6	Traceback-Objekte – traceback	722
20.7	Analyse des Laufzeitverhaltens	726
20.7.1	Laufzeitmessung – timeit	727
20.7.2	Profiling – cProfile	730
20.7.3	Tracing – trace	734

Teil IV Weiterführende Themen

21 Distribution von Python-Projekten 739

21.1	Erstellen von Distributionen – distutils	739
21.1.1	Schreiben des Moduls	740
21.1.2	Das Installationsscript	742
21.1.3	Erstellen einer Quellcodedistribution	746
21.1.4	Erstellen einer Binärdistribution	748
21.1.5	Distributionen installieren	749
21.2	Erstellen von EXE-Dateien – cx_Freeze	750

22 Optimierung 753

22.1	Die Optimize-Option	754
22.2	Mutable vs. immutable	754
22.3	Schleifen	755
22.4	Funktionsaufrufe	756

22.5	C	756
22.6	Lookup	757
22.7	Exceptions	757
22.8	Keyword Arguments	758
23	Grafische Benutzeroberflächen	759
23.1	Toolkits	759
23.1.1	Tkinter	760
23.1.2	PyGObject	760
23.1.3	PyQt	760
23.1.4	PySide	761
23.1.5	wxPython	761
23.2	Einführung in tkinter	762
23.2.1	Ein einfaches Beispiel	762
23.2.2	Steuerelementvariablen	764
23.2.3	Der Packer	767
23.2.4	Events	771
23.2.5	Die Steuerelemente	779
23.2.6	Die Klasse Tk	817
23.2.7	Weitere Module	818
23.3	Einführung in PyQt	826
23.3.1	Installation	826
23.3.2	Grundlegende Konzepte von Qt	827
23.3.3	Entwicklungsprozess	829
23.4	Signale und Slots	836
23.5	Wichtige Widgets	839
23.5.1	QCheckBox	840
23.5.2	QComboBox	841
23.5.3	QDateEdit, QTimeEdit, QDateTimeEdit	842
23.5.4	QDialog	843
23.5.5	QLineEdit	844
23.5.6	QListWidget, QListView	844
23.5.7	QProgressBar	845
23.5.8	QPushButton	846
23.5.9	QRadioButton	846
23.5.10	QSlider, QDial	847

23.5.11	QTextEdit	848
23.5.12	QWidget	848
23.6	Zeichenfunktionalität	850
23.6.1	Werkzeuge	851
23.6.2	Koordinatensystem	853
23.6.3	Einfache Formen	853
23.6.4	Grafiken	856
23.6.5	Text	857
23.6.6	Eye-Candy	859
23.7	Model-View-Architektur	863
23.7.1	Beispielprojekt: Ein Adressbuch	864
23.7.2	Auswählen von Einträgen	874
23.7.3	Bearbeiten von Einträgen	875
24	Wissenschaftliches Rechnen	881
24.1	Installation	882
24.2	Das Modellprogramm	883
24.2.1	Der Import von numpy, scipy und matplotlib	884
24.2.2	Vektorisierung und der Datentyp numpy.ndarray	885
24.2.3	Visualisieren von Daten mit matplotlib.pyplot	889
24.3	Überblicke über die Module numpy und scipy	891
24.3.1	Überblick über den Datentyp numpy.ndarray	891
24.3.2	Überblick über scipy	899
25	Anbindung an andere Programmiersprachen	903
25.1	Dynamisch ladbare Bibliotheken – ctypes	904
25.1.1	Ein einfaches Beispiel	904
25.1.2	Die eigene Bibliothek	905
25.1.3	Schnittstellenbeschreibung	909
25.1.4	Verwendung des Moduls	911
25.2	Schreiben von Extensions	914
25.2.1	Ein einfaches Beispiel	914
25.2.2	Exceptions	918
25.2.3	Erzeugen der Extension	919
25.2.4	Reference Counting	921

25.3	Python als eingebettete Scriptsprache	922
25.3.1	Ein einfaches Beispiel	922
25.3.2	Ein komplexeres Beispiel	924
25.3.3	Python-API-Referenz	927

26 Insiderwissen 933

26.1	URLs im Standardbrowser öffnen – webbrowser	933
26.2	Funktionsschnittstellen vereinfachen – functools	934
26.2.1	Funktionsschnittstellen vereinfachen	934
26.2.2	Caches	935
26.2.3	Ordnungsrelationen vervollständigen	937
26.3	Weitere Datentypen – collections	938
26.3.1	Zählen von Häufigkeiten	938
26.3.2	Dictionaries mit Standardwerten	941
26.3.3	Doppelt verkettete Listen	942
26.3.4	Benannte Tupel	944
26.3.5	Sortierte Dictionaries	945
26.4	Interpretieren von Binärdaten – struct	946
26.5	Versteckte Passworteingaben – getpass	949
26.6	Kommandozeilen-Interpreter – cmd	949
26.7	Dateiinterface für Strings – io.StringIO	952

27 Von Python 2 nach Python 3 955

27.1	Die wichtigsten Unterschiede	958
27.1.1	Ein-/Ausgabe	958
27.1.2	Iteratoren	959
27.1.3	Strings	960
27.1.4	Ganze Zahlen	961
27.1.5	Exception Handling	962
27.1.6	Standardbibliothek	962
27.1.7	Neue Sprachelemente in Python 3	963
27.2	Automatische Konvertierung	964

Anhang	969
A.1 Reservierte Wörter	969
A.2 Built-in Functions	969
 Index	 973

Über dieses Buch

»Der Anfang ist die Hälfte des Ganzen.«

– Aristoteles

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.

Warum haben wir dieses Buch geschrieben?

Wir, Peter Kaiser und Johannes Ernesti, sind vor einigen Jahren durch Zufall auf die Programmiersprache Python aufmerksam geworden und bis heute bei ihr geblieben. Die Einfachheit, Flexibilität und Eleganz von Python fasziniert uns nach wie vor. Mit Python lässt sich eine Idee in kurzer Zeit zu einem funktionierenden Programm fortentwickeln. Zudem braucht sich der Programmierer keine Gedanken über die Lauffähigkeit seines Codes auf verschiedenen Betriebssystemen zu machen, da Python-Code unmodifiziert auf allen wichtigen Betriebssystemen läuft. Die Programmiersprache Python erlaubt es, kurze, elegante und produktive Programme für komplexe Aufgaben zu schreiben, und vereinfacht damit den Programmieralltag erheblich. 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 finden sich Dokumentationen und Anleitungen, 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 Tutorien sogar möglich, innerhalb weniger Tage ein fundiertes Grundwissen aufzubauen, mit dem sich durchaus arbeiten lässt. Problematisch ist der Übergang zur fortgeschrittenen Programmierung, da man dann 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.

Ziel dieses Buchs ist es, diese Lücke zu schließen. Es 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.

Was leistet dieses Buch, was nicht?

Dieses Buch vermittelt dem Leser fundierte Python-Kenntnisse, mit denen 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 unser Ziel, 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 nicht als Einführung in die hinter der Programmierung stehende Theorie oder in die Informatik anzusehen, wir behandeln weder Datenstrukturen noch Algorithmen. Der Hauptfokus liegt auf der praktischen Arbeit mit Python.

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 Wert darauf, dass der Leser schon früh seine ersten eigenen Programme entwickeln und testen kann. Auch wenn sich das Buch selbst eher als Referenz versteht und des-

halb keine Übungsaufgaben stellt, empfehlen wir Ihnen, das in diesem Teil erworbene Wissen in eigenen Python-Programmen anzuwenden. 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 wird, 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 die Konzepte im Vordergrund, die die Arbeit mit Python so angenehm machen, allerdings für den unerfahrenen Leser auch völliges Neuland darstellen können. Als große Oberthemen sind dabei Modularisierung und Objektorientierung zu nennen, die in Python eine zentrale Rolle spielen. Außerdem werden moderne Programmiertechniken wie Exception Handling, Iteratoren und Generatoren behandelt.
3. Der dritte Teil konzentriert sich auf Pythons *Batteries-included*-Philosophie, wonach Python nach Möglichkeit alles in der Standardbibliothek mitbringen sollte, was für die Entwicklung eigener Anwendungen erforderlich ist. Wir werden in diesem Teil auf viele der mitgelieferten Module eingehen und auch das ein oder andere Drittanbietermodul besprechen. Insbesondere ist auch die Suche nach Fehlern in Python-Programmen und deren Behebung Thema dieses Teils.

Der dritte Teil ist als Nachschlagewerk zu konkreten Problemen gedacht. Es empfiehlt sich, jeweils die einleitenden Bemerkungen und Beispiele zu lesen. Dadurch erlangen Sie einen Überblick darüber, was die Standardbibliothek leistet und unter welchen Stichworten Sie mehr über eine bestimmte Funktionalität erfahren können.

4. Im letzten Teil behandeln wir weiterführende Themen wie die Weitergabe von fertigen Python-Programmen und -Modulen an Endanwender bzw. andere Entwickler. Neben der Programmoptimierung und der Auslagerung laufzeitkritischer Programmteile in effizientere Sprachen wie C wird auch die Entwicklung von grafischen Benutzeroberflächen mit Tkinter und Qt besprochen. Außerdem zeigen wir kleine Kniffe, die das Arbeiten mit Python noch effektiver machen können. Am Ende des Buchs besprechen wir die Unterschiede zwischen den Python-Generationen 2.x und 3.x und zeigen, wie man alte Programme unter den neuen Python-Versionen lauffähig macht.

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.

- ▶ Als Programmieranfänger oder Umsteiger mit grundlegenden Kenntnissen in einer anderen Programmiersprache sollten Sie die ersten beiden Teile des Buchs sorgfältig durcharbeiten. Dort wird ein umfassender Einstieg in die Sprache Python geboten. Die Einführung wird Sie früh in die Lage versetzen, einfache Python-Programme zu schreiben. Nutzen Sie diese Chance und programmieren Sie so viel wie möglich, bereits während Sie die Einführung lesen. Machen Sie sich darauf gefasst, dass der Anspruch ab Teil 2 rasch zunimmt, denn unser Buch soll Sie in die Lage versetzen, Python professionell einzusetzen.
- ▶ Wenn Sie bereits gut programmieren können und dieses Buch als Umstieg von einer anderen Sprache nach Python verwenden möchten, sollten Sie die ersten beiden Teile des Buchs lesen, um die Syntax und die Konzepte von Python kennen zu lernen und mögliche Wissenslücken zu füllen. Abhängig von Ihrem Kenntnisstand wird es Ihnen nicht schwer fallen, die Sprache Python zu erlernen. Im Anschluss daran können Sie Ihre Python-Kenntnisse in Teil 4 vertiefen. Die Besprechung zentraler Module ist in Teil 3 angesiedelt, der als Nachschlagewerk dient.
- ▶ Als letzte Zielgruppe kommen erfahrene Python-Programmierer in Betracht. Sollte der Umgang mit Python für Sie zum alltäglichen Geschäft gehören, können Sie den ersten und zweiten Teil des Buchs überfliegen. 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 bieten. Außerdem gibt dieses Buch einige interessante Praxistipps, mit denen Sie Ihre Ziele schneller als bisher erreichen können.

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.

Neuerungen in der dritten Auflage

Vier Jahre nach Erscheinen der ersten Auflage und drei Jahre nach der zweiten, freuen wir uns, die dritte Auflage unseres Python-Handbuchs zu präsentieren. Wieder hat sich während dieser Zeit in der Python-Welt viel verändert. So ist Python 3.x inzwischen weitgehend akzeptiert und wird von den meisten Drittanbieterbibliotheken unterstützt.

Für die dritte Auflage haben wir das Buch noch einmal vollständig überarbeitet und insbesondere im Hinblick auf Übersichtlichkeit und die Verwendung als Nachschlagewerk verbessert. Auch unter didaktischen Gesichtspunkten sind wir das Buch durchgegangen und haben teilweise ganze Kapitel umstrukturiert. Selbstverständlich wurden alle Erklärungen und Beispiele an die aktuellste Python-Version angepasst.

Abgesehen von den angesprochenen Verbesserungen gibt es zwei neue Kapitel: eines über die Programmierung grafischer Benutzeroberflächen mit PyQt und eines über wissenschaftliches Rechnen mit NumPy und SciPy. Das PyQt-Kapitel war bereits in der ersten Auflage enthalten und konnte nun, da PyQt in einer mit Python 3 kompatiblen Version erschienen ist, wieder aufgenommen werden.

Danksagung

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

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

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

Wir danken auch 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 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 unsere Lektorinnen Judith Stevens-Lemoine und Anne Scheibe, die uns geholfen haben, 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 beiden Auflagen herzlich bedanken, deren unzählige E-Mails mit Hinweisen, Kommentaren, Kritik und Fehlerkorrekturen in die dritte Auflage mit eingeflossen sind.

Johannes Ernesti – je@lpe-media.de

Peter Kaiser – p@penguin-p.de

Teil I

Einstieg in Python

Kapitel 1

Überblick über Python

*»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«

Im vorangegangenen Kapitel haben wir Ihnen einen Überblick über dieses Buch gegeben und besprochen, in welcher Weise Sie es lesen sollten. Jetzt wenden wir uns der Programmiersprache Python zu und beginnen in diesem Kapitel mit einer Einführung in die Geschichte und die grundlegenden Konzepte. Die beiden letzten Abschnitte dieses Kapitels behandeln die Einsatzmöglichkeiten und -gebiete von Python. Betrachten Sie dieses Kapitel also als erzählerische Einführung in die Thematik, die den darauffolgenden fachlichen Einstieg vorbereitet.

1.1 Geschichte und Entstehung

Python wurde Anfang der 90er Jahre von dem Niederländer *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, sodass sie problemlos einem interessierten Laien ohne Programmiererfahrung beigebracht werden kann. Die Erfahrung aus positiver und negativer Kritik an *ABC* nutzte van Rossum für die Entwicklung von Python. Er schuf somit eine Programmiersprache, die mächtig und zugleich einfach und leicht zu erlernen sein sollte.

Mit Python 3.0, das im Dezember 2008 erschien, wurde die Sprache von Grund auf überarbeitet. Dabei sind viele kleine Unschönheiten und Design-Fehler beseitigt worden, die man in bisherigen Versionen aufgrund der Abwärtskompatibilität stets in der Sprache behalten musste.

Zum Zeitpunkt der Drucklegung dieses Buchs im Mai 2012 liegt Python in den Versionen 2.7 und 3.2 vor.

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.

1.2 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*. Ein Python-Programm ist auf allen Betriebssystemen unmodifiziert lauffähig, die von dem Python-Interpreter unterstützt werden. Darunter fallen insbesondere die drei großen Desktop-Betriebssysteme Windows, Linux und Mac OS.

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 Python 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 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 Pro-

grammiersprache, nur eingeschränkte Möglichkeiten zur maschinennahen Programmierung bietet, können maschinennahe oder zeitkritische Erweiterungen problemlos in C geschrieben werden. Das ermöglicht die sogenannte *Python API*.

In Abbildung 1.1 ist das Zusammenwirken der bisher angesprochenen Konzepte Pythons zusammengefasst: Ein Python-Programm wird vom Python-Interpreter ausgeführt, der dabei eine umfangreiche Standardbibliothek bereitstellt, die vom Programm verwendet werden kann. Außerdem erlaubt es die Python API einem externen C-Programm, den Interpreter zu verwenden oder zu erweitern.

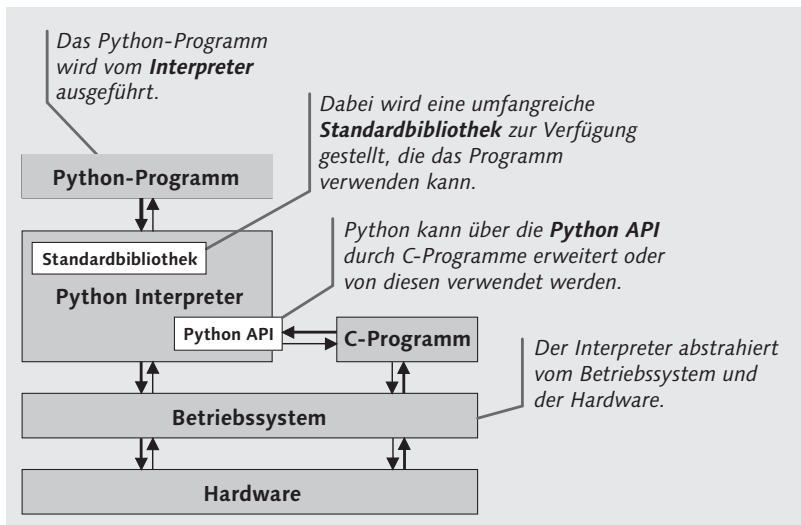


Abbildung 1.1 Veranschaulichung der grundlegenden Konzepte Pythons

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.

1.3 Einsatzmöglichkeiten und Stärken

In diesem Abschnitt stellen wir Ihnen einige Punkte vor, in denen sich Python positiv von anderen Programmiersprachen abhebt, und benennen Gebiete, in denen Python verwendet wird.

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 Marktes breitet sich Python beispielsweise im Embedded-Bereich aus. So existieren 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 auch 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.

1.4 Einsatzbeispiele

Python erfreut sich großer Bekanntheit und Verbreitung sowohl bei Software-Firmen als auch in der Open-Source-Gemeinschaft. Die Palette der Produkte, die zumindest zum Teil in Python geschrieben wurden, reicht von Web-Anwendungen (Google Mail, Google Maps, YouTube, Dropbox, reddit) über Filesharing-Plattformen (BitTorrent, Morpheus) und Entwicklungswerkzeuge (Mercurial, SCons) bis hin zu Computerspielen (Civilization IV, Battlefield 2, Eve Online).

Viele Anwendungen unterstützen Python als Skriptsprache für Erweiterungen. Beispiele hierfür sind die Grafikanwendungen Maya, Blender, Cinema 4D, Paint Shop Pro und GIMP.

Neben den Genannten gibt es unzählige weitere bekannte Anwendungen, die in Python geschrieben wurden oder in deren Umfeld Python eingesetzt wird. Lassen Sie sich anhand der obigen Beispiele sagen, dass Python eine beliebte, verbreitete und moderne Programmiersprache ist, die es sich lohnt zu erlernen.

Kapitel 2

Die Arbeit mit Python

»Python is more concerned with making it easy to write good programs than difficult to write bad ones.«

– Steve Holden auf `comp.lang.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.

2.1 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 3.2 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 unter anderem zum Lernen der Sprache Python interessant und wird deshalb in diesem Buch häufig verwendet.

Bei *IDLE* (Integrated DeveLopment Environment) handelt es sich um eine rudimentäre Python-Entwicklungsumgebung mit grafischer Benutzeroberfläche. Beim Starten von IDLE wird zunächst nur ein Fenster geöffnet, das eine Python-Shell beinhaltet. Zudem kann in IDLE über den Menüpunkt **FILE • NEW WINDOW** eine neue Python-Programmdatei erstellt und editiert werden. Nachdem die Programmdatei gespeichert wurde, kann sie über den Menüpunkt **RUN • RUN MODULE** in der Python-Shell von IDLE ausgeführt werden. Abgesehen davon bietet IDLE dem Programmierer einige

Komfortfunktionen wie beispielsweise das farbige Hervorheben bestimmter Code-Elemente (»Syntax Highlighting«) oder eine automatische Codevervollständigung.

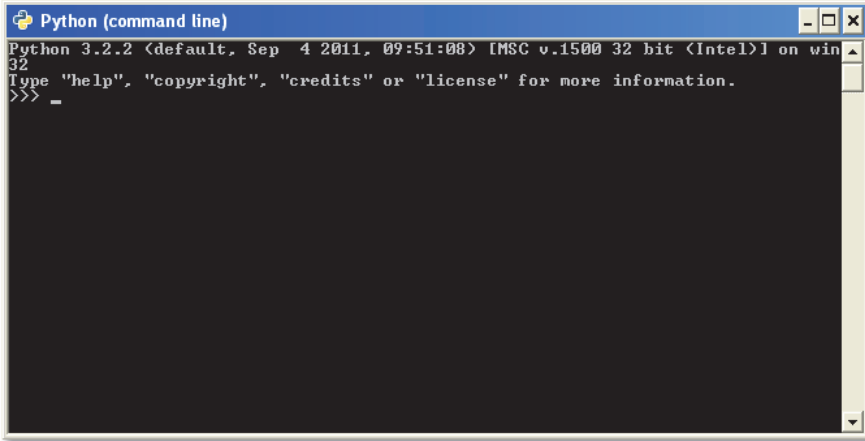


Abbildung 2.1 Python im interaktiven Modus (Python-Shell)

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

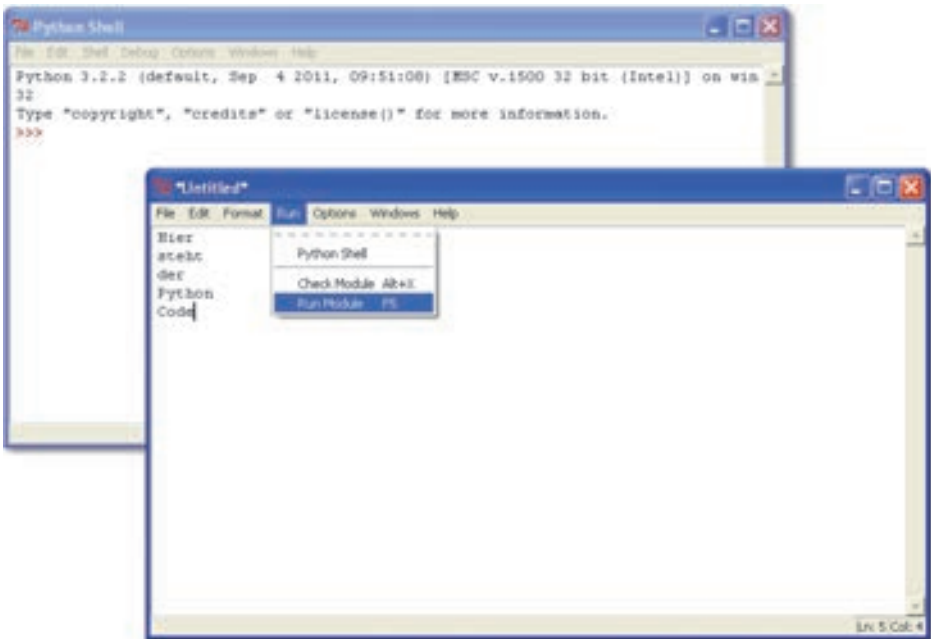


Abbildung 2.2 Die Entwicklungsumgebung IDLE

Die folgenden Abschnitte geben eine kurze Einführung darüber, wie Sie den interaktiven Modus und IDLE auf Ihrem System starten und verwenden. In Abschnitt 2.2 werden wir dann darauf eingehen, wie eine Python-Programmdatei erstellt und ausgeführt wird.

2.1.1 Windows

Sie finden die Windows-Installationsdatei von Python 3.2 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.

2.1.2 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 3.2 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 3.2 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 3.2 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.x mit einem anderen Befehl, beispielsweise `python3`, starten müssen, da diese Python 2.x und 3.x parallel installieren.



2.1.3 Mac OS X

Sie finden die Mac OS X-Installationsdatei von Python 3.2 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 `python3` bzw. `idle`.

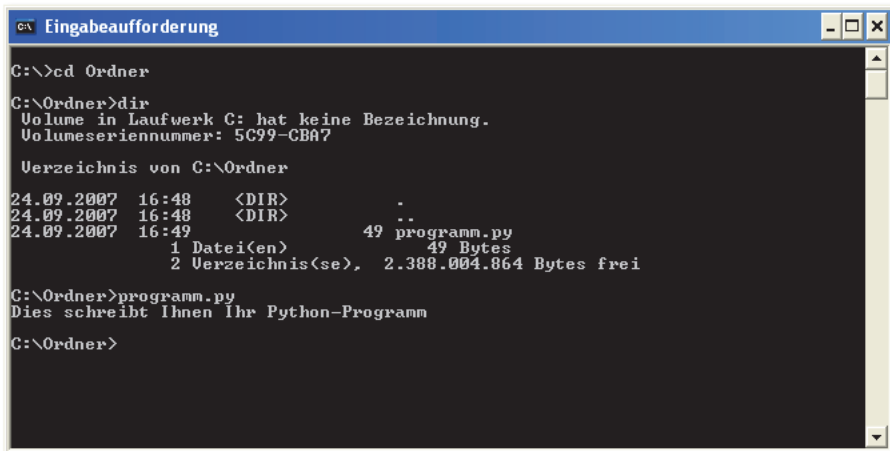
2.2 Tippen, kompilieren, testen

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 (auch *Shell*) 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 wurde, 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 Enter zu bestätigen. Im folgenden Beispiel, zu sehen in Abbildung 2.3, wird die Programmdatei *programm.py* im Ordner *C:\Ordner* ausgeführt. Dazu müssen Sie ein Konsolenfenster unter **START • PROGRAMME • ZUBEHÖR • EINGABEAUFFORDERUNG** starten.



```

C:\>cd Ordner
C:\Ordner>dir
Volume in Laufwerk C: hat keine Bezeichnung.
Volumeseriennummer: 5C99-CBA7

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 2.3 Ausführen eines Python-Programms unter Windows

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



Hinweis

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

Unter Unix-ähnlichen Betriebssystemen wie Linux oder Mac OS X 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 (Abbildung 2.4) wird die Programmdatei `programm.py` unter Linux ausgeführt, die sich im Verzeichnis `/home/user/ordner` befindet.



Abbildung 2.4 Ausführen eines Python-Programms unter Linux

Bitte beachten Sie den Hinweis aus Abschnitt 2.1.2, der besagt, dass das Kommando, mit dem Sie Python 3.2 starten, je nach Distribution von dem hier demonstrierten `python` abweichen kann.

2.2.1 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.

2.2.2 Interne Abläufe

Bislang haben Sie einen ungefähren Begriff davon, was Python ausmacht und wo die Stärken der Programmiersprache liegen. Außerdem wurde das 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. Abbildung 2.5 veranschaulicht, was beim Ausführen einer Programmdatei namens *programm.py* geschieht.

Wenn die Programmdatei *programm.py*, wie zu Beginn des Kapitels beschrieben, ausgeführt wird, passiert sie zunächst den sogenannten *Compiler*. Ein Compiler ist ein allgemeiner Begriff der Informatik und bezeichnet ein Programm, das von einer formalen Sprache in eine andere übersetzt. Im 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.

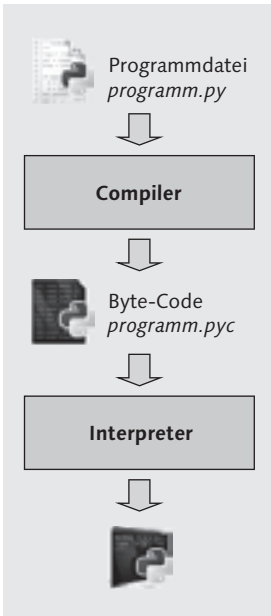


Abbildung 2.5 Kompilieren und Interpretieren einer Programmdatei

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

Kapitel 3

Der interaktive Modus

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

Startet man den Python-Interpreter ohne Argumente¹, 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 3.2.2 (default, Nov 21 2011, 16:50:59)
[GCC 4.6.2] 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 sei noch gesagt, dass er über eine *History-Funktion* verfügt. Das heißt, dass Sie über die - und -Tasten alte Eingaben

¹ Sie starten den Python-Interpreter ohne Argumente, indem Sie in einer Konsole Ihres Betriebssystems den Befehl `python` eingeben. Unter Windows findet sich der interaktive Modus auch im Startmenü unter dem Punkt **Python (command line)**.

Wenn Sie, statt den interaktiven Modus zu verwenden, ein konkretes Python-Programm starten möchten, rufen Sie den Interpreter mit dem Dateinamen der Programmdatei als Argument auf: `python programm.py`.

bequem wieder hervorholen können und nicht erneut eingeben müssen. Auch das Verändern der hervorgeholten Eingaben ist möglich.

Wir beginnen mit der Einführung einiger grundlegender *Datentypen*. Dabei beschränken wir uns zunächst auf ganze Zahlen, Gleitkommazahlen, Zeichenketten und Listen. 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*.

3.1 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 + 6
168
```

Hier zeigt sich, dass der Interpreter die gewohnten mathematischen Rechengesetze anwendet und das erwartete Ergebnis ausgibt. Das Divisionsergebnis zweier ganzer Zahlen ist nicht zwingend wieder eine ganze Zahl, weswegen der Divisionsoperator das Ergebnis stets als *Gleitkommazahl* zurückgibt:

```
>>> 3/2
1.5
>>> 2/3
0.6666666666666666
>>> 4/4
1.0
```

Hinweis

Diese Eigenschaft unterscheidet Python bereits von vielen anderen Programmiersprachen. Dort wird bei der Division zweier ganzer Zahlen eine ganzzahlige Division durchgeführt und das Ergebnis ebenfalls als ganze Zahl zurückgegeben. Diese sogenannte *Integer-Division* wurde in Python-Versionen vor 3.0 auch angewendet.²



3.2 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
```

² Die Integer-Division 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:

```
>>> 3//2
1
>>> 2//3
0
```

Soviel zunächst zu ganzen Zahlen und Gleitkommazahlen. In Kapitel 7 (»Basisdatentypen«) werden wir auf diese grundlegenden Datentypen zurückkommen und sie in aller Ausführlichkeit behandeln. Doch nun zu einem weiteren wichtigen Datentyp, den Zeichenketten.

3.3 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 String zu erzeugen, wird der zugehörige Text in doppelte Hochkommata geschrieben:

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

Die einfachen Hochkommata, die der Interpreter verwendet, um den Wert eines Strings auszugeben, sind eine äquivalente Schreibweise zu den von uns verwendeten doppelten Hochkommata, die Sie auch benutzen dürfen:

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

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

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



Hinweis

Im Zuge der Umstrukturierung der Sprache Python mit der Version 3.0 wurde der String-Datentyp im Hinblick auf Unicode stark überarbeitet. Der neue String-Datentyp repräsentiert nicht mehr eine Sequenz von Bytes, sondern abstrakten Text. Dieser Text kann durch Kodieren in eine Byte-Sequenz umgewandelt werden. Näheres zu diesem Thema erfahren Sie in Abschnitt 7.5.4 (S. 142).

3.4 Listen

Wenden wir uns nun dem vierten und vorerst letzten Datentyp zu, den Listen. Eine Liste ist eine geordnete Ansammlung von Elementen beliebigen Datentyps. Um eine Liste zu erzeugen, werden die Literale der Werte, die sie enthalten soll, durch Komma getrennt in eckige Klammern geschrieben:

```
>>> [1,2,3]
[1, 2, 3]
>>> ["Dies", "ist", "eine", "Liste"]
['Dies', 'ist', 'eine', 'Liste']
```

Die Elemente einer Liste müssen nicht alle den gleichen Typ haben und können insbesondere selbst wieder Listen sein, wie folgendes Beispiel zeigt:

```
>>> ["Python", 1, 2, -7 / 4, [1,2,3]]
['Python', 1, 2, -1.75, [1, 2, 3]]
```

Ähnlich wie Strings lassen sich Listen mit dem Operator + um Elemente erweitern, + bildet die Aneinanderreihung zweier Listen.

```
>>> [1,2,3] + ["Python", "ist", "super"]
[1, 2, 3, 'Python', 'ist', 'super']
```

Auf dieser grundlegenden Darstellung der vier Datentypen Ganzzahl, Gleitkommazahl, String und Liste werden wir in den folgenden Abschnitten aufbauen, bis wir in Kapitel 7 (»Basisdatentypen«, Seite 87) noch einmal ausführlich auf alle in Python eingebauten Datentypen eingehen werden.

3.5 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!"
>>> liste = [1,2,3]
```


Die mit den Namen verknüpften Werte können später ausgegeben oder in Berechnungen verwendet werden, indem der Name anstelle des jeweiligen Wertes 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.³ Beachten Sie, dass auch Umlaute und spezielle Buchstaben anderer Sprachen erlaubt sind, wie folgendes Beispiel zeigt:

```
>>> äöüßèè = 123
>>> äöüßèè
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 `ß` fallen auch bei deutschen Bezeichnern kaum ins Gewicht und wirken im Quellcode eher verwirrend als natürlich.

3 Häufig werden Variablen, die nur eine lokale und kurzfristige Bedeutung haben, mit einem kurzen, oft einbuchstabigen Namen versehen. Hierbei sollten Sie beachten, dass die Buchstaben »o«, »O«, »l« und »I« in manchen Schriftarten wie Zahlen aussehen und damit für einen Variablennamen ungeeignet sind.

Bestimmte sogenannte *Schlüsselwörter*⁴ sind in Python für die Sprache selbst reserviert und dürfen nicht als Bezeichner verwendet werden. Die folgende Tabelle liefert eine Übersicht über alle in Python reservierten Wörter.

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 3.1 Schlüsselwörter in Python

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

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

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

Hinweis

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.

[+]

3.6 Logische Ausdrücke

Abgesehen von den arithmetischen Operatoren, gibt es einen zweiten Satz von Operatoren, die das Vergleichen von Zahlen ermöglichen:

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

⁴ Unter einem Schlüsselwort versteht man in der Programmierung ein Wort, das eine bestimmte Bedeutung trägt, beispielsweise den Programmablauf steuert. In Python existieren zum Beispiel die Schlüsselworte `if` und `for`, die eine Fallunterscheidung bzw. eine Schleife einleiten.

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 die 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 3.2 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 sinnvoll, durch Klammerung die Zugehörigkeiten visuell eindeutig zu gestalten. Eine Tabelle mit den Prioritätsregeln für Operatoren, der sogenannten Operatorrangfolge, finden Sie in Abschnitt 7.1 (S. 88).

3.7 Funktionen und Methoden

In diesem Kapitel wird ein grundlegendes Wissen über Funktionen und einige Konzepte der objektorientierten Programmierung vermittelt. Dabei beschränken wir uns auf die Aspekte, die in den folgenden Kapiteln benötigt werden. Beide Themen werden in den Kapiteln 9 (S. 209) bzw. 11 (S. 269) noch einmal ausführlich behandelt.

3.7.1 Funktionen

In Python können Teile eines Programms in *Funktionen* gekapselt und danach über einen *Funktionsaufruf* ausgeführt werden. Das Ziel dieses Vorgehens ist es, Redun-

danz im Quellcode zu vermeiden. Funktionalität, die häufig benötigt wird, sollte stets nur einmal als Funktion implementiert und dann als solche im weiteren Programm verwendet werden. Außerdem kann der Einsatz von Funktionen die Les- und Wartbarkeit des Quellcodes deutlich erhöhen.

Python bietet einen Satz von eingebauten Funktionen (*Built-in Functions*), die der Programmierer zu jeder Zeit verwenden kann. Als Beispiel dient in diesem Abschnitt die eingebaute Funktion `max`, die das größte Element einer Liste bestimmt:

```
>>> max([1,5,2,7,9,3])
9
```

Eine Funktion wird aufgerufen, indem man den Funktionsnamen, gefolgt von den *Funktionsparametern* in Klammern, schreibt. Im Beispiel erwartet die Funktion `max` genau einen Parameter, nämlich eine Liste der zu betrachtenden Werte. Das Ergebnis der Berechnung wird als *Rückgabewert* der Funktion zurückgegeben. Sie können sich vorstellen, dass der Funktionsaufruf im Quelltext durch den Rückgabewert ersetzt wird.

Es gibt eine Variante der Funktion `max`, die statt des größten Elements einer Liste den größten ihr übergebenen Parameter bestimmt. Um einer Funktion mehrere Parameter zu übergeben, werden diese beim Funktionsaufruf durch Kommata getrennt in die Klammern geschrieben:

```
>>> max(1,5,3)
5
```

Selbstverständlich können Sie in Python eigene Funktionen definieren, hier genügt es jedoch zu wissen, wie bereits vorhandene Funktionen verwendet werden. In Kapitel 9 (S. 209) kommen wir noch einmal ausführlich auf Funktionen zu sprechen.

3.7.2 Methoden

Das Erzeugen eines Wertes eines bestimmten *Datentyps*, beispielsweise das Erzeugen einer ganzen Zahl über ihr Literal, wird *Instanziiieren* genannt und der entstandene Wert *Instanz*. So ist beispielsweise 1 eine Instanz des Datentyps »ganze Zahl« oder `[4,5,6]` eine Instanz des Datentyps »Liste«. Der Datentyp einer Instanz legt einerseits fest, was für Daten gespeichert werden, und definiert andererseits einen Satz von Operationen, die auf diesen Daten durchgeführt werden können. Ein Teil dieser Operationen wird durch Operatoren abgebildet, so bietet beispielsweise der Datentyp »Gleitkommazahl« den Operator `+` zum Addieren zweier Gleitkommazahlen an. Für die einfachen numerischen Datentypen sind einige wenige Operatoren ausreichend, um mit ihnen arbeiten zu können. Bei komplexeren Datentypen, bei-

spielsweise den Listen, sind eine ganze Reihe von Operationen denkbar, die allein über Operatoren nicht abgebildet werden können. Für solche Fälle können Datentypen sogenannte *Methoden* definieren. Dabei handelt es sich um Funktionen, die im Kontext einer bestimmten Instanz ausgeführt werden.

Der Datentyp »Liste« bietet zum Beispiel eine Methode `sort` an, mit deren Hilfe eine Liste sortiert werden kann. Um eine Methode aufzurufen, wird eine Instanz (oder eine Referenz darauf), gefolgt von einem Punkt und dem Methodenaufruf, geschrieben. Dieser ist wie ein Funktionsaufruf aufgebaut:

```
>>> liste = [2,7,3,2,7,8,4,2,5]
>>> liste.sort()
>>> liste
[2, 2, 2, 3, 4, 5, 7, 7, 8]
```

Ein weiteres Beispiel bietet die Methode `count` des Datentyps »String«, die zählt, wie oft ein Zeichen in einem String vorkommt.

```
>>> "Hallo Welt".count("l")
3
```

Das hier erworbene Wissen über Funktionen und Methoden wird zu gegebener Zeit vertieft. Im Folgenden wird die für den Anfang wohl wichtigste in Python eingebaute Funktion besprochen: `print`.

3.8 Bildschirmausgaben

Auch wenn wir hin und wieder auf den interaktiven Modus zurückgreifen werden, ist es 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 die Funktion `print` verwendet:

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

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

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

Auch hier ist es problemlos möglich, 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.



Hinweis

Die Funktion `print` wurde in Python 3.0 eingeführt. In früheren Python-Versionen konnten Bildschirmausgaben über das Schlüsselwort `print` erzeugt werden:

```
>>> print "Dies", "ist", "Python", 2
Dies ist Python 2
```

In den meisten Fällen sind die fehlenden Klammern der einzige Unterschied. Näheres zu den Unterschieden zwischen Python 2 und 3 finden Sie in Kapitel 27 (S. 955).

Kapitel 4

Grundlegendes zu Python-Programmen

4

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

4.1 Grundstruktur eines Python-Programms

Um Ihnen ein Gefühl für die Sprache Python zu vermitteln, möchten wir zunächst einen Überblick über ihre Syntax geben. 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*.

Python macht dem Programmierer sehr genaue Vorgaben, 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 4.1 Struktur einer mehrzeiligen Anweisung

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

```
if x > 10:
    print("x ist größer als 10")
    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.



Hinweis

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 `[Enter]`-Taste beendet werden.

4.2 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:

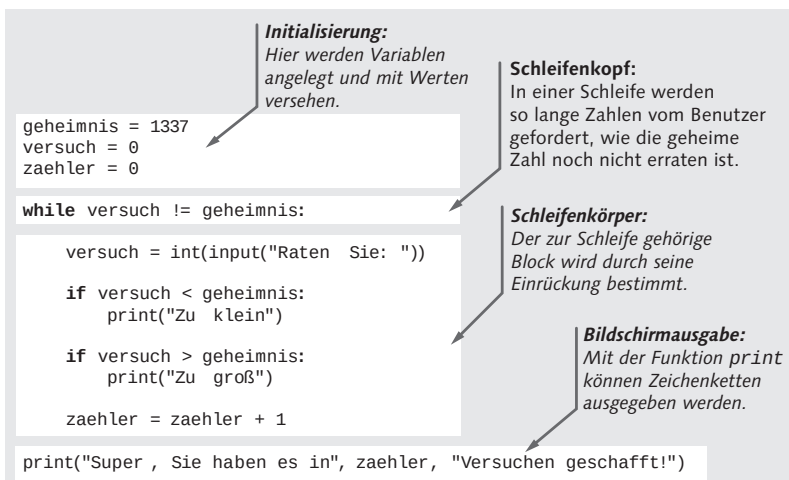


Abbildung 4.2 Zahlenraten, ein einfaches Beispiel

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 Datentypen, 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 Datentyp explizit anzugeben. Eine Variable kann im Laufe des Programms ihren Typ ändern.

In unserem Spiel werden Variablen für die gesuchte Zahl (`geheimnis`), die Benutzereingabe (`versuch`) und den Versuchszähler (`zaehler`) angelegt und mit Anfangswerten versehen. Dadurch, dass `versuch` und `geheimnis` 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 (`versuch != geheimnis`) erfüllt ist, also in diesem Fall, bis die Variablen `versuch` und `geheimnis` 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 `versuch` gespeichert. Dabei wird mithilfe von `input("Raten Sie: ")` die Eingabe des Benutzers eingelesen und mit `int` in eine ganze Zahl konvertiert (von engl. *integer*, »ganze Zahl«). 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 `versuch` größer oder kleiner als die gesuchte Zahl `geheimnis` ist, und mittels `print` eine entsprechende Meldung ausgegeben. Schlussendlich wird der Versuchszähler `zaehler` 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 *geheimnis*, und spielen Sie das Spiel.

4.3 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 (*"""*)¹:

```
""" 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.

¹ Eigentlich wird mit dieser Notation kein Blockkommentar erzeugt, sondern ein mehrzeiliger String, der sich aber auch dazu eignet, größere Quellcodebereiche »auszukomentieren«.

4.4 Der Fehlerfall

Vielleicht haben Sie bereits ein wenig mit dem Beispielprogramm aus Abschnitt 4.2 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 eingerückt")
```

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 eingerückt")
      ^
```

IndentationError: expected an indented block

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

Kapitel 5

Kontrollstrukturen

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

5

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

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

5.1 Fallunterscheidungen

In Python gibt es zwei Arten von Fallunterscheidungen: die klassische *if-Anweisung*¹ 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.

¹ Als Anweisung (engl. Statement) wird eine einzelne Vorschrift im Quellcode bezeichnet. Es gibt einzeilige Anweisungen, beispielsweise Zuweisungen, aber auch mehrzeilige Anweisungen, die weitere Anweisungen enthalten können. Ein Beispiel für eine mehrzeilige Anweisung ist die angesprochene *if-Anweisung*.

Beachten Sie, dass es einen Unterschied zwischen den Begriffen »Anweisung« und »Ausdruck« gibt. Im Gegensatz zu einer Anweisung hat ein Ausdruck immer einen Wert.

5.1.1 Die if-Anweisung

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 5.1 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 als Wahrheitswert (*True* oder *False*) interpretiert werden kann. Typischerweise werden hier die logischen Ausdrücke angewendet, die in Abschnitt 3.6 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 bräuchte die zweite Fallunterscheidung nicht mehr in Betracht gezogen zu werden, wenn die Bedingung der ersten bereits `True` ergeben hat. Die Variable `x` kann unter keinen Umständen sowohl den Wert 1 als auch 2 haben. Um solche Fälle aus Sicht des Interpreters performanter und aus Sicht des Programmierers übersichtlicher zu machen, kann eine `if`-Anweisung um einen oder mehrere sogenannte `elif`-Zweige (»`elif`« ist ein Kürzel für »else if«) erweitert werden.

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

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

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

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

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

Abbildung 5.2 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 5.3 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 fasst Abbildung 5.4 den Aufbau einer `if`-Anweisung noch einmal übersichtlich zusammen.

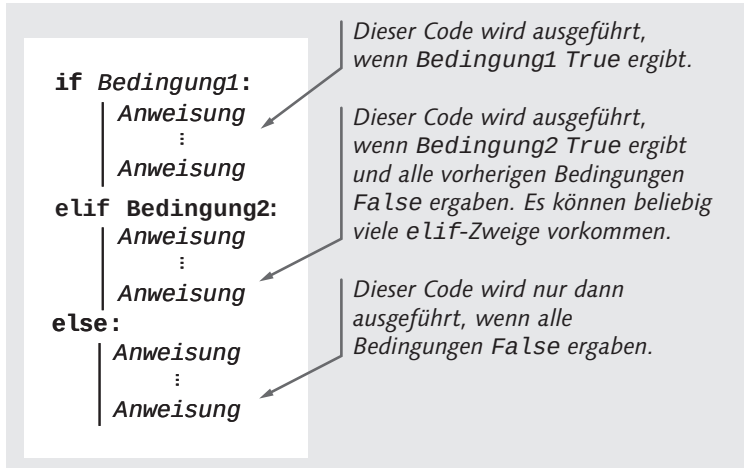


Abbildung 5.4 Aufbau einer if-Anweisung

5.1.2 Conditional Expressions

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

```

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

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

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

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

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

A if Bedingung else B

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

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

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

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

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

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

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



Hinweis

Die hier vorgestellten Conditional Expressions sind in Python 2.5 eingeführt worden und in früheren Versionen daher nicht verfügbar.

5.2 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.

5.2.1 Die 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 4.2 (S. 53) wurde mithilfe einer `while`-Schleife so lange eine neue Zahl vom Spieler eingelesen, bis die eingegebene Zahl mit der gesuchten Zahl übereinstimmte.

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

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

Abbildung 5.5 Struktur einer `while`-Schleife

Das folgende Beispiel ist eine etwas verknappte Variante des »Zahlenraten«-Spiels und soll die Verwendung der `while`-Schleife veranschaulichen:

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

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 `versuch` versehen. Dieser Prozess läuft so lange, bis die im Schleifenkopf genannte Bedingung erfüllt ist, bis also die Eingabe des Benutzers (`versuch`) mit der geheimen Zahl (`geheimnis`) übereinstimmt.

5.2.2 Vorzeitiger Abbruch einer Schleife

Da die im vorherigen Abschnitt eingeführte Variante des »Zahlenraten«-Spiels keine Hinweise darauf gibt, in welchen Bereichen die gesuchte Zahl liegt, kann ein Spiel recht lange dauern. Wir möchten dem Benutzer in diesem Abschnitt die Möglichkeit

geben, das Spiel durch Eingabe einer 0 vorzeitig abzubrechen. Dies lässt sich durch eine Modifikation der Schleifenbedingung zu

```
versuch != geheimnis and versuch != 0
```

erreichen. Das ist in diesem Fall eine annehmbare Lösung, doch wenn die Schleifenbedingung an sich bereits komplex ist und zusätzlich noch mehrere Abbruchbedingungen hinzugefügt werden, leidet die Lesbarkeit des Quellcodes stark.

Eine alternative Lösung für das obige Problem ist es, die Schleife über das Schlüsselwort `break` abzubrechen.

```
geheimnis = 1337
versuch = 0
while versuch != geheimnis:
    versuch = int(input("Raten Sie: "))
    if versuch == 0:
        print("Das Spiel wird beendet")
        break
print("Sie haben es geschafft!")
```

Direkt nach der Benutzereingabe wird mit einer `if`-Anweisung geprüft, ob es sich bei der Eingabe um eine 0 handelt. Sollte dies der Fall sein, wird eine entsprechende Meldung ausgegeben und die `while`-Schleife mit `break` beendet.

5.2.3 Erkennen eines Schleifenabbruchs

Im vorherigen Abschnitt wurde dem Benutzer die Möglichkeit gegeben, das »Zahlenraten«-Spiel durch Eingabe einer 0 vorzeitig zu beenden. Leider wird die Erfolgsmeldung, die dem Spieler eigentlich signalisieren soll, dass er die gesuchte Zahl erraten hat, in jedem Fall nach Beendigung der Schleife angezeigt, also auch nachdem der Benutzer das Spiel abgebrochen hat:

```
Raten Sie: 10
Raten Sie: 20
Raten Sie: 30
Raten Sie: 0
Das Spiel wird beendet
Sie haben es geschafft!
```

An dieser Stelle suchen wir also nach einer Möglichkeit, zu erkennen, ob die Schleife aufgrund der Schleifenbedingung oder aufgrund einer `break`-Anweisung beendet wurde. Dazu kann eine `while`-Schleife ähnlich wie eine `if`-Anweisung 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. Insbesondere wird der `else`-Zweig nicht ausgeführt, wenn die Schleife durch eine `break`-Anweisung vorzeitig abgebrochen wurde.

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

Abbildung 5.6 Struktur einer `while`-Schleife mit `else`-Zweig

Betrachten wir dies an einem konkreten Beispiel:

```
geheimnis = 1337
versuch = 0
while versuch != geheimnis:
    versuch = int(input("Raten Sie: "))
    if versuch == 0:
        print("Das Spiel wird beendet")
        break
else:
    print("Sie haben es geschafft!")
```

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

```
Raten Sie: 10
Raten Sie: 1337
Sie haben es geschafft!
```

Umgekehrt, wenn der Benutzer zum Spielabbruch die 0 eingibt, wird der `else`-Zweig nicht ausgeführt und damit auch keine Erfolgsmeldung ausgegeben:

```
Raten Sie: 10
Raten Sie: 0
Das Spiel wird beendet
```


5.2.4 Vorzeitiger Abbruch eines Schleifendurchlaufs

Wir haben mit `break` bereits eine Möglichkeit vorgestellt, den Ablauf einer Schleife zu beeinflussen. Eine zweite Möglichkeit bietet die `continue`-Anweisung, die im Gegensatz zu `break` nicht die gesamte Schleife, sondern nur den aktuellen Schleifendurchlauf abbricht. 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 Fakultät² 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
```

2 Die Fakultät einer natürlichen Zahl n ist das Produkt aller natürlichen Zahlen, die kleiner oder gleich n sind: $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n$.

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 das, was man in diesem Fall erwarten 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. 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 bedeutet, dass unmittelbar zum nächsten Schleifendurchlauf gesprungen wird, also die Schleifenbedingung geprüft und dann die nächste Zahl vom Benutzer eingelesen wird. 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
Geben Sie eine Zahl ein: 6
Ergebnis: 720
```

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

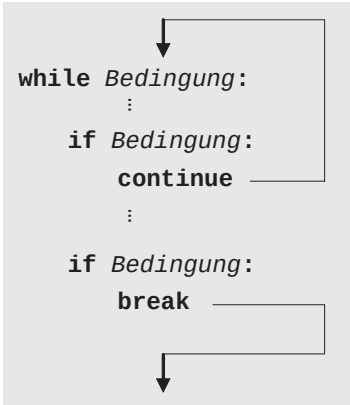


Abbildung 5.7 Eine Schleife mit break und continue

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

5.2.5 Die for-Schleife

Neben der bisher behandelten `while`-Schleife existiert in Python ein weiteres Schleifenkonstrukt, die sogenannte `for`-Schleife. Eine `for`-Schleife wird verwendet, um ein *iterierbares Objekt*³ zu durchlaufen. Dazu wird das Schlüsselwort `for`, gefolgt von einem Bezeichner, dem Schlüsselwort `in` und dem iterierbaren Objekt, geschrieben. Darauf folgt, eine Ebene weiter eingerückt, der Schleifenkörper.

```

for Variable in Objekt:
    Anweisung
    :
    Anweisung

```

Abbildung 5.8 Struktur einer for-Schleife

Konkret kann eine `for`-Schleife beispielsweise Listen oder Strings durchlaufen:

```

>>> for x in [1,2,3]:
...     print(x)
...
1
2

```

³ Ein iterierbares Objekt ist eine Instanz eines Datentyps, der das Iterator-Protokoll implementiert. Sie werden, abgesehen von den bereits bekannten iterierbaren Datentypen Listen und Strings, noch viele weitere kennen lernen, die sich mit einer `for`-Schleife durchlaufen lassen. Näheres zu iterierbaren Objekten finden Sie in Abschnitt 12.4 (S. 353)

```

3
>>> for c in "Python":
...     print(c)
...
P
y
t
h
o
n

```

Wir werden im Laufe dieses Buchs noch einige Datentypen kennen lernen, die auf diese Weise mit einer `for`-Schleife durchlaufen werden können.

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.



Die im Zusammenhang mit der `while`-Schleife besprochenen Schlüsselworte `break` und `continue` (siehe S. 65 und S. 68) zum Abbrechen einer Schleife bzw. eines Schleifendurchlaufs können auch mit der `for`-Schleife verwendet werden und haben dort dieselbe Bedeutung. Außerdem kann eine `for`-Schleife analog zur `while`-Schleife über einen `else`-Zweig verfügen, der genau dann ausgeführt wird, wenn die Schleife vollständig durchgelaufen ist und nicht mittels `break` vorzeitig abgebrochen wurde.

```

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

```

Abbildung 5.9 Struktur einer `for`-Schleife mit `else`-Zweig

5.2.6 Die `for`-Schleife als Zählschleife

Im Zusammenhang mit der `for`-Schleife ist die eingebaute Funktion `range` besonders interessant, die ein iterierbares Objekt erzeugt, das alle ganzen Zahlen eines bestimmten Bereichs durchläuft:

```
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.

5.3 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.

Kapitel 6

Das Laufzeitmodell

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

– H. L. Mencken

6

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.

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 Referenzen ins Spiel. »Referenz« bedeutet so viel wie »Verweis«. Erst durch Referenzen wird es möglich, mit Instanzen zu arbeiten, weil Referenzen uns den Zugriff auf diese ermöglichen. Die einfachste Form einer Referenz in Python ist ein *symbolischer Name* wie beispielsweise `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 `a` auf diese Instanz an. Dies lässt sich auch grafisch verdeutlichen:

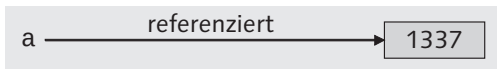


Abbildung 6.1 Schema der Referenz-Instanz-Beziehung

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

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

Grafisch veranschaulicht, sieht das Ergebnis so aus:



Abbildung 6.2 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
```

```

2674
>>> referenz2
1337

```

Bis zu den ersten beiden Ausgaben haben wir die in Abbildung 6.2 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 6.3 Die beiden Referenzen sind voneinander unabhängig.

Sie wissen nun, dass Sie Referenzen als Verweise auf Instanzen von den Instanzen selbst unterscheiden müssen. Im Folgenden werden wir uns mit den Eigenschaften von Instanzen im Detail beschäftigen.

6.1 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:

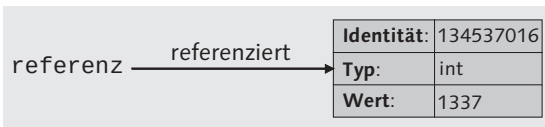


Abbildung 6.4 Eine Instanz mit ihren drei Eigenschaften

6.1.1 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

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 9 ab Seite 209 eingehend mit Funktionen beschäftigen und dort auch auf `type` zurückkommen.

```
>>> 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
>>> type(v1) == str
False
```

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 die Aussage: »`zuerst_ein_string` referenziert momentan eine Instanz des Typs `str`.«

6.1.2 Wert

Was den Wert einer Instanz konkret ausmacht, hängt von ihrem Typ ab. Dies können beispielsweise Zahlen, Zeichenketten oder Daten anderer Typen sein, die Sie später noch kennen lernen 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:

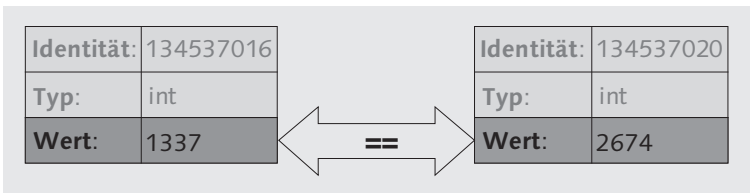


Abbildung 6.5 Wertevergleich zweier Instanzen (in diesem Fall False)

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.

6.1.3 Identität

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

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

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

```
>>> v1 = [1,2,3]
>>> v2 = v1
>>> v3 = [1,2,3]
>>> type(v1) == type(v3)
True
>>> v1 == v3
True
>>> id(v1) == id(v3)
False
>>> id(v1) == id(v2)
True
```

In diesem Beispiel hat Python zwei verschiedene Instanzen mit dem Typ `list` und dem Wert `[1,2,3]` angelegt, wobei `v1` und `v2` auf dieselbe Instanz verweisen. Abbildung 6.6 veranschaulicht dies grafisch.

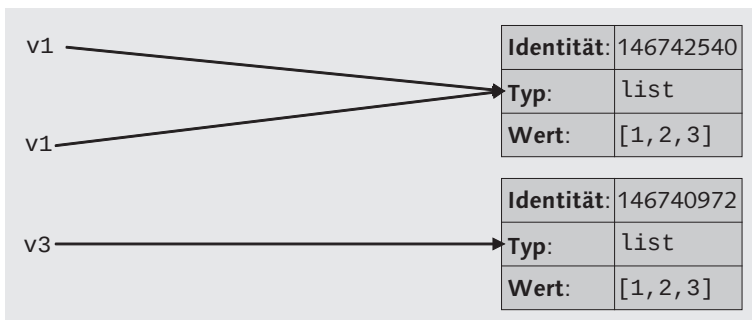


Abbildung 6.6 Drei Referenzen, zwei Instanzen.

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:

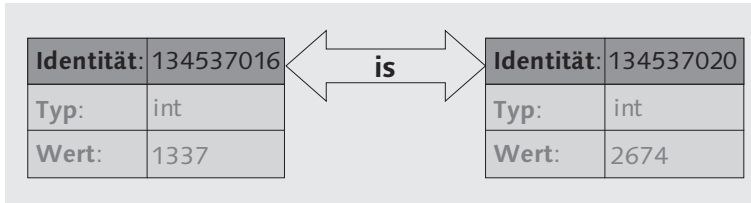


Abbildung 6.7 Identitätenvergleich zweier Instanzen

Der in Abbildung 6.7 gezeigte Vergleich ergibt den Wahrheitswert `False`, da sich die Identitäten der beiden Instanzen unterscheiden.

6.2 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*².

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

² 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.

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
```

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.

6.3 Mutable vs. immutable Datentypen

Vielleicht sind Sie beim Ausprobieren des gerade Beschriebenen schon auf den folgenden scheinbaren Widerspruch 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, dass Python grundlegend zwischen zwei Arten von Datentypen unterscheidet: zwischen *mutable* (dt. »veränderlichen«) Datentypen und *immutable* (dt. »unveränderlichen«) Datentypen. Wie die Namen schon sagen, besteht der Unterschied zwischen den beiden Arten darin, ob sich der Wert einer Instanz zur Laufzeit ändern kann, ob sie also veränderbar ist. Instanzen eines mutable Typs sind dazu in der Lage, nach ihrer Erzeugung andere Werte anzunehmen, während dies bei immutable Datentypen nicht der Fall ist.

Wenn sich der Wert einer Instanz aber nicht ändern kann, ergibt es auch keinen Sinn, mehrere immutable Instanzen des gleichen Wertes 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 solch einen immutable Datentyp, und Python hat aus Optimierungsgründen bei beiden Einsen auf dieselbe Instanz verweisen lassen. Auch Strings sind immutable.³

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 müssen.

³ Das bedeutet natürlich nicht, dass Strings und Ganzzahlen aus Sicht des Programmierers unveränderlich sind. Es wird nur bei jeder Manipulation eines immutablen Datentyps eine neue Instanz des Datentyps erzeugt, anstatt die alte zu verändern.

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.

6.3.1 Mutable Datentypen und Seiteneffekte

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.

Dieses unterschiedliche Verhalten beim Umgang mit *mutable* und *immutable* Datentypen führt dazu, dass der gleiche Code für verschiedene Datentypen fundamental anderes bewirken kann. Um Sie vor den damit verbundenen Schwierigkeiten zu warnen, dienen die folgenden Beispiele.

Zunächst führen wir den Operator `+=` ein, mit dem an einen bereits vorhandenen String ein weiterer angehängt werden kann:

```
>>> a = "Wasser"
>>> a += "flasche"
>>> a
'Wasserflasche'
```

Hier wird also zunächst ein String mit dem Wert "Wasser" erzeugt, an den dann der String "flasche" angehängt wird.

Genauso ist es möglich, mithilfe von `+=` an eine bestehende Liste die Elemente einer weiteren Liste anzuhängen:

```
>>> a = [1,2]
>>> a += [3,4]
>>> a
[1, 2, 3, 4]
```

Es deutet also alles darauf hin, dass der Operator `+=` für Strings und Listen das gleiche leistet. Tatsächlich besteht aber ein gravierender Unterschied.

Im nächsten Beispiel werden wir zweimal den gleichen Code ausführen, einmal für einen String und einmal für eine Liste.

```
>>> a = "Wasser"
>>> b = a
>>> a += "flasche"
>>> a
'Wasserflasche'
>>> b
'Wasser'
```

Wieder wird ein String mit dem Wert "Wasser" erzeugt. Dieses Mal lassen wir neben der Referenz `a` zusätzlich noch die Referenz `b` auf diesen String zeigen. Dann hängen wir an den von `a` referenzierten String erneut den String "flasche" an und sehen bei der Ausgabe, dass `b` immer noch den String "Wasser" referenziert, während `a` auf die Verkettung "Wasserflasche" zeigt.

Führen wir das gleiche mit Instanzen des Typs `list` aus, ergibt sich ein anderes Bild:

```
>>> a = [1,2]
>>> b = a
>>> a += [3,4]
>>> a
[1, 2, 3, 4]
>>> b
[1, 2, 3, 4]
```

Dieses Beispiel ist analog zum vorigen Beispiel mit den String-Instanzen: Die Liste `[1,2]` entspricht dem String "Wasser" und die Liste `[3,4]` dem String "flasche". Trotzdem hat sich die von `b` referenzierte Liste in diesem Beispiel verändert, was bei dem String-Beispiel nicht der Fall war.

Um diesen Unterschied zu verstehen, betrachten wir die Vorgänge der beiden Code-Sequenzen im Detail.

In Abbildung 6.8 sind der Beispielcode für Strings und die daraus resultierenden internen Vorgänge gegenübergestellt.

Der Knackpunkt befindet sich in der Zeile `a += "flasche"`. Da `str` ein immutable Datentyp ist, ist es nicht möglich, die bestehende `str`-Instanz "Wasser" durch Anhängen von "flasche" zu erweitern. Stattdessen wird an dieser Stelle eine neue `str`-Instanz mit dem Wert "Wasserflasche" erzeugt und anschließend von `a` referenziert. Die weiterhin von `b` referenzierte Instanz mit dem Wert "Wasser" ist davon nicht betroffen.

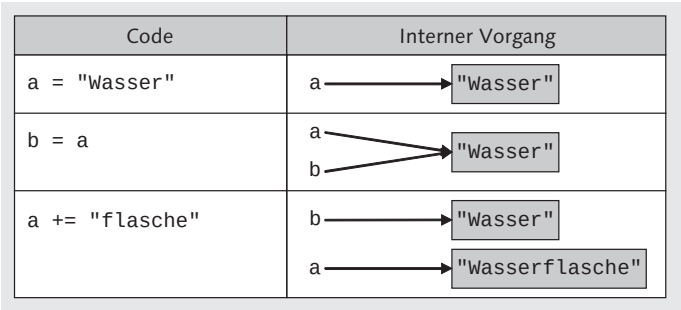


Abbildung 6.8 Der Operator += mit str-Instanzen

Anders verhält es sich beim Listenbeispiel, wie Abbildung 6.9 zeigt.

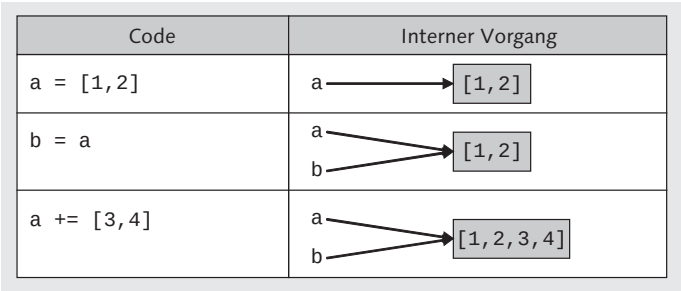


Abbildung 6.9 Der Operator += für list-Instanzen

Da list ein mutable Datentyp ist, kann die gemeinsam von a und b referenzierte list-Instanz verändert werden. Die Zeile `a += [3, 4]` bewirkt daher nicht, dass eine neue list-Instanz erzeugt wird sondern verändert die bereits bestehende Instanz. Dies hat zur Folge, dass a und b weiterhin dieselbe Liste referenzieren und somit auch b von der Manipulation an a betroffen ist.

Man nennt diesen Vorgang, bei dem etwas über Vorgänge an einer anderen Stelle beeinflusst wird, einen *Seiteneffekt*.

Seiteneffekte treten aus prinzipiellen Gründen ausschließlich bei mutable Datentypen auf und sind nicht auf den +=-Operator beschränkt. Sie kommen immer dann ins Spiel, wenn die Instanz eines mutable Datentyps verändert wird.

Während der weiteren Lektüre dieses Buchs werden sie eine ganze Reihe mutabler Datentypen und Operationen, mit denen sie verändert werden können, kennen lernen. Dabei sollten Sie das hier vorgestellte Verhalten im Hinterkopf behalten.

Kapitel 7

Basisdatentypen

»Alles ist Zahl.«
– Pythagoras

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 repräsentieren. Darüber hinaus existieren auch komplexere Datentypen, die eine Liste oder Zuordnung verschiedener 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. Die folgende Tabelle listet die Basisdatentypen auf.

Basisdatentyp	Beschreibung	Abschnitt	Seite
<i>NoneType</i>	nur der Wert None	7.2	92
Numerische Datentypen			
<i>int</i>	ganze Zahlen	7.3.4	96
<i>float</i>	Gleitkommazahlen	7.3.5	102
<i>bool</i>	boolesche Werte	7.3.6	104
<i>complex</i>	komplexe Zahlen	7.3.7	110
Sequentielle Datentypen			
<i>list</i>	Listen	7.5.2	128
<i>tuple</i>	unveränderliche Listen	7.5.3	139
<i>str</i>	Zeichenketten	7.5.4	142

Tabelle 7.1 Basisdatentypen

Basisdatentyp	Beschreibung	Abschnitt	Seite
Assoziative Datentypen			
<i>dict</i>	Schlüssel/Wert-Zuordnungen	7.6.1	174
Mengen			
<i>set</i>	Mengen	7.7.2	193
<i>frozenset</i>	unveränderliche Mengen	7.7.3	195

Tabelle 7.1 Basisdatentypen (Forts.)

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

7.1 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 Datentypen 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 können unterschiedliche Ergebnisse herauskommen:

$(2 * 3) + 4 = 10$
 $2 * (3 + 4) = 14$

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.

Die folgende Tabelle zeigt die Operatorrangfolge für die in Python definierten Operatoren. Zudem ist die übliche Bedeutung des Operators angegeben. Operatoren, die in der Tabelle weiter oben stehen, binden stärker als Operatoren, die weiter unten stehen. Operatoren, die in derselben Zeile stehen, haben eine gleich starke Bindigkeit.

Operator	Übliche Bedeutung
$x ** y$	y -te Potenz von x
$\sim x$	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

Tabelle 7.2 Die Operatorrangfolge

Operator	Übliche Bedeutung
$x + y$	Summe von x und y
$x - y$	Differenz von x und y
$x \ll n$	bitweise Verschiebung um n Stellen nach links
$x \gg n$	bitweise Verschiebung um n Stellen nach rechts
$x \& y$	bitweises UND zwischen x und y
$x \wedge y$	bitweises ausschließendes ODER zwischen x und y
x / y	bitweises nicht ausschließendes ODER zwischen x und y
$x < y$	Ist x kleiner als y ?
$x \leq y$	Ist x kleiner oder gleich y ?
$x > y$	Ist x größer als y ?
$x \geq y$	Ist x größer oder gleich y ?
$x \neq y$	Ist x ungleich y ?
$x == y$	Ist x gleich y ?
$x \text{ is } y$	Sind x und y identisch?
$x \text{ is not } y$	Sind x und y nicht identisch?
$x \text{ in } y$	Befindet sich x in y ?
$x \text{ not in } y$	Befindet sich x nicht in y ?
$\text{not } x$	logische Negierung
$x \text{ and } y$	logisches UND
$x \text{ or } y$	logisches ODER

Tabelle 7.2 Die Operatorrangfolge (Forts.)

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

$$a + b + c$$

$$a - b - c$$

In beiden Fällen ist die Auswertungsreihenfolge weder durch Klammern noch durch die Operatorrangfolge eindeutig geklärt. Sie sehen, dass dies für die Auswertung des

ersten Ausdrucks zwar kein Problem darstellt, doch spätestens beim zweiten Ausdruck ist eine Regelung vonnöten, da je nach Auswertungsreihenfolge zwei verschiedene Ergebnisse möglich sind. In einem solchen Fall gilt in Python die Regelung, dass Ausdrücke oder Teilausdrücke, die nur aus Operatoren gleicher Bindigkeit bestehen, *von links nach rechts* ausgewertet werden.

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

```
>>> 1 < 2.5
True
```

Denken Sie einmal über die Auswertungsreihenfolge dieses Ausdrucks nach:

```
a < b < c
```

Theoretisch ist 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 zum Beispiel `True`. Der nächste 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`.

7.2 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 der Funktion *print* bedienen.

Es wurde bereits gesagt, dass *None* die einzige Instanz des »Nichts« ist. Diese Besonderheit können wir uns zunutze machen, um 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.¹

¹ Näheres dazu finden Sie in Kapitel 6 (S. 75)

7.3 Numerische Datentypen

In diesem Abschnitt besprechen wir die erste große Gruppe von Datentypen, die numerischen Datentypen. Die folgende Tabelle listet alle zu dieser Gruppe gehörigen Datentypen auf und nennt ihren Zweck.

Datentyp	Beschreibung	Veränderlichkeit ²	Seite
<i>int</i>	ganze Zahlen	unveränderlich	96
<i>float</i>	Gleitkommazahlen	unveränderlich	102
<i>bool</i>	boolesche Werte	unveränderlich	104
<i>complex</i>	komplexe Zahlen	unveränderlich	110

Tabelle 7.3 Numerische Datentypen

7.3.1 Arithmetische Operatoren

Für alle numerischen Datentypen sind folgende arithmetische 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^3
$+x$	positives Vorzeichen
$-x$	negatives Vorzeichen
$x ** y$	x hoch y
$x // y$	abgerundeter Quotient von x und y^3

Tabelle 7.4 Gemeinsame Operatoren numerischer Datentypen

² Alle numerischen Datentypen sind unveränderlich. Dies bedeutet nicht, 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.

Näheres zu dem Unterschied zwischen veränderlichen und unveränderlichen Datentypen finden Sie in Abschnitt 6.3 (S. 82)

³ Die Operatoren `%` und `//` haben für komplexe Zahlen keine mathematische Bedeutung und sind deshalb für den Datentyp `complex` nicht definiert.



Hinweis

Zwei Anmerkungen für Leser, die bereits mit einer C-ähnlichen Programmiersprache vertraut sind:

- ▶ Es gibt in Python keine Entsprechungen für die Inkrementierungs- und Dekrementierungsoperatoren `x++` und `x--` aus C.
- ▶ Die Operatoren `%` und `//` können folgendermaßen beschrieben werden:

```
x // y = runden(x / y)
x % y = x - y * runden(x / y)
```

Python rundet dabei stets ab, während C zur Null hin rundet. Dieser Unterschied tritt nur auf, wenn einer der Operanden negativ und der andere positiv ist.

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
<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 /= 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 //= y</code>	<code>x = x // y</code>

Tabelle 7.5 Gemeinsame Operatoren numerischer Datentypen

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

7.3.2 Vergleichende Operatoren

Die folgende Tabelle listet die *vergleichenden Operatoren* auf, die für numerische Datentypen definiert sind.

Operator	Ergebnis
<code>==</code>	wahr, wenn x und y gleich sind
<code>!=</code>	wahr, wenn x und y verschieden sind
<code><</code>	wahr, wenn x kleiner ist als y ⁴
<code><=</code>	wahr, wenn x kleiner oder gleich y ist ⁴
<code>></code>	wahr, wenn x größer ist als y ⁴
<code>>=</code>	wahr, wenn x größer oder gleich y ist ⁴

Tabelle 7.6 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 verketteten. 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 7.3.6.

⁴ Da komplexe Zahlen prinzipiell nicht sinnvoll anzuordnen sind, lässt der Datentyp `complex` nur die Verwendung der ersten beiden Operatoren zu.

7.3.3 Konvertierung zwischen numerischen Datentypen

Numerische Datentypen können über die eingebauten Funktionen `int`, `float`, `bool` und `complex` ineinander umgeformt werden. Dabei können je nach Umformung Informationen verloren gehen. Als Beispiel betrachten wir einige Konvertierungen im interaktiven Modus:

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

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

7.3.4 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.⁵

⁵ 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, sodass die Python-Entwickler keinen Sinn mehr darin sahen, die beiden Datentypen zu trennen.

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

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

7

Zahlensysteme

Ganze Zahlen können in Python in mehreren Zahlensystemen⁶ 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. 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.⁷

- 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*, auch *Binärsystem* (Basis 2), von entscheidender Bedeutung. Seit Version 3.0 unterstützt

⁶ Sollten Sie nicht wissen, was ein Zahlensystem ist, können Sie diesen Abschnitt problemlos überspringen.

⁷ Bis zu Version 3.0 wurde in Python, wie beispielsweise in C auch, die »O« als Präfix für Oktalzahlen verwendet.

Python ein eigenes Literal für Dualzahlen. Diese werden analog zu den vorangegangenen Literalen durch das Präfix `0b` eingeleitet:

```
v_bin = 0b1101
```

Im Dualsystem dürfen nur die Ziffern 0 und 1 verwendet werden.

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).

Für alle Zahlensystem-Literale ist die Verwendung eines negativen Vorzeichens möglich:

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

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.

7

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
$x \& y$	bitweises UND von x und y (AND)
x / y	bitweises nicht ausschließendes ODER von x und y (OR)
$x \wedge y$	bitweises ausschließendes ODER von x und y (XOR)
$\sim x$	bitweises Komplement von x
$x \ll n$	Bitverschiebung um n Stellen nach links
$x \gg n$	Bitverschiebung um n Stellen nach rechts

Tabelle 7.7 Bit-Operatoren des Datentyps `int`

Auch hier sind erweiterte Zuweisungen mithilfe der folgenden Operatoren möglich:

Operator	Entsprechung
$x \&= y$	$x = x \& y$
$x /= y$	$x = x / y$
$x \wedge= y$	$x = x \wedge y$
$x \ll= n$	$x = x \ll n$
$x \gg= n$	$x = x \gg n$

Tabelle 7.8 Bit-Operatoren des Datentyps `int`

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 7.1:

	Dual							Dezimal
	1	1	0	1	0	1	1	107
&	0	0	1	1	0	0	1	25
	0	0	0	1	0	0	1	9

Abbildung 7.1 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:

```
>>> 107 & 25
9
```

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 7.2 veranschaulicht dies.

	Dual							Dezimal
	1	1	0	1	0	1	1	107
	0	0	1	1	0	0	1	25
	1	1	1	1	0	1	1	123

Abbildung 7.2 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 7.3.

Dual							Dezimal
1 1 0 1 0 1 1							107
$\wedge$ 0 0 1 1 0 0 1							25
1 1 1 0 0 1 0							114

Abbildung 7.3 Bitweises exklusives ODER

Das *bitweise Komplement* bildet das sogenannte Einerkomplement einer Dualzahl, das der Negation aller vorkommenden Bits entspricht. In Python ist dies auf Bitebene nicht möglich, da eine ganze Zahl in ihrer Länge unbegrenzt ist und das Komplement immer in einem abgeschlossenen Zahlenraum gebildet werden muss. Deswegen wird die eigentliche Bit-Operation zur arithmetischen Operation und ist folgendermaßen definiert⁸:

$$\sim x = -x - 1$$

Bei der *Bitverschiebung* wird die Bitfolge in der binären Darstellung des ersten Operanden um die durch den zweiten Operanden gegebene Anzahl Stellen nach links bzw. rechts verschoben. Auf der rechten Seite entstehende Lücken werden mit Nullen gefüllt, und das Vorzeichen des ersten Operanden bleibt erhalten. Abbildung 7.4 und Abbildung 7.5 veranschaulichen eine Verschiebung um zwei Stellen nach links bzw. nach rechts.

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

n = 2

Abbildung 7.4 Bitverschiebung um zwei Stellen nach links

⁸ 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: $-x = \text{Zweierkomplement von } x = \sim x + 1$

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

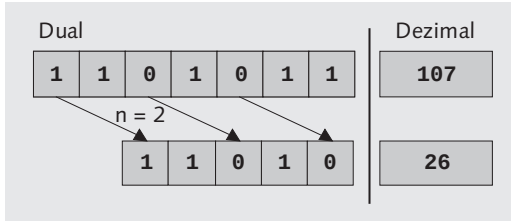


Abbildung 7.5 Bitverschiebung um zwei Stellen nach rechts

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

Die Bitverschiebung ist in Python ähnlich wie der Komplementoperator arithmetisch implementiert. Ein Shift um x Stellen nach rechts entspricht einer ganzzahligen Division durch 2^x . Ein Shift um x Stellen nach links entspricht einer Multiplikation mit 2^x .

Methoden

Der Datentyp `int` verfügt seit Python 3.2 über eine Methode, die sich auf die Binärdarstellung der ganzen Zahl bezieht. Die Methode `bit_length` berechnet die Anzahl Stellen, die für die Binärdarstellung der Zahl benötigt werden.

```
>>> (36).bit_length()
6
>>> (4345).bit_length()
13
```

Die Binärdarstellung der 36 ist 100100 und die der 4345 ist 1000011111001. Damit benötigen die beiden Zahlen 6 bzw. 13 Stellen für ihre Binärdarstellung.

Beachten Sie, dass die Klammern um die Zahlenlitterale bei ganzen Zahlen benötigt werden, da es sonst zu Doppeldeutigkeiten mit der Syntax für Gleitkommazahlen kommen könnte.

7.3.5 Gleitkommazahlen – float

Zu Beginn dieses Teils des Buchs 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 Exponential-schreibweise zu verwenden:

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

Durch ein kleines oder großes *e* wird die *Mantisse* (3.141) vom *Exponenten* (-12) getrennt. Übertragen in die mathematische Schreibweise, entspricht $3.141e-12$ $3,141 \cdot 10^{-12}$. Beachten Sie, dass sowohl die Mantisse als auch der Exponent im Dezimalsystem anzugeben sind. Andere Zahlensysteme sind nicht vorgesehen, was die gefahrlose Verwendung von führenden Nullen ermöglicht:

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

Es gibt noch weitere Varianten, eine gültige Gleitkommazahl zu definieren. Es handelt sich dabei um Spezialfälle der obigen Notation, weswegen sie etwas exotisch wirken. Sie 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:

```
>>> 1.1 + 2.2
3.3000000000000003
```

Aufgrund der Begrenztheit von `float` können reelle Zahlen nicht unendlich präzise gespeichert werden. Stattdessen werden sie mit einer bestimmten Genauigkeit angenähert. Es ist unter Verwendung der Basisdatentypen nicht möglich, mit beliebigen genauen Dezimalzahlen zu rechnen. Dazu muss die Standardbibliothek bemüht werden, was wir zu gegebener Zeit behandeln werden.⁹

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

9 Dabei handelt es sich um das Modul `decimal`, das in Abschnitt 13.3 behandelt wird.

Limit überschreiten, in Python nicht mehr dargestellt werden. Wenn das Limit überschritten wird, wird die Zahl als `inf` gespeichert¹⁰ bzw. als `-inf`, wenn das untere Limit unterschritten wurde. Es kommt also zu keinem Fehler, und es ist immer noch möglich, eine übergroße Zahl mit anderen zu vergleichen:

```
>>> 3.0e999
inf
>>> -3.0e999
-inf
>>> 3.0e999 < 12.0
False
>>> 3.0e999 > 12.0
True
>>> 3.0e999 == 3.0e9999999999999
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 vergleichbar mit `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.

7.3.6 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.

¹⁰ `inf` steht für *infinity* (dt. »unendlich«)

False. Deshalb ist es auf den ersten Blick absurd, `bool` den numerischen Datentypen unterzuordnen. Wie in vielen Programmiersprachen üblich, wird in Python `True` analog zur 1 und `False` analog zur 0 gesehen, sodass sich mit booleschen Werten genauso rechnen lässt wie beispielsweise 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
```

7

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*¹¹, und darunter sehen Sie weitere Erklärungen mit konkreten Beispielen.

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

Tabelle 7.9 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:

¹¹ Beachten Sie, dass es einen Unterschied gibt zwischen den logischen Operatoren, die im Zusammenhang mit booleschen Werten stehen, und den binären Operatoren, die sich auf die Binärdarstellung einer Zahl beziehen.

x	y	Ausdruck: a and b
<i>True</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>False</i>
<i>True</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>False</i>	<i>False</i>

Tabelle 7.10 Mögliche Fälle des logischen UND

In einem konkreten Beispiel würde die Anwendung des logischen UND 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 nicht¹². Folgende Tabelle listet alle möglichen Fälle auf:

x	y	Ausdruck: a or b
<i>True</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>False</i>	<i>False</i>

Tabelle 7.11 Mögliche Fälle des logischen ODER

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:

¹² Das exklusive ODER zwischen x und y lässt sich über (x or y) and not (x and y) nachbilden.

```
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*:

<i>x</i>	<i>y</i>	<i>z</i>	Ausdruck: <i>x</i> and <i>y</i> or ((<i>y</i> and <i>z</i>) and not <i>x</i>)
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>	<i>False</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>
<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>

Tabelle 7.12 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 Instanzen vergleichbarer Typen handeln, wie zum Beispiel *int*, *float* oder *bool*.

Wahrheitswerte anderer Datentypen

Mithilfe der Built-in Function *bool* lassen sich Instanzen eines jeden Basisdatentyps in einen booleschen Wert überführen.


```
>>> bool([1,2,3])
True
>>> bool("")
False
>>> bool(-7)
True
```

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 ist 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
<i>NoneType</i>	None	der Wert None
Numerische Datentypen		
<i>int</i>	0	der Wert Null
<i>float</i>	0.0	der Wert Null
<i>bool</i>	False	der boolesche Wert False
<i>complex</i>	0 + 0j	der Wert Null
Sequentielle Datentypen		
<i>str</i>	""	ein leerer String
<i>list</i>	[]	eine leere Liste

Tabelle 7.13 Wahrheitswerte der Basisdatentypen

Basisdatentyp	False-Wert	Beschreibung
<i>tuple</i>	()	ein leeres Tupel
Assoziative Datentypen		
<i>dict</i>	{}	ein leeres Dictionary
Mengen		
<i>set</i>	set()	eine leere Menge
<i>frozenset</i>	frozenset()	eine leere Menge

Tabelle 7.13 Wahrheitswerte der Basisdatentypen (Forts.)

Alle anderen Werte ergeben `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. Das folgende Beispiel demonstriert dieses Verhalten, das *Lazy Evaluation* (dt. »faule Auswertung«) genannt wird.

```
>>> a = True
>>> if a or print("Lazy "):
...     print("Evaluation")
...
Evaluation
```

Obwohl in der Bedingung der `if`-Anweisung die `print`-Funktion aufgerufen wird, wird diese Bildschirmausgabe nie durchgeführt, da der Wert der Bedingung bereits nach der Auswertung von `a` feststeht. Dieses Detail scheint unwichtig, kann aber insbesondere im Zusammenhang mit seiteneffektbehafteten¹³ Funktionen zu schwer auffindbaren Fehlern führen.

¹³ Siehe dazu Abschnitt 9.2.6 (S. 222)

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 nicht der Fall ist. Stattdessen gibt Python hier den ersten Operanden mit dem Wahrheitswert `True` zurück. Das ist 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'
```

7.3.7 Komplexe Zahlen – complex

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

Komplexe Zahlen bestehen aus einem reellen Realteil und einem Imaginärteil. Der Imaginärteil ist eine reelle Zahl, die mit der imaginären Einheit j multipliziert wird.¹⁴

¹⁴ 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 ist als Lösung der Gleichung

$$j^2 = -1$$

definiert. 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 Anordnung 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 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
<i>x.real</i>	Realteil von <i>x</i> als Gleitkommazahl
<i>x.imag</i>	Imaginärteil von <i>x</i> als Gleitkommazahl

Tabelle 7.14 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 Kapitel 11 darauf zurückkommen und näher darauf eingehen, was ein Attribut genau ist. Außer über seine zwei Attribute verfügt der Datentyp `complex` über eine Methode, die in der Tabelle exemplarisch für eine Referenz auf eine komplexe Zahl namens *x* erklärt wird.

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

Tabelle 7.15 Methoden des Datentyps `complex`

Das folgende Beispiel demonstriert die Verwendung der Methode `conjugate`:

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

Das Ergebnis von `conjugate` ist wieder eine komplexe Zahl und verfügt daher ebenfalls über die Methode `conjugate`:

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

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

Das Konjugieren einer komplexen Zahl ist eine selbstinverse Operation. Das bedeutet, dass das Ergebnis einer zweifachen Konjugation wieder die Ausgangszahl ist.

7.4 Parameter von Funktionen und Methoden

Die bisher behandelten numerischen Datentypen waren sehr einfach aufgebaut, und ihr Funktionsumfang war mit den für sie definierten Operatoren in der Regel vollständig beschrieben. In den folgenden Kapiteln werden wir uns komplexeren Datentypen widmen, die eine Reihe von Methoden bereitstellen, um mit ihnen zu arbeiten. Aus diesem Grund wird hier das in Abschnitt 3.7 (S. 47) erworbene Wissen über Parameter von Funktionen und Methoden aufgefrischt und ausgebaut.

7

7.4.1 Wiederholung

Bei einer *Funktion* handelt es sich um ein benanntes Unterprogramm, das eine häufig benötigte Funktionalität kapselt und über einen *Funktionsaufruf* ausgeführt werden kann. Ein Beispiel für eine Funktion ist die *eingebaute Funktion* oder *Built-in Function* `max` zur Bestimmung des größten Elements einer Liste:

```
>>> max([3,6,2,1,9])
9
```

Die Liste, deren größtes Element bestimmt werden soll, ist ein *Parameter* der Funktion `max` und wird beim Funktionsaufruf in die Klammern geschrieben.

Ein Funktionsaufruf kann ein Ergebnis zurückgeben. Dieser *Rückgabewert* kann als Instanz ganz normal weiterverwendet werden:

```
>>> wert = max([3,6,2,1,9])
>>> wert/2
4.5
```

Eine *Methode* ist eine Funktion, die im Kontext einer bestimmten Instanz ausgeführt wird. Listen verfügen beispielsweise über eine Methode `sort`, die die Liste, für die sie aufgerufen wird, sortiert.

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

Welche Methoden für eine Instanz verfügbar sind, hängt von ihrem Datentyp ab.

7.4.2 Parameter

Viele Methoden¹⁵ benötigen beim Aufruf neben der Instanz, auf die sich der Aufruf bezieht, weitere Informationen, um zu funktionieren. Hierfür gibt es sogenannte *Parameter*, die, durch Kommata getrennt, in die Klammern am Ende des Methodenaufrufs geschrieben werden. Als Parameter können formal sowohl Referenzen als auch Literale verwendet werden:

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

Wieviele und welche Parameter einer Methode übergeben werden dürfen, hängt von ihrer Definition ab und ist daher von Methode zu Methode verschieden.

Eine Methode zusammen mit den Parametern, die sie erwartet, wird *Schnittstelle* genannt. Wenn wir in diesem Buch eine Methode besprechen, werden wir stets die dazugehörige Schnittstelle in der folgenden Form angeben:

methode(parameter1, parameter2, parameter3)

In diesem Fall erwartet die Methode drei *positionsbezogene Parameter* namens *parameter1*, *parameter2* und *parameter3*. Positionsbezogen bedeutet, dass die beim Methodenaufruf übergebenen Instanzen entsprechend ihrer Position in der Parameterliste den Parametern zugeordnet werden. Im Falle des Methodenaufrufs

```
referenz.methode(1, 45, -7)
```

hätte *parameter1* den Wert 1, *parameter2* den Wert 45 und *parameter3* den Wert -7.

Man kann einer Methode Parameter auch als *Schlüsselwortparameter* (engl. *keyword arguments*) übergeben. Schlüsselwortparameter werden direkt mit dem formalen Parameternamen verknüpft, und ihre Reihenfolge in der Parameterliste spielt keine Rolle mehr. Um einen Wert als Schlüsselwortparameter zu übergeben, weist man dem Parameternamen innerhalb des Methodenaufrufs den zu übergebenden Wert mithilfe des Gleichheitszeichens zu. Die beiden folgenden Methodenaufrufe sind demnach gleichwertig:

```
referenz.methode(1, 2, 3)
referenz.methode(parameter2=2, parameter1=1, parameter3=3)
```

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:

¹⁵ Die Erklärungen und Beispiele in diesem Abschnitt beziehen sich auf Methoden, gelten aber analog für Funktionen.

```
referenz.methode(1, parameter3=3, parameter2=2)
```

Nur `parameter1` wurde als positionsbezogener Parameter übergeben, während `parameter2` und `parameter3` als Schlüsselwortparameter übergeben wurden.

Welche der beiden Übergabemethoden man in der Praxis bevorzugt, ist größtenteils Geschmackssache. In der Regel werden Parameter jedoch positionsbezogen übergeben, weil der Schreibaufwand geringer ist.

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

```
methode(parameter1, parameter2[, parameter3])
```

In diesem Beispiel sind `parameter1` und `parameter2` erforderliche Parameter und `parameter3` ein optionaler Parameter. Die Methode kann also in zwei verschiedenen Varianten aufgerufen werden:

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

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

7.5 Sequentielle Datentypen

Unter *sequentuellen 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 fünf sequentiellen Typen zur Verfügung: `str`, `bytes`, `bytearray`, `list` und `tuple`.

Datentyp	Speichert	Veränderlichkeit	Seite
<i>str</i>	Text als Sequenz von Buchstaben	<i>unveränderlich</i>	142
<i>bytes</i>	Binärdaten als Sequenz von Bytes	<i>unveränderlich</i>	142
<i>bytearray</i>	Binärdaten als Sequenz von Bytes	<i>veränderlich</i>	142
<i>list</i>	Listen beliebiger Instanzen	<i>veränderlich</i>	128

Tabelle 7.16 Liste der sequentiellen Datentypen

Datentyp	Speichert	Veränderlichkeit	Seite
<i>tuple</i>	Listen beliebiger Instanzen	<i>unveränderlich</i>	139

Tabelle 7.16 Liste der sequentiellen Datentypen (Forts.)

Der Datentyp `str` ist für die Speicherung und Verarbeitung von Texten vorgesehen. Daher sind die Elemente, die eine `str`-Instanz speichern kann, die Bausteine, aus denen Texte in menschlicher Sprache bestehen, also Buchstaben, Leer- und Interpunktionszeichen sowie Zeilenvorschübe. Bemerkenswert ist dabei, dass dies auch mit lokalen Sonderzeichen wie beispielsweise den deutschen Umlauten »ä«, »ü« und »ö« funktioniert.

Im Gegensatz dazu kann eine Instanz des Datentyps `bytes` einen binären Datenstrom, also eine Folge von Bytes speichern. Der Datentyp `bytearray` ist ebenfalls in der Lage, Binärdaten zu speichern. Allerdings sind `bytearray`-Instanzen anders als `bytes`-Instanzen mutable.

Die strukturelle Trennung von Textdaten und Binärdaten ist eine Eigenschaft, die Python von vielen anderen Programmiersprachen unterscheidet.



Hinweis

Dies ist eine der großen Neuerungen ab 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 häufig der alte Datentyp `str` zum Speichern von Text-Strings genutzt wurde, gab es einige Stolpersteine, wenn man Sonderzeichen mit Python-Programmen verarbeiten wollte. Durch die neue Typaufteilung ist der Umgang mit Zeichenketten klarer strukturiert. Allerdings muss man darauf achten, welchen Datentyp Funktionen der Standardbibliothek als Parameter erwarten bzw. als Rückgabewert zurückgeben und gegebenenfalls Umwandlungen durchführen.

Mehr dazu finden Sie im Abschnitt »Zeichensätze und Sonderzeichen« ab Seite 165.

Sowohl Instanzen des Datentyps `str` als auch des Datentyps `bytes` sind immutable, ihr Wert kann sich nach der Instanziierung also nicht *mehr* verändern. Trotzdem können Sie komfortabel mit Strings arbeiten. Bei Änderungen wird nur nicht der Ursprungsstring verändert, sondern stets ein neuer String erzeugt.

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

Für jede Instanz eines sequentiellen Datentyps gibt es einen Grundstock von Operatoren und Methoden, der immer verfügbar ist. Der Einfachheit halber werden wir diesen allgemein am Beispiel von `list`- und `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 ganze Zahlen; x ist eine Referenz auf eine beliebige Instanz):

Notation	Beschreibung	Seite
$x \text{ in } s$	Prüft, ob x in s enthalten ist. Das Ergebnis ist eine <code>bool</code> -Instanz.	118
$x \text{ not in } s$	Prüft, ob x nicht in s enthalten ist. Das Ergebnis ist eine <code>bool</code> -Instanz. Gleichwertig mit <code>not x in s</code> .	118
$s + t$	Das Ergebnis ist eine neue Sequenz, die die Verkettung von s und t enthält.	119
$s += t$	Erzeugt die Verkettung von s und t und weist sie s zu.	119
$s * n$ oder $n * s$	Liefert eine neue Sequenz, die die Verkettung von n Kopien von s enthält.	121
$s *= n$	Erzeugt das Produkt $s * n$ und weist es s zu.	121
$s[i]$	Liefert das i -te Element von s .	121
$s[i:j]$	Liefert den Ausschnitt aus s von i bis j .	121
$s[i:j:k]$	Liefert den Ausschnitt aus s von i bis j , wobei nur jedes k -te Element beachtet wird.	121
$\text{len}(s)$	Gibt die Anzahl der Elemente von s zurück.	126
$\text{max}(s)$	Liefert das größte Element von s , sofern eine Ordnungsrelation für die Elemente definiert ist.	126
$\text{min}(s)$	Liefert das kleinste Element von s , sofern eine Ordnungsrelation für die Elemente definiert ist.	126
$s.\text{index}(x, i[, j])$	Gibt den Index k des ersten Vorkommens von x in der Liste s im Bereich $i \leq k < j$ zurück.	127
$s.\text{count}(x)$	Zählt, wie oft x in der Sequenz s vorkommt.	128

Tabelle 7.17 Operationen auf Instanzen sequentieller Datentypen

Im Folgenden werden diese Operationen im Detail erklärt.

7.5.1 Operationen auf Instanzen sequentieller Datentypen

Ist ein Element vorhanden? – die Operatoren `in` und `not in`

Mithilfe von `in` lässt sich ermitteln, ob ein bestimmtes Element in einer Sequenz enthalten ist. Für eine Instanz des Datentyps `list`, die sowohl Strings als auch Zahlen enthält, sieht das folgendermaßen aus:

```
>>> l = ["eins", 2, 3.0, "vier", 5, "sechs", "sieben"]
>>> 3.0 in l
True
>>> "vier" in l
True
>>> 10 in l
False
```

Da die Elemente eines Strings Zeichen 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. Zeichen können Sie in Python durch Strings der Länge 1 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...
```

Es ist außerdem möglich, mit dem `in`-Operator zu prüfen, ob ein bestimmter Teilstring in einer Zeichenkette enthalten ist:

```
>>> s = "Dies ist unser Teststring"
>>> "ist" in s
True
>>> "Hallo" in s
False
```

Dies funktioniert nur mit Zeichenketten, also Instanzen der Typen `str`, `bytes` und `bytearray`. Mit dem `in`-Operator kann nicht geprüft werden, ob eine Teilliste in einer `list`-Instanz enthalten ist. Gleiches gilt für Instanzen des Typs `tuple`.

```
>>> [2,3] in [1,2,3,4]
False
```

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

7

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 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. Der Ausdruck `x not in s` liest sich im Gegensatz zu `not x in s` genau wie ein englischer Satz, während die andere Form schwieriger zu lesen ist.¹⁶

Verkettung von Sequenzen – die Operatoren `+` und `+=`

Um Sequenzen zu verketteten, dient der `+`-Operator. Im Beispiel werden Vor- und Nachname von Herrn Maier zusammen mit einem Leerzeichen zu einem neuen String verkettet:

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

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

¹⁶ 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, wird 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.

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

Dabei ist `s += t` für immutable Datentypen genauso zu lesen wie `s = s + t`, denn es wird tatsächlich eine neue Instanz mit dem Wert von `s + t` erzeugt, die dann von `s` referenziert wird. Es existieren also nach der Operation `s += t` drei verschiedene Instanzen im Speicher, nämlich mit den Werten `s`, `t` und `s + t`.

Für mutable Datentypen wie beispielsweise `list` gilt dies nicht. Hier wird keine weitere Instanz mit dem Wert `s + t` erzeugt, sondern die Instanz `s` verändert.

Zwei Beispiele verdeutlichen diesen Unterschied zwischen mutable und immutable Datentypen:

```
>>> s = "Musik"
>>> t = "lautsprecher"
>>> temp = s
>>> s += t
>>> s
'Musiklautsprecher'
>>> t
'lautsprecher'
>>> temp
'Musik'
```

Da mit der Anweisung `s += t` eine neue `str`-Instanz erzeugt wurde, hat sich die von `temp` referenzierte `str`-Instanz nicht verändert. Anders sieht dies beim mutable Datentyp `list` aus:

```
>>> s = [1,2]
>>> t = [3,4]
>>> temp = s
>>> s += t
>>> s
[1, 2, 3, 4]
>>> t
[3, 4]
>>> temp
[1, 2, 3, 4]
```

Hier verweisen `s` und `temp` auch nach der Anweisung `s += t` auf dieselbe `list`-Instanz, da die bestehende Liste verändert und kein neues Objekt erzeugt wurde.

Man sagt, der Operator `+=` arbeitet *in-place* (dt. »an der Stelle«). Dies gilt im Übrigen auch für den Operator `*=`, der im folgenden Abschnitt behandelt wird.

Wiederholung von Sequenzen – Die Operatoren `*` und `*=`

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"
'abcbabc'
>>> "xyz" * 5
'xyzxyzxyzxyzxyz'
```

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

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

Auf die gleiche Art und Weise lassen sich auch Listen mit ganzen Zahlen multiplizieren:

```
>>> [1,2] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Der Operator `*=` arbeitet genauso wie `+=` *in-place*. Was das genau bedeutet, erfahren Sie auf Seite 120, auf der dieser Begriff am Beispiel des `+=`-Operators erläutert wird.

Zugriff auf bestimmte Elemente einer Sequenz – der `[]`-Operator

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, als Speicher für Text 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'
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[3]
4
```

Um 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. Abbildung 7.6 veranschaulicht die zwei Indizierungsarten.

Index von vorne:	0	1	2	3	4	5
Elemente:	P	y	t	h	o	n
Index von hinten:	-6	-5	-4	-3	-2	-1

Abbildung 7.6 Indizierung von vorne und von hinten

```
>>> name = "Python"
>>> name[-2]
'o'
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[-1]
6
```

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 = "schrottWICHTIGschrott"
>>> s[7]
'W'
>>> s[14]
's'
>>> s[7:14]
'WICHTIG'
```

7

Abbildung 7.7 veranschaulicht den Zugriff auf die Teilsequenz.

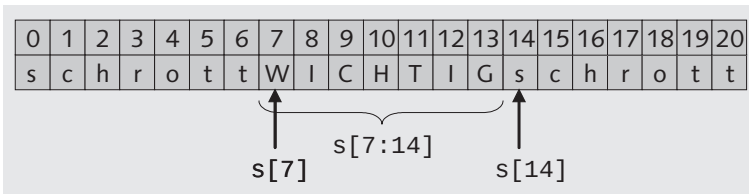


Abbildung 7.7 Extrahieren einer Teilsequenz

Analog extrahiert man Teile einer Liste:

```
>>> l = ["Ich", "bin", "eine", "Liste", "von", "Strings"]
>>> l[2:5]
['eine', 'Liste', 'von']
```

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'
>>> l = ["Ich", "bin", "eine", "Liste", "von", "Strings"]
>>> l[1:-1]
['bin', 'eine', 'Liste', 'von']
```

Die Indizes können auch 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 = ["Doktorarbeit"]
>>> s2 = s1
>>> s1 == s2
True
>>> s1 is s2
True
```

Wie erwartet, verweisen `s1` und `s2` auf dieselbe Instanz, sind also identisch. Anders sieht es im nächsten Beispiel aus, wo eine echte Kopie der Liste `["Doktorarbeit"]` im Speicher erzeugt wird.¹⁷ Dies zeigt sich beim Identitätsvergleich mit `is`:

```
>>> s1 = ["Doktorarbeit"]
>>> s2 = s1[:]
>>> s1 == s2
True
>>> s1 is s2
False
```



Hinweis

Wenn Sie im obigen Beispiel statt eines *mutable* Datentyps wie `list` einen *immutable* Datentyp wie `str` verwenden, kann es sein, dass in beiden Fällen der Ausdruck `s1 is s2` zu `True` ausgewertet wird. Das liegt daran, dass es für immutable Datentypen keinen Unterschied macht, ob die Instanz wirklich kopiert oder mit der ursprünglichen Instanz gearbeitet wird: Der Wert kann ohnehin nicht verändert werden.

Daher wird eine Instanz eines immutable Datentyps selbst bei Verwendung von `[ : ]` nicht zwingend kopiert, sondern aus Gründen der Effizienz stattdessen ein weiteres Mal referenziert.

¹⁷ Im wirklichen Leben sollte man natürlich niemals eine Doktorarbeit einfach kopieren ...

Verwendet man statt einer Liste einen String, wird der Unterschied zum obigen Beispiel deutlich:

```
>>> s1 = "Kopiere mich"
>>> s2 = s1[:]
>>> s2 is s1
True
```

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, also dem mit Index Null, 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 beispielsweise 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 Schrittweise größer als der Endindex, wird eine leere Sequenz zurückgegeben:

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

Länge einer Sequenz – die Built-in Function `len`

Die Anzahl der Elemente einer Sequenz definiert die *Länge der Sequenz*. Die Länge einer Sequenz ist eine positive ganze Zahl und lässt sich mit der Built-in Function `len` ermitteln:

```
>>> string = "Wie lang bin ich wohl?"
>>> len(string)
22
>>> len(["Hallo", 5, 2, 3, "Welt"])
5
```

Das kleinste und das größte Element einer Sequenz – `min` und `max`

Um das kleinste beziehungsweise größte Element einer Sequenz zu ermitteln, dienen die Built-in Functions `min` und `max`.

```
>>> l = [5, 1, 10, -9.5, 12, -5]
>>> max(l)
12
>>> min(l)
-9.5
```

Allerdings sind diese beiden Funktionen nur dann sinnvoll, wenn eine *Ordnungsrelation* für die Elemente der Sequenz existiert. In Abschnitt 7.3.7 über komplexe Zahlen auf Seite 110 wird zum Beispiel der Datentyp `complex` ohne Ordnungsrelation beschrieben. Ebenso ist es nicht möglich, komplett verschiedene Datentypen wie beispielsweise Strings und Zahlen zu vergleichen:

```
>>> l = [1,2, "welt"]
>>> min(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Trotzdem lassen sich `min` und `max` in sinnvoller Weise auf Strings anwenden, da für Buchstaben ihre Position im Alphabet als Ordnungsrelation benutzt wird.

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

Die Position eines Elements in der Sequenz – `s.index(x[, i[, j]])`

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

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

Um die Suche auf einen Teilbereich der Sequenz 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
>>> "Hallo Welt".index("l", 5, 100)
8
```

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
```

Anzahl der Vorkommen eines Elements der Sequenz – `s.count(x)`

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

```
>>> s = [1, 2, 2, 3, 2]
>>> s.count(2)
3
>>> "Hallo Welt".count("l")
3
```

Der nächste Abschnitt beschäftigt sich mit Operationen, die nur für mutable Sequenzen verfügbar sind.

7.5.2 Listen – »list«

In diesem Abschnitt werden Sie den ersten veränderbaren (mutable) Datentyp, die *Liste*, im Detail kennen lernen. Anders als die sequentiellen Datentypen `str`, `bytes` und `bytearray`, die nur gleichartige Elemente (Zeichen) speichern können, 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. Auf Seite 117 finden Sie eine Tabelle mit den Operationen, die für alle sequentiellen Datentypen verfügbar sind.

Im Gegensatz zu den bisher besprochenen sequentiellen Datentypen kann sich der Inhalt einer Liste auch nach ihrer Erzeugung ändern, weshalb eine Reihe weiterer Operatoren und Methoden für sie verfügbar sind:

Operator	Wirkung	Seite
$s[i] = x$	Das Element von s mit dem Index i wird durch x ersetzt.	129
$s[i:j] = t$	Der Teil $s[i:j]$ wird durch t ersetzt. Dabei muss t iterierbar sein.	129

Tabelle 7.18 Operatoren für den Datentyp `list`

Operator	Wirkung	Seite
$s[i:j:k] = t$	Die Elemente von $s[i:j:k]$ werden durch die von t ersetzt.	129
$del\ s[i]$	Das i -te Element von s wird entfernt.	130
$del\ s[i:j]$	Der Teil $s[i:j]$ wird aus s entfernt. Das ist äquivalent zu $s[i:j] = []$.	130
$del\ s[i:j:k]$	Die Elemente der Teilfolge $s[i:j:k]$ werden aus s entfernt.	130

Tabelle 7.18 Operatoren für den Datentyp list (Forts.)

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

Verändern eines Wertes innerhalb der Liste – Zuweisung mit []

Sie können 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 – Zuweisung mit []

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 – `del` zusammen mit `[]`

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:

```
>>> 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:¹⁸

¹⁸ Achtung: Wenn in der linken Spalte Parameter mit eckigen Klammern eingeklammert sind, bedeutet dies nach wie vor, dass es sich um optionale Parameter handelt. Diese eckigen Klammern haben nichts mit dem Erzeugen einer neuen Liste zu tun.

Methode	Wirkung	Seite
<i>s.append(x)</i>	Hängt <i>x</i> ans Ende der Liste <i>s</i> an.	131
<i>s.extend(t)</i>	Hängt alle Elemente der Liste <i>t</i> ans Ende der Liste <i>s</i> an.	131
<i>s.insert(i, x)</i>	Fügt <i>x</i> an der Stelle <i>i</i> in die Liste <i>s</i> ein. Anschließend hat <i>s[i]</i> den Wert von <i>x</i> , wobei alle folgenden Elemente um eine Stelle nach hinten aufrücken.	132
<i>s.pop([i])</i>	Gibt das <i>i</i> -te Element der Liste <i>s</i> zurück und entfernt es aus <i>s</i> . Ist <i>i</i> nicht angegeben, wird das letzte Element genommen.	132
<i>s.remove(x)</i>	Entfernt das erste Vorkommen von <i>x</i> aus der Liste <i>s</i> .	132
<i>s.reverse()</i>	Kehrt die Reihenfolge der Elemente in <i>s</i> um.	133
<i>s.sort([key[, reverse]])</i>	Sortiert die Liste <i>s</i> .	133

Tabelle 7.19 Methoden von list-Instanzen

Nun werden die Methoden im Detail besprochen.

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.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_luecke = [1, 2, 3, 5, 6, 7, 8]
>>> erst_mit_luecke.insert(3, 4)
>>> erst_mit_luecke
[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 erheblich weniger Rechenzeit und Speicher kostet, ist `reverse` der Slice-Notation vorzuziehen, wenn Sie die alte Liste nicht mehr 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 Instanzen vom Datentyp `complex`, führt der Aufruf von `sort` ohne Parameter zu einem `TypeError`:

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

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.

Das folgende Beispiel verdeutlicht diese Vorgehensweise.

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

Immer dann, wenn der Sortieralgorithmus zwei Elemente der Liste vergleicht, beispielsweise "Florian" und "Jan", werden nicht die Elemente selbst, sondern die zugehörigen Rückgabewerte der Funktion *key* verglichen. In unserem Beispiel werden somit `len("Florian")` und `len("Jan")`, also die Zahlen 7 und 3, verglichen. Daher ist der String "Jan" in diesem Beispiel vor dem String "Florian" einzuordnen. Grafisch kann man sich das obige Beispiel wie in Abbildung 7.8 veranschaulichen.

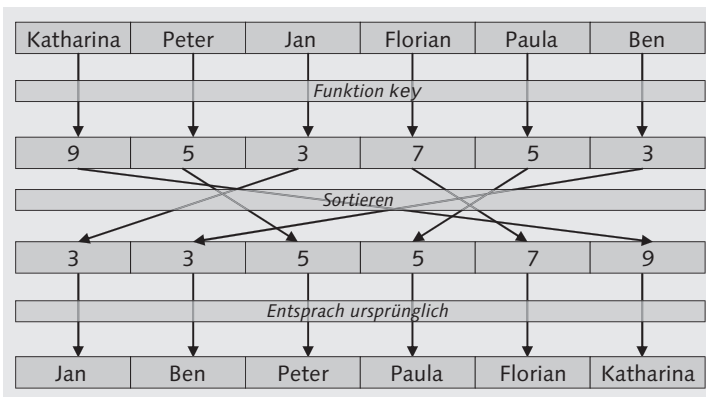


Abbildung 7.8 Sortieren mit key

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 9, »Funktionen«, ab Seite 209.

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]
```



Hinweis

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

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
```

7

Sie werden sich vielleicht fragen, wie `sort` mit solchen Werten verfährt, die bei der Sortierreihenfolge an der gleichen Stelle stehen. Im obigen Beispiel hatten »Jan« und »Ben« mit der Länge 3 sowie »Peter« und »Paula« mit der Länge 5 jeweils den gleichen Wert. Im nächsten Abschnitt lernen Sie, was man unter einem stabilen Sortierverfahren versteht und was das für diese Werte bedeutet.

Stabile Sortierverfahren

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

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

Tabelle 7.20 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
<i>Helmut</i>	<i>Schmidt</i>
Mathias	Schwarz
<i>Natalie</i>	<i>Schmidt</i>
Peter	Kaiser
<i>Ricarda</i>	<i>Schmidt</i>

Tabelle 7.21 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 7.22 Vollständig sortierte Namensliste

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

Weitere Eigenschaften von Listen

Im Zusammenhang mit Pythons `list`-Datentyp ergeben sich ein paar Besonderheiten, die nicht unmittelbar ersichtlich sind.¹⁹

¹⁹ Diese Besonderheiten gelten im Prinzip für alle mutablen Datentypen.

Sie erinnern sich sicherlich noch an die Besonderheit des Operators `+=`, die auf Seite 120 im Zusammenhang mit dem Begriff *in-place* erläutert wurde. Wir möchten dieses Verhalten nochmals aufgreifen und ein paar weitergehende Erläuterungen geben.

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*²⁰ sollten Sie bei der Arbeit mit Listen und anderen mutable Datentypen 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 den Ressourcenverbrauch gegen den Schutz vor Seiteneffekten abwägen, da die Kopien der Listen im Speicher erzeugt werden müssen. Das kostet insbesondere bei langen Listen Rechenzeit und Speicherplatz und kann somit das Programm ausbremsen.

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

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

Zu Beginn referenziert *a* eine Liste, in der eine weitere, leere Liste enthalten ist. Bei der anschließenden Multiplikation mit dem Faktor 4 wird die innere leere Liste nicht kopiert, sondern nur weitere drei Male referenziert. In der Ausgabe sehen wir also viermal dieselbe Liste. Wenn man das verstanden hat, ist es offensichtlich, warum die dem ersten Element von *a* angehängte 10 auch den anderen drei Listen hinzugefügt wird: Es handelt sich einfach um dieselbe Liste. Abbildung 7.9 verdeutlicht diese Tatsache.

²⁰ Seiteneffekte werden im Zusammenhang mit Funktionen auf Seite 222 eine wichtige Rolle spielen.

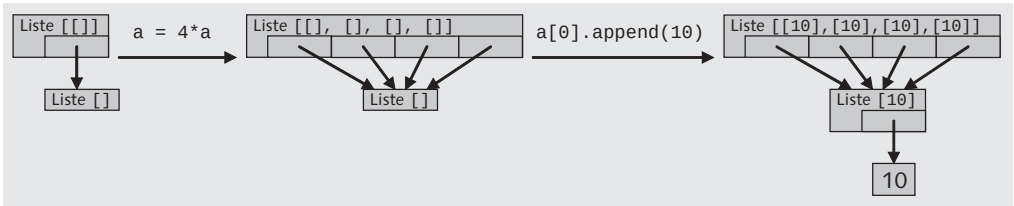


Abbildung 7.9 Seiteneffekt, wenn mehrere Elemente auf dieselbe Liste zeigen.

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.

7.5.3 Unveränderliche Listen – tuple

Der Datentyp `list` ist sehr flexibel und wird häufig verwendet. Seine Mächtigkeit und Flexibilität hat aber auch den Nachteil, dass die Verwaltung einer Liste intern relativ

ressourcenaufwendig ist. 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` immutabel 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, wie er auf Seite 117 beschrieben wird.

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. Versucht man, ein Tupel mit nur einem Element auf die oben beschriebene Weise zu erzeugen, ist der entstehende Programmtext 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 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 Elemente nach der Erzeugung des Tupels nicht mehr verändern.

7.5.4 Strings – str, bytes, bytearray

Dieser Abschnitt behandelt Pythons Umgang mit Zeichenketten und insbesondere die Eigenschaften der dafür bereitgestellten Datentypen `str`, `bytes` und `bytearray`.

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.²¹

Die Zeichen, die eine Instanz des Datentyps `str` speichern kann, sind alle atomaren Bestandteile menschlicher Sprache, wie Buchstaben, Satz- und Leerzeichen oder auch Umlaute. Im Gegensatz dazu sind die Datentypen `bytes` und `bytearray` für die Speicherung von Binärdaten vorgesehen. Daher bestehen Instanzen der Datentypen `bytes` und `bytearray` aus einer Folge von einzelnen Bytes, also ganzen Zahlen von 0 bis 255.

Wir werden uns bis auf weiteres nur mit `str`-Instanzen beschäftigen, da sich der Umgang mit `str` nicht wesentlich von dem mit `bytes` unterscheidet. Einzig beim Umwandeln von `str` nach `bytes` und umgekehrt gibt es einige Stolpersteine, die im Abschnitt über »Zeichensätze und Sonderzeichen« ab Seite 165 besprochen werden.

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 `Enter` gedrückt):

```
>>> s = "Erste Zeile
      File "<stdin>", line 1
        s = "Erste Zeile
              ^
SyntaxError: EOL while scanning string literal
```

²¹ Da `bytearray` im Unterschied zu `str` und `bytes` ein mutable Datentyp ist, werden zusätzlich zum Grundstock der Operationen für sequentielle Datentypen den in diesem Kapitel vorgestellten auch noch viele der Methoden des `list`-Datentyps unterstützt. Eine Auflistung der Methoden, die `bytearray` als mutablen sequentieller Datentyp bietet, finden Sie im Abschnitt 11.5.4, »Datentypen emulieren«, ab Seite 320.

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 = ("Stellen Sie sich einen schrecklich "
...      "komplizierten String vor, den man "
...      "auf keinen Fall in eine Zeile schreiben "
...      "kann.")
>>> a
'Stellen Sie sich einen schrecklich komplizierten String vor, den man auf
keinen Fall in eine Zeile schreiben kann.'
```

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 String-Literal 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` analog.

Um eine neue Instanz des Typs `bytearray` zu erzeugen, dient die gleichnamige Built-in Function `bytearray`.

```
>>> string1 = b"Hallo Welt"
>>> string2 = bytearray(string1)
>>> string2
bytearray(b'Hallo Welt')
```

Auf diese Weise lässt sich also eine `bytearray`-Instanz erzeugen, die ihren Wert von einer bestehenden `bytes`-Instanz übernimmt.

Übergibt man der Built-in Function `bytearray` eine ganze Zahl k als Parameter, so wird ein neues `bytearray` der Länge k erzeugt, wobei jedes der Bytes den Wert Null zugewiesen bekommt:

```
>>> bytearray(7)
bytearray(b'\x00\x00\x00\x00\x00\x00')
```

Was es mit der Darstellung »`\x00`« dieser Zeichen auf sich hat, erfahren Sie im nächsten Abschnitt sowie ab Seite 165 im Kontext der String-Kodierung.

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
\a	Bell (BEL) erzeugt einen Signalton.
\b	Backspace (BS) setzt die Ausgabeposition um ein Zeichen zurück.
\f	Formfeed (FF) erzeugt einen Seitenvorschub.
\n	Linefeed (LF) setzt die Ausgabeposition in die nächste Zeile.
\r	Carriage Return (CR) setzt die Ausgabeposition an den Anfang der nächsten Zeile.
\t	Horizontal Tab (TAB) hat die gleiche Bedeutung wie die Tabulatortaste.
\v	Vertikaler Tabulator (VT); dient zur vertikalen Einrückung.
\"	doppeltes Hochkomma
\'	einfaches Hochkomma
\\	Backslash, der wirklich als solcher in dem String erscheinen soll

Tabelle 7.23 Escape-Sequenzen für Steuerzeichen

Steuerzeichen stammen 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« ab Seite 165 werden wir auf Escape-Sequenzen zurückkommen, um damit beliebige Sonderzeichen wie beispielsweise 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 als maskierte Sonderzeichen interpretiert.

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:

String-Literal	Name
" "	Leerzeichen
"\n"	Zeilenvorschub
"\v"	Vertikaler Tabulator
"\t"	Horizontaler Tabulator
"\n"	Linefeed
"\f"	Formfeed
"\r"	Carriage Return

Tabelle 7.24 Liste der Whitespace-Zeichen

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.

Kategorie	Seite
Trennen eines Strings	147
Suchen eines Teilstrings innerhalb eines anderen Strings	149
Ersetzen eines Teilstrings innerhalb eines anderen Strings	150
Entfernen bestimmter Zeichen am Anfang oder am Ende eines Strings	152
Ausrichten eines Strings	153
Testen, ob ein String eine bestimmte Struktur hat, z. B. eine Zahl ist	154
Verkettung mit einem String als Trennzeichen	156
Stringformatierung	156

Tabelle 7.25 Kategorien der Stringmethoden

Auf den angegebenen Seiten finden Sie Tabellen, die einen Überblick über die zur jeweiligen Kategorie gehörigen Methoden geben.

Trennen von Strings

Um Strings nach bestimmten Regeln in mehrere Teile zu zerlegen, dienen folgende Methoden:

Methode	Beschreibung
<i>s.split([sep[, maxsplit]])</i>	Teilt <i>s</i> bei Vorkommen von <i>sep</i> . Die Suche beginnt am Stringanfang.
<i>s.rsplit([sep[, maxsplit]])</i>	Teilt <i>s</i> bei Vorkommen von <i>sep</i> . Die Suche beginnt am Stringende.
<i>s.splitlines([keepends])</i>	Teilt <i>s</i> bei Vorkommen von Zeilenvorschüben.
<i>s.partition(sep)</i>	Teilt <i>s</i> bei Vorkommen von <i>sep</i> . Die Suche beginnt am Stringanfang. Die resultierende Liste enthält neben den Spaltprodukten auch das Trennzeichen <i>sep</i> .
<i>s.rpartition(sep)</i>	Teilt <i>s</i> bei Vorkommen von <i>sep</i> . Die Suche beginnt am Stringende. Die resultierende Liste enthält neben den Spaltprodukten auch das Trennzeichen <i>sep</i> .

Tabelle 7.26 String-Methoden zum Teilen von Strings

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:

Methode	Beschreibung
<i>s.find(sub[, start[, end]])</i>	Sucht den String <i>sub</i> um String <i>s</i> . Die Suche beginnt am Stringanfang.
<i>s.rfind(sub[, start[, end]])</i>	Sucht den String <i>sub</i> um String <i>s</i> . Die Suche beginnt am Stringende.
<i>s.index(sub[, start[, end]])</i>	Sucht den String <i>sub</i> um String <i>s</i> . Die Suche beginnt am Stringanfang. Ist <i>sub</i> nicht in <i>s</i> vorhanden, wird eine Exception geworfen.
<i>s.rindex(sub[, start[, end]])</i>	Sucht den String <i>sub</i> um String <i>s</i> . Die Suche beginnt am Stringende. Ist <i>sub</i> nicht in <i>s</i> vorhanden, wird eine Exception geworfen.
<i>s.count(sub[, start[, end]])</i>	Zählt die Vorkommen von <i>sub</i> in <i>s</i> .

Tabelle 7.27 String-Methoden zum Suchen in Strings

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 bei *s[start]* beginnt und das Element *s[end]* nicht mehr enthält.

Um herauszufinden, ob und gegebenenfalls an welcher Stelle ein bestimmter String in einem anderen vorkommt, bietet Python die Methoden `find` und `index` mit ihren Gegenstücken `rfind` und `rindex` an. Die Methode `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.²²

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:

²² Sie können die Details in Abschnitt 12.1, »Exception Handling«, auf Seite 329 nachlesen.

Methode	Beschreibung
<code>s.replace(old, new[, count])</code>	Ersetzt die Vorkommen von <i>old</i> im String <i>s</i> durch <i>new</i> .
<code>s.lower()</code>	Ersetzt alle Großbuchstaben in <i>s</i> durch entsprechende Kleinbuchstaben.
<code>s.upper()</code>	Ersetzt alle Kleinbuchstaben in <i>s</i> durch entsprechende Großbuchstaben.
<code>s.swapcase()</code>	Ersetzt alle Großbuchstaben in <i>s</i> durch entsprechende Kleinbuchstaben und umgekehrt alle Kleinbuchstaben durch entsprechende Großbuchstaben.
<code>s.capitalize()</code>	Ersetzt den ersten Buchstaben von <i>s</i> durch den entsprechenden Großbuchstaben und alle folgenden Großbuchstaben durch entsprechende Kleinbuchstaben.
<code>s.title()</code>	Ändert die Groß-/Kleinschreibung von <i>s</i> dergestalt, dass alle Wörter bis auf den ersten Buchstaben klein geschrieben werden.
<code>s.expandtabs([tabsize])</code>	Rückt <i>s</i> ein, indem Tabs (" <code>\t</code> ") durch Leerzeichen ersetzt werden.

Tabelle 7.28 String-Methoden zum Ersetzen von Teilstrings

Die Methode `replace` gibt einen String zurück, in dem alle Vorkommen von *old* durch *new* ersetzt wurden:

```
>>> falsch = "Python ist nicht toll!"
>>> richtig = falsch.replace("nicht", "richtig")
>>> richtig
'Python ist richtig toll!'
```

Mit dem Parameter *count* kann die Anzahl der Ersetzungen begrenzt werden:

```
>>> s = "Bitte nur die ersten vier e ersetzen"
>>> s.replace("e", "E", 4)
'BitteE 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:

Methode	Beschreibung
<i>s.strip([chars])</i>	Entfernt bestimmte Zeichen am Anfang und am Ende des Strings <i>s</i> .
<i>s.lstrip([chars])</i>	Entfernt bestimmte Zeichen am Anfang des Strings <i>s</i> .
<i>s.rstrip([chars])</i>	Entfernt bestimmte Zeichen am Ende des Strings <i>s</i> .

Tabelle 7.29 String-Methoden zum Entfernen bestimmter Zeichen am Anfang oder Ende

Die Methode `strip` entfernt unerwünschte Zeichen auf beiden Seiten des Strings. Die Methode `rstrip` 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:

Methode	Beschreibung
<i>s.center(width[, fillchar])</i>	Zentriert <i>s</i> im resultierenden String.
<i>s.ljust(width[, fillchar])</i>	Richtet <i>s</i> im resultierenden String linksbündig aus.

Tabelle 7.30 String-Methoden zum Ausrichten

Methode	Beschreibung
<code>s.rjust(width[, fillchar])</code>	Richtet <code>s</code> im resultierenden String rechtsbündig aus.
<code>s.zfill(width)</code>	Richtet <code>s</code> rechtsbündig aus, indem links mit Nullen aufgefüllt wird.

Tabelle 7.30 String-Methoden zum Ausrichten (Forts.)

Mit dem Parameter `width` geben Sie die gewünschte Länge des neuen Strings an. Die Methode `center` zentriert `s` im neuen String, `ljust` richtet `s` linksbündig aus, `rjust` richtet `s` rechtsbündig 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'
```

Ist die Länge von `s` größer als der Wert des Parameters `width`, wird eine Kopie von `s` zurückgegeben, da in diesem Fall nicht genügend Raum zum Ausrichten vorhanden ist.

Die Methode `zfill` ist ein Spezialfall von `rjust` und für Strings gedacht, die numerische Werte enthalten. Ein Aufruf der Methode `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
<i>s.startswith(prefix[, start[, end]])</i>	True, wenn <i>s</i> mit dem String <i>prefix</i> beginnt
<i>s.endswith(suffix[, start[, end]])</i>	False, wenn <i>s</i> mit dem String <i>suffix</i> endet

Tabelle 7.31 Methoden für String-Tests

Da sich die ersten sieben Methoden 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`:

Die optionalen Parameter *start* und *end* begrenzen dabei – wie schon bei den Suchen- und Ersetzen-Methoden – die Abfrage auf den Bereich *s[start:end]*:

```
>>> 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. Für diesen Zweck stellt Python die Methode `join` zur Verfügung:

Methode	Beschreibung
<code>s.join(seq)</code>	Verkettet die Elemente der Sequenz <code>seq</code> zu einem neuen String, wobei <code>s</code> als Trennzeichen dient.

Tabelle 7.32 String-Methode zum Verketteten mehrerer Elemente mit einem Trennstring

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. Im folgenden Beispiel werden mehrere Namen, durch Komma getrennt, verkettet:

```
>>> kontaktliste = ["Fix", "Foxy", "Lupo", "Dr. Knox"]
>>> ", ".join(kontaktliste)
'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'
```

Die Methode `join` wird oft angewendet, um die Elemente einer Sequenz ohne ein Trennzeichen zu verketteten. In diesem Fall ruft man die Methode `join` des leeren Strings auf:

```
>>> "".join(["www", ".", "galileo-computing", ".", "de"])
'www.galileo-computing.de'
```

Im nächsten Abschnitt beschäftigen wir uns mit dem Thema String-Formatierung.

Formatierung von Strings

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`.²³ 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.

Es ist auch möglich, diese Nummerierung implizit vornehmen zu lassen, indem nichts zwischen die geschweiften Klammern geschrieben wird. Python nummeriert die Platzhalter dann automatisch, bei 0 beginnend:

```
>>> "Es ist {}:{}".format(13, 37)
'Es ist 13:37 Uhr'
```

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 wird, sie als Ganzzahlen zu interpretieren.

²³ Die Methode `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 String-Formatierung 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. Näheres zu den Unterschieden zwischen Python 2 und Python 3 finden Sie in Kapitel 27, »Von Python 2 nach Python 3« (S. 955).

Man kann auch nummerierte Platzhalter mit symbolischen Platzhaltern mischen:

```
>>> "Es ist {stunde}:{0} Uhr".format(37, stunde=13)
'Es ist 13:37 Uhr'
```

Dieses Mischen symbolischer und nummerierter Platzhalter funktioniert auch zusammen mit der impliziten Nummerierung. Dabei werden alle unbenannten Platzhalter, bei 0 beginnend, durchnummeriert.

```
>>> "{h}g Hefe, {g}g Mehl, {w}ml Wasser, {g}g Salz".format(
    50, 400, h=5, w=100)
'5g Hefe, 50g Mehl, 100ml Wasser, 400g Salz'
```

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.²⁴ 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:

```
>>> "Unformatiert: {{KeinPlatzhalter}}. Formatiert: {v}.".format(
    v="nur ein Test")
'Nicht formatiert: {KeinPlatzhalter}. Formatiert: nur ein Test.'
```

Zugriff auf Attribute 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:

²⁴ Näheres dazu, wie diese Konvertierung intern abläuft und beeinflusst werden kann, finden Sie in Abschnitt 11.5, »Magic Methods und Magic Attributes«, ab Seite 302.

```
>>> 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:

```
>>> 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 kennen gelernt 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 7.6, »Mappings«, ab Seite 173 behandeln werden.²⁵

Sowohl der Zugriff auf Attribute als auch der Operator `[]` funktionieren auch für die implizite Nummerierung von Platzhaltern:

```
>>> "Attribut: {0.imag}, Listenelement: {[1]}".format(
    1+4j, [1,2,3])
'Attribut: 4.0, Listenelement: 2'
```

Im Folgenden erfahren Sie, wie Sie die Ersetzung selbst beeinflussen können.

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'
```

²⁵ 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.

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 beispielsweise eine Tabelle mit Namen ausgeben und sicherstellen, dass alles bündig untereinander steht, 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:2}".format(lang="Ich bin laenger als zwei Zeichen!")
'Ich bin laenger als zwei 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 es im ersten der beiden obigen Beispiele 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. (Siehe Beispiel unten.) Diese Angabe ist ausschließlich bei numerischen Werten sinnvoll und führt bei anderen Datentypen zu einem <code>ValueError</code> .
^	Der Wert wird zentriert in den reservierten Platz eingefügt.

Tabelle 7.33 Ausrichtungsarten

Ein Beispiel soll die nicht ganz intuitive Wirkung der Ausrichtungsart »=`«` verdeutlichen. Dabei ist die Position des Vorzeichens interessant.

```
>>> "Temperatur: {:10}".format(-12.5)
'Temperatur:      -12.5'
>>> "Temperatur: {::=10}".format(-12.5)
'Temperatur: -    12.5'
```

Beachten Sie, dass eine Ausrichtungsangabe keinen Effekt hat, wenn der eingefügte Wert genauso lang wie oder länger als die gewünschte 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 die 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.
(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 7.34 Vorzeichenbehandlungsarten

Wir demonstrieren nun 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.

Zahlendarstellungstypen – type

Um bei Zahlenwerten die Ausgabe weiter anpassen zu können, gibt es verschiedene Ausgabebetypen, 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« ab Seite 165.
d	Die Zahl wird in Dezimaldarstellung ausgegeben. Dies ist das Standardverhalten.
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).

Tabelle 7.35 Ausgabebetypen 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 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.
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 7.36 Ausgabetypen für Gleitkommazahlen

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 Ausgabetyt, 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.'
```

Tausendertrennung – die Option »,«

Wird die Option »,« gesetzt, so werden Tausenderblöcke durch ein Komma voneinander getrennt:

```
>>> "Viel Geld: {:,d}".format(12345678900)
'Viel Geld: 12,345,678,900'
```

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 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*²⁶ 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 erhöhte, sodass $2^8 = 256$ verschiedene Zeichen gespeichert werden können. Damit wurde Platz für 128 weitere Sonderzeichen geschaffen. Welche Interpretation konkret für diese weiteren Plätze verwendet wird, legt die verwendete Codepage fest. Welche Codepage verwendet wird, hängt von der Konfiguration des jeweiligen Rechners ab. Insbesondere haben das Betriebssystem und die regionalen Einstellungen Einfluss auf die Auswahl der Codepage.

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:

```
>>> 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:

²⁶ »American Standard Code for Information Interchange« (dt. »Amerikanische Standardcodierung für den Informationsaustausch«)

```
>>> 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« deckt 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 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:

```
>>> '\xdcbpr\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"). Die Darstellungen 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. Letzterer Nachteil bedeutet, dass man immer genau wissen muss, mit welcher Codepage gearbeitet wurde, um einen kodierten String korrekt zu interpretieren. Dies ist insbesondere beim Datenaustausch oder beim Lesen und Schreiben von Dateien unbequem und fehleranfällig.

Ein wirklicher Mangel ist 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 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 besteht darin, den Speicherplatz pro Zeichen zu erhöhen, was aber neue Nachteile mit sich bringt. Verwendet man beispielsweise 16 Bits für jedes einzelne Zeichen, ist die Anzahl der Zeichen immer noch auf 65.536 begrenzt. Es ist davon auszugehen, dass die Sprachen sich weiterentwickeln werden und somit auch diese Anzahl auf Dauer nicht mehr ausreichen wird.²⁷ 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 *Unicode* erarbeitet, der variable Kodierungslängen für einzelne Zeichen vorsieht. Im Prinzip ist Unicode eine riesige Tabelle, die jedem bekannten Zeichen eine Zahl, den sogenannten *Codepoint*, zuweist. Diese Tabelle wird vom *Unicode Consortium*, einer gemeinnützigen Institution, gepflegt und ständig erweitert. Codepoints werden in der Regel als »U+x« geschrieben, wobei x die hexadezimale Repräsentation des Codepoints ist. Das wirklich Neue an Unicode ist das Verfahren *UTF* (*Unicode Transformation Format*), das Codepoints durch Byte-Folgen unterschiedlicher Länge darstellen kann. Es gibt verschiedene dieser Transformationsformate, aber das wichtigste und am weitesten verbreitete ist UTF-8. UTF-8 verwendet bis zu 7 Byte, um ein einzelnes Zeichen zu kodieren, wobei die tatsächliche Länge von der Häufigkeit des Zeichens in Texten abhängt. So lassen sich zum Beispiel alle Zeichen des ASCII-Standards mit jeweils einem Byte kodieren, das zusätzlich den gleichen Zahlenwert wie die entsprechende ASCII-Kodierung des Zeichens hat. Durch dieses Vorgehen wird erreicht, dass jeder mit ASCII kodierte String auch gültiger UTF-8-Code ist: UTF-8 ist zu ASCII abwärtskompatibel. Wir interessieren uns an dieser Stelle nicht weiter für die technische Umsetzung von Unicode, sondern vielmehr, 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

²⁷ Es ist tatsächlich so, dass 16 Bits schon heute nicht mehr ausreichen, um alle Zeichen der menschlichen Sprache zu kodieren.

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, hexadezimal: `0x20ac`), gibt es Escape-Sequenzen, die mit `\u` eingeleitet werden:

```
>>> s = "\u20ac"
>>> print(s)
€
```

Der neue Datentyp `str` eignet sich für die Arbeit mit Text-Strings in Python-Programmen und vereinfacht dabei den Umgang mit internationalen Schriftzeichen. Allerdings gibt es einige Besonderheiten, die bei der Verwendung des neuen `str` beachtet werden müssen.²⁸ 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 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. Instanzen des Typs `str` 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; €'
```

²⁸ Vor allem, wenn Sie den Umgang mit 8-Bit-Strings gewohnt sind, ist hier Vorsicht geboten.

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 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:

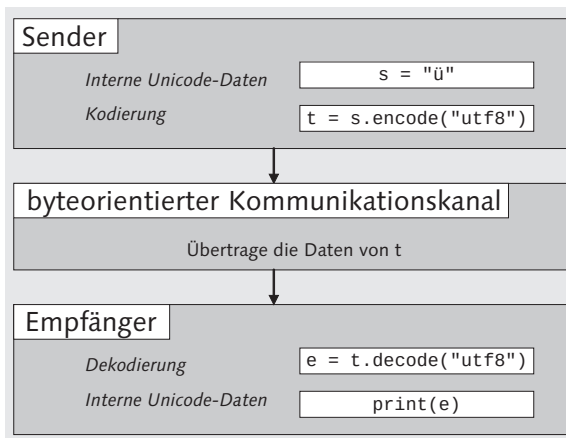


Abbildung 7.10 Schematische Darstellung eines Unicode-Transfers

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:

- ▶ Benutzen Sie `bytes` ausschließlich für Binärdaten.
- ▶ Verwenden Sie für alle Textdaten, die das Programm verwendet, `str`-Instanzen.

- Kodieren Sie `str`-Daten beim Speichern oder beim Datenversand zu anderen Programmen.
- 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
"iso-8859-15"	Kodierung der Zeichen für Westeuropa, wie sie von der Internationalen Organisation für Normung (ISO) 1999 standardisiert wurde
"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 7.37 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)  
Ω
```



```
>>> 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 das 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. ²⁹ (Nur bei <code>encode</code> möglich.)
"backslashreplace"	Nicht kodierbare Zeichen werden durch eine Escape-Sequenz ersetzt. (Nur bei <code>encode</code> möglich.)

Tabelle 7.38 Werte für den *errors*-Parameter von `encode` und `decode`

Wir betrachten das letzte Beispiel mit anderen Werten für *errors*:

²⁹ Dabei handelt es sich um spezielle Formatierungen zur Darstellung von Sonderzeichen in XML-Dateien. Näheres zu XML-Dateien erfahren Sie in Abschnitt 18.2 ab Seite 551.

```
>>> s = "\u03a9"
>>> print(s)
Ω
>>> s.encode("cp1252", "replace")
b'?'
>>> s.encode("cp1252", "xmlcharrefreplace")
b'&#937;'
>>> s.encode("cp1252", "backslashreplace")
b'\\u03a9'
```

7

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.³⁰ Ein Encoding ermöglicht es dem Python-Interpreter dann, diese Zeichen korrekt zuzuordnen. Eine Encoding-Deklaration steht in der Regel direkt unter der Shebang-Zeile³¹ bzw. in der ersten Zeile der Programmdatei und sieht folgendermaßen aus:

```
# -*- 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 unüblich wäre.

7.6 Mappings

Die Kategorie *Mappings* (dt. »Zuordnungen«) enthält Datentypen, die eine Zuordnung zwischen verschiedenen Objekten herstellen.

30 Oder Sie speichern Ihre Programme UTF-8-kodiert, was Python seit 3.0 standardmäßig erwartet.

Wie Sie Ihren Texteditor so konfigurieren können, dass er beim Speichern eine bestimmte Kodierung verwendet, entnehmen Sie bitte der Dokumentation des Editors.

31 Die Bedeutung einer Shebang-Zeile wird in Abschnitt 2.2.1 erklärt.

7.6.1 Dictionary – dict

Der einzige Datentyp der Kategorie Mappings ist das *Dictionary* (dt. »Wörterbuch«), 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 *assoziativer 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 wird ein `dict` mit zwei Einträgen angelegt, die durch ein Komma getrennt werden. Beim ersten Eintrag wird dem Schlüssel "Germany" der Wert "Deutschland" zugewiesen. Schlüssel und Wert sind durch einen Doppelpunkt voneinander getrennt. Es müssen nicht notwendigerweise alle Paare in eine Zeile geschrieben werden. Innerhalb der geschweiften Klammern kann der Quellcode beliebig formatiert werden:

```
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 wird aber nur das zweite Schlüssel-Wert-Paar ins Dictionary übernommen:

```
d = {
    "Germany" : "Deutschland",
    "Germany" : "Pusemuckel"
}
```

Im Gegensatz dazu brauchen die Werte eines Dictionarys nicht eindeutig zu sein, dürfen also 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 so sein:

```
mapping = {
    0 : 1,
    "abc" : 0.5,
    1.2e22 : [1,2,3,4],
    (1,3,3,7) : "def"
}
```

7

In einem Dictionary können beliebige Instanzen, seien sie mutable oder immutable, als Werte verwendet werden. Bei dem Schlüssel ist zu beachten, dass nur Instanzen unveränderlicher (immutable) Datentypen verwendet werden dürfen. Dabei handelt es sich um alle bisher besprochenen Datentypen mit Ausnahme der Listen und der Dictionarys selbst. Versuchen wir beispielsweise, ein Dictionary zu erstellen, in dem eine Liste als Schlüssel verwendet wird, so meldet sich der Interpreter mit einem entsprechenden Fehler:

```
>>> d = {[1,2,3] : "abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Diese Beschränkung rührt daher, dass die Schlüssel eines Dictionarys anhand eines aus ihrem Wert errechneten *Hash-Wertes* 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.³²

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:

³² In Abschnitt 11.5.1 auf Seite 305 erfahren Sie, wie Sie die Hash-Berechnung für eigene Datentypen implementieren können.

```
for key in woerterbuch:
    print(key)
```

Die Ausgabe des Codes sieht erwartungsgemäß folgendermaßen aus:

```
Germany
Spain
France
```



Hinweis

Bei einem Dictionary handelt es sich um einen ungeordneten Datentyp. Das bedeutet, dass die Schlüssel nicht zwingend in der Reihenfolge durchlaufen werden, in der sie dem Dictionary hinzugefügt wurden.

Selbstverständlich kann in einer Schleife auch auf die Werte des Dictionarys zugegriffen werden. Dazu bedient man sich des Zugriffsoperators, den wir im Folgenden behandeln werden. Beachten Sie, dass Sie die Größe des Dictionarys nicht verändern dürfen, während es in einer Schleife durchlaufen wird. Die Größe des Dictionarys würde zum Beispiel durch das Hinzufügen oder Löschen eines Schlüssel-Wert-Paares beeinflusst. Sollten Sie es dennoch versuchen, bekommen Sie folgende Fehlermeldung angezeigt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
```

Diese Beschränkung gilt ausschließlich für Operationen, die die Größe des Dictionarys beeinflussen, also beispielsweise das Hinzufügen und Entfernen von Einträgen. Sollten Sie in einer Schleife lediglich den zugehörigen Wert eines Schlüssels ändern, tritt kein Fehler auf.

Operatoren

Bisher haben Sie gelernt, was ein Dictionary ist und wie es erzeugt wird. Außerdem sind wir auf einige Besonderheiten eingegangen. Jetzt besprechen wir die für Dictionarys verfügbaren Operatoren.

Operator	Beschreibung
<code>len(d)</code>	Liefert die Anzahl aller im Dictionary <i>d</i> enthaltenen Schlüssel-Wert-Paare.

Tabelle 7.39 Operatoren eines Dictionarys

Operator	Beschreibung
<i>d[k]</i>	Zugriff auf den Wert mit dem Schlüssel <i>k</i>
<i>del d[k]</i>	Löschen des Schlüssels <i>k</i> und seines Wertes
<i>k in d</i>	True, wenn sich der Schlüssel <i>k</i> in <i>d</i> befindet
<i>k not in d</i>	True, wenn sich der Schlüssel <i>k</i> nicht in <i>d</i> befindet

Tabelle 7.39 Operatoren eines Dictionarys (Forts.)

Im Folgenden werden die Operatoren eines Dictionarys im Detail besprochen. Die meisten der Operatoren werden anhand des Dictionarys `woerterbuch` erklärt, das folgendermaßen definiert ist:

```
woerterbuch = {
    "Germany" : "Deutschland",
    "Spain" : "Spanien",
    "France" : "Frankreich"
}
```

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

Mithilfe des Zugriffsoperators kann auf einen Wert eines Dictionarys zugegriffen werden. Dazu schreiben Sie den entsprechenden Schlüssel in eckigen Klammern hinter den Namen des Dictionarys. Bei dem Beispiel-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

Ähnlich wie bei den Listen existieren für Dictionarys die Operatoren `in` und `not in`, die verwendet werden, um zu testen, ob ein Schlüssel in einem Dictionary vorhanden ist oder nicht. Sie geben das entsprechende Ergebnis als Wahrheitswert zurück:

```
>>> "France" in woerterbuch
True
>>> "Spain" not in woerterbuch
False
```

Methoden

Neben den Operatoren sind einige Methoden definiert, die die Arbeit mit Dictionarys erleichtern.

Methode	Beschreibung	Seite
<i>d.clear()</i>	Löscht den Inhalt des Dictionarys <i>d</i> . Das Dictionary selbst bleibt bestehen.	180
<i>d.copy()</i>	Erzeugt eine Kopie von <i>d</i> .	180
<i>d.get(k, x)</i>	Liefert <i>d[k]</i> , wenn der Schlüssel <i>k</i> vorhanden ist, ansonsten <i>x</i> .	181

Tabelle 7.40 Methoden eines Dictionarys

Methode	Beschreibung	Seite
<i>d.items()</i>	Gibt ein iterierbares Objekt zurück, das alle Schlüssel-Wert-Paare von <i>d</i> durchläuft. ³³	181
<i>d.keys()</i>	Gibt ein iterierbares Objekt zurück, das alle Schlüssel von <i>d</i> durchläuft. ³³	182
<i>d.pop(k)</i>	Gibt den zum Schlüssel <i>key</i> gehörigen Wert zurück und löscht das Schlüssel-Wert-Paar aus dem Dictionary <i>d</i> .	182
<i>d.popitem()</i>	Gibt ein willkürliches Schlüssel-Wert-Paar von <i>d</i> zurück und entfernt es aus dem Dictionary.	183
<i>d.setdefault(k[, x])</i>	Das Gegenteil von <i>get</i> . Setzt <i>d[k] = x</i> , wenn der Schlüssel <i>k</i> nicht vorhanden ist.	183
<i>d.update(d2)</i>	Fügt ein Dictionary <i>d2</i> zu <i>d</i> hinzu und überschreibt gegebenenfalls die Werte von bereits vorhandenen Schlüsseln.	184
<i>d.values()</i>	Gibt ein iterierbares Objekt zurück, das alle Werte von <i>d</i> durchläuft. ³³	184

Tabelle 7.40 Methoden eines Dictionarys (Forts.)

Abgesehen von diesen Methoden, stellt der Datentyp `dict` noch eine statische Methode bereit. Das ist eine Methode, die auch ohne konkrete Dictionary-Instanz aufgerufen werden kann.

Methode	Beschreibung	Seite
<i>dict.fromkeys(seq[, value])</i>	Erstellt ein neues Dictionary mit den Werten der Liste <i>seq</i> als Schlüssel und setzt jeden Wert initial auf <i>value</i> .	184

Tabelle 7.41 Statische Methoden eines Dictionarys

Jetzt werden alle Methoden detailliert und jeweils mit einem kurzen Beispiel im interaktiven Modus erläutert. Alle Beispiele verstehen sich dabei in folgendem Kontext:

³³ 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 (siehe Abschnitt 12.4, S. 353) lässt sich wie eine Liste in einer `for`-Schleife durchlaufen. Sollten Sie unbedingt eine Liste benötigen, verwenden Sie `list(d.items())`. Analoges gilt für die Methoden `keys` und `values`.


```
>>> d = {"k1" : "v1", "k2": "v2", "k3": "v3"}
```

Es ist also in jedem Beispiel ein Dictionary *d* mit drei Schlüssel-Wert-Paaren vorhanden. In den Beispielen werden wir das Dictionary verändern und uns vom Interpreter seinen Wert ausgeben lassen. Die Ausgabe des unveränderten Dictionarys sieht folgendermaßen aus:

```
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}
```

Sie können dabei jedes Beispiel für sich betrachten und von diesen Grundvoraussetzungen ausgehen. Änderungen, die in einem Beispiel an dem Dictionary *d* durchgeführt werden, wirken sich nicht auf die Folgebeispiele aus.

d.clear()

Die Methode *clear* löscht alle Schlüssel-Wert-Paare von *d*. Sie hat dabei nicht den gleichen Effekt wie *del d*, da das Dictionary selbst nicht gelöscht, sondern nur geleert wird:

```
>>> d.clear()
>>> d
{}
```

d.copy()

Die Methode *copy* erzeugt eine Kopie des Dictionarys *d*.

```
>>> e = d.copy()
>>> e
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1'}
```

Beachten Sie, dass zwar das Dictionary selbst kopiert wird, es sich bei den Werten aber nach wie vor um Referenzen auf dieselben Objekte handelt. Dies wird durch das folgende Beispiel verdeutlicht.

```
>>> d1 = {"key" : [1,2,3]}
>>> d2 = d1.copy()
>>> d2["key"].append(4)
>>> d2
{'key': [1, 2, 3, 4]}
>>> d1
{'key': [1, 2, 3, 4]}
```

Es wird ein Dictionary *d1* angelegt, das ein einziges Schlüssel-Wert-Paar mit einer Liste als Wert enthält. Das Dictionary *d1* wird durch einen Aufruf der Methode *copy*

kopiert und anschließend die in der Kopie `d2` als Wert referenzierte Liste um ein Element erweitert. Da die Methode `copy` nur eine oberflächliche Kopie durchführt, enthalten die beiden Dictionaries `d1` und `d2` eine Referenz auf dieselbe Liste:

```
>>> d1["key"] is d2["key"]
True
```

Die Veränderung dieser Liste führt zu einer Veränderung beider Dictionaries. Eine vollständige Kopie, die die Werte eines Dictionaries mit einbezieht, kann mit dem Modul `copy` der Standardbibliothek durchgeführt werden, das in Abschnitt 16.7 (S. 519) beschrieben wird.

d.get(k, x)

Die Methode `get` ermöglicht den Zugriff auf einen Wert des Dictionaries. Im Gegensatz zum Zugriffsoperator wird aber keine Exception erzeugt, wenn der Schlüssel nicht vorhanden ist. Stattdessen wird in diesem Fall der optionale Parameter `x` zurückgegeben. Sollte `x` nicht angegeben worden sein, so wird er als `None` angenommen. Die Zeile

```
wert = d.get(k,x)
```

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.items()

Die Methode `items` gibt ein iterierbares Objekt über alle Schlüssel-Wert-Paare des Dictionaries zurück. Dieses kann folgendermaßen mit einer `for`-Schleife durchlaufen werden:

```
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')
```

Beachten Sie, dass die Einträge in einem Dictionary nicht geordnet sind und auch in keiner festgelegten Reihenfolge durchlaufen werden.

d.keys()

Die Methode `keys` gibt ein iterierbares Objekt über alle Schlüssel des Dictionarys zurück. 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 durch folgenden Code umgangen werden:

```
for key in d:
    print(key)
```

Beide Beispiele sind äquivalent und erzeugen folgende Ausgabe:

```
k3
k2
k1
```

Dennoch hat die Methode `keys` ihre Berechtigung, beispielsweise um die Schlüssel eines Dictionarys in eine Liste zu schreiben:

```
>>> list(d.keys())
['k3', 'k2', 'k1']
```

Beachten Sie, dass die Einträge in einem Dictionary nicht geordnet sind und auch in keiner festgelegten Reihenfolge durchlaufen werden.

d.pop(k)

Die Methode `pop` löscht das Schlüssel-Wert-Paar mit dem Schlüssel `k` aus dem Dictionary und gibt den Wert dieses Paares 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:³⁴

```
>>> d.popitem()
('k3', 'v3')
>>> d
{'k2': 'v2', 'k1': 'v1'}
```

Sollte *d* leer sein, so wird eine entsprechende Exception erzeugt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

d.setdefault(k, x)

Die Methode *setdefault* fügt das Schlüssel-Wert-Paar $\{k : x\}$ zum Dictionary *d* hinzu, sollte der Schlüssel *k* nicht vorhanden sein. Der Parameter *x* ist optional und mit *None* vorbelegt.

```
>>> d.setdefault("k2", 1337)
'v2'
>>> d.setdefault("k5", 1337)
1337
>>> d
{'k3': 'v3', 'k2': 'v2', 'k1': 'v1', 'k5': 1337}
```

Unabhängig davon, ob das Schlüssel-Wert-Paar ins Dictionary übernommen wurde oder nicht, gibt die Methode *setdefault* den Wert $d[k]$ zurück.

³⁴ Das bedeutet, dass die Schlüssel-Wert-Paare in einer von der Implementierung abhängigen Reihenfolge von der Methode *popitem* zurückgegeben werden. Diese Reihenfolge ist aber nicht zufällig, es könnte beispielsweise stets das zuletzt eingefügte Schlüssel-Wert-Paar zurückgegeben werden.

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.values()

Die Methode *values* verhält sich ähnlich wie *keys*, mit dem Unterschied, dass alle Werte durchlaufen werden:

```
for value in d.values():
    print(value)
```

Das Beispiel erzeugt folgende Ausgabe:

```
v3
v2
v1
```

Beachten Sie, dass die Einträge in einem Dictionary nicht geordnet sind und auch in keiner festgelegten Reihenfolge durchlaufen werden.

dict.fromkeys(seq[, value])

Die statische Methode *fromkeys* erzeugt ein neues Dictionary und verwendet dabei die Einträge des iterierbaren Objekts *seq* als Schlüssel. Der Parameter *value* ist optional. Sollte er jedoch angegeben werden, so wird er als Wert eines jeden Schlüssel-Wert-Paars verwendet:

```
>>> dict.fromkeys([1,2,3], "python")
{1: 'python', 2: 'python', 3: 'python'}
```

Wird der Parameter *value* ausgelassen, so wird stets *None* als Wert eingetragen:

```
>>> dict.fromkeys([1,2,3])
{1: None, 2: None, 3: None}
```

7.7 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.

7.7.1 Die Datentypen `set` und `frozenset`

Eine leere Instanz der Datentypen `set` und `frozenset` wird folgendermaßen erzeugt:

```
s = set()
fs = frozenset()
```

Wenn die Menge bereits zum Zeitpunkt der Instanziierung Elemente enthalten soll, können diese den Konstruktoren als Elemente eines iterierbaren Objekts, beispielsweise eines Tuples, übergeben werden:

```
s = set(("A", "B", "C"))
fs = frozenset([True, 47, (1,2,3)])
```

Alternativ existiert seit Python 3.0 ein spezielles Literal zum Erzeugen von veränderlichen Mengen. Wie in der Mathematik werden die Elemente, die die Menge enthalten soll, durch Kommata getrennt, in geschweifte Klammern geschrieben.

```
>>> s = {1, 2, 3, 99, -7}
>>> s
{3, -7, 2, 99, 1}
```

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 – `{}` instanziiert stets ein leeres Dictionary. Leere Mengen müssen also, wie oben gezeigt, über `set()` instanziiert werden. Für den Datentyp `frozenset` existiert kein Literal, er muss stets über den Konstruktor `frozenset()` instanziiert werden.

Bei einer Menge handelt es sich um ein iterierbares Objekt, das 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
```

Beachten Sie hierbei, dass Mengen ungeordnete Datentypen sind, ihre Elemente also in keiner festgelegten Reihenfolge durchlaufen werden.

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	Seite
<code>len(s)</code>	Liefert die Anzahl aller in der Menge <code>s</code> enthaltenen Elemente.	188
<code>x in s</code>	<code>True</code> , wenn <code>x</code> in der Menge <code>s</code> enthalten ist, andernfalls <code>False</code>	188
<code>x not in s</code>	<code>True</code> , wenn <code>x</code> nicht in der Menge <code>s</code> enthalten ist, andernfalls <code>False</code>	188
<code>s <= t</code>	<code>True</code> , wenn es sich bei der Menge <code>s</code> um eine Teilmenge der Menge <code>t</code> handelt, andernfalls <code>False</code>	188
<code>s < t</code>	<code>True</code> , wenn es sich bei der Menge <code>s</code> um eine echte Teilmenge ³⁵ der Menge <code>t</code> handelt, andernfalls <code>False</code>	188
<code>s >= t</code>	<code>True</code> , wenn es sich bei der Menge <code>t</code> um eine Teilmenge der Menge <code>s</code> handelt, andernfalls <code>False</code>	188
<code>s > t</code>	<code>True</code> , wenn es sich bei der Menge <code>t</code> um eine echte Teilmenge der Menge <code>s</code> handelt, andernfalls <code>False</code>	188

Tabelle 7.42 Operatoren der Datentypen `set` und `frozenset`

³⁵ Eine Menge `T` wird »echte Teilmenge« einer zweiten Menge `M` genannt, wenn `T` Teilmenge von `M` ist und es mindestens ein Element gibt, das in `M`, aber nicht in `T` enthalten ist.

Operator	Beschreibung	Seite
s / t	Erzeugt eine neue Menge, die alle Elemente von s und t enthält. Diese Operation bildet also die Vereinigungsmenge zweier Mengen.	189
$s \& t$	Erzeugt eine neue Menge, die die Objekte enthält, die sowohl Element der Menge s als auch Element der Menge t sind. Diese Operation bildet also die Schnittmenge zweier Sets.	190
$s - t$	Erzeugt eine neue Menge mit allen Elementen von s , außer denen, die auch in t enthalten sind. Diese Operation erzeugt also die Differenz zweier Mengen.	190
$s \wedge t$	Erzeugt eine neue Menge, die alle Objekte enthält, die entweder in s oder in t vorkommen, nicht aber in beiden. Diese Operation bildet also die symmetrische Differenz ³⁶ zweier Mengen.	191

Tabelle 7.42 Operatoren der Datentypen set und frozenset (Forts.)

Für einige dieser Operatoren existieren auch erweiterte Zuweisungen.

Operator	Entsprechung
$s /= t$	$s = s / t$
$s \&= t$	$s = s \& t$
$s -= t$	$s = s - t$
$s \wedge= t$	$s = s \wedge t$

Tabelle 7.43 Operatoren des Datentyps set

Diese Operatoren gibt es auch für den Datentyp `frozenset`. Sie verändern aber nicht 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.

Im Folgenden werden alle Operatoren anhand von Beispielen anschaulich beschrieben. Die Beispiele sind dabei in diesem Kontext zu sehen:

³⁶ Die symmetrische Differenz zweier Mengen A und B enthält alle Elemente von A und B , die nur in genau einer der beiden Mengen enthalten sind. Der Operator $\wedge$ lässt sich also folgendermaßen nachbilden:

$$A \wedge B = (A \mid B) - (A \& B)$$


```
>>> 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 7.11 veranschaulicht werden. Der dunkelgraue Bereich entspricht der Schnittmenge von *s* und *t*.

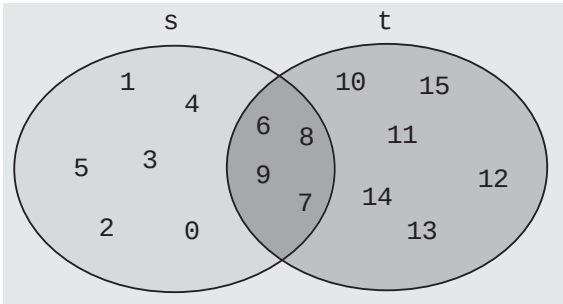


Abbildung 7.11 Die Ausgangssituation

Anzahl der Elemente

Um die Anzahl der Elemente zu bestimmen, die in einer Menge enthalten sind, wird – wie schon bei den sequentiellen Datentypen und dem Dictionary – die eingebaute Funktion `len` verwendet:

```
>>> len(s)
10
```

Ist ein Element im Set enthalten?

Zum Test, ob ein Element in einer Menge 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 einer Menge um eine Teilmenge einer anderen Menge 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 zusammenzuführen, bietet Python den Operator `|`. Er erzeugt eine neue Menge, die 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 7.12 veranschaulicht dies.

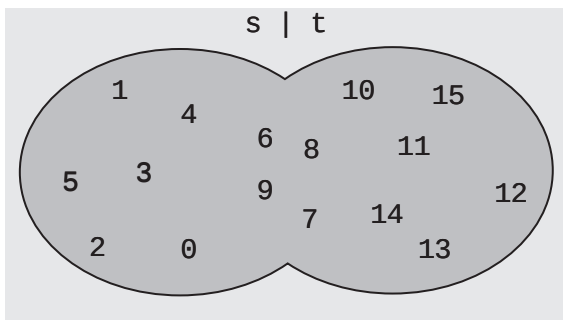


Abbildung 7.12 Vereinigungsmenge von `s` und `t`

Für den Vereinigungsoperator wird dasselbe Symbol verwendet, wie für das bitweise ODER zweier ganzer Zahlen. Diese Wahl ist darin begründet, dass die Vereinigung der

Mengen s und t alle Elemente enthält, die in s ODER in t enthalten sind. Beachten Sie, dass es sich hierbei, wie bei dem bitweisen Operator, um ein nicht ausschließendes ODER handelt, die Vereinigungsmenge also auch die Elemente enthält, die sowohl in s , als auch in t vorkommen.

Schnittmenge

Um die Schnittmenge zweier Mengen zu bestimmen, wird der Operator $\&$ verwendet. Er erzeugt eine neue Menge, die 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 7.13):

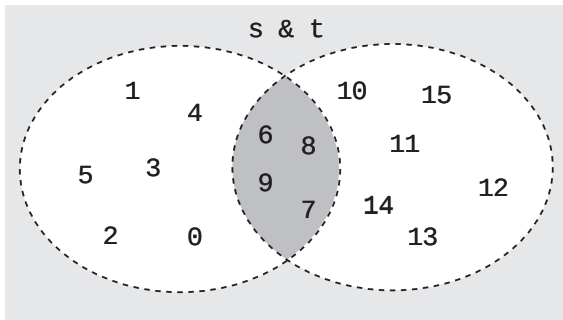


Abbildung 7.13 Schnittmenge von s und t

Die Schnittmenge zweier Mengen s und t enthält alle Elemente, die in s UND t vorkommen. Daraus erschließt sich die Wahl des Operatorsymbols $\&$, das auch für das bitweise UND verwendet wird.

Differenz zweier Mengen

Um die Differenz zweier Mengen zu bestimmen, wird der Operator $-$ verwendet. Es wird eine neue Menge gebildet, die 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 7.14 dargestellt.

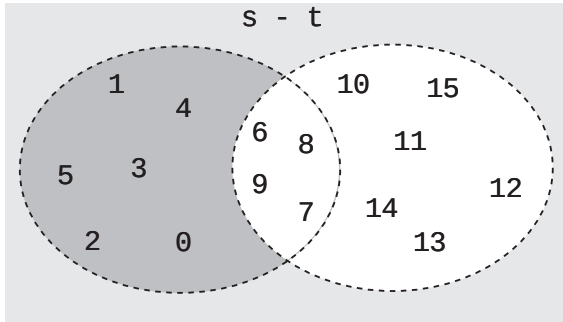


Abbildung 7.14 Differenz von s und t

Symmetrische Differenz zweier Mengen

Um die symmetrische Differenz zweier Mengen zu bestimmen, nutzen Sie den Operator $\wedge$. Er erzeugt eine neue Menge, die 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 7.15:

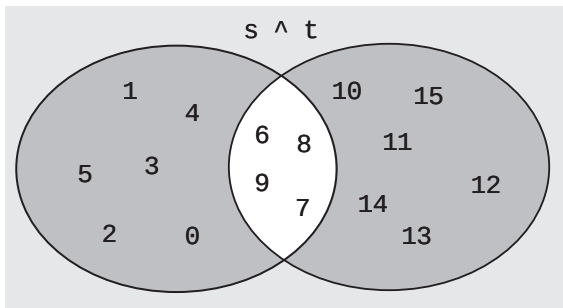


Abbildung 7.15 Symmetrische Differenz von s und t

Die symmetrische Differenzmenge zweier Mengen s und t enthält diejenigen Elemente, die entweder in s oder in t (nicht aber in beiden Mengen) enthalten sind. Daraus erschließt sich die Wahl des Operatorsymbols $\wedge$, das auch für das bitweise ausschließende ODER (XOR) verwendet wird.

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.

Methode	Beschreibung
<i>s.issubset(t)</i>	Äquivalent zu $s \leq t$
<i>s.issuperset(t)</i>	Äquivalent zu $s \geq t$
<i>s.isdisjoint(t)</i>	Prüft, ob die Mengen <i>s</i> und <i>t</i> disjunkt sind, das heißt, ob sie eine leere Schnittmenge haben.
<i>s.union(t)</i>	Äquivalent zu $s \cup t$
<i>s.intersection(t)</i>	Äquivalent zu $s \cap t$
<i>s.difference(t)</i>	Äquivalent zu $s - t$
<i>s.symmetric_difference(t)</i>	Äquivalent zu $s \Delta t$
<i>s.copy()</i>	Erzeugt eine Kopie des Sets <i>s</i> .

Tabelle 7.44 Methoden der Datentypen set und frozenset

Der Unterschied zwischen den Operatoren $\leq$, $\geq$, $\cup$, $\cap$, $-$ und Δ und den jeweils bedeutungsgleichen Methoden `issubset`, `issuperset`, `union`, `intersection`, `difference` und `symmetric_difference` ist, dass bei Verwendung der Operatoren beide Operanden `set`- oder `frozenset`-Instanzen sein müssen, während die Methoden für den Parameter *t* einen beliebigen sequentiellen Datentyp akzeptieren:

```
>>> {1,2,3} | frozenset([4,5,6])
{1, 2, 3, 4, 5, 6}
>>> {1,2,3} | "ABC"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for |: 'set' and 'str'
>>> {1,2,3}.union("ABC")
{'A', 1, 2, 3, 'C', 'B'}
```

Abgesehen von diesem Unterschied, verhalten sich die Methoden wie die zugehörigen Operatoren und benötigen deshalb keine weiteren Erklärungen.

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 die Menge 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 7.5.2, »Listen – list«, geläufig.

7

7.7.2 Veränderliche 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.add(e)</code>	Fügt das Objekt <i>e</i> als Element in die Menge <i>s</i> ein.
<code>s.clear()</code>	Löscht alle Elemente der Menge <i>s</i> , jedoch nicht die Menge selbst.
<code>s.difference_update(t)</code>	Äquivalent zu $s -= t$
<code>s.discard(e)</code>	Löscht das Element <i>e</i> aus der Menge <i>s</i> . Sollte <i>e</i> nicht vorhanden sein, wird dies ignoriert.
<code>s.intersection_update(t)</code>	Äquivalent zu $s \&= t$
<code>s.remove(e)</code>	Löscht das Element <i>e</i> aus der Menge <i>s</i> . Sollte <i>e</i> nicht vorhanden sein, wird eine Exception erzeugt.
<code>s.symmetric_difference_update(t)</code>	Äquivalent zu $s \wedge= t$
<code>s.update(t)</code>	Äquivalent zu $s = t$

Tabelle 7.45 Methoden des Datentyps `set`

Diese Methoden möchten wir nachfolgend anhand einiger Beispiele erläutern.

s.add(e)

Die Methode `add` fügt ein Element *e* in die Menge *s* ein:

```
>>> s = {1,2,3,4,5}
>>> s.add(6)
>>> s
{1, 2, 3, 4, 5, 6}
```

Sollte e bereits in der Menge vorhanden sein, so verändert sich diese nicht.

s.clear()

Die Methode *clear* entfernt alle Elemente aus der Menge s . Die Menge selbst bleibt nach dem Aufruf von *clear* weiterhin erhalten:

```
>>> s = {1,2,3,4,5}
>>> s.clear()
>>> s
set()
```

s.discard(e)

Die Methode *discard* löscht ein Element e aus der Menge s . Der einzige Unterschied zur Methode *remove* besteht darin, dass keine Fehlermeldung erzeugt wird, wenn e nicht in s vorhanden ist:

```
>>> s = {1,2,3,4,5}
>>> s.discard(5)
>>> s
{1, 2, 3, 4}
>>> s.discard(17)
>>> s
{1, 2, 3, 4}
```

s.remove(e)

Die Methode *remove* löscht das Element e aus der Menge s :

```
>>> s = {1,2,3,4,5}
>>> s.remove(5)
>>> s
{1, 2, 3, 4}
```

Sollte das zu löschende Element nicht in der Menge vorhanden sein, so wird eine Fehlermeldung erzeugt:

```
>>> s.remove(17)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 17
```

7.7.3 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 instanziiert werden kann. Die Instanziierung eines `frozenset` 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.³⁷ Innerhalb eines Dictionarys kann also ein `frozenset` sowohl als

³⁷ Auch hier aufgrund des zu berechnenden Hash-Wertes. In Abschnitt 11.5.1 auf Seite 305 erfahren Sie, wie Sie die Hash-Berechnung für eigene Datentypen implementieren können.

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 einen weiteren Weg kennen lernen, mit dem Benutzer bzw. dem System zu interagieren: das Lesen und Schreiben von Dateien.

Kapitel 8

Dateien

»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

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 gehört zum Standardrepertoire eines jeden Programmierers – 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.

8.1 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.

Abbildung 8.1 veranschaulicht das Konzept der Datenströme anhand eines Beispiels, in dem ein Programm Daten von einem eingehenden Strom empfängt, aus diesen ein Ergebnis berechnet und das Ergebnis auf einem ausgehenden Strom ausgibt. Sie können sich beispielsweise vorstellen, dass das Programm eine Datei ausliest und jedes zweite in der Datei enthaltene Wort auf dem Bildschirm ausgibt.

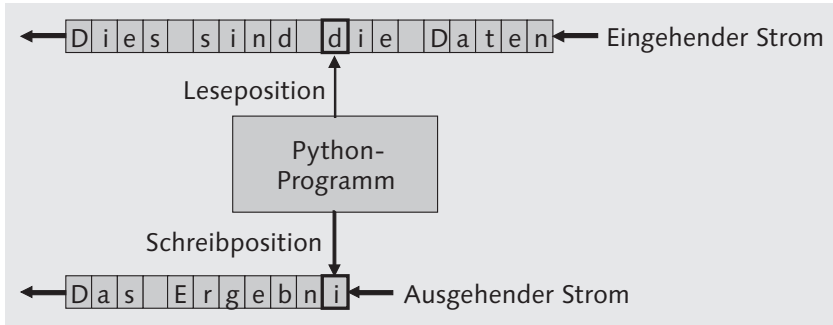


Abbildung 8.1 Ein Programm liest Daten von einem eingehenden Datenstrom und schreibt in einen ausgehenden Strom.

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`. Auf den Ausgabestrom `stdout` kann mit der eingebauten Funktion `print` geschrieben und von dem Eingabestrom `stdin` mittels `input` gelesen werden.

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, ist der Standardeingabestrom (`stdin`) oder eine Netzwerkverbindung.

8.2 Daten aus einer Datei auslesen

Zunächst besprechen wir, wie Daten aus einer Datei ausgelesen werden können. 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 möchten wir die Daten in dieser Datei 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* (engl. *file object*) zurück:

```
fobj = open("woerterbuch.txt", "r")
```

Als ersten Parameter von `open` übergeben wir einen String, der den Pfad zur gewünschten Datei enthält. Beachten Sie, dass hier sowohl relative als auch absolute Pfade erlaubt sind.¹ Der zweite Parameter ist ebenfalls ein String und spezifiziert den Modus, in dem die Datei geöffnet werden soll, wobei "r" für »read« steht und bedeutet, dass die Datei zum Lesen geöffnet wird. Das von der Funktion zurückgegebene Dateiobjekt 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'
```

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()
```

¹ Ein absoluter Pfad identifiziert eine Datei ausgehend von der Wurzel im Dateisystembaum. Unter Windows könnte ein absoluter Pfad folgendermaßen aussehen:

```
C:\Programme\TestProgramm\woerterbuch.txt
```

Ein relativer Pfad bezieht sich auf das aktuelle Arbeitsverzeichnis des Programms. Hier kann die Verknüpfung »..« für das übergeordnete Verzeichnis verwendet werden. Im Beispiel ist ein relativer Pfad angegeben, die Datei *woerterbuch.txt* muss sich also im gleichen Verzeichnis befinden wie das Programm.

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 Dateiojekts 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-Zeichen², 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, Übersetzungsanfragen 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.

² Whitespaces sind Zeichen, die am Bildschirm typischerweise nicht dargestellt werden. Beispiele für Whitespaces sind Leerzeichen, Tabulatorzeichen oder Zeilenumbrüche. Näheres zu Whitespaces erfahren Sie in Abschnitt 7.5.4 (S. 146)

```

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 Dictionarys 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 12.7 die `with`-Anweisung kennen lernen, 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 Variante Ihnen besser gefällt.

8.3 Daten in eine Datei schreiben

Im letzten Abschnitt haben wir uns dem Lesen von Dateien gewidmet. Dass es auch andersherum geht, soll jetzt 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. Eine nicht vorhandene Datei wird erstellt.

```
fobj = open("ausgabe.txt", "w")
```

Nachdem alle Daten in die Datei geschrieben wurden, muss das Dateiojekt durch Aufruf der Methode `close` geschlossen werden:

```
fobj.close()
```

Das Schreiben eines Strings in die geöffnete Datei erfolgt durch Aufruf der Methode `write` des Dateiojekts. Das folgende Beispielprogramm versteht sich als Gegenstück zu dem Beispiel 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("{0} {1}\n".format(engl, woerter[engl]))
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 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.



8.4 Verwendung des Dateiobjekts

Das Dateiobjekt besitzt, wie beispielsweise die komplexeren Basisdatentypen auch, Methoden und Attribute. Einige von ihnen haben wir in den beiden vorherigen Abschnitten bereits besprochen. Wir möchten die auf das Dateiobjekt bezogenen 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 Dateiobjekt zurück. Mithilfe dieses Dateiobjekts können Sie nachher die gewünschten Operationen auf 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.
"rb", "wb", "ab", "r+b", "w+b", "a+b"	Die Datei wird im Binärmodus geöffnet. Beachten Sie, dass in diesem Fall bytes-Instanzen statt Strings verwendet werden müssen (b steht für »binary«).

Tabelle 8.1 Dateimodi

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, was auch das Verhalten ist, wenn der Parameter nicht angegeben wird.³

Der 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.

Methode	Beschreibung
<i>close()</i>	Schließt ein bestehendes Dateiobjekt. Beachten Sie, dass danach keine Lese- oder Schreiboperationen mehr durchgeführt werden dürfen.
<i>fileno()</i>	Gibt den Deskriptor ⁴ der geöffneten Datei als ganze Zahl zurück.
<i>flush()</i>	Verfügt, dass anstehende Schreiboperationen sofort ausgeführt werden. ⁵
<i>isatty()</i>	<code>True</code> , wenn das Dateiobjekt auf einem Datenstrom geöffnet wurde, der nicht an beliebiger Stelle geschrieben oder gelesen werden kann
<i>next()</i>	Liest die nächste Zeile der Datei ein und gibt sie als String zurück.

Tabelle 8.2 Attribute eines Dateiobjekts

3 Näheres zu den Parametern *encoding* und *errors* erfahren Sie im Abschnitt 7.5.4 über Strings im Teil über Codecs.

4 Ein Dateideskriptor (engl. *file descriptor*) ist eine vom Betriebssystem vergebene Identifikationsnummer für geöffnete Dateien. Die Standardströme `stdin` und `stdout` haben die Deskriptoren 0 bzw. 1.

5 Das Betriebssystem kann anstehende Dateioperationen aus Effizienzgründen puffern und zu einem späteren Zeitpunkt ausführen. Dies ist zum Beispiel der Grund dafür, dass USB-Sticks nicht herausgezogen werden sollten, ohne sie beim Betriebssystem abzumelden.

Methode	Beschreibung
<code>read([size])</code>	Liest <i>size</i> Bytes der Datei ein, oder weniger, wenn vorher das Ende der Datei erreicht wurde. Sollte <i>size</i> nicht angegeben sein, so wird die Datei vollständig eingelesen. Die Daten werden als String zurückgegeben.
<code>readline([size])</code>	Liest eine Zeile der Datei ein. Durch Angabe von <i>size</i> lässt sich die Anzahl der zu lesenden Bytes begrenzen.
<code>readlines([sizehint])</code>	Liest alle Zeilen und gibt sie in Form einer Liste von Strings zurück. Sollte <i>sizehint</i> angegeben sein, so wird nur gelesen, bis ungefähr <i>sizehint</i> Bytes gelesen wurden. ⁶
<code>seek(offset[, whence])</code>	Setzt die aktuelle Schreib-/Leseposition in der Datei auf <i>offset</i> . Eine ausführliche Beschreibung von <i>seek</i> finden Sie am Ende des Kapitels auf Seite 207.
<code>tell()</code>	Liefert die aktuelle Schreib-/Leseposition in der Datei.
<code>truncate([size])</code>	Löscht in der Datei alle Daten, die hinter der aktuellen Schreib-/Leseposition stehen, bzw. – sofern angegeben – alles außer den ersten <i>size</i> Bytes.
<code>write(str)</code>	Schreibt den String <i>str</i> in die Datei.
<code>writelines(iterable)</code>	Schreibt mehrere Zeilen in die Datei. Das iterierbare Objekt <i>iterable</i> muss Strings durchlaufen, möglich ist zum Beispiel eine Liste von Strings.

Tabelle 8.2 Attribute eines Dateiobjekts (Forts.)

Darüber hinaus enthält das Dateiobjekt folgende Attribute:

Attribut	Beschreibung
<code>closed</code>	True, wenn die Datei geschlossen ist, andernfalls False
<code>encoding</code>	Enthält das Encoding, das genutzt wird, um eine Datei im Textmodus zu schreiben bzw. zu lesen. Ein Wert von None bedeutet, dass der Systemdefault verwendet wird.

Tabelle 8.3 Methoden eines Dateiobjekts

⁶ In diesem Zusammenhang bedeutet »ungefähr«, dass die Anzahl der zu lesenden Bytes möglicherweise zu einer internen Puffergröße aufgerundet wird.

Attribut	Beschreibung
<i>errors</i>	Beschreibt das Verhalten des Dateiobjekts bei einem Encoding-Fehler. Dabei sind dieselben Werte wie beim Parameter <i>errors</i> der Funktion <i>open</i> möglich.
<i>mode</i>	Enthält den Modus, der beim Öffnen der Datei angegeben wurde.
<i>name</i>	Enthält den Namen der geöffneten Datei.
<i>newlines</i>	Dieses Attribut enthält alle Typen von Newline-Zeichen, die beim Lesen der Datei bisher vorgekommen sind, da diese von System zu System verschieden sind.

Tabelle 8.3 Methoden eines Dateiobjekts (Forts.)

Viele der oben beschriebenen Methoden sind durch vorangegangene Beispiele oder den erklärenden Text ausreichend beschrieben. Wir möchten uns trotzdem noch einmal mit der Methode `seek` befassen:

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+").⁷

Sollte die Datei im Binärmodus geöffnet worden sein, wird der Parameter *offset* in Bytes vom Dateianfang aus gezählt. Diese Interpretation von *offset* lässt sich durch den optionalen Parameter *whence* beeinflussen:

Wert von whence	Interpretation von offset
0	Anzahl Bytes relativ zum Dateianfang
1	Anzahl Bytes relativ zur aktuellen Schreib-/Leseposition
2	Anzahl Bytes relativ zum Dateiende

Tabelle 8.4 Der Parameter whence

Sie können `seek` nicht so unbeschwert verwenden, 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.

⁷ Im Modus "a+" können mittels `seek` Leseoperationen an beliebigen Stellen in der Datei durchgeführt werden. Bevor Schreiboperationen durchgeführt werden, wird die Schreib-/Leseposition jedoch wieder auf das Dateiende gesetzt.

Im folgenden Beispiel wird die Methode *seek* verwendet, um Breite, Höhe und Farbtiefe einer Bitmap-Grafik⁸ zu bestimmen.

```
def bytes2int(b):
    res = 0
    for x in b[::-1]:
        res = (res << 8) + x
    return res

f = open("cover.bmp", "rb")
f.seek(18)
print("Breite:", bytes2int(f.read(4)), "px")
print("Höhe:", bytes2int(f.read(4)), "px")

f.seek(2, 1)
print("Farbtiefe:", bytes2int(f.read(2)), "bpp")
f.close()
```

Der Spezifikation des Bitmap-Dateiformats⁹ kann man entnehmen, dass sich die gesuchten Informationen an den Offsets 18, 22 und 28, jeweils in Form von 4- bzw. 2-Byte-Werten, befinden. Wir öffnen daher die Datei *bild.bmp* zum Lesen im Binärmodus und überspringen mithilfe der Methode *seek* die ersten 18 Bytes. An dieser Stelle können wir Breite und Höhe der Grafik auslesen und ausgeben.¹⁰

Von der aktuellen Leseposition aus überspringen wir zwei weitere Bytes (der Parameter *whence* ist beim *seek*-Aufruf auf 1 gesetzt) und können zwei Bytes auslesen, die die Farbtiefe des Bildes enthalten.¹¹

Die mittels *read* eingelesenen Werte werden als bytes-String zurückgegeben und müssen daher noch mithilfe von *bytes2int* in ganze Zahlen konvertiert werden. Die Funktion *bytes2int* durchläuft den übergebenen bytes-String rückwärts, also mit dem höchstwertigen Byte zuerst. Das aktuelle Zwischenergebnis wird immer binär um acht Stellen nach links geschoben und dann das nächstniedere Byte aufaddiert.

8 Das Windows-Bitmap-Dateiformat (Endung .bmp) ist ein weit verbreitetes Rastergrafikformat.

9 Diese lässt sich beispielsweise in der Wikipedia finden: http://de.wikipedia.org/wiki/Windows_Bitmap

10 Sollte bei Ihnen eine negative Höhe ausgegeben werden, so ist dies kein Fehler im Programm, sondern eine Eigenart des Bitmap-Dateiformats: Eine negative Höheninformation bedeutet, dass die Bilddaten von oben nach unten gespeichert sind, statt, wie üblich, von unten nach oben.

11 Die Farbtiefe wird in »Bits per Pixel« (BPP) angegeben.

Kapitel 9

Funktionen

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

9

Aus der Mathematik kennen Sie den Begriff der *Funktion*, mit dem eine Zuordnungsvorschrift bezeichnet wird. Die Funktion $f(x) = x^2$ ordnet beispielsweise dem Parameter x sein Quadriertes zu. Eine Funktion im mathematischen Sinne besteht aus einem Namen, einer Liste von Parametern und einer Berechnungsvorschrift für den Funktionswert.

In der Programmierung findet sich das mathematische Konzept der Funktion wieder. Wir haben beispielsweise bereits die eingebaute Funktion `len` besprochen, die die Länge eines iterierbaren Objekts berechnet. Dazu bekommt sie das entsprechende Objekt als Parameter übergeben und gibt das Ergebnis in Form eines Rückgabewertes zurück.

```
>>> len("Dieser String ist ein Parameter")  
31
```

Offensichtlich besteht hier eine gewisse Analogie zum mathematischen Begriff der Funktion. Eine Funktion in der Programmierung besteht aus einem *Funktionsnamen*, einer Liste von *Funktionsparametern* und einem Codeblock, dem *Funktionskörper*. Bei einem Funktionsaufruf wird dann der Funktionskörper unter Berücksichtigung der übergebenen Parameter ausgeführt. Eine Funktion in Python kann, wie `len`, einen *Rückgabewert* haben oder nicht.¹

Funktionen werden in der Programmierung dazu eingesetzt, um Redundanzen im Quellcode zu vermeiden. Das bedeutet, dass Codestücke, die in der gleichen oder einer ähnlichen Form öfter im Programm benötigt werden, nicht jedes Mal neu geschrieben, sondern in einer Funktion gekapselt werden. Diese Funktion kann dann an den Stellen, an denen sie benötigt wird, aufgerufen werden. Darüber hinaus bilden Funktionen ein elegantes Hilfsmittel, um einen langen Quellcode sinnvoll in Unterprogramme aufzuteilen. Das erhöht die Les- und Wartbarkeit des Codes.

¹ In Python wird nicht, wie beispielsweise in PASCAL, zwischen den Begriffen Funktion und Prozedur unterschieden. Unter einer Prozedur versteht man eine Funktion, die keinen Rückgabewert hat.

Im Folgenden wird die Handhabung einer bestehenden Funktion am Beispiel von `range` erläutert. Vieles des hier Gesagten kennen Sie bereits aus dem Abschnitt 3.7 (S. 47), wir möchten es an dieser Stelle trotzdem noch einmal wiederholen.

Die eingebaute Funktion `range` wurde in Abschnitt 5.2.6 eingeführt und erzeugt ein iterierbares Objekt über eine begrenzte Anzahl fortlaufender ganzer Zahlen:

```
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 damit befassen, wie Sie eine eigene Funktion erstellen.

9.1 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 ist:

- ▶ 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.²
- ▶ 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(parameter_1, ..., parameter_n):
    | Anweisung
    | :
    | Anweisung
```

Abbildung 9.1 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.

² Das bedeutet, dass sich der Funktionsname aus großen und kleinen Buchstaben, Zahlen sowie dem Unterstrich (`»_«`) zusammensetzen, allerdings nicht mit einer Zahl beginnen darf. Seit Python 3.0 dürfen auch Buchstaben verwendet werden, die nicht im englischen Alphabet enthalten sind.

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 gut nachvollziehen, wie der Parameter `zahl` im Funktionskörper verarbeitet wird. Nachdem die Berechnung erfolgt ist, wird `ergebnis` mittels `print` ausgegeben. Die Referenz `zahl` ist nur innerhalb des Funktionskörpers definiert und hat nichts mit anderen Referenzen außerhalb der Funktion zu tun.

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 mithilfe der soeben definierten Funktion:

```
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 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 die Funktion `fak` mit der eingegebenen Zahl als Parameter aufgerufen wird.

Betrachten Sie noch einmal die beiden Komponenten des Programms. Es wäre im Sinne der Kapselung der Funktionalität 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 in dieses Konzept, dass `fak` das Ergebnis der Berechnung selbst ausgibt.

Idealerweise sollte unsere Funktion `fak` die Berechnung abschließen und das Ergebnis an das Hauptprogramm zurückgeben, sodass die Ausgabe dort erfolgen kann. Dies 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, was nicht unbedingt nötig ist. Der folgende Code ist 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 Betrages einer Zahl gibt es in Python auch die Built-in Funktion `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
```

Durch die Definition einer Funktion wird ein sogenanntes *Funktionsobjekt* erzeugt, das über den Funktionsnamen referenziert wird.

9.2 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 können.

9.2.1 Optionale Parameter

Zu Beginn dieses Kapitels wurde die Verwendung einer Funktion anhand der Built-in Function `range` erklärt. Sicherlich wissen Sie aus Abschnitt 5.2.5 (S. 70) über die `for`-Schleife noch, dass der letzte der drei Parameter der `range`-Funktion optional ist. 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 überflüssigen Parameter anzugeben. Deswegen sind vordefinierte Parameterwerte oft eine sinnvolle Ergänzung einer 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 Funktions-

aufruf nicht explizit angegeben werden, so fließen sie nicht in die Addition ein. Die Funktion kann folgendermaßen aufgerufen werden:

```
>>> summe(1, 2)
3
>>> summe(1, 2, 3)
6
>>> summe(1, 2, 3, 4)
10
```

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 zugeordnet werden können.

9.2.2 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 um eine alternative 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 Arguments übergeben wurden.

Zum Schluss möchten wir noch anmerken, dass optionale Parameter auch unter Verwendung von Keyword Arguments wie erwartet funktionieren.

9.2.3 Beliebige Anzahl von Parametern

Rufen Sie sich noch einmal die Verwendung der eingebauten Funktion `print` in Erinnerung:

```
>>> print("P")
P
>>> print("P", "y", "t", "h", "o", "n")
P y t h o n
>>> print("P", "y", "t", "h", "o", "n", " ", "i", "s", "t", " ",
... "s", "u", "p", "e", "r")
P y t h o n   i s t   s u p e r
```

Offensichtlich ist es möglich, der Funktion `print` eine beliebige Anzahl von Parametern zu übergeben. Diese Eigenschaft ist nicht exklusiv für die `print`-Funktion, sondern es können auch eigene Funktionen definiert werden, denen beliebig viele Parameter übergeben werden können.

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 ist:

```
Feste Parameter: 1 2
Weitere Parameter: ()
```

Der Parameter `weitere` referenziert also ein leeres Tupel. Im Falle des zweiten Aufrufs sieht 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, in der sie übergeben wurden.

An dieser Stelle möchten wir die im vorangegangenen Beispiel definierte Funktion `summe` dahingehend erweitern, dass sie die Summe einer vom Benutzer festgelegten Zahl von Parametern berechnen kann:

```
def summe(*parameter):
    s = 0
    for p in parameter:
        s += p
    return s
```

Das folgende Beispiel demonstriert die Anwendung der weiterentwickelten Funktion `summe` im interaktiven Modus:

```
>>> summe(1, 2, 3, 4, 5)
15
>>> summe(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
78
```

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 sieht folgendermaßen aus:

```
Feste Parameter: 1 2
Weitere Parameter: {}
```

Der Parameter `weitere` referenziert also ein leeres Dictionary. Nach dem zweiten Aufruf sieht 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:", positional)
    print("Keyword:", keyword)
```

Der Funktionsaufruf

```
funktion(1, 2, 3, 4, hallo="welt", key="word")
```

gibt diese Werte aus:

```
Positional: (1, 2, 3, 4)
Keyword: {'hallo': 'welt', 'key': 'word'}
```

Sie sehen, dass `positional` ein Tupel mit allen Positions- und `keyword` ein Dictionary mit allen Schlüsselwortparametern referenziert.

9.2.4 Reine Schlüsselwortparameter

Es ist möglich, Parameter zu definieren, die ausschließlich in Form von Schlüsselwortparametern übergeben werden dürfen. Solche reinen Schlüsselwortparameter³ werden bei der Funktionsdefinition nach dem Parameter geschrieben, der beliebig viele Positionsargumente aufnimmt:

³ Engl. *keyword-only parameters*


```
def f(a, b, *c, d, e):
    print(a, b, c, d, e)
```

In diesem Fall besteht die Funktionsschnittstelle aus den beiden Positionsparametern `a` und `b`, der Möglichkeit für weitere Positionsparameter `*c` und den beiden reinen Schlüsselwortparametern `d` und `e`. Es gibt keine Möglichkeit, die Parameter `d` und `e` zu übergeben, außer in Form von Schlüsselwortparametern.

```
>>> f(1, 2, 3, 4, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() needs keyword-only argument d
>>> f(1, 2, 3, 4, 5, d=4, e=5)
1 2 (3, 4, 5) 4 5
```

Wie bei Positionsparametern müssen reine Schlüsselwortparameter angegeben werden, sofern sie nicht mit einem Defaultwert belegt sind:

```
>>> def f(a, b, *c, d=4, e=5):
...     print(a, b, c, d, e)
...
>>> f(1, 2, 3, 4, 5)
1 2 (3, 4, 5) 4 5
```

Wenn zusätzlich das Übergeben beliebig vieler Schlüsselwortparameter ermöglicht werden soll, so folgt die dazu notwendige `**`-Notation nach den reinen Schlüsselwortparametern am Ende der Funktionsdefinition:

```
def f(a, b, *args, d, e, **kwargs):
    print(a, b, args, d, e, kwargs)
```

Es ist auch möglich, reine Schlüsselwortparameter zu definieren, ohne gleichzeitig beliebig viele Positionsparameter zuzulassen. Dazu werden die reinen Schlüsselwortparameter in der Funktionsschnittstelle durch einen `*` von den Positionsparametern getrennt.

```
>>> def f(a, b, *, c, d):
...     print(a, b, c, d)
...
>>> f(1, 2, 3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```

TypeError: f() takes exactly 2 positional arguments (4 given)
>>> f(1, 2, c=3, d=4)
1 2 3 4

```

9.2.5 Entpacken einer Parameterliste

In diesem Abschnitt wird eine weitere Möglichkeit besprochen, Parameter an eine Funktion zu übergeben. Dazu stellen wir uns vor, wir wollten mithilfe der in Abschnitt 9.2.3 definierten erweiterten Version der `summe`-Funktion die Summe aller Einträge eines Tupels bestimmen. Dazu ist momentan die folgende Notation nötig:

```

>>> t = (1, 4, 3, 7, 9, 2)
>>> summe(t[0], t[1], t[2], t[3], t[4], t[5])
26

```

Das ist sehr umständlich. Zudem laufen wir der Allgemeinheit der Funktion `summe` zuwider, denn die Anzahl der Elemente des Tupels `t` muss stets bekannt sein. Wünschenswert ist ein Weg, eine in einem iterierbaren Objekt gespeicherte Liste von Argumenten direkt einer Funktion übergeben zu können. Dieser Vorgang wird *Entpacken* genannt.

Das Entpacken eines iterierbaren Objekts geschieht dadurch, dass der Funktion das Objekt mit einem vorangestellten Sternchen (*) übergeben wird. Im folgenden Beispiel wird das von der eingebauten Funktion `range` erzeugte iterierbare Objekt verwendet, um mithilfe der Funktion `summe` die Summe der ersten 100 natürlichen Zahlen zu berechnen:⁴

```

>>> summe(*range(101))
5050

```

Beim Funktionsaufruf wird der Funktion jedes Element des iterierbaren Objekts, in diesem Fall also die Zahlen von 0 bis 100, als gesonderter Parameter übergeben. Das Entpacken einer Parameterliste funktioniert nicht nur im Zusammenhang mit einer Funktion, die beliebig viele Parameter erwartet, sondern kann auch mit der ursprünglichen Funktion `summe`, die die Summe von maximal vier Parametern bestimmt, verwendet werden:

```

def summe(a, b, c=0, d=0):
    return a + b + c + d

```

⁴ Das beim Funktionsaufruf von `range(n)` zurückgegebene iterierbare Objekt durchläuft alle ganzen Zahlen von 0 bis einschließlich `n-1`. Daher muss im Beispiel 101 statt 100 übergeben werden.

Dabei ist zu beachten, dass das zu entpackende iterierbare Objekt auch maximal vier (und mindestens zwei) Elemente bereitstellt:

```
>>> t = (6, 3, 9, 12)
>>> summe(*t)
30
>>> summe(*[4, 6, 12, 7, 9])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: summe() takes at most 4 positional arguments (5 given)
```

Analog zum Entpacken eines Tupels zu einer Liste von Positionsparametern kann ein Dictionary zu einer Liste von Schlüsselwortparametern entpackt werden. Der Unterschied in der Notation besteht darin, dass zum Entpacken eines Dictionarys zwei Sternchen vorangestellt werden müssen:

```
>>> d = {"a" : 7, "b" : 3, "c" : 4}
>>> summe(**d)
14
```

Abschließend ist noch zu erwähnen, dass die Techniken zum Entpacken von Parameterlisten miteinander kombiniert werden können, wie folgendes Beispiel zeigt:

```
>>> summe1(1, *(2,3), **{"d" : 4})
10
```

Beachten Sie allgemein, dass die hier vorgestellten Notationen nur innerhalb eines Funktionsaufrufs verwendet werden dürfen und außerhalb dessen zu einem Fehler führen.

9.2.6 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.

In Python werden bei einem Funktionsaufruf keine Kopien der als Parameter übergebenen Instanzen erzeugt, sondern es wird funktionsintern mit Referenzen auf die Argumente gearbeitet.⁵ Betrachten Sie dazu folgendes Beispiel:

⁵ Diese Methode der Parameterübergabe wird *Call by Reference* genannt. Demgegenüber steht das Prinzip *Call by Value*, bei dem funktionsintern auf Kopien der Argumente gearbeitet wird. Letztere Variante ist frei von Seiteneffekten, aber aufgrund des Kopierens langsamer.

```
>>> def f(a, b):
...     print(id(a))
...     print(id(b))
...
>>> p = 1
>>> q = [1,2,3]
>>> id(p)
134537016
>>> id(q)
134537004
>>> f(p, q)
134537016
134537004
```

Im interaktiven Modus definieren wir zuerst eine Funktion `f`, die zwei Parameter `a` und `b` erwartet und deren jeweilige Identität ausgibt. Nachfolgend werden zwei Referenzen `p` und `q` angelegt, die eine ganze Zahl bzw. eine Liste 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. Dabei macht es zunächst einmal keinen Unterschied, ob die referenzierten Objekte Instanzen eines veränderlichen oder unveränderlichen Datentyps sind.

Trotzdem ist die Verwendung eines unveränderlichen 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 führt dazu, dass bei der Verwendung von unveränderlichen Datentypen in Funktionsschnittstellen keine Seiteneffekte auftreten können.⁶

Diese Sicherheit können uns veränderliche Datentypen, wie Listen oder Dictionaries, nicht geben. Dazu folgendes Beispiel:

```
def f(liste):
    liste[0] = 42
    liste += [5,6,7,8,9]

zahlen = [1,2,3,4]
```

⁶ Beachten Sie, dass dies nicht für unveränderliche Instanzen gilt, die veränderliche Instanzen enthalten. So können bei der Parameterübergabe eines Tupels, das eine Liste enthält, durchaus Seiteneffekte auftreten.

```
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. Dieses Phänomen wird Seiteneffekt genannt. 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 kann in Bezug auf das obige Beispiel so aussehen:⁷

```
f(zahlen[:])
```

Neben den bisher besprochenen Referenzparametern existiert eine weitere, seltenere Form von Seiteneffekten, die auftritt, wenn ein veränderlicher Datentyp als Defaultwert eines Parameters verwendet wird:

```
>>> def f(a=[1,2,3]):
...     a += [4,5]
...     print(a)
...
>>> f()
[1, 2, 3, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5, 4, 5]
```

⁷ Sie erinnern sich, dass beim Slicen einer Liste stets eine Kopie derselben erzeugt wird. Im Beispiel wurde das Slicing ohne Angabe von Start- und Endindex verwendet, um eine vollständige Kopie der Liste zu erzeugen.

Wir definieren im interaktiven Modus eine Funktion, die einen einzigen Parameter erwartet, der mit einer Liste vorbelegt ist. Im Funktionskörper wird diese Liste um zwei Elemente vergrößert und ausgegeben. Nach mehrmaligem Aufrufen der Funktion ist zu erkennen, dass es sich bei dem Defaultwert augenscheinlich immer um dieselbe Instanz gehandelt hat.

Das liegt daran, dass eine Instanz, die als Defaultwert genutzt wird, nur einmalig und nicht bei jedem Funktionsaufruf neu erzeugt wird. Grundsätzlich sollten Sie also darauf verzichten, Instanzen veränderlicher 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.

9.3 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.

9.3.1 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. Abbildung 9.2 fasst das Konzept der Namensräume zusammen.

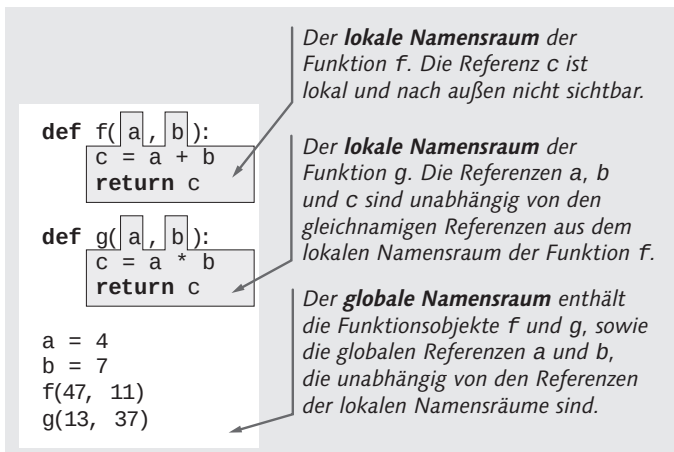


Abbildung 9.2 Abgrenzung lokaler Namensräume vom globalen Namensraum anhand eines Beispiels

9.3.2 Zugriff auf den globalen Namensraum

Im lokalen Namensraum eines 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.

9.3.3 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 9.4, »Lokale Funktionen«, dürfen auch lokale Funktionen innerhalb von Funktionen definiert werden. Lokale Funktio-

nen bringen natürlich wieder ihren eigenen lokalen Namensraum im lokalen Namensraum der übergeordneten Funktion mit. Bei 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 gibt `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` gibt 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.

9.4 Lokale Funktionen

Es ist möglich, sogenannte *lokale Funktionen* zu definieren. Das sind Funktionen, die im lokalen Namensraum einer anderen Funktion angelegt werden und nur dort gültig sind. Das folgende Beispiel zeigt eine solche Funktion:

```

def globale_funktion(n):

    def lokale_funktion(n):
        return n**2

    return lokale_funktion(n)

```

Innerhalb der globalen Funktion `globale_funktion` wurde eine lokale Funktion namens `lokale_funktion` definiert. Beachten Sie, dass der jeweilige Parameter `n` trotz des gleichen Namens nicht zwangsläufig denselben Wert referenziert. Die lokale Funktion kann im Namensraum der globalen Funktion völlig selbstverständlich wie jede andere Funktion auch aufgerufen werden.

Da sie einen eigenen Namensraum besitzt, hat die lokale Funktion keinen Zugriff auf lokale Referenzen der globalen Funktion. Um dennoch einige ausgewählte Referenzen an die lokale Funktion durchzuschleusen, bedient man sich eines Tricks mit vorgelegten 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.

9.5 Anonyme Funktionen

Mithilfe des Schlüsselwortes `lambda` können kleine, anonyme Funktionen erstellt werden. Solche Funktionen werden üblicherweise für häufig auftretende Berechnungen verwendet, um sich alle Vorteile einer echten Funktion zu erhalten, diese gleichzeitig aber nicht aufwendig definieren zu müssen. Eine anonyme Funktion wird zum Beispiel folgendermaßen erzeugt:

```
f = lambda x: x * 3 + 7
```

Auf das Schlüsselwort `lambda` folgen eine Parameterliste und ein Doppelpunkt. Hinter dem Doppelpunkt muss ein beliebiger arithmetischer oder logischer Ausdruck stehen, der nach seiner Auswertung im Rückgabewert der Funktion mündet. Beachten Sie, dass die Beschränkung auf einen arithmetischen Ausdruck zwar die Verwendung von Kontrollstrukturen ausschließt, nicht aber die Verwendung einer Conditional Expression.

Eine `lambda`-Form ergibt ein Funktionsobjekt und kann, wie im Beispiel geschehen, referenziert werden. Der Aufruf der Funktion läuft wie gewohnt ab:

```
r = f(10)
```

Der Rückgabewert wäre in diesem Fall 37. Betrachten wir noch ein etwas komplexeres Beispiel einer anonymen Funktion mit drei Parametern:

```
f = lambda x, y, z: (x - y) * z
```

Jede `lambda`-Form kann ebenso durch eine »echte« Funktion ersetzt werden. Das entsprechende Gegenstück zum obigen Beispiel sieht 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)
```

9.6 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. 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.⁸

9.7 Eingebaute Funktionen

Es war im Laufe des Buches schon oft von sogenannten *eingebauten Funktionen* oder *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 erleichtern. Sie kennen zum Beispiel bereits die Built-in Functions `len` und `range`. Im Folgenden werden alle bisher relevanten Built-in Functions ausführlich beschrieben. Die folgende Tabelle listet alle in diesem Abschnitt besprochenen Built-in Functions auf und gibt eine kurze Erklärung.

⁸ 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.

Built-in Function	Beschreibung	Seite
<code>__import__(name[, globals[, locals[, fromlist[, level]]]])</code>	Importiert das Modul <i>name</i> .	267
<code>abs(x)</code>	Berechnet den Betrag der Zahl <i>x</i> .	235
<code>all(iterable)</code>	Prüft, ob alle Elemente des iterierbaren Objekts <i>iterable</i> den Wert <code>True</code> ergeben.	235
<code>any(iterable)</code>	Prüft, ob mindestens ein Element des iterierbaren Objekts <i>iterable</i> den Wert <code>True</code> ergibt.	235
<code>ascii(object)</code>	Erzeugt einen druckbaren String, der das Objekt <i>object</i> beschreibt. Dabei werden Sonderzeichen maskiert, sodass die Ausgabe nur ASCII-Zeichen enthält.	236
<code>bin(x)</code>	Gibt einen String zurück, der die Ganzzahl <i>x</i> als Binärzahl darstellt.	236
<code>bool([x])</code>	Erzeugt einen booleschen Wert.	236
<code>bytearray([source[, encoding[, errors]]])</code>	Erzeugt eine neue <code>bytearray</code> -Instanz.	236
<code>bytes([source[, encoding[, errors]]])</code>	Erzeugt eine neue <code>bytes</code> -Instanz.	237
<code>chr(i)</code>	Gibt das Zeichen mit dem Unicode-Codepoint <i>i</i> zurück.	238
<code>compile(source, filename, mode[, flags[, dont_inherit[, optimize]]])</code>	Übersetzt einen übergebenen String oder eine Datei in ein ausführbares Objekt.	–
<code>complex([real[, imag]])</code>	Erzeugt eine komplexe Zahl.	238
<code>dict([arg])</code>	Erzeugt ein Dictionary.	239
<code>divmod(a, b)</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>	239
<code>enumerate(iterable[, start])</code>	Gibt einen Aufzählungsiterator für das übergebene iterierbare Objekt zurück.	239

Tabelle 9.1 Built-in Functions, die in diesem Abschnitt besprochen werden

Built-in Function	Beschreibung	Seite
<i>eval(expression[, globals[, locals]])</i>	Wertet den Python-Ausdruck <i>expression</i> aus.	367
<i>exec(object[, globals[, locals]])</i>	Führt einen Python-Code aus.	366
<i>filter(function, iterable)</i>	Ermöglicht es, bestimmte Elemente eines iterierbaren Objekt herauszufiltern.	240
<i>float([x])</i>	Erzeugt eine Gleitkommazahl.	241
<i>format(value[, format_spec])</i>	Formatiert einen Wert <i>value</i> mit der Formatangabe <i>format_spec</i> .	241
<i>frozenset([iterable])</i>	Erzeugt eine unveränderliche Menge.	241
<i>globals()</i>	Gibt ein Dictionary mit allen Referenzen des globalen Namensraums zurück.	242
<i>hash(object)</i>	Gibt den Hash-Wert der Instanz <i>object</i> zurück.	242
<i>help([object])</i>	Startet die eingebaute interaktive Hilfe von Python.	243
<i>hex(x)</i>	Gibt den Hexadezimalwert der ganzen Zahl <i>x</i> in Form eines Strings zurück.	243
<i>id(object)</i>	Gibt die Identität der Instanz <i>object</i> zurück.	243
<i>input([prompt])</i>	Liest einen String von der Tastatur ein.	243
<i>int(x[, base])</i>	Erzeugt eine ganze Zahl.	244
<i>len(s)</i>	Gibt die Länge einer Instanz <i>s</i> zurück.	244
<i>list([iterable])</i>	Erzeugt eine Liste.	245
<i>locals()</i>	Gibt ein Dictionary zurück, das alle Referenzen des lokalen Namensraums enthält.	245
<i>map(function, iterable, ...)</i>	Wendet die Funktion <i>function</i> auf jedes Element der übergebenen iterierbaren Objekte an.	245
<i>max(iterable[, args], *[, key])</i>	Gibt das größte Element von <i>iterable</i> zurück.	247
<i>memoryview(obj)</i>	Erzeugt ein <i>memoryview</i> -Objekt.	–

Tabelle 9.1 Built-in Functions, die in diesem Abschnitt besprochen werden (Forts.)

Built-in Function	Beschreibung	Seite
<i>min(iterable[, args], *, key)</i>	Gibt das kleinste Element von <i>iterable</i> zurück.	248
<i>oct(x)</i>	Gibt den Oktalwert der ganzen Zahl <i>x</i> in Form eines Strings zurück.	248
<i>open(file[, mode[, buffering[, encoding[, errors[, newline[, closefd]]]]]])</i>	Erzeugt ein Dateiojekt.	204
<i>ord(c)</i>	Gibt den Unicode-Code des Zeichens <i>c</i> zurück.	248
<i>pow(x, y[, z])</i>	Führt eine Potenzoperation durch.	248
<i>print([object, ...], *[, sep][, end][, file])</i>	Gibt die übergebenen Objekte auf dem Bildschirm oder in andere Ausgabeströme aus.	248
<i>range([start,]stop[, step])</i>	Erzeugt einen Iterator über eine Zahlenfolge von <i>start</i> bis <i>stop</i> .	249
<i>repr(object)</i>	Gibt eine String-Repräsentation der Instanz <i>object</i> zurück.	250
<i>reversed(seq)</i>	Erzeugt einen Iterator, der das iterierbare Objekt <i>seq</i> rückwärts durchläuft.	251
<i>round(x[, n])</i>	Rundet die Zahl <i>x</i> auf <i>n</i> Nachkommastellen.	251
<i>set([iterable])</i>	Erzeugt eine Menge.	251
<i>sorted(iterable[, key][, reverse])</i>	Sortiert das iterierbare Objekt <i>iterable</i> .	251
<i>str([object[, encoding[, errors]]])</i>	Erzeugt einen String.	252
<i>sum(iterable[, start])</i>	Gibt die Summe aller Elemente des iterierbaren Objekts <i>iterable</i> zurück.	253
<i>tuple([iterable])</i>	Erzeugt ein Tupel.	253
<i>type(object)</i>	Gibt den Datentyp einer Instanz zurück.	254
<i>zip(*iterables)</i>	Fasst mehrere Sequenzen zu Tupeln zusammen, um sie beispielsweise mit einer <code>for</code> -Schleife zu durchlaufen.	254

Tabelle 9.1 Built-in Functions, die in diesem Abschnitt besprochen werden (Forts.)

Beachten Sie, dass es noch weitere eingebaute Funktionen gibt, die an dieser Stelle nicht besprochen werden können, da sie Konzepte der objektorientierten Programmierung voraussetzen. Eine vollständige Übersicht über alle in Python eingebauten Funktionen finden Sie im Anhang dieses Buchs (Abschnitt A.2, S. 969).

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
```

Das übergebene iterierbare Objekt muss nicht zwingend nur `bool`-Instanzen durchlaufen. Instanzen anderer Datentypen werden nach den Regeln aus Abschnitt 7.3.6 (S. 107) in Wahrheitswerte überführt.

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
```


Das übergebene iterierbare Objekt muss nicht zwingend nur `bool`-Instanzen durchlaufen. Instanzen anderer Datentypen werden nach den Regeln aus Abschnitt 7.3.6 (S. 107) in Wahrheitswerte überführt.

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` (S. 250) 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)

Die Funktion `bin` 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])

Hiermit wird eine Instanz des Datentyps `bool` mit dem Wahrheitswert der Instanz `x` erzeugt. Der Wahrheitswert einer Instanz wird nach den in Abschnitt 7.3.6 (S. 107) festgelegten Regeln bestimmt.

Wenn kein Parameter übergeben wurde, gibt die Funktion `bool` den booleschen Wert `False` zurück.

bytearray([source[, encoding[, errors]]])

Die Funktion `bytearray` 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 *source* 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* (S. 252).

Wenn für *source* eine ganze Zahl übergeben wird, wird ein Byte-Array der Länge *source* angelegt und mit Nullen gefüllt.

Wenn für *source* ein iterierbares Objekt, beispielsweise eine Liste, übergeben wird, wird das Byte-Array mit den Elementen gefüllt, über die *source* iteriert. Beachten Sie, dass es sich dabei um ganze Zahlen aus dem Zahlenbereich von 0 bis 255 handeln muss.

Außerdem kann für *source* eine beliebige Instanz eines Datentyps übergeben werden, der das sogenannte Buffer-Protokoll unterstützt. Das sind beispielsweise die Datentypen *bytes* und *bytearray* selbst.

```
>>> bytearray("äü", "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')
```

Näheres zum Datentyp *bytearray* erfahren Sie in Abschnitt 7.5.4 (ab S. 142).

bytes([source[, encoding[, errors]])

Hiermit wird eine Instanz des Datentyps *bytes*⁹ erzeugt, 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 *source*, *encoding* und *errors* werden wie bei der Built-in Function *bytearray* (S. 236) 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("äü", "utf-8")
b'\xc3\xa4\xc3\xb6\xc3\xbc'
```

⁹ Siehe Abschnitt 7.5.4 (S. 142)

chr(i)

Die Funktion *chr* gibt einen String der Länge 1 zurück, der das Zeichen mit dem Unicode-Codepoint *i* enthält:

```
>>> chr(65)
'A'
>>> chr(33)
'!'
>>> chr(8364)
'€'
```

complex([real[, imag]])

Hiermit wird eine Instanz des Datentyps `complex`¹⁰ zur Speicherung einer komplexen Zahl erzeugt. Die erzeugte Instanz hat den komplexen Wert $real + imag \cdot j$. 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.

¹⁰ Siehe Abschnitt 7.3.7 (S. 110)

dict([source])

Hiermit wird eine Instanz des Datentyps `dict`¹¹ erzeugt. 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* ein über Tupel iterierendes Objekt sein, wobei jedes Tupel zwei Elemente enthalten muss: 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 Form `(i, iterable[i])`. Dabei ist *i* ein Schleifenzähler, der bei 0 beginnt. Die Schleife wird beendet, wenn *i* den Wert

¹¹ Siehe Abschnitt 7.6.1 (S. 174)

`len(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 wird für eine Liste `iterable = [1,2,3,4,5]` ausgeführt, so kommt 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, iterable)

Die Funktion *filter* erwartet ein Funktionsobjekt als ersten und ein iterierbares Objekt als zweiten Parameter. Der Parameter *function* muss eine Funktion oder Lambda-Form (siehe Abschnitt 9.5, S. 230) sein, die einen Parameter erwartet und einen booleschen Wert zurückgibt.

Die Funktion *filter* ruft für jedes Element des iterierbaren Objekts *iterable* die Funktion *function* auf und erzeugt ein iterierbares Objekt, das alle Elemente von *list* durchläuft, für die *function* den Wert `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:

```
>>> filterobj = filter(lambda x: x%2 == 0, range(21))
>>> print(list(filterobj))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

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])

Hiermit wird eine Instanz des Datentyps `float`¹² erzeugt. 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])

Die Funktion `format` gibt den Wert `value` gemäß der Formatangabe `format_spec` zurück. 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 Abschnitt 7.5.4 über Stringformatierungen.

frozenset([iterable])

Hiermit wird eine Instanz des Datentyps `frozenset`¹³ zum Speichern einer unveränderlichen Menge erzeugt. 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.

¹² Siehe Abschnitt 7.3.5 (S. 102)

¹³ Siehe Abschnitt 7.7.3 (S. 195)

```
>>> 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)

Die Funktion *hash* berechnet den Hash-Wert der Instanz *object* und gibt ihn zurück. Bei einem Hash-Wert handelt es sich um eine ganze Zahl, die aus Typ und Wert der Instanz erzeugt wird. Ein solcher Wert wird verwendet, um effektiv zwei komplexere Instanzen auf Gleichheit prüfen zu können. So werden beispielsweise die Schlüssel eines Dictionarys intern durch ihre Hash-Werte verwaltet.

```
>>> hash(12345)
12345
>>> hash("Hallo Welt")
-962533610
>>> hash((1,2,3,4))
89902565
```

Beachten Sie den Unterschied zwischen veränderlichen (mutable) und unveränderlichen (immutable) Instanzen. Aus Letzteren kann zwar formal auch ein Hash-Wert errechnet werden, dieser wäre aber nur so lange gültig, wie die Instanz nicht verändert wurde. Aus diesem Grund ist es nicht sinnvoll, Hash-Werte von veränderlichen Instanzen zu berechnen; veränderliche Instanzen sind »unhashable«:

```
>>> hash([1,2,3,4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 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)

Die Funktion *hex* 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
```

Näheres zu Identitäten erfahren Sie in Abschnitt 6.1.3 (S. 80).

input([prompt])

Die Funktion *input* 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 Function `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[, base]])`

Hiermit wird eine Instanz des Datentyps `int`¹⁴ erzeugt. 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 `base` 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)`

Die Funktion `len` gibt die Länge bzw. die Anzahl der Elemente von `s` zurück. Für `s` können Sequenzen, Mappings oder Mengen übergeben werden.

```
>>> len("Hallo Welt")
10
>>> len([1,2,3,4,5])
5
```

¹⁴ Siehe Abschnitt 7.3.4 (S. 96)

list([sequence])

Hiermit wird eine Instanz des Datentyps `list`¹⁵ aus den Elementen von *sequence* erzeugt. 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}
```

Der Aufruf von *locals* im Namensraum des Hauptprogramms ist äquivalent zum Aufruf von *globals*.

map(function, iterable, ...)

Diese Funktion erwartet ein Funktionsobjekt als ersten und ein iterierbares Objekt als zweiten Parameter. Optional können weitere iterierbare Objekte übergeben werden, die aber die gleiche Länge wie das erste haben müssen. Die Funktion *function*

¹⁵ Siehe Abschnitt 7.5.2 (S. 128)

muss genauso viele Parameter erwarten, wie iterierbare Objekte übergeben wurden, und aus den Parametern einen Rückgabewert erzeugen.

Die Funktion *map* ruft *function* für jedes Element von *iterable* auf und gibt ein iterierbares Objekt zurück, das die jeweiligen Rückgabewerte von *function* durchläuft. Sollten mehrere iterierbare Objekte übergeben werden, so werden *function* die jeweils n-ten Elemente dieser Objekte übergeben.

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.

Das letzte Beispiel wird durch Abbildung 9.3 veranschaulicht. Die eingehenden und ausgehenden iterierbaren Objekte sind jeweils senkrecht dargestellt.

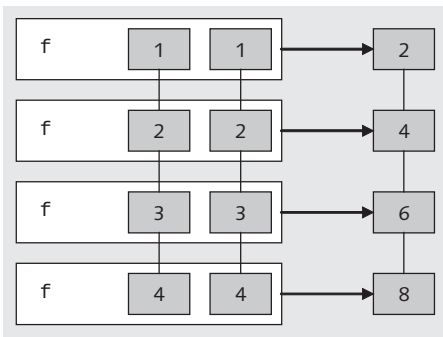


Abbildung 9.3 Arbeitsweise der Built-in Function *map*

In beiden Beispielen wurden Listen verwendet, die ausschließlich numerische Elemente enthielten. Das muss nicht unbedingt sein. Welche Elemente ein an *map* über-

gegebenes iterierbares Objekt durchlaufen darf, hängt davon ab, welche Instanzen für *function* als Parameter verwendet werden dürfen.

max(iterable[, args...][key])

Wenn keine zusätzlichen Parameter übergeben werden, erwartet *max* ein iterierbares Objekt und gibt ihr größtes Element zurück.

```
>>> 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'
```

Der Funktion *max* kann optional über den Schlüsselwortparameter *key* ein Funktionsobjekt übergeben werden. Das Maximum wird dann durch das Vergleichen der Rückgabewerte dieser Funktion bestimmt. Mit dem Parameter *key* lässt sich also eine eigene Ordnungsrelation festlegen. 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(iterable[, 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'
```

ord(c)

Die Funktion *ord* erwartet einen String der Länge 1 und gibt den Unicode-Codepoint des enthaltenen Zeichens zurück.

```
>>> ord("P")
80
>>> ord("€")
8364
```

Näheres zu Unicode und Codepoints erfahren Sie in Abschnitt 7.5.4 (S. 165).

pow(x, y[, z])

Berechnet $x ** y$ oder, wenn *z* angegeben wurde, $x ** y \% z$. Diese Berechnung ist unter Verwendung des Parameters *z* performanter als die Ausdrücke $pow(x, y) \% z$ 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*. Bisher 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 Dateiojekt 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
```

9

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]
```



Hinweis

In Python-Versionen vor 3.0 existieren die eingebauten Funktionen `range` und `xrange`. Die alte `range`-Funktion gibt das Ergebnis in Form einer Liste zurück, während die `xrange`-Funktion so funktioniert wie die `range`-Funktion in aktuellen Python-Versionen.

`repr(object)`

Die Funktion `repr` 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(sequence)

Mit *reversed* kann eine Instanz *sequence* eines sequentiellen Datentyps effizient rückwärts durchlaufen werden.

```
>>> for i in reversed([1, 2, 3, 4, 5, 6]):
...     print(i)
6
5
4
3
2
1
```

round(x[, n])

Die Funktion *round* rundet die Gleitkommazahl *x* auf *n* Nachkommastellen. Der Parameter *n* ist optional und mit 0 vorbelegt.

```
>>> round(-0.5)
-1.0
>>> round(0.5234234234234, 5)
0.52342
>>> round(0.5, 4)
0.5
```

set([iterable])

Hiermit wird eine Instanz des Datentyps *set*¹⁶ erzeugt. Wenn angegeben, werden alle Elemente des iterierbaren Objekts *iterable* in das Set übernommen. Beachten Sie, dass ein Set keine Dubletten enthalten darf, jedes in *iterable* mehrfach vorkommende Element also nur einmal eingetragen wird.

```
>>> set()
set()
>>> set("Hallo Welt")
set({'a', ' ', 'e', 'H', 'l', 'o', 't', 'W'})
>>> set({1,2,3,4})
set({1, 2, 3, 4})
```

sorted(iterable[, key[, reverse]])

Die Funktion *sorted* erzeugt aus den Elementen von *iterable* eine sortierte Liste:

¹⁶ Siehe Abschnitt 7.7.2 (S. 193)


```
>>> sorted([3,1,6,2,9,1,8])
[1, 1, 2, 3, 6, 8, 9]
>>> sorted("Hallo Welt")
[' ', 'H', 'W', 'a', 'e', 'l', 'l', 'o', 't']
```

Die Funktionsweise von *sorted* ist identisch zur Methode *sort* der sequentiellen Datentypen, die im Abschnitt 7.5 (S. 133) erklärt wird.

str([object[, encoding[, errors]]])

Hiermit wird ein String erzeugt,¹⁷ 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 der Parameter *encoding* 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:

¹⁷ Siehe Abschnitt 7.5.4 (S. 142)

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 9.2 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 7.5.4 (S. 142).



`sum(iterable[, start])`

Die Funktion `sum` berechnet die Summe aller Elemente des iterierbaren Objekts *iterable* und gibt das Ergebnis zurück. Wenn der optionale Parameter *start* angegeben wurde, 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([iterable])`

Hiermit wird eine Instanz des Datentyps `tuple`¹⁸ aus den Elementen von *iterable* erzeugt.

¹⁸ Siehe Abschnitt 7.5.3 (S. 139)

```
>>> 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(*iterables)

Die Funktion *zip* nimmt beliebig viele, gleich lange iterierbare Objekte als Parameter. Sollten nicht alle die gleiche Länge haben, werden die längeren nur bis zur Länge des kürzesten dieser Objekte betrachtet.

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')]
```

Bei der bereits besprochenen Funktion *enumerate* handelt es sich um einen Spezialfall der *zip*-Funktion:

```
>>> s = "Python"
>>> list(zip(range(len(s)), s))
[(0, 'P'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

Damit ist der erste Teil der eingebauten Funktionen besprochen. In Kapitel 11 (S. 269) werden die Konzepte der objektorientierten Programmierung besprochen. Dort findet sich in Abschnitt 11.6 eine Beschreibung der Build-in Functions mit objektorientiertem Hintergrund.

Teil II

Fortgeschrittene Programmiertechniken

Kapitel 10

Modularisierung

»Divide et impera!«
– Julius Caesar

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

- Zum einen kann jedes Python-Programm *globale Module* einbinden. Ein globales Modul stellt üblicherweise Datentypen und Funktionen bereit, die einem bestimmten Zweck dienen, beispielsweise der Arbeit mit Dateien eines bestimmten Dateiformats. Globale Module werden systemweit installiert und stehen allen Python-Programmen gleichermaßen zur Verfügung. Es ist möglich, eigene globale Module zu schreiben oder ein globales Modul eines Drittanbieters zu installieren.
- 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, da sie dem Programmierer die Möglichkeit gibt, langen Programmcode überschaubar auf verschiedene Programmdateien aufzuteilen.

In Python besteht der einzige Unterschied zwischen lokalen und globalen Modulen darin, wo sie gespeichert sind. Während sich lokale Module in der Regel im Programmverzeichnis bzw. in einem seiner Unterverzeichnisse befinden, sind globale Module in einigen festgelegten Verzeichnissen der Python-Installation gespeichert.¹

10.1 Einbinden globaler Module

Ein globales Modul, sei es ein Teil der Standardbibliothek oder ein selbstgeschriebenes, kann mithilfe der `import`-Anweisung eingebunden werden. Wir werden in den Beispielen hauptsächlich das Modul `math` der Standardbibliothek verwenden. Das ist

¹ Selbstgeschriebene Bibliotheken können Sie in das Unterverzeichnis *site-packages* der Python-Installation speichern. Dort werden üblicherweise auch Bibliotheken von Drittanbietern installiert.

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, wird die `import`-Anweisung in der folgenden Form verwendet:

```
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 ein Modul eingebunden wurde, wird ein neuer *Namensraum* mit seinem Namen erstellt. Über diesen Namensraum sind alle Funktionen, Datentypen und Werte des Moduls im Programm nutzbar. Mit einem Namensraum kann wie mit einer Instanz umgegangen werden, und die Funktionen des Moduls können wie Methoden des Namensraums verwendet werden. So bindet folgendes Beispielprogramm das Modul `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))
```

Dieser neue Name ist keine zusätzliche Option. Das Modul `math` ist nun ausschließlich über den Namensraum `mathematik` erreichbar.

Des Weiteren kann die `import`-Anweisung so verwendet werden, dass kein eigener Namensraum für das eingebundene Modul erzeugt wird, sondern alle Elemente des Moduls in den globalen Namensraum des Programms eingebunden werden:

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

Wenn die `import`-Anweisung in dieser Weise verwendet wird, sollten Sie beachten, dass keine Referenzen oder Funktionen 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 den Inhalt eines Moduls, zu kapseln und über einen gemeinsamen Namen anzusprechen. Wenn Sie den kompletten Inhalt eines Moduls 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, ein Modul, wenn es 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 beschriebene Form der `from/import`-Anweisung nicht praktikabel. Es ist aber möglich, statt des Sterns eine Liste von zu importierenden Elementen des Moduls 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 globaler Module. Sie werden die Standardbibliothek von Python im dritten Teil dieses Buches noch ausführlich kennen lernen.



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 die einzubindenden Module bei einer normalen `import`-Anweisung nicht in Klammern gesetzt werden dürfen.

10.2 Lokale Module

Nachdem Sie in die `import`-Anweisung eingeführt wurden, möchten wir uns damit beschäftigen, wie lokale Module selbst erstellt und eingebunden werden können. Beachten Sie, dass es sich hier nicht um ein globales Modul handelt, das 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 lokale und globale Module 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 *mathehelfer.py*:

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

def kehr(n):
    return 1 / n
```

Die Funktionen sollten selbsterklärend sein. Die Datei *mathehelfer.py* führt selbst keinerlei Code aus, sondern stellt nur Funktionen bereit, die aus anderen Modulen heraus aufgerufen werden können.

Jetzt erstellen wir eine Programmdatei namens *programm.py*, in der das Hauptprogramm stehen soll. Beide Dateien müssen sich im selben Verzeichnis befinden. Im Hauptprogramm importieren wir zunächst das lokale Modul *mathehelfer*. Der Modulname eines lokalen Moduls entspricht dem Dateinamen der zugehörigen Programmdatei ohne Dateiendung. Der Modulname muss den Regeln der Namensgebung eines Bezeichners folgen. Das bedeutet insbesondere, dass, abgesehen von dem Punkt vor der Dateiendung, kein Punkt im Dateinamen erlaubt ist.

```
import mathehelfer

while True:
    zahl = int(input("Geben Sie eine ganze Zahl ein: "))
    print("Fakultät:", mathehelfer.fak(zahl))
    print("Kehrwert:", mathehelfer.kehr(zahl))
```

Sie sehen, dass Sie das lokale Modul im Hauptprogramm wie ein globales Modul importieren und verwenden können.

Hinweis

Beim Einbinden eines Moduls wird auch unter Windows zwischen Groß- und Kleinschreibung unterschrieben. Ein Modul namens *ABC.py* lässt sich also mit der Anweisung `import abc` nicht einbinden. Stattdessen muss `import ABC` geschrieben werden.



10.2.1 Namenskonflikte

Durch das Erstellen eigener Module kann es leicht zu Namenskonflikten mit globalen Modulen kommen. Beispielsweise hätten wir unsere obige Programmdatei auch *math.py* und das Modul demzufolge *math* nennen können. Dieses Modul stünde im Konflikt mit dem Modul *math* der Standardbibliothek. Für solche Fälle ist dem Interpreter eine Reihenfolge vorgegeben, nach der er zu verfahren hat, wenn ein Modul 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 globale Module ausgeweitet.
- ▶ Wenn auch kein globales Modul 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
```

10.2.2 Modulinterne Referenzen

In jedem Modul existieren Variablen, die Informationen über das Modul selbst enthalten. An dieser Stelle soll ein Überblick über diese 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 10.1 Globale Variablen in einem Modul

[+]

Hinweis

Mithilfe der Referenz `__name__` lässt sich feststellen, ob ein Modul über eine `import`-Anweisung eingebunden oder über einen Interpreteraufruf direkt ausgeführt wurde. In letzterem Fall referenziert `__name__` nicht den Modulnamen, sondern den String `"__main__"`.

10.3 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 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 Pakets einmalig ausgeführt wird. Ein Programm mit mehreren Paketen und Unterpaketen hat also eine ähnliche Verzeichnisstruktur wie in Abbildung 10.1 dargestellt.

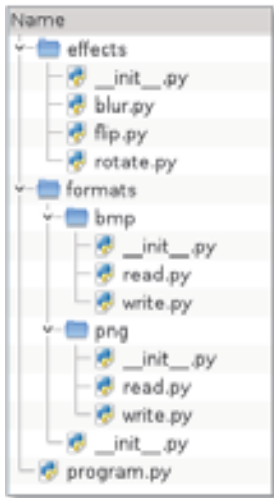


Abbildung 10.1 Paketstruktur eines Beispielprogramms

Es handelt sich um die Verzeichnisstruktur eines fiktiven Bildbearbeitungsprogramms. Das Hauptprogramm befindet sich in der Datei *program.py*. Neben dem Hauptprogramm existieren im Programmverzeichnis zwei Pakete:

- Das Paket *effects* 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 *formats* soll dazu in der Lage sein, bestimmte Grafikformate zu lesen und schreiben. Dazu nehmen wir an, dass in seiner *__init__.py* zwei Funktionen namens *readImage* und *writeImage* definiert sind. Es soll hier nicht näher auf Funktionsschnittstellen oder Ähnliches eingegangen werden. Damit das Lesen und Schreiben von Grafiken diverser Formate möglich ist, enthält das Paket *formats* zwei Unterpakete namens *bmp* und *png*, die je zwei Module zum Lesen bzw. Schreiben des entsprechenden Formats enthalten.

Im Hauptprogramm sollen zunächst die Pakete *effects* und *formats* eingebunden und verwendet werden. Dies ermöglicht die *import*-Anweisung:

```
import effects, formats
```

bzw.:

```
import effects
```

```
import formats
```



Hinweis

Es kommt zu einem Namenskonflikt, wenn neben dem Paket `effects` ein Modul gleichen Namens, also eine Programmdatei namens `effects.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 Pakets 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 `readImage` und `writeImage` zugreifen:

```
formats.readImage()
formats.writeImage()
```

Um das nun geladene Bild zu modifizieren, soll diesmal ein Modul des Pakets `effects` 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 effects.blur
```

In diesem Fall wurde das Paket `effects` vorher eingebunden. Wenn dies nicht der Fall gewesen wäre, so würde das Importieren von `effects.blur` dafür sorgen, dass zunächst das Paket `effects` eingebunden und die dazugehörige `__init__.py` ausgeführt würde. Danach wird das Untermodul `blur` eingebunden. Das Modul kann fortan wie jedes andere verwendet werden:

```
effects.blur.blurImage()
```

Das Verhalten der hier besprochenen Version der `import`-Anweisung verändert sich, wenn man sich in einer Paketstruktur befindet. Dies soll das Thema des nächsten Abschnitts sein.

10.3.1 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. Dafür 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 ...mathematik.konstanten import pi
```

importiert beispielsweise das Objekt `pi` aus dem Modul `konstanten` des Pakets `mathematik`, 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
```

Die klassische `import`-Anweisung, wie sie in den vorangegangenen Abschnitten besprochen wurde, wird auch *absolute import-Anweisung* genannt und kann innerhalb einer Paketstruktur ausschließlich dazu genutzt werden, globale Module einzubinden. In allen anderen Fällen muss eine *relative import-Anweisung* verwendet werden.

Die eingangs besprochenen Möglichkeiten zur Umbenennung eines eingebundenen Pakets oder Moduls funktionieren auch bei relativen `import`-Anweisungen wie erwartet:

```
from . import xyz as bla
```

Diese Anweisung bindet das Modul oder das Paket `xyz` aus dem lokalen Paketverzeichnis unter dem Namen `bla` ein.



Hinweis

Die relative `import`-Anweisung wurde in Python 2.5 eingeführt. Seit Python 3.0 ist es innerhalb einer Paketstruktur nicht mehr möglich, über die einfache `import`-Anweisung Module desselben Pakets einzubinden. Das Verhalten wurde geändert, weil in komplexen Paketen Module der Standardbibliothek oft unabsichtlich durch lokale Module des Pakets verdeckt wurden.

10.3.2 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 Ordernamen 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. Alle in dieser Datei angelegten Referenzen 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.

10.4 Built-in Functions

Es existiert eine Built-in Function, die sich auf Modularisierung, also auf das Einbinden von Modulen und Paketen, bezieht. Diese Funktion wurde in Abschnitt 9.7, »Eingebaute Funktionen« (S. 231), nicht erläutert, da das Konzept der Modularisierung zu diesem Zeitpunkt noch nicht eingeführt war. Aus diesem Grund soll die Beschreibung der Built-in Function `__import__` hier nachgeholt werden. Beachten Sie, dass diese Funktion nur in wenigen Fällen benötigt und daher an dieser Stelle nur oberflächlich erläutert wird.

10

`__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 10.3.1). Der voreingestellte Wert von 0 weist die Funktion `__import__` dazu an, absolutes Importverhalten zu zeigen. Ein Wert größer Null legt die Anzahl der übergeordneten Verzeichnisse fest, 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 der zentralen Themen bei der Arbeit mit Python zu: der objektorientierten Programmierung.

Kapitel 11

Objektorientierung

»Abstraction is selective ignorance.«
– Andrew Koenig

11

In diesem Kapitel lassen wir endlich die Katze aus dem Sack: Sie werden in das wichtigste und grundlegendste 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 dann ein zweites Mal, diesmal aber unter Verwendung der objektorientierten Mechanismen von Python, 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 regelt. Ein möglicher Ansatz ist, dass wir für jedes Bankkonto ein Dictionary anlegen, in dem alle Informationen über den Kunden und seinen Finanzstatus gespeichert sind. Um die gewünschten Operationen zu unterstützen, definieren wir Funktionen. Ein Dictionary für ein vereinfachtes Konto sieht dann folgendermaßen aus:¹

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

¹ Wir verwenden hier `float`-Instanzen zum Speichern von Geldbeträgen, um die Beispiele einfach zu halten. In echten Programmen sollten Sie stattdessen die Centbeträge als `int`-Instanzen speichern, da sonst bei großen Beträgen Informationen verloren gehen können.

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.

Ausgehend von dieser Datenstruktur, werden 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 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 zur Durchführung der Überweisung der gewünschte Geldbetrag bekannt sein. Die Funktion wird also drei Parameter erwarten: *quelle*, *ziel* und *betrag*. Nach unseren Voraussetzungen ist eine Überweisung nur dann möglich, wenn 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, betrag):
    # Hier erfolgt der Test, ob der Transfer möglich ist
    if(betrag < 0 or
       quelle["UmsatzHeute"] + betrag > quelle["MaxTagesumsatz"] or
       ziel["UmsatzHeute"] + betrag > ziel["MaxTagesumsatz"]):
        # Transfer unmöglich
        return False
```

```

else:
    # Alles OK - Auf geht's
    quelle["Kontostand"] -= betrag
    quelle["UmsatzHeute"] += betrag
    ziel["Kontostand"] += betrag
    ziel["UmsatzHeute"] += betrag
    return True

```

Die Funktion überprüft zuerst, ob der Transfer durchführbar ist, und beendet den Funktionsaufruf frühzeitig mit dem Rückgabewert `False`, wenn dies nicht der Fall ist. Wenn für den Betrag ein gültiger Wert übergeben wurde 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 (betrag < 0 or
        konto["UmsatzHeute"] + betrag > konto["MaxTagesumsatz"]):
        # Tageslimit überschritten oder ungültiger Betrag
        return False
    else:
        konto["Kontostand"] += betrag
        konto["UmsatzHeute"] += betrag
        return True

def auszahlen(konto, betrag):
    if (betrag < 0 or
        konto["UmsatzHeute"] + betrag > konto["MaxTagesumsatz"]):
        # Tageslimit überschritten oder ungültiger Betrag
        return False
    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 {0}".format(konto["Inhaber"]))
    print("Aktueller Kontostand: {0:.2f} Euro".format(
        konto["Kontostand"]))
    print("(Heute schon {0:.2f} von {1} Euro 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 Euro umgesetzt)
>>> zeige_konto(k2)
Konto von Erwin Schmidt
Aktueller Kontostand: 14160.00 Euro
(Heute schon 1160.00 von 1500 Euro umgesetzt)
```

Zuerst eröffnet Heinz Meier ein neues Konto `k1` mit der Kontonummer 567123 mit einem 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 Meier einen DVD-Recorder von Herrn Schmidt 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 Schmidt in den Abendstunden stark an der Heimkinoanlage von Herrn Meier 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, die 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.

Genau diese Wünsche befriedigt die Objektorientierung, indem sie Daten und Verarbeitungsfunktionen zu *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 lässt sich das Objekt eines Kontos also folgendermaßen darstellen:

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

Tabelle 11.1 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 das dazugehörige Member durch einen Punkt getrennt aufschreiben.

Angenommen, `k1` und `k2` sind Konto-Objekte, wie sie das obige Schema zeigt, mit den Daten von Herrn Meier und Herrn Schmidt; dann können wir das letzte Beispiel folgendermaßen formulieren (der Code ist so natürlich noch nicht lauffähig, da die Definition für die Konto-Objekte fehlt, die erst im Folgenden erarbeitet wird):

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

Die Methoden `geldtransfer` und `zeige` haben nun beim Aufruf einen Parameter weniger, da das Konto, auf das sie sich jeweils beziehen, jetzt am Anfang des Aufrufs steht. Da sich die Methode `zeige` nun automatisch auf ein Konto bezieht, haben wir den Namen der Methode entsprechend verkürzt.

Seit der Einführung der Basisdatentypen sind Sie bereits mit dem Umgang von Objekten und der Verwendung ihrer Attribute und Methoden vertraut. In diesem Kapitel werden Sie lernen, wie Sie Ihre eigenen Objekte mithilfe von Klassen erzeugen können.

11.1 Klassen

Objekte werden über *Klassen* erzeugt. Eine Klasse ist dabei eine formale Beschreibung der Struktur eines Objekts, also welche Attribute und Methoden es besitzt.

Mit einer Klasse allein kann man noch nicht sinnvoll arbeiten, da sie nur die Beschreibung eines Objekttyps darstellt, selbst aber kein Objekt ist.² Man kann das Verhältnis von Klasse und Objekt mit dem von Backrezept und Kuchen vergleichen:

² Streng genommen, sind in Python auch Klassen Instanzen sogenannter Metaklassen. Dies soll hier aber keine Rolle spielen.

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 *Instanzieren* genannt. So, wie es zu einem Rezept mehrere Kuchen geben kann, so können auch mehrere Instanzen einer Klasse erzeugt werden:

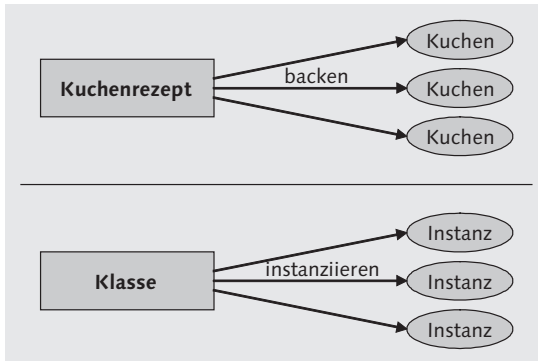


Abbildung 11.1 Analogie von Rezept/Kuchen und Klasse/Objekt

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, lässt sich mit einer Klasse allein nicht arbeiten, weil sie nur eine abstrakte Beschreibung ist. Deshalb wollen wir nun eine Instanz der noch leeren Beispielsklasse `Konto` erzeugen. Um eine Klasse zu instanziiieren, 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 Ausgabe teilt uns mit, dass der Rückgabewert von `Konto()` eine Instanz der Klasse `Konto` im Namensraum `__main__` ist und im Speicher unter der Adresse `0xb787776c` abgelegt wurde – uns reicht als Information aus, dass eine neue Instanz der Klasse `Konto` erzeugt worden ist.

11.1.1 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(self):
        pass
```

Beachten Sie den Parameter `self` am Anfang der Parameterliste 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 lernen, wie Sie in den Erzeugungsprozess neuer Objekte eingreifen und neue Attribute anlegen können.

11.1.2 Der Konstruktor und die Erzeugung von Attributen

Der Lebenszyklus jeder Instanz sieht gleich aus: Sie wird erzeugt, benutzt und anschließend wieder beseitigt. Dabei ist die Klasse, also der Bauplan, dafür verantwortlich, dass sich die Instanz zu jeder Zeit in einem wohldefinierten Zustand befindet. Aus diesem Grund gibt es eine spezielle Methode, die automatisch beim Instanziiieren eines Objekts aufgerufen wird, um das Objekt in einen gültigen Initialzustand zu versetzen. Man nennt diese Methode den *Konstruktor* (engl. *construct*, dt. »erstellen«) einer Klasse.

Um einer Klasse einen Konstruktor zu geben, muss man eine Methode mit dem Namen³ `__init__` definieren.

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

11

Wenn wir jetzt eine Instanz der Klasse `Beispielklasse` erzeugen, wird implizit die Methode `__init__` 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.

Hinweis

Falls Sie bereits andere objektorientierte Programmiersprachen beherrschen und sich wundern, wie Sie in Python einen Destruktor implementieren können, sei Ihnen an dieser Stelle gesagt, dass es in Python keinen Destruktor gibt, der garantiert am Ende der Lebenszeit einer Instanz gerufen wird.

Ein ähnliches Verhalten kann mithilfe der Methode `__del__` realisiert werden, die auf Seite 304 beschrieben wird.



³ Dabei ist das Wort »init« sowohl von links als auch von rechts mit zwei Unterstrichen »_« umgeben. Methoden, die nach dem Schema `__WORT__` aufgebaut sind, haben in Python eine besondere Bedeutung. Wir werden später im Abschnitt 11.5, »Magic Methods und Magic Attributes«, ab Seite 302 ausführlicher darauf eingehen.

Neue Attribute anlegen

Da es die Hauptaufgabe eines Konstruktors ist, einen konsistenten Initialzustand der Instanz herzustellen, sollten alle Attribute einer Klasse auch dort definiert werden.⁴ Die Definition neuer Attribute erfolgt durch eine 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:

```
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.

Im folgenden Listing sehen Sie die vollständige Klasse `Konto`.

```
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 geldtransfer(self, ziel, betrag):
        # Hier erfolgt der Test, ob der Transfer möglich ist
        if (betrag < 0 or
            self.UmsatzHeute + betrag > self.MaxTagesumsatz or
```

⁴ Es gibt wenige Sonderfälle, in denen von dieser Regel abgewichen werden muss. Sie sollten im Regelfall alle Attribute Ihrer Klassen im Konstruktor anlegen.

```

        ziel.UmsatzHeute + betrag > ziel.MaxTagesumsatz):
            # Transfer unmöglich
            return False
    else:
        # Alles OK - Auf geht's
        self.Kontostand -= betrag
        self.UmsatzHeute += betrag
        ziel.Kontostand += betrag
        ziel.UmsatzHeute += betrag
        return True

def einzahlen(self, betrag):
    if (betrag < 0 or
        self.UmsatzHeute + betrag > self.MaxTagesumsatz):
        # Tageslimit überschritten oder ungültiger Betrag
        return False
    else:
        self.Kontostand += betrag
        self.UmsatzHeute += betrag
        return True

def auszahlen(self, betrag):
    if (betrag < 0 or
        self.UmsatzHeute + betrag > self.MaxTagesumsatz):
        # Tageslimit überschritten oder ungültiger Betrag
        return False
    else:
        self.Kontostand -= betrag
        self.UmsatzHeute += betrag
        return True

def zeige(self):
    print("Konto von {0}".format(self.Inhaber))
    print("Aktueller Kontostand: {0:.2f} Euro".format(
        self.Kontostand))
    print("(Heute schon {0:.2f} von {1} Euro umgesetzt)".format(
        self.UmsatzHeute, self.MaxTagesumsatz))

```

An dieser Stelle haben wir unser Ziel erreicht, die Kontodaten und die dazugehörigen Verarbeitungsfunktionen zu einer Einheit zu verbinden.

Mithilfe der neuen Kontoklasse können wir die Geschäfte von Herrn Schmidt und Herrn Meier vom Beginn des Kapitels erneut durchführen:

```
>>> k1 = Konto("Heinz Meier", 567123, 12350.0)
>>> k2 = Konto("Erwin Schmidt", 396754, 15000.0)
>>> 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 Euro umgesetzt)
>>> k2.zeige_konto()
Konto von Erwin Schmidt
Aktueller Kontostand: 14160.00 Euro
(Heute schon 1160.00 von 1500 Euro umgesetzt)
```

Im folgenden Abschnitt werden wir uns darüber Gedanken machen, wie wir unser Beispiel so strukturieren können, dass es sich leicht auf neue Problemstellungen verallgemeinern lässt.

11.2 Vererbung

Neben der strukturellen Verschmelzung von Daten und den darauf arbeitenden Methoden zu einer Einheit zielt das Konzept der Objektorientierung darauf ab, die Wiederverwendbarkeit von Programmcode zu verbessern. Damit ist gemeint, dass ein Programm mit geringem Aufwand an Probleme angepasst werden kann, die dem Problem ähnlich sind, für das das Programm ursprünglich entwickelt wurde.

Konkret bedeutet dies, dass man von bereits bestehenden Klassen neue Klassen ableitet, um diese um zusätzliche Funktionalität zu erweitern. Dabei übernimmt die abgeleitete Klasse alle Fähigkeiten von ihrer *Basisklasse*, sodass sie zunächst eine Kopie dieser Klasse ist. Man sagt, die Basisklasse *vererbt* ihre Fähigkeiten an eine Tochterklasse. Nach diesem Vererbungsschritt kann man die abgeleitete Klasse an die neuen Anforderungen anpassen.

Bevor wir Vererbung auf unser konkretes Beispiel anwenden, werden wir Ihnen an einigen abstrakten Beispielen zeigen, wie Python dieses Konzept technisch umsetzt.

11.2.1 Technische Grundlagen

Um eine Klasse von einer anderen erben zu lassen, schreibt man bei der Definition der Tochterklasse die Basisklasse in Klammern hinter den Klassennamen. Im folgenden Beispiel erbt also die Klasse `B` von der Klasse `A`:

```
class A:
    pass

class B(A):
    pass
```

Diese Klassen `A` und `B` sind noch sehr langweilig, da sie keine Methoden oder Attribute besitzen. Daher erweitern wir unsere Klassen folgendermaßen:

```
class A:
    def __init__(self):
        self.X = 1337
        print("Konstruktor von A")

    def m(self):
        print("Methode m von A. Es ist self.X =", self.X)

class B(A):
    def n(self):
        print("Methode n von B")

b = B()
b.n()
b.m()
```

In diesem Beispiel wird die Klasse `A` um einen Konstruktor erweitert, der ein Attribut `X` mit dem Wert 1337 erzeugt. Zusätzlich erhält die Klasse `A` eine Methode `m`. Sowohl der Konstruktor als auch die Methode `m` geben jeweils eine Meldung auf dem Bildschirm aus. Außerdem versehen wir die Klasse `B` mit einer Methode `n`, die ebenfalls eine Meldung ausgibt. Am Ende des kleinen Programms werden eine Instanz der Klasse `B` erzeugt und ihre Methoden `n` und `m` gerufen.

Die Ausgabe zeigt, dass `B` sowohl den Konstruktor als auch die Methode `m` von der Klasse `A` geerbt hat. Auch das Attribut `X` wurde ordnungsgemäß angelegt.

```
Konstruktor von A
Methode n von B
Methode m von A. Es ist self.X = 1337
```

Der Konstruktor einer Klasse hat die Aufgabe, die Klasse in einen wohldefinierten Initialzustand zu bringen. Wie die Ausgabe des obigen Programms zeigt, wurde beim Erzeugen einer Instanz der Klasse `B` der Konstruktor der Klasse `A` gerufen. Nun ist es in der Praxis häufig so, dass eine abgeleitete Klasse einen anderen Konstruktor als ihre Basisklasse benötigt, um eigene Initialisierungen vorzunehmen.

Überschreiben von Methoden

Wir erweitern daher unsere Klasse `B` um einen eigenen Konstruktor, der ein Attribut `Y` anlegt und auch eine Ausgabe erzeugt. Zusätzlich erweitern wir die Methode `n` so, dass sie den Wert des Attributs `Y` ausgibt.

```
class B(A):
    def __init__(self):
        self.Y = 10000
        print("Konstruktor von B")

    def n(self):
        print("Methode n von B. Es ist self.Y =", self.Y)

b = B()
b.n()
b.m()
```

Die Ausgabe dieses Beispiels überrascht uns mit einer Fehlermeldung:

```
Konstruktor von B
Methode n von B. Es ist self.Y = 10000
Traceback (most recent call last):
...
AttributeError: 'B' object has no attribute 'X'
```

Laut der Bildschirmausgabe werden der Konstruktor von `B` sowie die Methoden `n` und `m` gerufen. Allerdings beschwert sich die Methode `m` darüber, dass die Instanz kein Attribut `X` besitzt.

Dies ist nicht verwunderlich, da der Konstruktor von `A`, der für das Anlegen des Attributs `X` zuständig ist, nicht aufgerufen wird. Dieses Verhalten ist folgendermaßen begründet:

Die Klasse `B` hat die Methode `__init__`, also den Konstruktor, zunächst von der Klasse `A` geerbt, sie aber dann mit ihrem eigenen Konstruktor *überschrieben*. Infolgedessen wird beim Erzeugen einer Instanz der Klasse `B` nur noch der neue von `B` definierte Konstruktor gerufen, wohingegen der Konstruktor von `A` nicht zum Zuge kommt.

Generell spricht man vom *Überschreiben einer Methode*, wenn eine Klasse eine Methode erneut implementiert, die sie bereits von ihrer Basisklasse geerbt hat.

Im Allgemeinen ist es aber erforderlich, dass der überschriebene Konstruktor der Basisklasse gerufen wird, um die Instanz in einen konsistenten Zustand zu versetzen. Daher ist es möglich, überschriebene Methoden der Basisklasse explizit zu rufen:

```
class B(A):
    def __init__(self):
        A.__init__(self)
        self.Y = 10000
        print("Konstruktor von B")

    def n(self):
        print("Methode n von B. Es ist self.Y =", self.Y)

b = B()
b.n()
b.m()
```

11

Mit der Zeile `A.__init__(self)` rufen wir im Konstruktor der Klasse `B` explizit den Konstruktor der Basisklasse `A`. Wichtig ist an dieser Stelle, dass der sonst implizit übergebene Parameter `self` explizit angegeben werden muss, wenn eine Methode der Basisklasse gerufen wird.

Die Ausgabe des obigen Codes zeigt, dass der Konstruktor von `A` nun wie gewünscht aufgerufen wird, und auch der Aufruf der Methode `m` funktioniert wieder.

```
Konstruktor von A
Konstruktor von B
Methode n von B. Es ist self.Y = 10000
Methode m von A. Es ist self.X = 1337
```

Dieses Überschreiben von Methoden ist nicht auf den Konstruktor beschränkt, und es kann auch jede beliebige Methode der Basisklasse wie der Konstruktor im obigen Beispiel explizit gerufen werden.

Nun haben wir das Werkzeug an der Hand, um das Konzept Vererbung auf unser Kontobeispiel anzuwenden. Dabei werden wir unser Programm in mehrere Klassen zerlegen, die voneinander erben.

11.2.2 Die Klasse GirokontoMitTagesumsatz

Objektorientierte Programmierung zielt darauf ab, Vorhandenes erneut zu verwenden bzw. Code bereitzustellen, der einfach an neue Anforderungen angepasst werden

kann. Dies hat zur Folge, dass man bei der Entwicklung eines objektorientierten Programms immer darauf achten sollte, seine Klassen möglichst universell zu halten. Erst dadurch wird es möglich, Teile des Programms durch geschickte Vererbung für die Lösung neuer Probleme zu übernehmen.

Es ist eine Wissenschaft für sich, wie ein elegantes objektorientiertes Programm strukturiert werden sollte. Da wir an dieser Stelle keine erschöpfende Einführung in die Theorie der objektorientierten Programmierung geben möchten, werden wir die Prinzipien an einem Beispiel demonstrieren.

Wir werden dazu eine Klasse `GirokontoMitTagesumsatz` entwickeln, die das Gleiche leistet wie die oben präsentierte Klasse `Konto`. Allerdings werden wir diesmal darauf achten, unseren Programmcode so zu strukturieren, dass er leicht für ähnliche Aufgaben verwendet werden kann.

Ausgangspunkt unseres Programms ist die Klasse `Konto` auf Seite 278, deren Attribute sich zunächst in zwei Kategorien einteilen lassen:

1. Daten, die Umgang mit dem Geld auf dem Konto betreffen (`Kontostand`, `MaxTagesumsatz`, `UmsatzHeute`)
2. Daten, die den Kunden betreffen (`Inhaber`, `Kontonummer`)

Alle Methoden mit Ausnahme der Methode `zeige` verwenden nur Attribute der ersten Kategorie. Daher nehmen wir an dieser Stelle die erste strukturelle Trennung vor, indem wir ein Konto in zwei Teile aufspalten.

Der eine Teil soll sich um die Verwaltung des Kontostandes kümmern, und der andere Teil soll die Kundendaten speichern.

Die Klasse `VerwalteterGeldbetrag`

Abstrakt gesehen, muss eine Klasse, die den Kontostand unseres Kontos verwaltet, Einzahlungen, Auszahlungen und Geldtransfers zu anderen Konten unterstützen. Diese Operationen müssen an bestimmte Bedingungen gekoppelt werden können, nämlich, ob die jeweiligen maximalen Tagesumsätze eingehalten werden oder nicht.

Neben Konten gibt es aber weitere Gebilde, die einen Geldbetrag nach bestimmten Regeln verwalten. Beispielsweise lässt sich das Geld, das sich in einer Geldbörse befindet, als Kontostand interpretieren. Die Operationen Einzahlen und Auszahlen beschreiben dann den Vorgang, Bargeld in die Geldbörse zu geben bzw. Bargeld aus dieser zu entnehmen. Ähnlich verhält es sich bei einem Tresor oder dem Guthaben auf einer Prepaid-Karte.

Es ist daher sinnvoll, eine Klasse zu implementieren, die es ermöglicht, einen Geldbetrag nach bestimmten Regeln zu verwalten. Diese Klasse `VerwalteterGeldbetrag` wird

dann als Basis für unsere GirokontoMitTagesumsatz-Klasse dienen, bleibt aber weiterhin nützlich für andere Anwendungen.

```
class VerwalteterGeldbetrag:
    def __init__(self, anfangsbetrag):
        self.Betrag = anfangsbetrag

    def einzahlenMoeglich(self, betrag):
        return True

    def auszahlenMoeglich(self, betrag):
        return True

    def einzahlen(self, betrag):
        if betrag < 0 or not self.einzahlenMoeglich(betrag):
            return False
        else:
            self.Betrag += betrag
            return True

    def auszahlen(self, betrag):
        if betrag < 0 or not self.auszahlenMoeglich(betrag):
            return False
        else:
            self.Betrag -= betrag
            return True

    def zeige(self):
        print("Betrag: {:.2f}".format(self.Betrag))
```

11

Im Konstruktor der Klasse wird das Attribut `Betrag` angelegt und auf den übergebenen Initialwert gesetzt. Über die Methoden `einzahlen` und `auszahlen` kann der Betrag verändert werden, wobei jeweils `True` zurückgegeben wird, wenn die Operation erfolgreich war, und `False`, falls ein Problem aufgetreten ist. Die Methode `zeige` gibt den aktuell vorhandenen Betrag auf dem Bildschirm aus.

Der Clou der Klasse `VerwalteterGeldbetrag` liegt in den Methoden `einzahlenMoeglich` und `auszahlenMoeglich`, mit denen die Methoden `einzahlen` bzw. `auszahlen` prüfen, ob die jeweilige Operation ausgeführt werden kann.

Sie sind dazu gedacht, von abgeleiteten Klassen *überschrieben* zu werden, um die gewünschten Bedingungen festzulegen. Da sie in der Klasse `VerwalteterGeldbetrag`

den Wert `True` zurückgeben, sind Einzahlungen und Auszahlungen ohne Einschränkungen möglich, solange diese Methoden nicht überschrieben werden.

Die Klasse `AllgemeinesKonto`

Unserer Klasse `VerwalteterGeldbetrag` fehlt unter anderem noch die Möglichkeit, Geld zwischen verschiedenen Instanzen zu transferieren, um die Funktionalität unserer Ausgangsklasse `Konto` nachzubilden. Da dies ein Vorgang ist, der von sämtlichen Konten beherrscht werden soll, werden wir nun eine Klasse `AllgemeinesKonto` von `VerwalteterGeldbetrag` ableiten und sie um eine Methode `geldtransfer` erweitern.

Außerdem gehören zu einem Konto immer die Kundendaten des jeweiligen Kontoinhabers. Diese werden wir in dem Attribut `Kundendaten` ablegen, dessen Wert den ersten Parameter des Konstruktors festlegt. Um die Definition der Klasse, mit der die Kundendaten gespeichert werden, kümmern wir uns später.

```
class AllgemeinesKonto(VerwalteterGeldbetrag):
    def __init__(self, kundendaten, kontostand):
        VerwalteterGeldbetrag.__init__(self, kontostand)
        self.Kundendaten = kundendaten

    def geldtransfer(self, ziel, betrag):
        if (self.auszahlenMoeglich(betrag) and
            ziel.einzahlenMoeglich(betrag)):
            self.auszahlen(betrag)
            ziel.einzahlen(betrag)
            return True
        else:
            return False

    def zeige(self):
        self.Kundendaten.zeige()
        VerwalteterGeldbetrag.zeige(self)
```

Die neue Methode `geldtransfer` greift auf die Methoden `auszahlenMoeglich` und `einzahlenMoeglich` zurück, um die Machbarkeit des Transfers zu prüfen. Für den Transfer selbst werden die Methoden `auszahlen` und `einzahlen` verwendet.

Um eine Instanz der Klasse `AllgemeinesKonto` auszugeben, wird die Methode `zeige` überschrieben, sodass zunächst die Kundendaten ausgegeben werden und anschließend die Methode `zeige` der Basisklasse `VerwalteterGeldbetrag` gerufen wird. Dabei wird vorausgesetzt, dass die Instanz, die vom Attribut `Kundendaten` referenziert wird, eine Methode namens `zeige` besitzt.

Die Klasse AllgemeinesKontoMitTagesumsatz

Nun ist es an der Zeit, die Klasse `AllgemeinesKonto` um die Fähigkeit zu erweitern, den Tagesumsatz zu begrenzen. Zu diesem Zweck leiten wir die Klasse `AllgemeinesKontoMitTagesumsatz` von `AllgemeinesKonto` ab und überschreiben einige der Methoden.

```
class AllgemeinesKontoMitTagesumsatz(AllgemeinesKonto):
    def __init__(self, kundendaten, kontostand,
                  max_tagesumsatz=1500):
        AllgemeinesKonto.__init__(self, kundendaten, kontostand)
        self.MaxTagesumsatz = max_tagesumsatz
        self.UmsatzHeute = 0.0

    def transferMoeglich(self, betrag):
        return (self.UmsatzHeute + betrag <= self.MaxTagesumsatz)

    def auszahlenMoeglich(self, betrag):
        return self.transferMoeglich(betrag)

    def einzahlenMoeglich(self, betrag):
        return self.transferMoeglich(betrag)

    def einzahlen(self, betrag):
        if AllgemeinesKonto.einzahlen(self, betrag):
            self.UmsatzHeute += betrag
            return True
        else:
            return False

    def auszahlen(self, betrag):
        if AllgemeinesKonto.auszahlen(self, betrag):
            self.UmsatzHeute += betrag
            return True
        else:
            return False

    def zeige(self):
        AllgemeinesKonto.zeige(self)
        print("Heute schon {:.2f} von {:.2f} Euro umgesetzt".format(
            self.UmsatzHeute, self.MaxTagesumsatz))
```

Es werden die Methoden `einzahlenMoeglich` und `auszahlenMoeglich` überschrieben, sodass sie, abhängig vom Tagesumsatz, Einzahlungen und Auszahlungen ermöglichen oder blockieren. Beide Methoden greifen dafür auf die neue Methode `transferMoeglich` zurück.

Die Methoden `einzahlen` und `auszahlen` werden so angepasst, dass sie das Attribut `UmsatzHeute` gegebenenfalls aktualisieren. Zu guter Letzt fügt die `zeige`-Methode der Ausgabe von `AllgemeinesKonto.zeige` Informationen über den Tagesumsatz hinzu.

Damit verfügt die Klasse `AllgemeinesKontoMitTagesumsatz` über die gleiche Funktionalität, den Kontostand zu verwalten, wie unsere Ausgangsklasse `Konto`. Was noch fehlt, ist die Verwaltung der Kundendaten.

Die Klasse `GirokontoDaten`

Die mit einem `Girokonto` assoziierten Kundendaten werden in Instanzen der Klasse `GirokontoKundendaten` abgelegt. Neben zwei Attributen, die den Namen des Kontoinhabers sowie die Kontonummer speichern, verfügt auch diese Klasse über eine Methode `zeige`, um die Informationen auf dem Bildschirm auszugeben.

```
class GirokontoKundendaten:
    def __init__(self, inhaber, kontonummer):
        self.Inhaber = inhaber
        self.Kontonummer = kontonummer

    def zeige(self):
        print("Inhaber:", self.Inhaber)
        print("Kontonummer:", self.Kontonummer)
```

Nun können wir die Klasse `GirokontoMitTagesumsatz` definieren.

Die Klasse `GirokontoMitTagesumsatz`

Abschließend leiten wir die Klasse `GirokontoMitTagesumsatz` von der Klasse `AllgemeinesKontoMitTagesumsatz` ab und versehen sie durch Überschreiben des Konstruktors mit passenden Kundendaten.

```
class GirokontoMitTagesumsatz(AllgemeinesKontoMitTagesumsatz):
    def __init__(self, inhaber, kontonummer, kontostand,
                  max_tagesumsatz=1500):
        kundendaten = GirokontoKundendaten(inhaber, kontonummer)
        AllgemeinesKontoMitTagesumsatz.__init__(self,
                                                  kundendaten, kontostand, max_tagesumsatz)
```

Diese Klasse bildet alle Funktionalität der Klasse `Konto` ab, sodass wir unser Eingangsbeispiel von Herrn Meier und Herrn Schmidt⁵ ausführen können.

⁵ Siehe S. 272 und S. 280.

```

>>> k1 = GirokontoMitTagesUmsatz("Heinz Meier", 567123, 12350.0)
>>> k2 = GirokontoMitTagesUmsatz("Erwin Schmidt", 396754, 15000.0)

>>> k1.geldtransfer(k2, 160)
True
>>> k2.geldtransfer(k1, 1000)
True
>>> k2.geldtransfer(k1, 500)
False
>>> k2.einzahlen(500)
False

>>> k1.zeige()
Inhaber: Heinz Meier
Kontonummer: 567123
Betrag: 13190.00
Heute schon 1160.00 von 1500.00 Euro umgesetzt

>>> k2.zeige()
Inhaber: Erwin Schmidt
Kontonummer: 396754
Betrag: 14160.00
Heute schon 1160.00 von 1500.00 Euro umgesetzt

```

Im nächsten Absatz werden wir verdeutlichen, dass unser Programm durch diese Strukturierung leicht zu erweitern ist.

11.2.3 Beispiele möglicher Erweiterungen

Wir haben nun durch stückweise Verfeinerung mittels Vererbung aus unserer anfänglichen Idee des verwalteten Geldbetrags die Klasse `GirokontoMitTagesumsatz` abgeleitet. Diese Klasse verfügt nun über die gleiche Funktionalität wie die Klasse `Konto`.

Abbildung 11.2 veranschaulicht die entstehende Klassenhierarchie grafisch.

Der Nutzen dieser Strukturierung wird deutlich, wenn wir neue Klassen einführen, die auf bereits vorhandene Funktionalität zurückgreifen können. Als Beispiel dienen dazu die Klassen `Geldboerse`, `Tresor`, `Girokonto`, `Nummernkonto` und `NummernkontoMitTagesumsatz`.

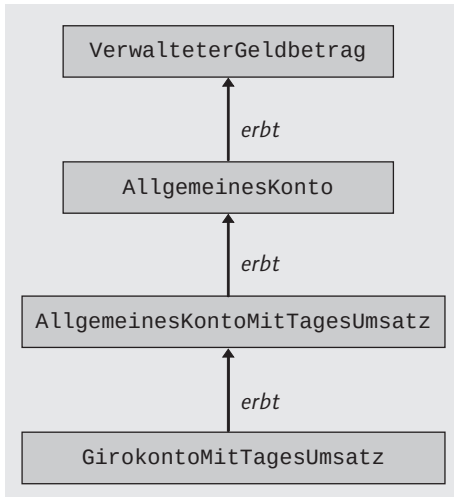


Abbildung 11.2 Die Klassenhierarchie des Kontobeispiels

Bevor wir die Beschreibung und Implementation dieser Klassen besprechen, werfen wir einen Blick auf die neu entstehende Klassenhierarchie, wie sie Abbildung 11.3 zeigt.

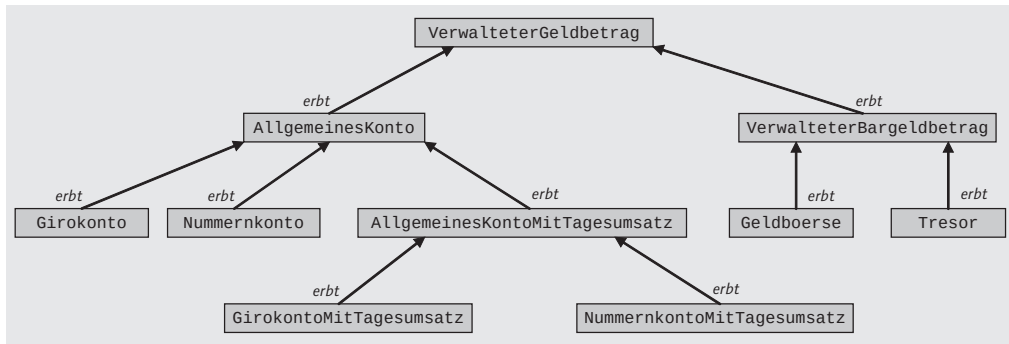


Abbildung 11.3 Eine erweiterte Klassenhierarchie des Kontobeispiels

Die Klassen `Geldboerse` und `Tresor` verwalten jeweils einen Bargeldbetrag, weshalb wir von der Klasse `VerwalteterGeldbetrag` zunächst eine Klasse `VerwalteterBargeldbetrag` ableiten. Im Unterschied zum allgemeinen verwalteten Geldbetrag kann ein Bargeldbetrag nicht negativ sein. Daher überschreibt die Klasse `VerwalteterBargeldbetrag` die Methode `auszahlenMoeglich`, um negative Beträge zu verhindern.

Zusätzlich zu dem Girokonto, dessen Transaktionen durch einen maximalen Tagesumsatz limitiert sind, modellieren wir nun Girokonten ohne Limitierung der Umsätze durch die Klasse `Girokonto`. Diese Klasse wird direkt von der Klasse `AllgemeinesKonto` abgeleitet und verwendet dieselben Kundendaten wie `GirokontoMitTagesumsatz`.

Um neben Girokonten auch Nummernkonten verwalten zu können, legen wir eine neue Klasse an, mit der die Kundendaten eines Nummernkontos verwaltet werden können.⁶ Damit lassen sich dann die Klassen `Nummernkonto` und `NummernkontoMitTagesumsatz` von den Klassen `AllgemeinesKonto` bzw. `AllgemeinesKontoMitTagesumsatz` ableiten.

Nun schauen wir uns an, wie die Klassen `Geldboerse`, `Tresor`, `Girokonto`, `Nummernkonto` und `NummernkontoMitTagesumsatz` implementiert werden können.

Die Klassen `VerwalteterBargelddbetrag`, `Geldboerse` und `Tresor`

Die Klasse `VerwalteterBargelddbetrag` passt die Klasse `VerwalteterGeldbetrag` so an, dass sie einen negativen Wert für das Attribut `Betrag` verhindert.

```
class VerwalteterBargelddbetrag(VerwalteterGeldbetrag):
    def __init__(self, bargelddbetrag):
        if bargelddbetrag < 0:
            bargelddbetrag = 0
        VerwalteterGeldbetrag.__init__(self, bargelddbetrag)

    def auszahlenMoeglich(self, betrag):
        return (self.Betrag >= betrag)
```

11

Im Konstruktor wird dafür gesorgt, dass der Betrag nicht mit einem negativen Wert initialisiert werden kann, und die Methode `auszahlenMoeglich` liefert genau dann `True` zurück, wenn der Betrag in der Geldbörse mindestens so groß ist wie der Betrag, der ausgezahlt werden soll.

Die Klassen `Geldboerse` und `Tresor` erben nun von der Klasse `VerwalteterBargelddbetrag`.

```
class Geldboerse(VerwalteterBargelddbetrag):
    # TODO: Spezielle Methoden fuer eine Geldboerse
    pass

class Tresor(VerwalteterBargelddbetrag):
    # TODO: Spezielle Methoden fuer einen Tresor
    pass
```

Mit den beiden Kommentaren soll angedeutet werden, dass an dieser Stelle noch Methoden fehlen, die eine Geldbörse und einen Tresor zu besonderen verwalteten

⁶ Diese Klasse `NummernkontoKundendaten` ist genau wie die Klasse `GirokontoKundendaten` nicht in Abbildung 11.3 aufgeführt.

Geldbeträgen machen. Da wir an dieser Stelle keine vollständige Software entwickeln, sondern Ihnen die prinzipielle Erweiterbarkeit des Programms demonstrieren möchten, verzichten wir auf diese Details. Sie können sich als Übung einmal selbst überlegen, welche Funktionalität in den beiden Fällen sinnvoll ist.⁷

Die Klassen Girokonto, Nummernkonto und NummernkontoMitTagesumsatz

Die Klasse Girokonto erbt direkt von der Klasse AllgemeinesKonto.

```
class Girokonto(AllgemeinesKonto):
    def __init__(self, inhaber, kontonummer, kontostand):
        kundendaten = GirokontoKundendaten(inhaber, kontonummer)
        AllgemeinesKonto.__init__(self, kundendaten, kontostand)
```

Analog zur Klasse GirokontoKundendaten führen wir die Klasse NummernkontoKundendaten ein, um die Kundendaten eines Nummernkontos zu verwalten. In unserem Modell wird ein Nummernkonto durch eine Identifikationsnummer beschrieben.

```
class NummerkontoKundendaten:
    def __init__(self, identifikationsnummer):
        self.Identifikationsnummer = identifikationsnummer

    def zeige(self):
        print("Identifikationsnummer:",
              self.Identifikationsnummer)
```

Mithilfe dieser Klasse können wir die Klassen Nummernkonto und NummernkontoMitTagesumsatz definieren.

```
class Nummernkonto(AllgemeinesKonto):
    def __init__(self, identifikationsnummer, kontostand):
        kundendaten = NummerkontoKundendaten(identifikationsnummer)
        AllgemeinesKonto.__init__(self, kundendaten, kontostand)

class NummernkontoMitTagesumsatz(AllgemeinesKontoMitTagesumsatz):
    def __init__(self, kontonummer, kontostand, max_tagesumsatz):
        kundendaten = NummerkontoKundendaten(kontonummer)
        AllgemeinesKontoMitTagesumsatz.__init__(self,
            kundendaten, kontostand, max_tagesumsatz)
        NummerkontoKundendaten.__init__(self, kontonummer)
```

⁷ Beispielsweise kann man nur dann Geld in einen Tresor hineinlegen oder herausnehmen, wenn die Tür nicht verriegelt ist ...

Zur Demonstration verwenden wir die beiden Klassen in einem kleinen Beispielprogramm.

```
>>> nk1 = Nummernkonto(113427613185, 5000)
>>> nk2 = NummernkontoMitTagesumsatz(45657364234, 12000, 3000)
>>> nk1.auszahlen(1000)
True
>>> nk2.einzahlen(1500)
True
>>> nk1.geldtransfer(nk2, 2000)
False
>>> nk1.zeige()
Identifikationsnummer: 113427613185
Betrag: 4000.00
>>> nk2.zeige()
Identifikationsnummer: 45657364234
Betrag: 13500.00
Heute schon 1500.00 von 3000.00 Euro umgesetzt
```

Es werden sowohl eine Instanz der Klasse `Nummernkonto` als auch der Klasse `NummernkontoMitTagesumsatz` erzeugt. Anschließend werden von dem ersten Konto 1000 € abgeboben und 1500 € auf das zweite eingezahlt. Schlussendlich versuchen wir, 2000 € von dem Konto `nk1` auf das Konto `nk2` zu überweisen. Da der Tagesumsatz von `nk2` damit überschritten würde, schlägt dies fehl.

Wie die Ausgabe zeigt, arbeiten die beiden Klassen genauso wie die anderen Kontoklassen.

11.2.4 Ausblick

Der große Vorteil der Vererbung ist es, dass man aus vorhandenen Klassen neue Klassen ableiten kann, um diese dann an die zu lösende Problemstellung anzupassen. Dabei kann die abgeleitete Klasse auf die gesamte Funktionalität zurückgreifen, die von der Basisklasse zur Verfügung gestellt wird. Folglich müssen nur noch die Methoden implementiert bzw. überschrieben werden, die nicht zur neuen Problemstellung passen.

Würden wir beispielsweise, ausgehend von der Klasse `Konto`, die Klassen `Girokonto`, `Nummernkonto`, `GirokontoMitTagesumsatz` und `NummernkontoMitTagesumsatz` entwickeln, ohne auf Vererbung zurückzugreifen, müssten wir die Methoden zum Ein- und Auszahlen in jeder dieser Klassen neu implementieren. Dies hätte dazu geführt, dass an mehreren Stellen unseres Programms sehr ähnlicher Code stehen würde.

Diese Dopplung von Code bläht den Umfang eines Programms unnötig auf. Dadurch werden die Wartung und Weiterentwicklung erschwert, da immer an mehreren Stellen parallel gearbeitet bzw. korrigiert werden muss.

Durch geschickte Strukturierung mittels Vererbung sind Programme möglich, die mit einem Minimum an Funktionalitätsdopplung auskommen.

In großen Softwareprojekten haben wir es nicht wie in unserem Modellbeispiel mit einer Hand voll Klassen zu tun, sondern es kommen hunderte oder tausende Klassen zum Einsatz. In einem solchen Umfeld fallen die durch Vererbung gemachten Einsparungen noch deutlicher ins Gewicht.

11.2.5 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, schreiben Sie die Basisklassen durch Kommata getrennt in die Klammern hinter den Klassennamen:

```
class NeueKlasse(Basisklasse1, Basisklasse2, Basisklasse3):
    # Definition von Methoden und Attributen
    pass
```

In diesem Beispiel erbt die Klasse `NeueKlasse` von den drei Klassen `Basisklasse1`, `Basisklasse2` und `Basisklasse3`.

Mehrfachvererbung ist ein sehr komplexes Thema, weshalb wir uns hier nur auf ein abstraktes Beispiel beschränken möchten, um Ihnen die dahinter stehende Idee zu verdeutlichen.

Wir nehmen an, wir hätten zwei Klassen zur Beschreibung von Geländefahrzeugen und Wasserfahrzeugen, nämlich `Gelaendefahrzeug` und `Wasserfahrzeug`. Wenn wir nun eine Klasse `Amphibienfahrzeug` definieren möchten, kommen sowohl die Klasse `Gelaendefahrzeug` als auch die Klasse `Wasserfahrzeug` als Basisklasse infrage, denn ein `Amphibienfahrzeug` ist sowohl das eine als auch das andere.

Es ist daher nur konsequent, die Klasse `Amphibienfahrzeug` von beiden dieser Klassen erben zu lassen, wie es Abbildung 11.4 veranschaulicht.

Im Ergebnis erbt die Klasse `Amphibienfahrzeug` die Methoden beider Klassen `Gelaendefahrzeug` und `Wasserfahrzeug`.

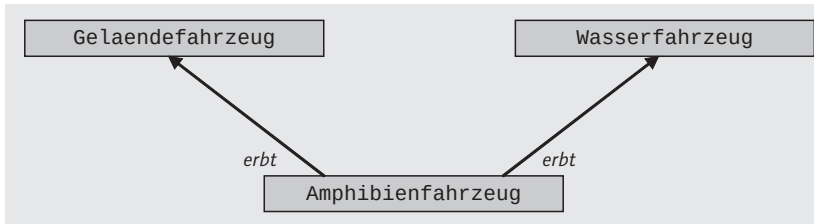


Abbildung 11.4 Mehrfachvererbung am Beispiel eines Amphibienfahrzeugs

Mögliche Probleme der Mehrfachvererbung

Es ist kein Zufall, dass nur wenige Sprachen das Konzept der Mehrfachvererbung unterstützen, da es eine Reihe prinzipieller Probleme gibt.

Beispielsweise kommt es vor, dass mehrere Basisklassen eine Methode mit dem gleichen Namen implementieren. Die erbende Klasse erbt diese Methode dann von derjenigen Basisklasse, die am weitesten links in der Liste der Basisklassen steht.

Nun müssen zwei Methoden mit demselben Namen aber keinesfalls die gleiche Aufgabe erfüllen. Im schlimmsten Fall kann es also passieren, dass die erbende Klasse unbenutzbar wird, weil sie nur eine der in Konflikt stehenden Methoden erben kann.

In der Praxis lässt sich Mehrfachvererbung in der Regel umgehen, weshalb wir hier nicht näher darauf eingehen.

11

11.3 Setter und Getter und Property Attributes

Manchmal ist es wünschenswert, den Zugriff auf die Attribute einer Klasse nach bestimmten Regeln zu beeinflussen. So gibt es Attribute, für die nur bestimmte Werte Sinn ergeben. In unserer Klasse `Konto` (siehe Seite 278) gibt es zum Beispiel das Attribut `Tagesumsatz`, dessen Werte sinnvollerweise immer positiv sein sollten.

Ein Konzept der objektorientierten Programmierung, um den Zugriff auf Attribute zu steuern, stellen die *Setter-Methoden* und *Getter-Methoden* dar. Anstatt direkt auf das Attribut zuzugreifen, wird der Zugriff dabei über spezielle Methoden geregelt.

Um das folgende Beispiel übersichtlich zu halten, werden wir eine neue Klasse `A` einführen, die ein über Setter- und Getter-Methoden verwaltetes Attribut besitzt. Die Übertragung auf das Kontobeispiel ist dann eine einfache Übungsaufgabe.

```

class A:
    def __init__(self):
        self.x = 100
  
```

```

def getX(self):
    return self._X

def setX(self, wert):
    if wert < 0:
        return
    self._X = wert

```

Mithilfe der Getter-Methode `getX` und der Setter-Methode `setX` kann nun mit dem Attribut `_X` von `A` gearbeitet werden, wobei negative Belegungen von `_X` durch `setX` verhindert werden:

```

a = A()
print(a.getX())
a.setX(300)
print(a.getX())
a.setX(-20)
print(a.getX())

```

Die Ausgabe dieses Beispiels zeigt, dass der Versuch, `_X` auf den Wert `-20` zu setzen, fehlschlägt.

```

100
300
300

```



Hinweis

Python bietet keinen technischen Schutz davor, dass Attribute ohne den Einsatz von Setter- und Getter-Methoden direkt verwendet werden. Daher ist es auch im obigen Beispiel weiterhin möglich, direkt mit `a._X` auf das Attribut `_X` zuzugreifen.

Es ist eine Konvention unter Python-Entwicklern, Attribute und Methoden, die mit einem Unterstrich beginnen, nach Möglichkeit nicht von außen zu verwenden. Solche Attribute und Methoden werden als Implementierungsdetail betrachtet und gehören nicht zur Schnittstelle der Klasse. Dies ist der Grund, warum wir in unserem Beispiel den Namen `_X` für das Attribut gewählt haben.

Property-Attribute

Die explizite Verwaltung von Attributen mittels Setter- und Getter-Methoden ist unschön, da man nun bei der Benutzung einer Klasse zwischen Attributen unter-

scheiden muss, die man direkt verwenden darf, und solchen, deren Zugriffe durch Setter- und Getter-Methoden gesteuert werden müssen.

Die sogenannten *Property-Attribute* beheben dieses Problem, indem Setter- und Getter-Methoden beim Schreiben bzw. Lesen eines Attributs implizit aufgerufen werden.

Wir können damit unsere Klasse so anpassen, dass sie sich folgendermaßen verwenden lässt und wir trotzdem für gültige Belegungen von `X` sorgen können.

```
a = A()
a.X = 300
print(a.X)
a.X = -20
print(a.X)
```

11

Um dies zu erreichen, müssen wir `X` mithilfe der Built-in Function `property` als Property-Attribut anlegen.

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

Für den Parameter *fget* wird eine Referenz auf eine Getter-Methode für das neue Attribut erwartet. Der Parameter *fset* gibt die dazugehörige Setter-Methode an. Mit dem Parameter *fdel* kann zusätzlich eine Methode angegeben werden, die dann ausgeführt werden soll, wenn das Attribut per `del` gelöscht wird. Über den Parameter *doc* kann das Attribut mit einem sogenannten *Docstring* versehen werden. Was ein Docstring ist, können Sie in Abschnitt 12.5, »Docstrings«, ab Seite 363 nachlesen.

Wir erweitern die Klasse `A` nun mithilfe von `property` um ein Property-Attribut `X`, das die Methoden `getX` und `setX` für den Zugriff verwendet. Intern wird der Wert von `X` weiterhin in dem Attribut `_X` gespeichert. Außerdem fügen wir den Methoden `getX` und `setX` Ausgaben hinzu, um zu sehen, dass sie tatsächlich implizit aufgerufen werden.

```
class A:
    def __init__(self):
        self._X = 100

    def getX(self):
        print("Getter gerufen")
        return self._X

    def setX(self, wert):
        print("Setter gerufen")
        if wert < 0:
```

```

        return
        self._X = wert

X = property(getX, setX)

a = A()
a.X = 300
print(a.X)
a.X = -20
print(a.X)

```

Wie wir der Ausgabe entnehmen, wurden die Setter- und die Getter-Methode implizit beim Zugriff auf das Attribut gerufen. Außerdem wurde die ungültige Zuweisung des Wertes -20 verhindert.

```

Getter gerufen
100
Setter gerufen
Getter gerufen
300
Setter gerufen
Getter gerufen
300

```

Sie sollten bei der Verwendung von Setter- und Getter-Methoden im Hinterkopf behalten, dass dadurch bei jedem Attributzugriff eine Methode gerufen wird, was das Programm bei sehr vielen Zugriffen ausbremsen kann.

11.4 Klassenattribute und Klassenmethoden sowie statische Methoden

Die Methoden und Attribute, die wir bisher für unsere Klassen definiert haben, haben sich immer auf konkrete Instanzen bezogen. Insbesondere besitzt jede Instanz der Klasse `Konto` (siehe Seite 278) ihre eigenen Werte für ihre Attribute `Inhaber`, `Kontonummer`, `Kontostand`, `MaxTagesumsatz` und `UmsatzHeute`, und die Methoden `geldtransfer`, `einzahlen`, `auszahlen` und `zeige` können nur sinnvoll in Zusammenhang mit einer bereits bestehenden Instanz der Klasse `Konto` gerufen werden.

Solche Methoden und Attribute, die sich immer auf konkrete Instanzen beziehen, werden als *nicht-statisch* bezeichnet. Demgegenüber stehen die *statischen* Methoden und Attribute, die sich alle Instanzen einer Klasse teilen.

Statische Methoden

Zur Definition einer statischen Methode dient die Built-in Function `staticmethod`. Im folgenden Beispiel wird eine Klasse `A` definiert, die eine statische Methode `m` besitzt.

```
>>> def m():
...     print("Hallo statische Methode!")
>>> class A:
...     m = staticmethod(m)
>>> A.m()
Hallo statische Methode!
```

Eine statische Methode wird zunächst wie eine normale Funktion definiert und erst durch die Funktion `staticmethod` als statische Methode an eine Klasse gebunden.

Da eine statische Methode sich nicht auf eine Instanz der Klasse bezieht, benötigt sie keinen `self`-Parameter. Außerdem kann sie direkt von der Klasse aus gerufen werden, ohne dass zuvor eine Instanz erzeugt werden muss.

In Python werden statische Methoden häufig dazu genutzt, alternative Konstruktor anzubieten. Beispielsweise können wir unsere Klasse `Konto` um eine statische Methode erweitern, die ein Juniorkonto erzeugt. Dabei zeichnet sich ein Juniorkonto durch ein voreingestelltes niedriges Tageslimit aus.

```
def Juniorkonto(inhaber, kontonummer, kontostand):
    return Konto(inhaber, kontonummer, kontostand, 20)

class Konto:
    # Hier gehoeren die restlichen Methoden hin

    Juniorkonto = staticmethod(Juniorkonto)

J = Konto.Juniorkonto("Emil Peters", 436574, 67)
J.zeige()
```

Die Ausgabe dieses Programms sieht folgendermaßen aus.

```
Konto von Emil Peters
Aktueller Kontostand: 67.00 Euro
(Heute schon 0.00 von 20 Euro umgesetzt)
```

Die Methode `Juniorkonto` erzeugt also ihrerseits eine neue Instanz der Klasse `Konto`, wobei für das Tageslimit der feste Wert 20 übergeben wird.

Einen solchen alternativen Konstruktor bezeichnet man auch als *Factory-Function*.

Klassenmethoden und Klassenattribute

Neben den statischen Methoden, die losgelöst von einer Klasse existieren können, gibt es noch eine andere Art von Methoden, die sich nicht auf eine Instanz einer Klasse beziehen. Diese sogenannten *Klassenmethoden* erwarten als ersten Parameter eine Referenz auf die Klasse, für die sie aufgerufen werden. Um eine Klassenmethode zu definieren, dient die Built-in Function `classmethod`.

Im folgenden Beispiel werden drei Klassen A, B und C definiert, wobei C und B jeweils von A erben. Die Klasse A besitzt eine Klassenmethode `m`, die ausgibt, von welchem Typ die Instanz ist, mit der die Methode aufgerufen wurde.

```
class A:
    def m(cls):
        print("Ich bin", cls)
    m = classmethod(m)
```

```
class B(A):
    pass
```

```
class C(A):
    pass
```

```
A.m()
```

```
a = A()
```

```
b = B()
```

```
c = C()
```

```
a.m()
```

```
b.m()
```

```
c.m()
```

Die Ausgabe des Programms sieht folgendermaßen aus:

```
Ich bin <class '__main__.A'>
Ich bin <class '__main__.A'>
Ich bin <class '__main__.B'>
Ich bin <class '__main__.C'>
```

Mit `A.m()` rufen wir die Klassenmethode `m` von A auf, ohne uns dabei auf eine Instanz der Klasse zu beziehen. Wie die erste Zeile der Ausgabe uns zeigt, wurde für den ersten Parameter `cls` von `m` die Klasse A selbst übergeben.

Anschließend erzeugen wir Instanzen der Klassen A, B und C und rufen die Methode `m` für diese neuen Instanzen auf. Auch hier würde für den Parameter `cls` jeweils die Klasse der Instanz, mit der sie aufgerufen wurde, übergeben. Bei einer gewöhnlichen Methode wäre eine Referenz auf die Instanz selbst als erster Parameter übergeben worden.

Klassenmethoden sind ein sehr spezielles Konzept, das hauptsächlich in Zusammenhang mit *Metaklassen*⁸ verwendet wird. Da wir in diesem Buch keine Metaklassen behandeln, verweisen wir Sie auf die Python-Dokumentation.

Hinweis

Sowohl `staticmethod` als auch `classmethod` werden typischerweise als *Function Decorator* verwendet. Mehr zu diesem Thema erfahren Sie im Abschnitt 12.9, »Function Decorator«, ab Seite 373.



Neben den Klassenmethoden gibt es auch Attribute, die sich nicht auf Instanzen der Klasse, sondern auf die Klasse selbst beziehen. Auf diese Attribute kann sowohl über die Klasse selbst als auch über ihre Instanzen zugegriffen werden.

Das nachstehende Beispiel definiert eine Klasse `D` mit einem Klassenattribut `X`.

```
class D:
    X = 10

print(D.X)

d = D()
print(d.X)
```

Ein Klassenattribut kann also direkt durch eine Zuweisung innerhalb des Körpers der `class`-Anweisung erzeugt werden. Das Beispiel produziert die folgende Ausgabe:

```
10
10
```

Man kann also wieder sowohl direkt über die Klasse als auch über Instanzen der Klasse auf Klassenattribute zugreifen.

⁸ Eine Metaklasse ist eine Klasse, deren Instanz selbst wieder eine Klasse ist.

11.5 Magic Methods und Magic Attributes

Es gibt in Python eine Reihe spezieller Methoden und Attribute, um Klassen besondere Fähigkeiten zu geben. Die Namen dieser Methoden und Attribute beginnen und enden jeweils mit zwei Unterstrichen `__`. Im Laufe der letzten Abschnitte haben Sie bereits eine dieser sogenannten *Magic Methods* bzw. *Magic Attributes* kennen gelernt, nämlich den Konstruktor namens `__init__`.

Der Umgang mit diesen Methoden und Attributen ist insofern »magisch«, als dass sie in der Regel nicht direkt mit ihrem Namen benutzt, sondern bei Bedarf implizit im Hintergrund verwendet werden. Der Konstruktor `__init__` wird beispielsweise immer dann aufgerufen, wenn ein neues Objekt einer Klasse erzeugt wird, auch wenn kein expliziter Aufruf mit zum Beispiel `Klassenname.__init__()` an der entsprechenden Stelle steht.

Mit vielen Magic Methods lässt sich das Verhalten von Built-in Functions und Operatoren für die eigenen Klassen anpassen, sodass die Instanzen Ihrer Klassen beispielsweise sinnvoll mit den Vergleichsoperatoren `<` und `>` verglichen werden können.

Wir werden Ihnen im Folgenden eine Liste präsentieren, die häufig genutzte Magic Methods 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.

11.5.1 Allgemeine Magic Methods

Name	Beschreibung	Seite
<code>__init__(self, ...)</code>	Der Konstruktor einer Klasse. Wird beim Erzeugen einer neuen Instanz aufgerufen. Näheres können Sie in Abschnitt 11.1.2, »Der Konstruktor und die Erzeugung von Attributen«, ab Seite 277 nachlesen.	–
<code>__del__(self)</code>	Der Destruktor einer Klasse. Wird beim Zerstören einer Instanz aufgerufen.	304
<code>__repr__(self)</code>	Der Rückgabewert von <code>obj.__repr__</code> gibt an, was <code>repr(obj)</code> zurückgeben soll. Dies sollte nach Möglichkeit gültiger Python-Code sein, der beim Ausführen die Instanz <code>obj</code> erzeugt.	–

Tabelle 11.2 Allgemeine Magic Methods

Name	Beschreibung	Seite
<code>__str__(self)</code>	Der Rückgabewert von <code>obj.__str__</code> gibt an, was <code>str(obj)</code> zurückgeben soll. Dies sollte nach Möglichkeit eine für den Menschen lesbare Repräsentation von <code>obj</code> in Form einer <code>str</code> -Instanz sein.	–
<code>__bytes__(self)</code>	Der Rückgabewert von <code>obj.__bytes__</code> gibt an, was <code>bytes(obj)</code> zurückgeben soll. Dies sollte eine <code>bytes</code> -Instanz sein.	–
<code>__bool__(self)</code>	Die <code>__bool__</code> -Methode sollte einen Wahrheitswert (<code>True</code> oder <code>False</code>) zurückgeben, der angibt, wie das Objekt in eine <code>bool</code> -Instanz umzuwandeln ist. Ist <code>__bool__</code> nicht implementiert, wird stattdessen der Rückgabewert von <code>__len__</code> verwendet. Sind beide Methoden nicht vorhanden, werden alle Instanzen der betreffenden Klasse als <code>True</code> behandelt.	–
<code>__call__(self[, args...])</code>	Mit der <code>__call__</code> -Methode werden die Instanzen einer Klasse wie Funktionen aufrufbar.	304
<code>__complex__(self)</code>	Legt fest, welchen Wert die Built-in Function <code>complex</code> für eine Instanz der Klasse zurückgeben soll.	–
<code>__int__(self)</code>	Legt fest, welchen Wert die Built-in Function <code>int</code> für eine Instanz der Klasse zurückgeben soll.	–
<code>__float__(self)</code>	Legt fest, welchen Wert die Built-in Function <code>float</code> für eine Instanz der Klasse zurückgeben soll.	–
<code>__round__(self[, n])</code>	Legt fest, welchen Wert die Built-in Function <code>round</code> für eine Instanz der Klasse zurückgeben soll. Der Parameter <code>n</code> gibt dabei an, auf wie viele Nachkommastellen gerundet werden soll.	–
<code>__hash__(self)</code>	Die <code>__hash__</code> -Methode einer Instanz bestimmt, welchen Wert die Built-in Function <code>hash</code> für die Instanz zurückgeben soll.	305

Tabelle 11.2 Allgemeine Magic Methods (Forts.)

Name	Beschreibung	Seite
<code>__index__(self)</code>	Wenn ein Datentyp außerdem als Index benutzt werden soll, wie er beispielsweise für das Slicing benötigt wird, muss er die parameterlose Methode <code>__index__(self)</code> überschreiben. Der Rückgabewert von <code>__index__</code> muss eine Ganzzahl (<code>int</code>) sein.	–

Tabelle 11.2 Allgemeine Magic Methods (Forts.)

Nun werden einige der Methoden im Detail besprochen.

`__del__(self)`

Der Destruktor einer Klasse. Wird beim Zerstören einer Instanz aufgerufen.



Hinweis

Der Destruktor `__del__` wird erst dann gerufen, wenn keine Referenz mehr auf eine Instanz zeigt. Im folgenden Beispiel wird der Destruktor der Klasse A daher nur einmal gerufen:

```
>>> class A:
...     def __del__(self):
...         print("Hier spricht der Destruktor.")
...
>>> a = A()
>>> b = a
>>> del a
>>> del b
Hier spricht der Destruktor.
```

Die Anweisung `del x` ruft also nicht sofort `x.__del__`.

Außerdem wird *nicht* garantiert, dass der Destruktor aller noch im Speicher existierender Instanzen gerufen wird, wenn der Python-Interpreter beendet wird. Er eignet sich daher nur bedingt für Aufräumarbeiten wie dem Schließen von Dateien oder dem Beenden von Netzwerkverbindungen.

`__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 Konstruktor 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
```

`__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.

Bedingung für einen gültigen Hash-Wert ist, dass Objekte, die bei Vergleichen mit `==` als gleich angesehen werden, auch den gleichen Hash-Wert besitzen. Außerdem darf sich der Hash-Wert einer Instanz nicht zur Laufzeit ändern, weshalb er nur für immutable Datentypen sinnvoll definiert werden kann.

Hinweis

Eine Klasse, die `__hash__` implementiert, sollte zusätzlich die Methode `__eq__` implementieren.

Eine Instanz, die sowohl eine Methode `__hash__` also auch `__eq__` besitzt, wird *hashable* genannt. Es können nur *hashable* Instanzen als Schlüssel für ein Dictionary verwendet oder in Mengen gespeichert werden.



11.5.2 Zugriff auf Attribute anpassen

Die Methoden und Attribute 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 und Attribute nur selten überschreiben.

Name	Beschreibung	Seite
<code>__dict__</code>	Jede Instanz besitzt ein Attribut namens <code>__dict__</code> , das die Member der Instanz in einem Dictionary speichert.	306
<code>__getattr__(self, name)</code>	Wird dann aufgerufen, wenn das Attribut mit dem Namen <i>name</i> gelesen wird, aber nicht existiert. Die Methode <code>__getattr__</code> sollte entweder einen Wert zurückgeben, der für das Attribut gelten soll, oder einen <code>AttributeError</code> erzeugen.	–
<code>__getattribute__(self, name)</code>	Wird immer aufgerufen, wenn der Wert des Attributs mit dem Namen <i>name</i> gelesen wird, auch wenn das Attribut bereits existiert.	307
<code>__setattr__(self, name, value)</code>	Die Methode <code>__setattr__</code> wird immer dann aufgerufen, wenn der Wert eines Attributs per Zuweisung geändert oder ein neues Attribut erzeugt wird.	307
<code>__delattr__(self, name)</code>	Wird aufgerufen, wenn das Attribut mit dem Namen <i>name</i> per <code>del</code> gelöscht wird.	–
<code>__slots__</code>	Weist Python an, die Attribute einer Klasse speicherschonend zu verwalten.	308

Tabelle 11.3 Methoden und Attribute, um den Zugriff auf Attribute zu regeln

Für einige der Tabelleneinträge folgt eine ausführlichere Besprechung.

`__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"
```

`__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.

Hinweis

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")`.



11

`__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.

Hinweis

Verwenden Sie innerhalb von `__setattr__` niemals eine Zuweisung der Form `self.attribut = wert`, um die Attribute auf bestimmte Werte zu setzen, da dies eine endlose Rekursion bewirken würde: Bei jeder Zuweisung würde `__setattr__` erneut aufgerufen.

Um Attributwerte mit `__setattr__` zu verändern, können Sie auf das Attribut `__dict__` zurückgreifen: `self.__dict__["attribut"] = wert`.



`__slots__`

Instanzen in Python sind flexibel und mächtig, was das Arbeiten mit Python angenehm macht. Beispielsweise können Sie zur Laufzeit Attribute dynamisch hinzufügen.

```
>>> class A:
...     pass
>>> a = A()
>>> a.X = 10
>>> a.X
10
```

Diese Flexibilität wird durch Rechenzeit und Speicher erkauft, da für jede Instanz ein dict-Objekt erzeugt wird, um die Attribute zu verwalten.

Wenn Sie eine einfache Klasse mit wenigen Attributen definieren, von der es zur Laufzeit eine sehr große Anzahl von Instanzen gibt, kann dies unnötig Speicher vergeuden.

Um in einem solchen Fall Speicher zu sparen, kann man die Attribute der Instanzen einer Klasse bei der Klassendefinition einschränken. Dadurch geht zwar die Flexibilität verloren, dynamisch neue Attribute anlegen zu können, aber der Python-Interpreter kann die Attribute dann effizienter verwalten, sodass Speicher eingespart wird.

Im folgenden Beispiel wird eine Klasse B definiert, deren Instanzen nur die Attribute X und Y haben können.

```
>>> class B:
...     __slots__ = ("X", "Y")
...     def __init__(self):
...         self.X = 1
...         self.Y = 2
>>> b = B()
>>> b.X
1
>>> b.Y
2
>>> b.Z = 3
Traceback (most recent call last):
AttributeError: 'B' object has no attribute 'Z'
```

Wie Sie sehen, ist es nicht möglich, ein weiteres Attribut `Z` anzulegen. Dafür verbrauchen Instanzen der Klasse `B` weniger Speicher als die einer Klasse ohne `__slots__`-Definition.

Hinweis

Es gibt einige Besonderheiten, die den Umgang mit `__slots__` betreffen. Beispielsweise lässt sich eine `__slots__`-Definition nicht auf Subklassen vererben.

Genauere Informationen dazu finden Sie in der Python-Dokumentation.



11.5.3 Operatoren überladen

Ein *Operator* ist eine Vorschrift, die aus einer Reihe von *Operanden* einen neuen Wert berechnet. Ihnen sind in diesem Buch schon häufiger Operatoren begegnet, beispielsweise in Form von Rechenzeichen.

11

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

In diesem Beispiel wurde der Operator `+` verwendet, um die Summer zweier `int`-Instanzen zu berechnen. Der Operator `+` kann aber auch verwendet werden, um beispielsweise Strings miteinander zu verketteten.

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

Diese Mehrfachbelegung eines Operators wird dadurch ermöglicht, dass intern eine spezielle Methode gerufen wird, die festlegt, was der Operator bewirken soll. Im Falle des Operators `+` ist dies die Methode `__add__`. Die beiden folgenden Ausdrücke sind daher gleichwertig.

```
>>> 1 + 2
3
>>> (1).__add__(2)
3
```

Die Klammern um die `1` sind deshalb notwendig, da ein Punkt direkt hinter der `1` als Dezimalpunkt interpretiert werden würde.

Sie können also auch für Ihre eigenen Klassen Operatoren definieren, indem Sie die dahinter stehenden Methoden überschreiben.

Als Beispiel werden wir eine kleine Klasse zum Verwalten von Längenangaben mit Einheiten implementieren, die die Operatoren für Addition und Subtraktion unterstützt. Dabei dient die Methode `__sub__` für die Implementation des Operators `-`.

Die Klasse wird intern alle Maße für die Berechnungen in Meter umwandeln. Ihre Definition sieht dann folgendermaßen aus:

```
class Laenge:
    Umrechnung = {"m" : 1, "dm" : 0.1, "cm" : 0.01,
                  "mm" : 0.001, "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 Längemaße 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.

Betrachten wir einmal folgende Anwendung der Klasse `Laenge`.

```
>>> a1 = Laenge(5, "cm")
>>> a2 = Laenge(3, "dm")
>>> print(a1 + a2)
35.000000 cm
>>> print(a2 + a1)
3.500000 dm
```

Wie Sie sehen, funktionieren die Rechnungen wie gewünscht. Bemerkenswert ist, dass sich die Einheit in der Ausgabe je nach Operandenreihenfolge verändert. Dies resultiert daraus, dass unsere Klasse `Laenge` immer die Einheit des weiter links stehenden Operanden als Einheit des Ergebnisses verwendet.

Neben den Operatoren `+` und `-` gibt es in Python eine Reihe weiterer Operatoren. Dabei unterscheiden wir mehrere Typen von Operatoren, wie es die nachstehende Tabelle zeigt.

Kategorie	Beschreibung	Beispiele	Seite
Vergleichsoperatoren	Vergleichen zwei Instanzen miteinander und liefern eine <code>bool</code> -Instanz als Ergebnis.	<code><, >, =</code>	311
binäre arithmetische Operatoren	Operatoren, die auf zwei Operanden angewendet werden. Der Rückgabotyp hängt von dem Operator und den Operanden ab.	<code>+, -, *, /</code>	313
binäre Operatoren mit umgekehrter Operandenreihenfolge	Operatoren, die auf zwei Operanden angewendet werden. Der Rückgabotyp hängt von dem Operator und den Operanden ab.	<code>+, -, *, /</code>	314
erweiterte Zuweisungen	Operatoren, die eine Operation und eine Zuweisung verbinden	<code>+=, -=, *=, /=</code>	315
unäre Operatoren	Operatoren mit nur einem Operanden, wie beispielsweise Vorzeichen	<code>+, -</code>	316

Tabelle 11.4 Arten von Operatoren

Vergleichsoperatoren

Die folgenden *Magic Methods* dienen dazu, das Verhalten der Vergleichsoperatoren für die Klasse anzupassen.

Um beispielsweise zwei Instanzen der Klasse `Konto` (siehe Seite 278) zu vergleichen, kann die Kontonummer herangezogen werden. Damit gibt es eine sinnvolle Interpretation für den Vergleich mit `==` bei Konten. Die Methode für Vergleiche mit `==` heißt `__eq__` (von engl. *equals* »ist 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 Herkunftsspalte kann Ihnen helfen, sich die Methodennamen und ihre Bedeutung besser zu merken.

Operator	Methode	Herkunft
<	<code>__lt__(self, other)</code>	<i>less than</i> (dt. »kleiner als«)
<=	<code>__le__(self, other)</code>	<i>less or equal</i> (dt. »kleiner oder gleich«)
==	<code>__eq__(self, other)</code>	<i>equal</i> (dt. »gleich«)
!=	<code>__ne__(self, other)</code>	<i>not equal</i> (dt. »ungleich«)
>	<code>__gt__(self, other)</code>	<i>greater than</i> (dt. »größer als«)
>=	<code>__ge__(self, other)</code>	<i>greater or equal</i> (dt. »größer oder gleich«)

Tabelle 11.5 Die Magic Methods für Vergleiche

Hinweis

Wenn eine Klasse keine der Methoden `__eq__` oder `__ne__` implementiert, werden Instanzen der Klasse mittels `==` und `!=` anhand ihrer Identität miteinander verglichen.



Ist es nicht möglich, die von `self` referenzierte Instanz mit `other` zu vergleichen, sollte `NotImplemented` zurückgegeben werden.

Binäre Operatoren

Ein binärer Operator ist ein Operator, der zwei Operanden verarbeitet. Beispiele für binäre Operatoren sind `+`, `-`, `*` und `/`.

Alle Methoden zum Überladen von binären Operatoren erwarten einen Parameter, der den zweiten Operanden referenziert. Als erster Operator wird immer diejenige Instanz verwendet, die für den Parameter `self` übergeben wurde.

Ihr Rückgabewert muss eine neue Instanz sein, die das Ergebnis der Rechnung enthält.

Ein Beispiel für die Verwendung binärer Operatoren finden Sie auf Seite 310.

In der nachstehenden Tabelle sind alle binären Operatoren⁹ und die entsprechenden Magic Methods aufgelistet:

⁹ Selbstverständlich sind die Vergleichsoperatoren auch binäre Operatoren. Aus Gründen der Übersicht haben wir sie aber separat auf Seite 311 besprochen.

Operation	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>divmod()</code>	<code>__divmod__(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 11.6 Magic Methods für binäre Operatoren

Binäre arithmetische 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.

Allerdings muss der zweite Operand eine spezielle Methode für vertauschte Operanden implementieren.¹⁰ Die folgende Tabelle listet alle dafür verfügbaren Methodenamen und die entsprechenden Operatoren auf, wobei es für jeden der binären Operatoren eine Entsprechung gibt.

¹⁰ Das hier auf die Reihenfolge geachtet wird, ist wichtig, denn nicht bei allen Operationen ist die Reihenfolge der Operanden egal. Beispielsweise macht es einen Unterschied, ob `"x" + "y"` oder `"y" + "x"` ausgewertet wird.

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>divmod()</code>	<code>__rdivmod__(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 11.7 Magic Methods für binäre Operatoren des rechten Operanden

Für nicht unterstützte Werte von `other` sollte auch hier `NotImplemented` zurückgegeben werden.

Erweiterte Zuweisungen

Es können auch die erweiterten Zuweisungen überladen werden, die eine arithmetische Operation mit einer Zuweisung verbinden. Bei einer erweiterten Zuweisung wird dem jeweiligen Operator ein Gleichheitszeichen nachgestellt:

```
>>> a = 10
>>> a += 5
>>> a
15
```

Standardmäßig verwendet Python für solche Zuweisungen den Operator selbst, sodass `a += 5` intern wie `a = a + 5` ausgeführt wird. Diese Vorgehensweise hat für komplexe Datentypen wie beispielsweise Listen den Nachteil, dass immer eine neue Liste erzeugt werden muss. Deshalb können Sie gezielt die erweiterten Zuweisungen

anpassen, um die Effizienz des Programms zu verbessern. Außerdem wird bei der Anwendung einer erweiterten Zuweisung auf eine Instanz eines mutable Datentyps erwartet, dass die Instanz selbst verändert und keine neue erzeugt wird.

In der folgenden Tabelle stehen alle Operatoren für erweiterte Zuweisungen und die entsprechenden Methoden:

Operator	Magic Method
<code>+=</code>	<code>__iadd__(self, other)</code>
<code>-=</code>	<code>__isub__(self, other)</code>
<code>*=</code>	<code>__imul__(self, other)</code>
<code>/=</code>	<code>__itruediv__(self, other)</code>
<code>//=</code>	<code>__ifloordiv__(self, other)</code>
<code>**=</code>	<code>__ipow__(self, other[, modulo])</code>
<code>%=</code>	<code>__imod__(self, other)</code>
<code>>>=</code>	<code>__ilshift__(self, other)</code>
<code><<=</code>	<code>__irshift__(self, other)</code>
<code>&=</code>	<code>__iand__(self, other)</code>
<code> =</code>	<code>__ior__(self, other)</code>
<code>^=</code>	<code>__ixor__(self, other)</code>

Tabelle 11.8 Methoden für die erweiterte Zuweisung



Hinweis

Auch wenn die Operatoren für die erweiterte Zuweisung die Instanz `self` verändern, müssen sie eine Referenz auf das Ergebnis der Berechnung, in diesem Fall also `self`, zurückgeben.

Unäre Operatoren

Mit den folgenden Methoden werden die unären Operatoren überladen. Unäre Operatoren erwarten im Gegensatz zu den binären Operatoren nur einen Operanden.

Zu den unären Operatoren zählen die Vorzeichen `+` und `-`, die Built-in Function `abs` zur Bestimmung des absoluten Wertes 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 11.9 Magic Methods für die unären Operatoren

Die Methoden sollten bei erfolgreicher Rechnung das Ergebnis zurückgeben. Ist es nicht möglich, den Operanden `other` zu verarbeiten, sollte `NotImplemented` zurückgegeben werden.

11.5.4 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.

Duck-Typing

Dieses Konzept, den Typ einer Instanz anhand der vorhandenen Methoden und nicht der Klasse zu beurteilen, wird *Duck-Typing* genannt. Die Bezeichnung ist an ein Gedicht von James Whitcomb Riley¹¹ angelehnt, dessen deutsche Übersetzung folgendermaßen lautet:

»Wenn ich einen Vogel sehe, der wie eine Ente läuft, schwimmt und quakt, so nenne ich diesen Vogel eine Ente.«¹²

Übertragen auf die Instanzen in einem Programm, bedeutet dies, dass beispielsweise alle Instanzen wie Zahlen behandelt werden, die sich genauso wie andere Zahlen verhalten. Ob es sich um Instanzen der Typen `int`, `float` oder `complex` handelt, ist dabei egal. Insbesondere ist es möglich, eine eigene Klasse zu definieren, die sich ebenfalls wie eine Zahl verhält und somit auch wie eine solche behandelt wird.

¹¹ James Whitcomb Riley (1849-1916) war ein US-amerikanischer Dichter.

¹² Englisch Original: *»When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.«*

Dabei gilt das Prinzip, nicht mit Gewalt möglichst alle Operatoren und Methoden zu implementieren, sondern nur solche, die für die Klasse Sinn ergeben. Alle anderen Methoden sollten entweder gar nicht implementiert werden oder `NotImplemented` zurückgeben.

Sie werden im Folgenden die Methoden kennen lernen, die ein Datentyp implementieren muss, um nach dem Duck-Typing ein numerischer Datentyp zu sein. Außerdem werden die Schnittstellen von Sequenzen und Mappings behandelt.

Numerische Datentypen emulieren

Ein numerischer Datentyp sollte möglichst viele arithmetische Operatoren implementieren. Außerdem kann er Methoden definieren, um ihn in andere numerische Datentypen zu überführen, falls dies möglich ist.

Die folgende Tabelle gibt einen Überblick über die möglichen Methoden.

Name oder Kurzbeschreibung	Beschreibung	Seite
arithmetische Operatoren	Rechenoperatoren für die Summe, Differenz, den Quotienten usw.	309
<code>__complex__</code>	Umwandlung nach <code>complex</code>	–
<code>__int__</code>	Umwandlung nach <code>int</code>	–
<code>__float__</code>	Umwandlung nach <code>float</code>	–
<code>__round__</code>	Rundung des Wertes	–
<code>__index__</code>	Verwendung als Index	–

Tabelle 11.10 Besondere Methoden, die ein numerischer Datentyp nach Möglichkeit definieren sollte.

Kontextmanager implementieren

Unter einem *Kontextmanager* versteht man eine Instanz, die in Zusammenhang mit der `with`-Anweisung verwendet werden kann. Näheres zu `with` erfahren Sie in Abschnitt 12.7, »Die `with`-Anweisung«, ab Seite 368.

Um mit `with` als Kontextmanager verwendet werden zu können, müssen zwei Methoden implementiert werden, die die folgende Tabelle auflistet.

Name	Beschreibung	Seite
<code>__enter__(self)</code>	Baut den Kontext auf und gibt das Objekt zurück, mit dem gearbeitet werden soll.	370
<code>__exit__(self, ...)</code>	Räumt nach Verlassen des Körpers der <code>with</code> -Anweisung auf.	371

Tabelle 11.11 Methoden für Kontextmanager

Container emulieren

Mithilfe der folgenden Methoden ist es möglich, eigene Container-Datentypen zu erzeugen. Unter einem *Container* (engl. *to contain*, dt. »enthalten«), versteht man eine Instanz, die ihrerseits weitere Instanzen enthalten kann. Beispiele sind die Liste, das Dictionary oder die Menge.

Dabei wird grundsätzlich unterschieden zwischen *sequentiellen Containern*, deren Elemente sich über ganze Zahlen¹³ ansprechen lassen, und *Mapping-Containern*, deren Indizes beliebige Gestalt haben können.

Methoden für allgemeine Container

Zunächst gibt es einen Satz von Methoden, den sowohl sequentielle als auch Mapping-Container implementieren sollten.

Methode	Beschreibung	Seite
<code>__len__(self)</code>	Liefert die Anzahl der Elemente in dem Container als <code>int</code> -Instanz zurück.	–
<code>__getitem__(self, key)</code>	Liest ein Element oder mehrere Elemente aus dem Container, wenn der Operator <code>[]</code> verwendet wird.	–
<code>__setitem__(self, key, value)</code>	Verändert das Element des Containers, das dem Schlüssel <code>key</code> zugeordnet ist.	–
<code>__delitem__(self, key)</code>	Entfernt das Element mit dem Index <code>key</code> aus dem Container.	–

Tabelle 11.12 Methoden, die alle Container-Datentypen implementieren können

¹³ Dabei sollten die Elemente, bei 0 beginnend, fortlaufend durchnummeriert sein.

Methode	Beschreibung	Seite
<code>__iter__(self)</code>	Muss einen Iterator über die Werte des sequentiellen Containers bzw. die Schlüssel des Mapping-Containers zurückgeben. Näheres über Iteratoren erfahren Sie im Abschnitt 12.4, »Iteratoren«, ab Seite 353.	–
<code>__contains__(self, item)</code>	Prüft, ob ein Element in dem Container enthalten ist.	–

Tabelle 11.12 Methoden, die alle Container-Datentypen implementieren können (Forts.)

Nun stellen wir Ihnen die Methoden vor, die speziell für sequentielle Container vorgesehen sind.

Methoden für sequentielle Container

Alle sequentiellen Container sollten zusätzlich zu den allgemeinen Methoden für Container die Methoden für die Addition (Verkettung) und Multiplikation (Wiederholung) implementieren

Methoden	Beschreibung	Seite
<code>__add__(self, other)</code>	Verkettet die Sequenz mit der Sequenz <i>other</i> .	313
<code>__radd__(self, other)</code>	Dabei sollte <code>__iadd__</code> im Fall eines mutable Datentyps die von <i>self</i> referenzierte Instanz verändern, also <i>in-place</i> arbeiten.	314
<code>__iadd__(self, other)</code>		315
<code>__mul__(self, other)</code>	Sollte eine Sequenz erzeugen, die aus der von <i>self</i> referenzierten Instanz dadurch hervorgeht, dass sie <i>other</i> -mal wiederholt wird.	313
<code>__rmul__(self, other)</code>	Für Strings sieht das beispielsweise folgendermaßen aus.	314
<code>__imul__(self, other)</code>	<pre>>>> 5*"a" 'aaaaa'</pre>	315

Tabelle 11.13 Allgemeine Methoden für sequentielle Container

Mutable Sequenzen sollten zusätzlich noch die folgenden Methoden definieren. Für Beispiele dieser Methoden können Sie sich den Datentyp `list` im Abschnitt 7.5.2, »Listen – `list`«, ab Seite 128 anschauen.

Methoden	Beschreibung	Seite
<i>append(x)</i>	Hängt <i>x</i> an das Ende der Sequenz an.	–
<i>count(x)</i>	Zählt die Vorkommen von <i>x</i> in der Sequenz.	–
<i>index(x[, i[, j]])</i>	Liefert den Index des ersten Vorkommens von <i>x</i> in der Sequenz. Mit den optionalen Parametern <i>i</i> und <i>j</i> kann dabei der Suchbereich eingegrenzt werden.	–
<i>extend(s)</i>	Erweitert die Sequenz um die Elemente der Sequenz <i>s</i> .	–
<i>insert(i, x)</i>	Fügt das Element <i>x</i> an der Stelle <i>i</i> in die Sequenz ein.	–
<i>pop([i])</i>	Liefert das <i>i</i> -te Element der Sequenz und entfernt es aus dieser. Wird <i>i</i> nicht angegeben, wird das letzte Element zurückgegeben und anschließend entfernt.	–
<i>remove(x)</i>	Entfernt das erste Vorkommen von <i>x</i> in der Sequenz aus dieser.	–
<i>__reversed__(self)</i>	Liefert einen Iterator zum umgekehrten Durchlaufen des sequentiellen Datentyps. Dabei wird die Sequenz nicht verändert	–
<i>reverse()</i>	Dreht die Reihenfolge der Sequenz <i>in-place</i> um.	–
<i>sort([key[, reverse]])</i>	Sortiert die Sequenz <i>in-place</i> .	

Tabelle 11.14 Methoden für mutable Sequenzen

Methoden für Mapping-Container

Alle Mapping-Datentypen sollten zusätzlich zu den Methoden für allgemeine Container weitere Methoden implementieren, die in der nachstehenden Tabelle aufgelistet sind:¹⁴

¹⁴ Wenn Ihnen die hier angegebenen Beschreibungen nicht ausführlich genug sind, können Sie sich noch einmal den Abschnitt 7.6.1, »Dictionary – dict«, ab Seite 174 ansehen.

Methode	Bedeutung
<i>m.keys()</i>	Gibt einen Iterator über die Schlüssel von <i>m</i> zurück.
<i>m.values()</i>	Gibt einen Iterator über die Werte von <i>m</i> zurück.
<i>m.items()</i>	Gibt einen Iterator über die Schlüssel-Wert-Paare von <i>m</i> zurück.
<i>m.has_key(k)</i>	Prüft, ob der Schlüssel <i>k</i> in <i>m</i> existiert.
<i>m.get(k[, d])</i>	Wenn der Schlüssel <i>k</i> in <i>m</i> existiert, wird <i>m[k]</i> zurückgegeben, ansonsten <i>d</i> .
<i>m.clear()</i>	Entfernt alle Elemente aus <i>m</i> .
<i>m.setdefault(k[, x])</i>	Wenn der Schlüssel <i>k</i> in <i>m</i> existiert, wird <i>m[k]</i> zurückgegeben. Gibt es den Schlüssel <i>k</i> nicht in <i>m</i> , wird <i>m[k]</i> auf den Wert <i>x</i> gesetzt und <i>x</i> zurückgegeben.
<i>m.pop(k[, d])</i>	Wenn der Schlüssel <i>k</i> in <i>m</i> existiert, wird <i>m[k]</i> zurückgegeben und danach mit <i>del</i> gelöscht. Gibt es den Schlüssel <i>k</i> nicht in <i>m</i> , so wird <i>d</i> zurückgegeben.
<i>m.popitem()</i>	Gibt ein willkürlich ausgewähltes Schlüssel-Wert-Paar von <i>m</i> zurück und entfernt es anschließend aus <i>m</i> .
<i>m.copy()</i>	Gibt eine Kopie von <i>m</i> zurück.
<i>m.update(b)</i>	Übernimmt alle Schlüssel-Wert-Paare von <i>b</i> in <i>m</i> . Vorhandene Einträge werden dabei überschrieben.

Tabelle 11.15 Methoden für Mapping-Typen

11.6 Built-in Functions für Objektorientierung

Die folgende Tabelle listet diejenigen Built-in Functions auf, die sich speziell auf Objekte und Klasse beziehen.

Name	Beschreibung	Seite
<i>getattr(object, name[, default])</i>	Liefert den Wert des Attributs <i>name</i> von der Instanz <i>object</i> .	323
<i>setattr(object, name, value)</i>	Setzt den Wert des Attributs <i>name</i> von der Instanz <i>object</i> auf den Wert <i>value</i> .	324

Tabelle 11.16 Built-in Functions für Objektorientierung

Name	Beschreibung	Seite
<i>hasattr(object, name)</i>	Prüft, ob die Instanz <i>object</i> das Attribut <i>name</i> besitzt. Ist das Attribut vorhanden, wird <code>True</code> , ansonsten <code>False</code> zurückgegeben.	–
<i>delattr(object, name)</i>	Entfernt das Attribut <i>name</i> von der Instanz <i>object</i> .	324
<i>isinstance(object, classinfo)</i>	Prüft, ob die Instanz <i>object</i> eine Instanz der von <i>classinfo</i> beschriebenen Klasse(n) ist.	325
<i>issubclass(class_, classinfo)</i>	Prüft, ob die Klasse <i>class_</i> eine Tochterklasse der von <i>classinfo</i> beschriebenen Klasse(n) ist.	326

Tabelle 11.16 Built-in Functions für Objektorientierung (Forts.)

11.6.1 Funktionen für die Verwaltung der Attribute einer Instanz

Um die Attribute einer Instanz zu verwalten, dienen die Funktionen `setattr`, `getattr` und `delattr`. Dabei kann auf ein Attribut zugegriffen werden, indem der Name des Attributs als String übergeben wird.

getattr(object, name[, default])

Diese Funktion gibt das Attribut mit dem Namen *name* von der Instanz *object* zurück, sofern dieses Attribut existiert. Falls *default* übergeben wurde, wird der Wert von *default* zurückgegeben, wenn das Attribut *name* nicht vorhanden ist.

Das folgende Beispiel greift mittels `getattr` auf ein vorhandenes und ein nichtvorhandenes Attribut zu. Dabei ist der Aufruf `getattr(a, "X")` gleichbedeutend mit `a.X`.

```
class A:
    def __init__(self):
        self.X = 42

a = A()
print(getattr(a, "X"))
42
print(getattr(a, "Y", 404))
404
```

Ist das Attribut nicht vorhanden und wird kein Wert für *default* übergeben, wirft `getattr` eine `AttributeError`-Exception.


```
print(getattr(a, "Y"))
Traceback (most recent call last):
AttributeError: 'A' object has no attribute 'Y'
```

setattr(object, name, value)

Diese Funktion setzt den Wert des Attributs *name* von der Instanz *object* auf den Wert *value*.

Das folgende Beispiel definiert eine Klasse, die zehn Attribute in einer Schleife anlegt.

```
class B:
    def __init__(self):
        for i in range(10):
            setattr(self, "X{}".format(i), i)

b = B()
print(b.X3)
3
print(b.X8)
8
```

Ein Aufruf der Form `setattr(a, "X", wert)` ist dabei äquivalent zu `a.X = wert`.

delattr(object, name)

Mit `delattr` können Attribute einer Instanz gezielt gelöscht werden. Wir betrachten erneut die Klasse `B` aus dem obigen Beispiel zu `setattr`:

```
b = B()
print(b.X4)
4
delattr(b, "X4")
print(b.X4)
Traceback (most recent call last):
AttributeError: 'B' object has no attribute 'X4'
```

Der Aufruf `delattr(b, "X4")` hat dieselbe Wirkung wie `del b.X4`.

11.6.2 Funktionen für Informationen über die Klassenhierarchie

Um die folgenden Methoden zu demonstrieren, gehen wir von folgendem Beispielprogramm aus:

```

class A:
    pass

class B(A):
    pass

class C(B):
    pass

class D:

    pass

a = A()
b = B()
c = C()
d = D()

```

11

isinstance(object, classinfo)

Diese Funktion prüft, ob *object* eine Instanz der Klasse(n) *classinfo* ist, und liefert dementsprechend entweder `True` oder `False` zurück. Dabei kann für den Parameter *classinfo* entweder eine einzelne Klasse oder ein Tupel mehrerer Klassen übergeben werden.

```

>>> isinstance(a, A)
True
>>> isinstance(a, (B,C))
False
>>> isinstance(a, (A,B,C))
True

```

Der Rückgabewert von `isinstance` ist auch dann `True`, wenn *object* die Instanz einer Klasse ist, die von einer der Klassen in *classinfo* erbt.

```

>>> isinstance(c, A)
True
>>> isinstance(c, (B,D))
True

```

issubclass(class_, classinfo)

Mit der Funktion *issubclass* kann geprüft werden, ob die Klasse¹⁵ *class_* von einer der Klassen in *classinfo* abgeleitet wurde. Genau wie bei *isinstance* kann für *classinfo* entweder eine einzelne Klasse oder ein Tupel mehrerer Klassen übergeben werden.

```
>>> issubclass(B,A)
True
>>> issubclass(B,(D,A))
True
>>> issubclass(A,C)
False
>>> issubclass(D,(A,B,C))
False
```

11.7 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 Klassen¹⁶ 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 implementiert eine Subklasse von *list*, die automatisch ihre Elemente nach jeder Veränderung sortiert. Dazu werden alle Methoden von *list*, die Elemente in die Liste einfügen, so überschrieben, dass im Anschluss an die jeweilige Operation die Liste sortiert wird. Da die Liste sortiert ist, soll ihre Reihenfolge nicht umgekehrt werden können, weshalb der Methode *reverse* ihre Funktionalität genommen wird.

Für die Realisierung der Methoden wird natürlich weiterhin auf die Implementation von *list* zurückgegriffen.

```
class SortierteListe(list):
    def __init__(self, *args, **kwargs):
        list.__init__(self, *args, **kwargs)
        self.sort()
```

¹⁵ Der Unterstrich am Ende von *class_* wurde deshalb eingefügt, damit der Parameter nicht mit dem Schlüsselwort *class* zur Definition einer Klasse kollidiert.

¹⁶ Der Datentyp von Klassen-Instanzen sind sogenannte Metaklassen, deren Verwendung in diesem Buch nicht behandelt wird.

```

def __setitem__(self, key, value):
    list.__setitem__(self, key, value)
    self.sort()

def append(self, value):
    list.append(self, value)
    self.sort()

def extend(self, sequence):
    list.extend(self, sequence)
    self.sort()

def insert(self, i, x):
    list.insert(self, i, x)
    self.sort()

def reverse(self):
    pass

def __iadd__(self, s):
    erg = list.__iadd__(self, s)
    self.sort()
    return erg

def __imul__(self, n):
    erg = list.__imul__(self, n)
    self.sort()
    return erg

```

11

Das nächste Beispielprogramm illustriert die Verwendung der neuen Klasse.

```

l = SortierteListe([6,4,3])
print(l)
l.append(2)
print(l)
l.extend([67,0,-56])
print(l)
l += [100,5]
print(l)
l *= 2
print(l)

```

Wie Sie sehen, kann die neue Klasse genauso wie `list` verwendet werden.¹⁷ Erst die Ausgaben verdeutlichen den Unterschied in der Funktionsweise.

```
[3, 4, 6]
```

```
[2, 3, 4, 6]
```

```
[-56, 0, 2, 3, 4, 6, 67]
```

```
[-56, 0, 2, 3, 4, 5, 6, 67, 100]
```

```
[-56, -56, 0, 0, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 67, 67, 100, 100]
```

Obwohl in willkürlicher Reihenfolge eingefügt wurde, sind die Elemente der Liste nach jeder Operation sortiert.

¹⁷ Wir nehmen dabei an, dass eine Ordnungsrelation für die Elemente der Liste existiert. Wenn Sie Elemente einfügen, die sich nicht sortieren lassen, führt dies zu einem Fehler.

Kapitel 12

Weitere Spracheigenschaften

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

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.

12

12.1 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. Die Information, dass ein Fehler aufgetreten ist, muss also durch die Aufrufhierarchie nach oben geschleust werden, 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 ist 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 seltene Fälle zu behandeln.

Für solche Fälle unterstützt Python ein Programmierkonzept, das *Exception Handling* (dt. »Ausnahmebehandlung«) genannt wird. Im Fehlerfall erzeugt unsere Unterfunktion dann eine sogenannte Exception und wirft sie, bildlich gesprochen, nach oben. 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 4.4, »Der Fehlerfall« (S. 56), 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 ein Objekt, das Attribute und Methoden zur Klassifizierung und Bearbeitung eines 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.

12.1.1 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. Sie 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 folgende Tabelle listet die eingebauten Exception-Typen auf und nennt übliche Umstände, unter denen sie auftreten. Die Vererbungshierarchie ist dabei durch die Einrückungstiefe des Typnamens angedeutet.

Exception-Typ	Auftreten
<i>BaseException</i>	Basisklasse aller eingebauter Exception-Typen (siehe S. 334)
<i>SystemExit</i>	Beim Beenden des Programms durch <code>sys.exit</code> . Diese Exception verursacht keinen Traceback.
<i>KeyboardInterrupt</i>	beim Beenden des Programms per Tastenkombination (üblicherweise mit <code>[Strg] + [C]</code>)
<i>GeneratorExit</i>	beim Beenden eines Generators
<i>Exception</i>	Basisklasse aller »normalen« Exceptions. Auch alle selbstdefinierten Exception-Typen sollten von <code>Exception</code> erben.
<i>StopIteration</i>	beim Aufruf der Methode <code>next</code> eines Iterators, wenn kein nächstes Element existiert
<i>ArithmeticError</i>	Basisklasse der Exception-Typen für Fehler in arithmetischen Operationen
<i>FloatingPointError</i>	bei einer fehlschlagenden Gleitkommaoperation
<i>OverflowError</i>	bei einer arithmetischen Berechnung, deren Ergebnis zu groß für den verwendeten Datentyp ist
<i>ZeroDivisionError</i>	bei einer Division durch Null
<i>AssertionError</i>	bei einer fehlschlagenden <code>assert</code> -Anweisung (siehe Abschnitt 12.10)
<i>AttributeError</i>	beim Zugriff auf ein nicht existierendes Attribut
<i>BufferError</i>	bei einer fehlgeschlagenen Operation auf einem Buffer-Datentyp
<i>EnvironmentError</i>	Basisklasse der Exception-Typen für Fehler, die außerhalb des Python-Systems passieren

Tabelle 12.1 Hierarchie der eingebauten Exception-Typen

Exception-Typ	Auftreten
<i>IOError</i>	bei einer fehlschlagenden I/O-Operation, beispielsweise einer Bildschirmein- oder -ausgabe
<i>OSError</i>	bei Fehlern im Modul <code>os</code>
<i>WindowsError</i>	bei einem Windows-spezifischen Fehler
<i>VMSError</i>	bei einem VMS ¹ -spezifischen Fehler
<i>EOFError</i>	wenn die eingebaute Funktion <code>input</code> ein Dateiende (EOF für <i>End-of-File</i>) signalisiert bekommt, ohne vorher Daten lesen zu können
<i>ImportError</i>	bei einer fehlschlagenden <code>import</code> -Anweisung, beispielsweise, wenn kein Modul mit dem angegebenen Namen existiert
<i>LookupError</i>	Basisklasse der Exception-Typen für Fehler bei Schlüssel- oder Indexzugriffen
<i>IndexError</i>	bei einem Sequenzzugriff mit einem ungültigen Index
<i>KeyError</i>	bei einem Dictionaryzugriff mit einem ungültigen Schlüssel
<i>MemoryError</i>	wenn nicht genügend Speicher zur Ausführung einer Operation vorhanden ist
<i>NameError</i>	wenn ein unbekannter lokaler oder globaler Bezeichner verwendet wird
<i>UnboundLocalError</i>	wenn in einer Funktion oder Methode eine lokale Referenz verwendet wird, bevor ihr eine Instanz zugewiesen wurde
<i>ReferenceError</i>	beim Zugriff auf ein Objekt, das von der Garbage Collection bereits freigegeben wurde
<i>RuntimeError</i>	bei einem Fehler, der durch keinen der anderen Exception-Typen abgedeckt wird

Tabelle 12.1 Hierarchie der eingebauten Exception-Typen (Forts.)

¹ VMS (für *Virtual Memory System*) ist ein Betriebssystem der Firma DEC, welches heutzutage unter dem Namen OpenVMS von HP weiterentwickelt wird.

Exception-Typ	Auftreten
<i>NotImplementedError</i>	Basisklassen werfen diese Exception aus Methoden heraus, die von einer abgeleiteten Klasse implementiert werden müssen.
<i>SyntaxError</i>	bei einem Syntaxfehler im Programm bzw. in einem an <code>exec</code> oder <code>eval</code> übergebenen Code
<i>IndentationError</i>	Basisklasse der Exception-Typen für Fehler durch unkorrektes Einrücken des Quellcodes
<i>TabError</i>	bei fehlerhafter Vermischung von Tabulatoren und Leerzeichen zur Einrückung
<i>SystemError</i>	bei einem internen Fehler im Python-Interpreter
<i>TypeError</i>	bei einer Operation auf einer Instanz eines dafür ungeeigneten Datentyps
<i>ValueError</i>	bei einer Operation auf einer Instanz, die zwar einen passenden Typ, aber einen unpassenden Wert hat
<i>UnicodeError</i>	bei fehlerhafter En- oder Dekodierung von Unicode-Zeichen
<i>UnicodeDecodeError</i>	bei einem Unicode-spezifischen Fehler in einer Dekodierung
<i>UnicodeEncodeError</i>	bei einem Unicode-spezifischen Fehler in einer Enkodierung
<i>UnicodeTranslateError</i>	bei einem Unicode-spezifischen Fehler in einer Übersetzung
<i>Warning</i>	Basisklasse der Warnungs-Typen
<i>DeprecationWarning</i>	bei der Verwendung einer als veraltet eingestuft Funktionalität
<i>PendingDeprecationWarning</i>	bei der Verwendung einer als zukünftig veraltet eingestuften Funktionalität
<i>RuntimeWarning</i>	bei Umständen, die zu Laufzeitfehlern führen könnten, beispielsweise Versionsdifferenzen
<i>SyntaxWarning</i>	bei einer Syntax, die problematisch, aber gültig ist

Tabelle 12.1 Hierarchie der eingebauten Exception-Typen (Forts.)

Exception-Typ	Auftreten
<i>UserWarning</i>	Basisklasse für selbstdefinierte Warnungen
<i>FutureWarning</i>	bei der Verwendung von Funktionalitäten, die sich in zukünftigen Versionen verändern werden
<i>ImportWarning</i>	bei Problemen, die durch das Einbinden eines Moduls entstehen
<i>UnicodeWarning</i>	bei Unicode-spezifischen Problemen
<i>BytesWarning</i>	bei bytes-spezifischen Problemen
<i>ResourceWarning</i>	bei Problemen mit der Nutzung von Ressourcen

Tabelle 12.1 Hierarchie der eingebauten Exception-Typen (Forts.)

BaseException

Die Klasse `BaseException` ist die Basisklasse aller Exceptions und stellt damit eine Grundfunktionalität bereit, die für alle Exception-Typen vorhanden ist. Aus diesem Grund soll sie hier besprochen werden.

Die Grundfunktionalität, die `BaseException` bereitstellt, besteht aus einem wesentlichen Attribut namens `args`. Dabei handelt es sich um ein Tupel, in dem alle Parameter abgelegt werden, die der Exception bei ihrer Instanziierung 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.

12.1.2 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 von `BaseException` abgeleiteten Klasse 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, sodass sie nicht in einem Traceback enden, sondern zur Ausnahmebehandlung eingesetzt werden können. Wir werden sowohl in diesem als auch im nächsten Abschnitt bei den eingebauten Exceptions bleiben. Selbstdefinierte Exceptions werden das Thema von Abschnitt 12.1.4, »Eigene Exceptions« (S. 340), sein.

12.1.3 Abfangen einer Exception

In diesem Abschnitt geht es darum, wie eine in einer Unterfunktion geworfene Exception in den darüberliegenden Aufrufebenen abgefangen werden kann. Das Fangen einer Exception ist notwendig, um auf den aufgetretenen Fehler reagieren zu können. Stellen Sie sich ein Programm vor, das Daten aus einer vom Benutzer festgelegten Datei liest. Dazu verwendet das Programm die folgende, im Moment noch sehr simple Funktion `get`, die das geöffnete Dateiobjekt zurückgibt:

```
def get(name):
    return open(name)
```

Sollte keine Datei mit dem angegebenen Namen existieren, wirft die eingebaute Funktion `open` eine `IOError`-Exception. Da die Funktion `get` nicht auf diese Exception reagiert, wird sie in der Aufrufhierarchie weiter nach oben gereicht und verursacht schlussendlich ein vorzeitiges Beenden des Programms.

Nun sind fehlerhafte Benutzereingaben Probleme, die man beim Schreiben eines interaktiven Programms berücksichtigen sollte. Die folgende Variante der Funktion `get` fängt eine von `open` geworfene `IOError`-Exception ab und gibt in diesem Fall statt des geöffneten Dateiobjekts den Wert `None` zurück.

```
def get(name):
    try:
        return open(name)
    except IOError:
        return None
```

Zum Abfangen einer Exception wird eine `try/except`-Anweisung verwendet. Eine solche Anweisung besteht zunächst 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 auf die gefangene Exception-Instanz im `except`-Zweig zugreifen 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.

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 12.1 Struktur einer `try/except`-Anweisung

Kommen wir zurück zu unserer Beispielfunktion `get`. Es ist durchaus möglich, dass bei einem Funktionsaufruf für `name` fälschlicherweise kein String, sondern zum Beispiel

eine Liste übergeben wird. In einem solchen Fall wird kein `IOError`, sondern ein `TypeError` geworfen, der von der `try/except`-Anweisung bislang nicht abgefangen wird:

```
>>> get([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in get
TypeError: invalid file: [1, 2, 3]
```

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 ist es, die Liste der abzufangenden Exception-Typen im vorhandenen `except`-Zweig um den `TypeError` zu erweitern. Beachten Sie dabei, dass zwei oder mehr Exception-Typen im Kopf eines `except`-Zweiges als Tupel angegeben werden müssen.

```
def get(name):
    try:
        return open(name)
    except (IOError, TypeError):
        return None
```

Dies ist einfach und führt im gewählten Beispiel zu dem gewünschten Resultat. Stellen Sie sich jedoch 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.

```
def get(name):
    try:
        return open(name)
    except IndexError:
        return None
    except TypeError:
        return None
```

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

```
def get(name):
    try:
        return open(name)
    except:
        return None
```



Hinweis

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

Eine Exception ist nichts anderes als eine Instanz einer bestimmten Klasse. 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 als *Bezeichner*-Teils im Kopf des `except`-Zweiges möglich. Unter dem dort angegebenen Namen können wir nun innerhalb des Codeblocks auf die geworfene Exception-Instanz zugreifen:²

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 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 entsprechenden `else`-Zweiges 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.

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

Abbildung 12.2 zeigt eine vollständige `try/except`-Anweisung.

² Die möglicherweise verwirrende Schreibweise `print([1,2,3][10])` ist gleichbedeutend mit

```
lst = [1,2,3]
print(lst[10])
```

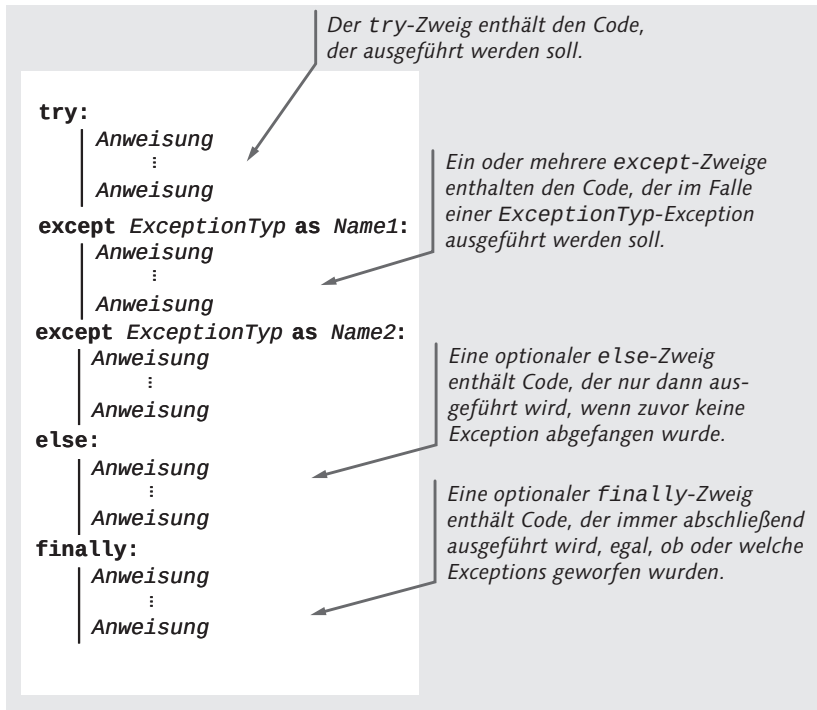


Abbildung 12.2 Eine vollständige try/except-Anweisung

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 der else-Zweig. Der finally-Zweig wird in jedem Fall zum Schluss ausgeführt.

Exceptions, die innerhalb eines except-, else- oder finally-Zweiges geworfen werden, werden so behandelt, als würde 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.

12.1.4 Eigene Exceptions

Beim Werfen und Abfangen von Exceptions sind Sie nicht auf den eingebauten Satz von Exception-Typen beschränkt, vielmehr können Sie selbst 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 Exception hinzufügen.

Im Folgenden soll das Erstellen eines eigenen Exception-Typs anhand eines Beispiels demonstriert werden. Dazu definieren wir zunächst eine rudimentäre Kontoklasse, die als einzige Operation das Abheben eines bestimmten Geldbetrags unterstützt.

```

class Konto:

    def __init__(self, betrag):
        self.kontostand = betrag

    def abheben(self, betrag):
        self.kontostand -= betrag

```

In dieser Implementierung der Klasse ist es möglich, das Konto beliebig zu überziehen. In einer etwas raffinierteren Variante soll das Überziehen des Kontos unterbunden und beim Versuch, mehr Geld abzuheben, als vorhanden ist, eine selbstdefinierte Exception geworfen werden. Dazu definieren wir zunächst eine von der Basisklasse `Exception` abgeleitete Klasse und fügen Attribute für den Kontostand und den abzuhebenden Betrag hinzu.

```

class KontoException(Exception):

    def __init__(self, kontostand, betrag):
        self.kontostand = kontostand
        self.betrag = betrag

```

Dann modifizieren wir die Methode `abheben` der Klasse `Konto` dahingehend, dass bei einem ungültigen Abhebevorgang eine `KontoException`-Instanz geworfen wird.

```
def abheben(self, betrag):
    if betrag > self.kontostand:
        raise KontoException(self.kontostand, betrag)
    self.kontostand -= betrag
```

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

```
>>> k = Konto(1000)
>>> k.abheben(2000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in abheben
__main__.KontoException
```

12

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

```
try:
    k.abheben(2000)
except KontoException as e:
    print("Kontostand: {}€".format(e.kontostand))
    print("Abheben von {}€ nicht möglich.".format(e.betrag))
```

Dieser Code fängt die entstandene Exception ab und gibt daraufhin eine Fehlermeldung aus. Anhand der zusätzlichen Informationen, die die Klasse durch die Attribute `kontostand` und `betrag` bereitstellt, lässt sich der vorausgegangene Abhebevorgang rekonstruieren. Die Ausgabe des Beispiels lautet:

```
Kontostand: 1000€
Abheben von 2000€ nicht möglich.
```

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

```
class KontoException(Exception):

    def __init__(self, kontostand, betrag):
        self.kontostand = kontostand
        self.betrag = betrag
```

```
def __str__(self):
    return "Kontostand zu niedrig"
```

Ein Traceback, der durch diese Exception verursacht wird, sieht folgendermaßen aus:

```
>>> k = Konto(1000)
>>> k.abheben(2000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in abheben
__main__.KontoException: Kontostand zu niedrig
```

12.1.5 Erneutes Werfen einer Exception

In manchen 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 Dateiobjekt geschlossen werden. Nachdem `funktion1` ihre funktionsinterne Fehlerbehandlung durchgeführt hat, soll die Exception weiter nach oben gereicht werden. Dazu wird sie erneut geworfen, wie im folgenden Beispiel:

```
def funktion3():
    raise TypeError

def funktion2():
    funktion3()

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

funktion1()
```

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

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

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 TypeError

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.

12.1.6 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 verloren geht. Das ist problematisch, da möglicherweise genau das Auftreten dieser Exception dazu beigetra-

gen 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
```

Bei der endgültigen Exception handelt es sich um eine `RuntimeError`-Exception. Sie kann nicht von einem `except`-Zweig gefangen werden, der `IndexError`-Exceptions behandelt.

Die Verwendung der `raise/from`-Syntax war im obigen Beispiel eigentlich nicht notwendig, da bei Exceptions, die aus `except`-Zweigen heraus geworfen werden, automatisch die vorangegangene Exception angehängt wird. Sollten Sie das Exception Chaining aber außerhalb von `else`-Zweigen verwenden wollen, müssen Sie die `from`-Syntax benutzen.

Abschließend sei gesagt, dass die hier vorgestellten Techniken zum Exception Handling ungemein beim Schreiben von strukturiertem und lesbarem Code helfen, sodass Sie sie verinnerlichen sollten. Wir werden auch im Laufe dieses Buches immer wieder Exceptions verwenden.

12.2 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.

12.2.1 List Comprehensions

Es ist ein häufig auftretendes Problem, dass man aus den Elementen einer bestehenden Liste nach einer bestimmten Berechnungsvorschrift eine neue Liste erstellen möchte. Bisher würden Sie dies entweder 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 flexiblere Syntax, die für 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 legt fest, mit welchem Bezeichner über welche Liste iteriert wird – in diesem Fall mit dem Bezeichner `x` über die Liste `lst`. Der angegebene Bezeichner kann im Ausdruck zu Beginn der List Comprehension verwendet werden. Das Ergebnis einer List Comprehension ist eine neue Liste, die als Elemente die Ergebnisse des Ausdrucks in jedem Iterationsschritt enthält. Die Funktionsweise der obigen List Comprehension lässt sich folgendermaßen zusammenfassen:

Für jedes Element `x` der Liste `lst` bilde das Quadrat von `x`, und füge das Ergebnis in die Ergebnisliste ein.

Dies ist die einfachste Form der List Comprehension. Der `for/in`-Bereich lässt sich um eine Fallunterscheidung erweitern, sodass nur bestimmte Elemente in die neue Liste übernommen werden. So könnten wir die obige List Comprehension bei-

spielsweise 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 dreidimensionale Vektoren zu addieren. Die Addition von 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, 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.

12.2.2 Dict Comprehensions

Seit Version 3.0 bietet Python einen zu den List Comprehensions analogen Weg an, um ein Dictionary zu erzeugen. Dies nennt sich *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:

```
>>> namen = ["Donald", "Dagobert", "Daisy", "Gustav"]
>>> {k:len(k) for k in namen}
{'Donald': 6, 'Gustav': 6, 'Dagobert': 8, 'Daisy': 5}
>>> {k:len(k) for k in namen if k[0] == "D"}
{'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.

12

12.2.3 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, abgesehen von den geschweiften Klammern, völlig analog zur List Comprehension. Es bedarf also keiner weiteren Beispiele, um sie erfolgreich einzusetzen.

12.3 Generatoren

In diesem Abschnitt werden wir uns mit dem Konzept der Generatoren beschäftigen, die eine komfortable Möglichkeit anbieten, Folgen 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 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 virtuellen³ 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 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 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.

```
>>> square_generator(10)
<generator object square_generator at 0xb6fc8c34>
```

3 Mit »virtuell« ist dabei gemeint, dass diese Sequenz zu keiner Zeit komplett im Speicher abgelegt werden muss und trotzdem 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 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 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 "Carina"
    yield "Lina"
    if not auch_jungen:
        return
    yield "Ramin"
    yield "Sven"
```

Mithilfe der Built-in Function `list` können wir aus den Werten des Generators eine Liste erstellen, die entweder nur "Carina" und "Lina" oder zusätzlich "Ramin" und "Sven" enthält:

```
>>> list(namen())
['Carina', 'Lina', 'Ramin', 'Sven']
>>> list(namen(False))
['Carina', 'Lina']
```

Generator Expressions

Sie erinnern sich sicherlich noch an die sogenannten List Comprehensions (siehe Seite 346), mit denen Sie auf einfache Weise Listen erzeugen können. Mit solchen List Comprehensions können 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 wollen, können 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 wird 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 `[]` runde 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 kennen lernen, denn hinter den Generatoren und der `for`-Schleife steht das Konzept der Iteratoren.

12.4 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 können. In diesem Abschnitt werden wir uns nun mit den Hintergründen und Funktionsweisen dieser Objekte befassen.

Ein sogenannter *Iterator* ist eine Abstraktionsschicht, die es ermöglicht, die Elemente eines Containers⁴ über eine standardisierte Schnittstelle aufzuzählen.

Dabei werden die Eigenheiten des Containers, auf seine Elemente zuzugreifen, hinter dem Iterator versteckt, sodass mit demselben Code beliebige iterierbare Objekte

⁴ Ein Container ist dabei die Instanz eines Datentyps, der eine Menge von Elementen speichert. Beispiele für Container sind Listen, Mengen und Dictionaries.

durchlaufen werden können. Wie der Container die Elemente speichert und wie sonst darauf zugegriffen werden kann, spielt dann keine Rolle mehr.

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 liefert. 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-Folge⁵ 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 12.2 Die ersten 14 Elemente der Fibonacci-Folge

```
class Fibonacci:
    def __init__(self, max_n):
        self.MaxN = max_n
        self.N = 0
        self.A = 0
        self.B = 0
    def __iter__(self):
```

5 Viele Prozesse in der Natur, wie beispielsweise die Anzahl von Samen einer Sonnenblumenblüte, lassen sich durch die Fibonacci-Zahlen beschreiben. Außerdem konvergiert der Quotient von aufeinanderfolgenden Elementen für große n gegen den Goldenen Schnitt ($\Phi = 1,618...$), einem Verhältnis, das oft mit Schönheit assoziiert wird.

```

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.⁶ Außerdem müssen beim Beginn des Durchlaufens die Speicher für das letzte nächste Element mit ihren Anfangswerten 0 bzw. 1 belegt und der `N`-Zähler auf 0 gesetzt werden. Die `__next__`-Methode kümmert sich um die Berechnung des aktuellen Elements der Folge und aktualisiert die Zwischenspeicher und den Zähler. Ist das Ende der gewünschten Teilfolge erreicht, wird `StopIteration` geworfen.

Die Klasse lässt sich nun mit allen Konstrukten verarbeiten, die das Iterator-Protokoll unterstützen, wie beispielsweise die `for`-Schleife und die Built-in Functions `list` oder `sum`:

```

>>> for f in Fibonacci(14):
        print(f, 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

```

⁶ Dies hat zur Folge, dass es nur einen Iterator auf jede Instanz der Klasse `Fibonacci` geben kann. Mehr dazu erfahren Sie auf den folgenden Seiten, insbesondere ab Seite 359.

Mit einer kleinen Subklasse von `Fibonacci` können wir auch einen Iterator erzeugen, der uns die Verhältnisse zweier aufeinander folgender Fibonacci-Zahlen durchlaufen lässt. Dabei sieht man, 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
```

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
1.619048 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037
```

Es ist möglich, 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 lässt 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.

Verwendung 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
```

12

Wie Sie bereits wissen, benötigen wir zum Durchlaufen einer Sequenz das dazugehörige Iterator-Objekt. Dieses liefert uns die Built-in Function `iter`, die, wie schon im 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, sondern 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
```

Mehrere Iteratoren für dieselbe Instanz

Im Umkehrschluss bedeutet dies, dass die Built-in Function `iter` bei Aufrufen für dasselbe iterierbare Objekt verschiedene Iteratoren zurückgeben kann, was auch sinnvoll ist, um beispielsweise eine Liste mit zwei verschiedenen Iteratoren zu durchlaufen:

```
l = [1,2,3]
for i in l:
    for j in l:
        print(i,j, end=" ")
    print()
1 1, 1 2, 1 3,
2 1, 2 2, 2 3,
3 1, 3 2, 3 3,
```

In diesem Beispiel wird jedes Element der Liste `l` mit jedem anderen Element der Liste kombiniert, und die entstehenden Paare werden ausgegeben. Dazu wurde jeweils eine `for`-Schleife verwendet.

Versuchen wir, denselben Code mit einer Instanz der `Fibonacci`-Klasse vom Anfang dieses Abschnitts auszuführen, erhalten wir ein anderes Ergebnis:

```
l = Fibonacci(3)
for i in l:
    for j in l:
        print(i,j, end=" ")
    print()
1 1, 1 1, 1 2,
```

Wir haben erwartet, dass auch hier jede der ersten drei Fibonacci-Zahlen 1,1,2, mit jeder anderen kombiniert, ausgegeben wird, also eine Ausgabe wie folgt:

```
1 1, 1 1, 1 2,
1 1, 1 1, 1 2,
2 1, 2 1, 2 2,
```

Dieses abweichende Verhalten rührt daher, dass die `__iter__`-Methode der Klasse `Fibonacci` keinen neuen Iterator erzeugt, sondern eine Referenz auf das jeweilige Objekt selbst zurückgibt. Daher kann es zu einer Instanz der Klasse `Fibonacci` zur gleichen Zeit nur einen Iterator geben, den sich im obigen Beispiel die beiden `for`-Schleifen teilen müssen.

Der genaue Ablauf sieht folgendermaßen aus:

Die äußere Schleife initialisiert durch Aufruf der `__iter__`-Methode die Instanz `l` für die Iteration, das heißt, die Attribute `l.N` und `l.A` werden auf 0, das Attribut `l.B` auf 1

gesetzt. Dann wird durch Aufrufen der Methode `__next__` die erste Fibonacci-Zahl 1 ermittelt und mit `i` referenziert.

Nun startet die innere Schleife, die ihrerseits wieder die `__iter__`-Methode der Instanz `l` ruft und dadurch erneut die Attribute `l.N`, `l.A` auf 0 und `l.B` auf 1 setzt. Anschließend durchläuft `j` in der inneren Schleife die ersten drei Fibonacci-Zahlen 1, 1, 2 und kombiniert sie mit dem Wert 1 von `i`. Nach der dritten Zahl wirft die Methode `__next__` die `StopIteration`-Exception, sodass die innere Schleife stoppt.

Im Anschluss ruft die äußere Schleife erneut die Methode `__next__` von `l`, die immer noch die `StopIteration`-Exception wirft, da das Ende des gemeinsamen Iterators schon in der inneren Schleife erreicht wurde. Somit endet das Programm an dieser Stelle.

Wie Sie sehen, kann unerwartetes Verhalten auftreten, wenn implizit ein Iterator über mehrere Schleifen geteilt wird. Abhilfe kann dadurch geschaffen werden, dass wir die Klasse `Fibonacci` so anpassen, dass sie bei jedem Aufruf von `__iter__` ein neues Iterator-Objekt zurückgibt:

```
class Fibonacci3:
    class FibonacciIterator:
        def __init__(self, max_n):
            self.MaxN = max_n
            self.N, self.A, self.B = 0, 0, 1

        def __iter__(self):
            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

    def __init__(self, max_n):
        self.MaxN = max_n

    def __iter__(self):
        return self.FibonacciIterator(self.MaxN)
```

Die neue Klasse `Fibonacci3` definiert eine weitere Klasse `FibonacciIterator`, die das eigentliche Iterieren übernimmt und genauso definiert wurde wie die Klasse

Fibonacci. Jedes Mal, wenn ein neuer Iterator über eine Instanz der Klasse `Fibonacci3` angefordert wird, wird ein neues Objekt der Klasse `FibonacciIterator` erzeugt.

Dadurch liefern die beiden verschachtelten Schleifen von oben das erwartete Ergebnis:

```
l = Fibonacci3(3)
for i in l:
    for j in l:
        print(i,j, end=" ")
    print()
```

```
1 1, 1 1, 1 2,
1 1, 1 1, 1 2,
2 1, 2 1, 2 2,
```

Aufgrund dieser möglichen Probleme ist es grundsätzlich eine gute Idee, bei jedem Aufruf von `__iter__` einen neuen Iterator zu erzeugen.

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 gut, um alle Elemente einer Sequenz zu durchlaufen und dies einheitlich für alle Container-Datentypen umzusetzen. Im Gegensatz dazu kann mit Indizes in beliebiger Reihenfolge lesend und schreibend auf die Elemente eines Containers zugegriffen werden, 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 vielen Folgen und Containern möglich ist, die Elemente durczunummerieren und über ganzzahlige Indizes anzusprechen, ist ein Objekt schon dann iterierbar, wenn man seine Elemente über die `__getitem__`-Methode, also den `[]`-Operator, über ganzzahlige Indizes ansprechen kann. Ruft man die Built-in Function `iter` mit einer solchen Instanz als Parameter auf, kümmert Python sich um die Erzeugung des Iterators. Bei jedem Aufruf der `__next__`-Methode des erzeugten Iterators wird die `__getitem__`-Methode der iterierbaren Instanz aufgerufen, wobei immer eine Ganzzahl als Parameter über-

geben 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 kann eine Klasse zum Iterieren über die ersten `max_n` Quadratzahlen folgendermaßen aussehen, wenn sie zudem noch das Bestimmen ihrer Länge mittels `len` unterstützt:

```
class Quadrate:
    def __init__(self, max_n):
        self.MaxN = max_n

    def __getitem__(self, index):
        index += 1 # 0*0 ist nicht sehr interessant...
        if index > len(self) or index < 1:
            raise IndexError
        return index*index

    def __len__(self):
        return self.MaxN
```

Zur Demonstration dieses versteckten Iterators lassen wir uns eine Liste mit den ersten zwanzig Quadratzahlen ausgeben:

```
>>> list(Quadrate(20))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
324, 361, 400]
```

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 besonderen Wert, den *Terminator* der Folge, zurückgibt. Einen Funktionsiterator erzeugen Sie mit der Built-in Funktion `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.
```

12.5 Docstrings

In Abschnitt 4.3, »Kommentare«, wurde ein in drei einfachen oder doppelten Hochkommata eingefasster Text als Blockkommentar bezeichnet:

```
"""
Dies ist ein Blockkommentar.
Er kann mehrere Zeilen umfassen.
"""
```

Der Name Blockkommentar wird den Möglichkeiten, die diese Notation bietet, jedoch nicht 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 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
>>> print(math.__doc__)
This module is always available. It provides access to the
mathematical functions defined by the C standard.

```

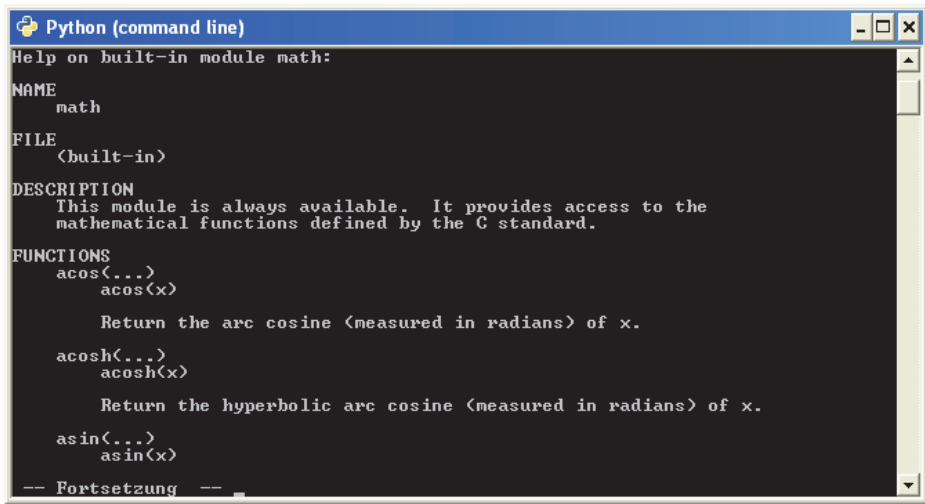
Die eingebaute Funktion `help` erzeugt aus den in einem Objekt enthaltenen Docstrings eine Hilfeseite und zeigt diese im interaktiven Modus an:

```

>>> import math
>>> help(math)

```

In Abbildung 12.3 ist die durch das obige Beispiel erzeugte Hilfeseite zu sehen.



```

Python (command line)
Help on built-in module math:

NAME
  math

FILE
  <built-in>

DESCRIPTION
  This module is always available. It provides access to the
  mathematical functions defined by the C standard.

FUNCTIONS
  acos(...)
    acos(x)

    Return the arc cosine <measured in radians> of x.

  acosh(...)
    acosh(x)

    Return the hyperbolic arc cosine <measured in radians> of x.

  asin(...)
    asin(x)

-- Fortsetzung --

```

12

Abbildung 12.3 Mittels help erzeugte Hilfeseite

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.

12.6 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 Built-in `exec` 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 Programmablauf 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.⁷

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.

⁷ Stellen Sie sich einmal vor, Sie speichern ein Passwort oder sonstige sensible Daten in einer Variablen, die der Benutzer dann einsehen könnte.

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, sieht 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
```

12

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(source [, globals[, locals]])
```

Ein einfaches Beispiel für die Funktion `eval` sieht folgendermaßen aus:

```
>>> eval("5 * 4")
20
```

Auch der von `eval` ausgewertete Ausdruck hat standardmäßig Zugriff auf alle Variablen des aktuellen Kontextes. 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 aktuelle Kontext verwendet wurde. Dem zweiten Aufruf hingegen übergaben wir ein leeres Dictionary als Kontext, weshalb der versuchte Zugriff auf `x` mit einer Exception quittiert wurde.

12.7 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 Dateiobjekts gerufen wird, selbst wenn zwischen dem Aufruf von `open` und dem der `close`-Methode des Dateiobjekts eine Exception geworfen wurde. Dazu ist mit den herkömmlichen Sprachelementen Pythons folgende `try/finally`-Anweisung nötig:

```
f = open("datei.txt", "r")
try:
    print(f.read())
finally:
    f.close()
```

Zunächst wird eine Datei namens *datei.txt* zum Lesen geöffnet. Die darauffolgende `try/finally`-Anweisung stellt sicher, dass `f.close` in jedem Fall aufgerufen wird. Der Nachteil dieser Schreibweise ist, dass sich der Programmierer darum kümmern muss, dass das Dateiojekt korrekt deinitialisiert wird. Die `with`-Anweisung überträgt diese Verantwortung an das Objekt selbst und erlaubt eine kurze und elegante Alternative für den obigen Code:

```
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.

Statt mehrere with-Anweisungen zu verschachteln, können die jeweiligen Instanzen und Bezeichner auch, durch Kommata getrennt, in eine with-Anweisung geschrieben werden. So ist der Code

```
with open("file1.txt", "r") as f1, open("file2.txt", "r") as f2:
    print(f1.read())
    print(f2.read())
```

äquivalent zu:

```
with open("file1.txt") as f1:
    with open("file2.txt", "r") as f2:
        print(f1.read())
        print(f2.read())
```

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 die Methode `write` des Dateiobjekts auf. Beachten Sie, dass die Methode `eintrag` nur innerhalb einer `with`-Anweisung aufgerufen werden kann, da das Dateiobjekt erst in den folgenden Magic Methods geöffnet und geschlossen wird.

Die angesprochenen Magic Methods `__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 Beispielsklasse wird die Datei `self.logfile` zum Schreiben geöffnet und mit `return self` eine Referenz auf die eigene Instanz zurückgegeben.

Die zweite Magic Method `__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 Beispielsklasse 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:

```
with MeinLogfile("logfile.txt") 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 werden die Magic Methods `__enter__` und `__exit__` vollständig erläutert.

`__enter__(self)`

Diese Magic Method wird einmalig zum Öffnen des Kontextes 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 Method `__exit__` wird einmalig zum Schließen des Kontextes aufgerufen, nachdem der Körper der `with`-Anweisung ausgeführt worden ist. Die drei Parameter `exc_type`, `exc_value` und `traceback` spezifizieren Typ, Wert und Traceback-Objekt⁸ 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: Gibt die Methode `True` zurück, wird die Exception unterdrückt. Bei einem Rückgabewert von `False` wird die Exception erneut geworfen.

12.8 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 optional, und die Funktionsschnittstelle darf auch 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.

Annotations ändern an der Ausführung einer Funktion nichts, man könnte sagen: Dem Python-Interpreter sind Annotations egal. Das Interessante an Function Annotations ist, dass man sie über das Attribut `__annotations__` des Funktionsobjekts 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.

⁸ Näheres zum Umgang mit Traceback-Objekten erfahren Sie in Abschnitt 20.6 (S. 722).

"return" für die Annotation des Rückgabewertes als Schlüssel und die jeweiligen Annotation-Ausdrücke als Werte. Für die obige schematische Funktionsdefinition sieht 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. Dazu definieren wir zunächst eine Funktion samt Annotations:

```
def strmult(s: str, n: int) -> str:
    return s*n
```

Die Funktion `strmult` hat die Aufgabe, einen String `s` `n`-mal hintereinander geschrieben zurückzugeben. Das geschieht durch Multiplikation von `s` und `n`.

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 not isinstance(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 Funktionsobjekts `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 gut gegangen, 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 ungültigen Typ
```

Um die Überprüfung auf den Rückgabewert testen zu können, muss natürlich die Definition der Funktion `strmult` verändert werden.

12.9 Function Decorator

Aus Kapitel 11, »Objektorientierung«, kennen Sie sicherlich noch die Built-in Function `staticmethod`, die folgendermaßen verwendet wurde:

```
class MeineKlasse:

    def methode():
        pass

methode = 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. Function Decorators sind nicht auf den Einsatz mit `staticmethod` beschränkt, Sie können 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 folgende Beispiel soll einen interessanten Ansatz zum *Cachen* (dt. »zwischen speichern«) 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 allen Funktionsaufrufen funktionieren, bei

denen nur hashbare Instanzen übergeben werden. 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 instanziiert wird. Sie erinnern sich sicherlich, dass eine Klasse durch Implementieren der Magic Method `__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,⁹ können also wie ein Funktionsobjekt verwendet werden. Um als Function Decorator verwendet werden zu können, muss die Methode `__call__` ein

9 Näheres dazu erfahren Sie in Abschnitt 11.5.1 (S. 304).

Funktionsobjekt als Parameter akzeptieren und ein Funktionsobjekt zurückgeben. Das zurückgegebene Funktionsobjekt wird dann als veränderte Version der ursprünglich übergebenen Funktion mit dieser assoziiert. In unserem Beispiel gibt `__call__` das Funktionsobjekt der Methode `cachedFunc` zurück.

Die Methode `cachedFunc` soll jetzt also 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 so viele Funktionsschnittstellen wie möglich funktionieren soll.¹⁰ 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 Caching bietet je nach Anwendungsbereich und Komplexität der Berechnung erhebliche Geschwindigkeitsvorteile. Beachten Sie aber, dass keine Logik zum Löschen von Cache-Einträgen implementiert ist. Der Cache wird also bei Benutzung immer weiter anwachsen.

¹⁰ Da die bei einem Funktionsaufruf übergebenen Parameter als Schlüssel für das interne Cache-Dictionary verwendet werden, dürfen nur Instanzen hashbarer Datentypen übergeben werden.

12.10 assert

Mithilfe des Schlüsselworts `assert` lassen sich Konsistenzabfragen in ein Python-Programm integrieren. Durch das Schreiben einer `assert`-Anweisung legt der Programmierer eine Bedingung fest, die für die Ausführung des Programms essentiell 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 praktisches Hilfsmittel zum Aufspüren von Fehlern und ermöglicht es, den Programmlauf zu beenden, wenn bestimmte Voraussetzungen nicht gegeben sind. Häufig prüft man an Schlüsselstellen im Programm mit `assert`, ob alle Referenzen die erwarteten Werte referenzieren, um eventuelle Fehlrechnungen rechtzeitig 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 Konstante¹¹ `__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.

¹¹ Bei `__debug__` handelt es sich tatsächlich um eine Konstante, ihr Wert kann aus einem laufenden Programm heraus nicht verändert werden.

12.11 Weitere Aspekte der Syntax

Das Thema dieses Abschnitts sollen kleinere Aspekte der Python-Syntax sein, die bisher vernachlässigt wurden. Allgemein gilt, dass die hier besprochenen Notationen keineswegs notwendig oder unumgänglich sind. Entscheiden Sie ganz nach Ihren Vorlieben, ob und in welchem Umfang Sie sie einsetzen möchten.

12.11.1 Umbrechen langer Zeilen

Sicherlich haben Sie bereits einige eigene Python-Programme geschrieben, und dabei ist die ein oder andere lange Quellcodezeile entstanden. Viele Programmierer beschränken die Länge ihrer Quellcodezeilen, damit beispielsweise mehrere Quellcode-dateien nebeneinander auf den Bildschirm passen oder der Code auch auf Geräten mit einer festen Zeilenbreite angenehm zu lesen ist. Eine geläufige maximale Zeilenlänge ist 80 Zeichen. Doch welche Möglichkeiten bietet Python, überlange Zeilen umzubrechen, sodass eine maximale Zeilenlänge eingehalten werden kann?

Sie wissen bereits, dass Sie Ihren Quellcode innerhalb von Klammern beliebig umbrechen dürfen, doch an vielen anderen Stellen sind Sie an die strengen syntaktischen Regeln von Python gebunden. Durch Einsatz der Backslash-Notation ist es möglich, Quellcode an nahezu beliebigen Stellen in eine neue Zeile umzubrechen:

```
>>> var \
... = \
... 10
>>> var
10
```

Grundsätzlich kann ein Backslash überall da stehen, wo auch ein Leerzeichen hätte stehen können. Somit ist auch ein Backslash innerhalb eines Strings möglich:

```
>>> "Hallo \
... Welt"
'Hallo Welt'
```

Beachten Sie dabei aber, dass eine Einrückung des 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'
```

12.11.2 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 durch ein Semikolon zusammengefasst werden:

```
>>> x = True
>>> if x: print("Hallo"); print("Welt")
...
Hallo
Welt
```

Alle durch ein Semikolon zusammengefügte Anweisungen werden so behandelt, als wären sie gleich weit eingerückt. Allein ein Doppelpunkt vergrößert die Einrückungstiefe. 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.

Hinweis

Beim Einsatz des Backslashes und vor allem des Semikolons entsteht schnell unleserlicher Code. Verwenden Sie beide Notationen daher nur, wenn Sie meinen, dass es der Lesbarkeit und Übersichtlichkeit dienlich ist.



12.12 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 beispielsweise durch das bloße Hinzufügen eines Schlüsselwortes schon derjenige Code inkompatibel geworden, der das neue Schlüsselwort als normalen Bezeichner verwendet.

Der Interpreter besitzt einen 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 sehr 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.

Teil III

Die Standardbibliothek

Herzlich willkommen zum dritten Teil dieses Buchs. Hier möchten wir uns intensiv mit der Standardbibliothek von Python auseinandersetzen und alle wichtigen Module besprechen.

Die Tabelle auf den folgenden Seiten listet die Module auf, die in diesem Teil besprochen werden.

Modul	Beschreibung	Seite
Mathematik		
<i>math</i>	mathematische Funktionen	385
<i>cmath</i>	mathematische Funktionen für komplexe Zahlen	385
<i>random</i>	Erzeugen von Zufallszahlen oder zufälligen Auswahlen aus gegebenen Mengen	394
<i>decimal</i>	präzisere Repräsentation von Dezimalzahlen	399
<i>itertools</i>	spezielle Generatoren, beispielsweise für Kombinationen oder Permutationen	405
Strings		
<i>re</i>	reguläre Ausdrücke	415
<i>gettext</i>	Lokalisierung von Programmen	442
<i>hashlib</i>	verschiedene bekannte Hashfunktionen	446
Datum und Zeit		
<i>time</i>	elementare Funktionen zum Umgang mit Unix-Timestamps	451
<i>datetime</i>	komfortable Funktionen für Datums- und Zeitangaben	460
Schnittstelle zum Betriebssystem		
<i>os</i>	Zugriff auf das Betriebssystem	481
<i>os.path</i>	Funktionen zur Manipulation von Dateisystem-Pfaden	490
<i>shutil</i>	Zugriff auf das Dateisystem	495
<i>sys</i>	Zugriff auf die Laufzeitumgebung	501
<i>platform</i>	Zugriff auf Systeminformationen	511
<i>argparse</i>	Funktionen zur Verwendung von Kommandozeilenparametern	512
<i>copy</i>	Kopieren von Instanzen	519
<i>atexit</i>	Erkennen und Behandeln des Programmendes	523

Modul	Beschreibung	Seite
Parallele Programmierung		
<i>_thread</i>	einfaches Modul zur parallelen Programmierung	528
<i>threading</i>	objektorientierter Ansatz zur parallelen Programmierung	536
Datenspeicherung		
<i>gzip</i>	Lesen und Schreiben von komprimierten Dateien	549
<i>xml.dom.minidom</i>	DOM-Modell zum Lesen und Schreiben von XML-Dateien	553
<i>xml.sax</i>	SAX-Modell zum Lesen von XML-Dateien	565
<i>xml.etree.ElementTree</i>	Python-spezifischer Ansatz zum Lesen und Schreiben von XML-Dateien	570
<i>sqlite3</i>	Umgang mit SQLite Datenbanken	579
<i>pickle</i>	Speichern und Laden beliebiger Python-Instanzen	595
<i>csv</i>	Lesen und Schreiben von CSV-Dateien	599
<i>tempfile</i>	Umgang mit temporären Dateien	604
Netzwerkcommunication		
<i>socket</i>	grundlegende Funktionalität zur Netzwerkkommunikation	609
<i>select</i>	Modul für multiplexende Server	627
<i>socketserver</i>	komfortables Modul zum Schreiben eines Servers	630
<i>urllib.request</i>	Zugriff auf Internetressourcen	634
<i>urllib.parse</i>	Funktionen zur Manipulation von URLs	640
<i>ftplib</i>	Kommunikation mit FTP-Servern zum Dateitransfer	646
<i>smtplib</i>	Kommunikation mit SMTP-Servern zum E-Mail-Versand	655
<i>poplib</i>	Kommunikation mit POP3-Servern zum Abholen von E-Mails	660

Modul	Beschreibung	Seite
<i>imaplib</i>	Kommunikation mit IMAP4-Servern zum Abholen und Verwalten von E-Mails	665
<i>email</i>	Modul zum Erstellen komplexer E-Mails	671
<i>telnetlib</i>	Kommunikation über das Telnet-Protokoll	676
<i>xmlrpc</i>	Modul zum entfernten Funktionsaufruf über das XMLRPC-Protokoll	680
Debugging		
<i>inspect</i>	detaillierte Informationen über Instanzen	694
<i>pprint</i>	formatierte Ausgabe von Instanzen	703
<i>logging</i>	Funktionen zum Schreiben einer Logdatei	706
<i>doctest</i>	automatisierte Testfälle in Docstrings	713
<i>unittest</i>	Modul für Unittests	718
<i>traceback</i>	Funktionen zum Umgang mit Traceback-Objekten	722
<i>timeit</i>	Funktionen zur Laufzeitmessung	727
<i>cProfile</i>	Modul zum Profiling von Python-Programmen	730
<i>trace</i>	Modul zur Durchführung einer Überdeckungsanalyse in Python-Programmen	734
Sonstiges		
<i>distutils</i>	Modul zur Distribution von Python-Programmen	739
<i>tkinter</i>	grafische Benutzeroberflächen	762
<i>ctypes</i>	Verwenden von dynamisch ladbaren Bibliotheken	904
<i>webbrowser</i>	Verwenden des Standardbrowsers	933
<i>functools</i>	Vereinfachen von Funktionsschnittstellen	934
<i>getpass</i>	verdeckte Passworteingaben	949
<i>cmd</i>	Schreiben von Kommandozeilen-Interpretern	949

Kapitel 13

Mathematik

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

Wir beginnen mit den Modulen der Standardbibliothek, mit deren Hilfe sich im weitesten Sinne mathematische Berechnungen durchführen lassen. Die folgende Tabelle listet die Module auf, die in diesem Kapitel besprochen werden.

Modul	Beschreibung	Seite
<i>math</i>	mathematische Funktionen	385
<i>cmath</i>	mathematische Funktionen für komplexe Zahlen	385
<i>random</i>	Erzeugen von Zufallszahlen oder zufälligen Auswahlen aus gegebenen Mengen	394
<i>decimal</i>	präzisere Repräsentation von Dezimalzahlen	399
<i>itertools</i>	spezielle Generatoren, beispielsweise für Kombinationen oder Permutationen	405

Tabelle 13.1 Mathematische Module der Standardbibliothek

13.1 Mathematische Funktionen – *math*, *cmath*

Das Modul *math* ist Teil der Standardbibliothek und stellt mathematische Funktionen und Konstanten bereit.

Hinweis

Das Modul *math* berücksichtigt den komplexen Zahlenraum – und damit den Datentyp *complex* – nicht. 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 wird der Inhalt der Module `math` und `cmath` zunächst tabellarisch besprochen. Die Spalte `cmath` gibt an, ob das besprochene Element auch im Modul `cmath` enthalten ist.

Konstante	Beschreibung	cmath	Seite
<i>e</i>	die Eulersche Zahl <i>e</i> .	Ja	–
<i>pi</i>	die Kreiszahl Pi (π).	Ja	–

Tabelle 13.2 Konstanten der Module `math` und `cmath`

Funktion	Beschreibung	cmath	Seite
Zahlentheoretische Funktionen			
<i>ceil(x)</i>	Rundet auf die nächsthöhere Ganzzahl auf.	Nein	389
<i>copysign(x,y)</i>	Überträgt das Vorzeichen von <i>y</i> auf <i>x</i> und gibt das Ergebnis zurück.	Nein	389
<i>fabs(x)</i>	Gibt den Betrag von <i>x</i> zurück. Das Ergebnis ist immer eine Gleitkommazahl.	Nein	–
<i>factorial(x)</i>	Berechnet die Fakultät der ganzen Zahl <i>x</i> .	Nein	–
<i>floor(x)</i>	Rundet auf die nächstniedrigere Ganzzahl ab.	Nein	390
<i>fmod(x,y)</i>	Berechnet <i>x</i> Modulo <i>y</i> .	Nein	390
<i>frexp(x)</i>	Extrahiert Mantisse und Exponent von <i>x</i>	Nein	390
<i>fsum(iterable)</i>	Berechnet die Summe der in <i>iterable</i> enthaltenen Gleitkommazahlen.	Nein	390
<i>isfinite(x)</i>	Gibt genau dann <code>True</code> zurück, wenn <i>x</i> keinen der Werte <code>inf</code> , <code>-inf</code> oder <code>nan</code> referenziert. ¹	Ja	–

Tabelle 13.3 Funktionen der Module `math` und `cmath`

1 Diese Werte stehen für positiv und negativ unendlich bzw. »Not a Number«. Näheres dazu finden Sie in Abschnitt 7.3.5, »Gleitkommazahlen – float«, (S. 102).

Funktion	Beschreibung	cmath	Seite
<i>isinf(x)</i>	Gibt genau dann <code>True</code> zurück, wenn x positiv oder negativ unendlich ist.	Ja	–
<i>isnan(x)</i>	Gibt genau dann <code>True</code> zurück, wenn x den Wert <code>nan</code> hat.	Ja	–
<i>ldexp(m,e)</i>	Bestimmt eine Gleitkommazahl aus Mantisse m und Exponent e .	Nein	391
<i>modf(x)</i>	Gibt ein Tupel mit dem Nachkomma- und dem Vorkommaanteil der Gleitkommazahl x zurück.	Nein	–
<i>trunc(x)</i>	Gibt den Vorkommaanteil von x als ganze Zahl zurück. Verhält sich bei positiven Werten von x wie <i>floor</i> und bei negativen Werten von x wie <i>ceil</i> .	Nein	–
Exponential- und Logarithmusfunktionen			
<i>exp(x)</i>	Berechnet e^x .	Ja	391
<i>expm1(x)</i>	Berechnet $e^x - 1$. Das Ergebnis ist genauer als das von $\exp(x) - 1$.	Ja	–
<i>log(x[,base])</i>	Berechnet den Logarithmus von x zur Basis <i>base</i> .	Ja	391
<i>log10(x)</i>	Berechnet den Logarithmus von x zur Basis 10.	Ja	–
<i>log1p(x)</i>	Berechnet den Logarithmus von $1+x$ zur Basis e . Das Ergebnis ist genauer als das von $\log(1+x)$.	Ja	–
<i>pow(x,y)</i>	Berechnet x^y .	Nein	392
<i>sqrt(x)</i>	Berechnet die Quadratwurzel von x .	Ja	392
Trigonometrische Funktionen			
<i>acos(x)</i>	Berechnet den Arkuskosinus von x .	Ja	392
<i>asin(x)</i>	Berechnet den Arkussinus von x .	Ja	392
<i>atan(x)</i>	Berechnet den Arkustangens von x .	Ja	393
<i>atan2(y,x)</i>	Berechnet $\text{atan}(x / y)$.	Nein	393
<i>cos(x)</i>	Berechnet den Kosinus von x .	Ja	–

Tabelle 13.3 Funktionen der Module math und cmath (Forts.)

Funktion	Beschreibung	cmath	Seite
$\text{hypot}(x,y)$	Berechnet die Euklidische Norm des Vektors (x,y) .	Nein	393
$\sin(x)$	Berechnet den Sinus von x .	Ja	–
$\tan(x)$	Berechnet den Tangens von x .	Ja	–
Umrechnung von Winkeln			
$\text{degrees}(x)$	Rechnet den Winkel x vom Bogenmaß in Grad um.	Nein	–
$\text{radians}(x)$	Rechnet den Winkel x von Grad ins Bogenmaß um.	Nein	–
Hyperbolische Funktionen			
$\text{acosh}(x)$	Berechnet den Areakosinus Hyperbolicus von x . Das ist die Umkehrfunktion des Kosinus Hyperbolicus.	Ja	–
$\text{asinh}(x)$	Berechnet den Areasinus Hyperbolicus von x . Das ist die Umkehrfunktion des Sinus Hyperbolicus.	Ja	–
$\text{atanh}(x)$	Berechnet den Areatangens Hyperbolicus von x . Das ist die Umkehrfunktion des Tangens Hyperbolicus.	Ja	–
$\text{cosh}(x)$	Berechnet den Kosinus Hyperbolicus von x .	Ja	–
$\text{sinh}(x)$	Berechnet den Sinus Hyperbolicus von x .	Ja	–
$\text{tanh}(x)$	Berechnet den Tangens Hyperbolicus von x .	Ja	–
Spezielle Funktionen			
$\text{erf}(x)$	Berechnet die Gaußsche Fehlerfunktion an der Stelle x .	Nein	–
$\text{erfc}(x)$	Berechnet $1.0 - \text{erf}(x)$, ist aber für große Werte von x genauer als die direkte Berechnung.	Nein	–
$\text{gamma}(x)$	Berechnet die Gammafunktion an der Stelle x .	Nein	–
$\text{lgamma}(x)$	Berechnet den natürlichen Logarithmus des Betrags der Gammafunktion an der Stelle x .	Nein	–

Tabelle 13.3 Funktionen der Module math und cmath (Forts.)

Funktion	Beschreibung	cmath	Seite
Funktionen aus cmath, die nicht in math enthalten sind			
<i>phase(x)</i>	Berechnet die Phase der komplexen Zahl x .	Ja	394
<i>polar(x)</i>	Berechnet die Polardarstellung der komplexen Zahl x .	Ja	394
<i>rect(r,phi)</i>	Konvertiert eine in Polardarstellung gegebene komplexe Zahl mit dem Radius r und dem Winkel phi in ihre kartesische Darstellung.	Ja	394

Tabelle 13.3 Funktionen der Module math und cmath (Forts.)

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

```
>>> import math
```

13

13.1.1 Zahlentheoretische Funktionen

ceil(x)

Die Funktion *ceil* (für engl. *ceiling*, dt. »Zimmerdecke«) gibt die kleinste ganze Zahl zurück, die größer oder gleich x ist. Sie rundet eine Zahl also auf die nächstgrößere ganze Zahl auf. 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
```

copysign(x,y)

Die Funktion *copysign* gibt den Wert von x mit dem Vorzeichen von y zurück.

```
>>> math.copysign(5, -1)
-5.0
>>> math.copysign(-1.5, 0)
1.5
>>> math.copysign(-1.5, -0.0)
-1.5
```

Das letzte Beispiel setzt voraus, dass das verwendete System ein Vorzeichen für den Gleitkommawert 0.0 unterstützt.

floor(x)

Die Funktion *floor* (dt. »Fußboden«) gibt die größte ganze Zahl zurück, die kleiner oder gleich x ist. Sie rundet eine Zahl also auf die nächstkleinere ganze Zahl ab. 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
```

fmod(x, y)

Die Funktion *fmod* berechnet x Modulo y . Beachten Sie, dass diese Funktion nicht immer dasselbe Ergebnis berechnet wie $x \% y$. So gibt *fmod* das Ergebnis 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
```

frexp(x)

Die Funktion *frexp* 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 verstehen.

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

fsum(iterable)

Die Funktion *fsum* berechnet die Summe der in *iterable* enthaltenen Gleitkommazahlen.

```
>>> math.fsum([1.5, 7.34, 2, 4.78])
15.620000000000001
```

Der Vorteil von *fsum* gegenüber der Built-in Function *sum*, die prinzipiell für den gleichen Zweck eingesetzt werden kann, ist, dass *fsum* versucht, die durch die Summation in der Gleitkommaarithmetik entstehenden Fehler möglichst gering zu halten. Das Ergebnis von *fsum* ist also genauer als das Ergebnis von *sum*.

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
```

13.1.2 Exponential- und Logarithmusfunktionen

13

exp(x)

Die Funktion *exp* 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
>>> cmath.exp(1j * math.pi)
(-1+1.2246063538223773e-16j)
```

Das letzte Beispiel zeigt die Verwendung der Exponentialfunktion mit einem komplexen Exponenten. Bei der Auswertung der Gleichung $e^{i\pi} = -1$ zeigt sich erneut, dass beim Rechnen mit Gleitkommazahlen immer ein gewisser Fehler berücksichtigt werden sollte.

log(x[, base])

Die Funktion *log* berechnet den Logarithmus² von *x* zur Basis *base*. Wenn *base* nicht angegeben wurde, wird der Logarithmus Naturalis, also der Logarithmus zur Basis *e*, berechnet.

² Der Logarithmus von *a* zur Basis *b* ist die Lösung *x* der Gleichung $bx = a$.

```
>>> math.log(math.e)
1.0
>>> math.log(32, 2)
5.0
```

pow(x, y)

Diese Funktion *pow* berechnet x^y . Es können für x , insbesondere aber auch für y , negative Zahlen oder Gleitkommazahlen übergeben werden. Beachten Sie, dass *pow* stets eine reelle Zahl zurückgibt und im Falle eines komplexen Ergebnisses eine *ValueError-Exception* wirft.

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

Diese Funktion ist äquivalent zur Built-in Function *pow*.

sqrt(x)

Die Funktion *sqrt* berechnet die Quadratwurzel von x , wobei x in der reellen Funktion *sqrt* größer oder gleich 0 sein muss. Für die komplexe Variante von *sqrt* im Modul *cmath* gilt diese Einschränkung nicht.

```
>>> math.sqrt(100)
10.0
>>> cmath.sqrt(-1)
1j
```

Das Ergebnis ist immer eine Gleitkommazahl.

13.1.3 Trigonometrische Funktionen

acos(x)

Die Funktion *acos* berechnet den Arkuskosinus von x . Der Arkuskosinus ist die Umkehrfunktion des Kosinus. Der Parameter x muss eine Gleitkommazahl im Intervall von -1 bis 1 sein. Der Rückgabewert von *acos* ist ebenfalls eine Gleitkommazahl, wird im Bogenmaß angegeben und liegt im Bereich von 0 bis π .

asin(x)

Die Funktion *asin* berechnet den Arkussinus von x . Der Arkussinus ist die Umkehrfunktion des Sinus. Der Parameter x muss eine Gleitkommazahl im Intervall von -1

bis 1 sein. Der Rückgabewert von *asin* ist ebenfalls eine Gleitkommazahl, wird im Bogenmaß angegeben und liegt im Bereich von $-\pi/2$ bis $+\pi/2$.

atan(x)

Die Funktion *atan* 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 $+\pi/2$.

atan2(y, x)

Die Funktion *atan2* 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 kartesische Koordinaten elegant in Polarkoordinaten umrechnen.

```
>>> math.atan2(1, 1)
0.78539816339744828
>>> math.pi / 4
0.7853981633974483
```

13

Im Beispiel wurde die Phase φ des kartesischen Koordinatenpaares (1,1) in Polardarstellung berechnet. Zur Probe ist zusätzlich das exakte Ergebnis von $\frac{\pi}{4}$ ausgegeben.

hypot(x, y)

Die Funktion *hypot* berechnet die Euklidische Norm des Vektors (*x*,*y*). Die Euklidische Norm eines Vektors 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
```

In Kombination mit der Funktion *atan2* lässt sich *hypot* benutzen, um die Polardarstellung eines in kartesischen Koordinaten gegebenen Punktes zu bestimmen:

```
>>> punkt = (1,1)
>>> r = math.hypot(*punkt)
>>> phi = math.atan2(*punkt)
>>> (r, phi)
(1.4142135623730951, 0.7853981633974483)
```

13.1.4 Funktionen aus cmath

In diesem Abschnitt werden die Funktionen aus `cmath` vorgestellt, die keine Entsprechung im Modul `math` haben.

Die folgenden drei Funktionen eignen sich dazu, verschiedene Darstellungsformen komplexer Zahlen ineinander umzurechnen. Dazu wird eine komplexe Zahl $a + bj$ als Punkt (a, b) im zweidimensionalen Raum aufgefasst. Dieser Raum wird die *Gaußsche Zahlenebene* genannt. Die komplexe Zahl lässt sich jetzt entweder in kartesischen Koordinaten über die Werte a und b ausdrücken oder in Polarkoordinaten mithilfe des Winkels φ und des Radius r . Abbildung 13.1 zeigt die beiden Darstellungsformen von komplexen Zahlen an einem Beispiel.

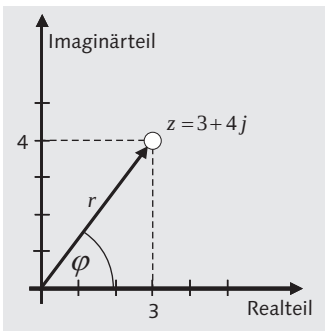


Abbildung 13.1 Die zwei Darstellungen der komplexen Zahl $3+4j$

`phase(x)`

Die Funktion *phase* gibt die Phase, häufig auch Argument oder Winkel genannt, der komplexen Zahl x zurück.

`polar(x)`

Die Funktion *polar* 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)`

Die Funktion *rect* ist das Gegenstück zu *polar*. Die Funktion *rect* konvertiert eine in Polardarstellung durch den Radius r und den Winkel *phi* gegebene komplexe Zahl in ihre kartesische Darstellung. Das Ergebnis wird als `complex`-Instanz zurückgegeben.

13.2 Zufallszahlengenerator – random

Das Modul `random` der Standardbibliothek erzeugt Pseudozufallszahlen und bietet zudem einige zusätzliche Funktionen, um zufallsgesteuerte Operationen auf Basisdatentypen anzuwenden.



Hinweis

Das Modul `random` erzeugt keine echten Zufallszahlen, 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.

Die folgende Tabelle listet die im Modul `random` enthaltenen Funktionen auf und weist gegebenenfalls auf eine detaillierte Erläuterung.

Funktion	Beschreibung	Seite
Steuerungsfunktionen		
<code>getrandbits(k)</code>	Erzeugt eine ganze Zahl, deren Bitfolge aus k zufälligen Bits besteht.	–
<code>getstate()</code>	Gibt den Status des Zufallszahlengenerators zurück.	397
<code>seed([x],version)</code>	Initialisiert den Zufallszahlengenerator.	396
<code>setstate(state)</code>	Setzt den Status des Zufallszahlengenerators.	397
Funktionen für ganze Zahlen		
<code>randint(a,b)</code>	Erzeugt eine ganze Zufallszahl im angegebenen Bereich.	397
<code>randrange([start], stop[,step])</code>	Erzeugt eine Zufallszahl im angegebenen Bereich.	397
Funktionen für Sequenzen		
<code>choice(seq)</code>	Wählt ein zufälliges Element einer Sequenz.	398
<code>sample(population,k)</code>	Wählt zufällig k Elemente aus <i>population</i> .	398
<code>shuffle(x[,random])</code>	Ordnet die Elemente von x zufällig an.	398

Tabelle 13.4 Funktionen des Moduls `random`

Funktion	Beschreibung	Seite
Verteilungen		
<i>random()</i>	Gibt eine gleichverteilte Zufallszahl zwischen 0.0 und 1.0 zurück.	–
<i>uniform(a,b)</i>	Gibt eine gleichverteilte Zufallszahl zwischen <i>a</i> und <i>b</i> zurück.	–
<i>betavariate(alpha,beta)</i>	Gibt eine betaverteilte Zufallszahl zurück.	–
<i>expovariate(lambd)</i>	Gibt eine exponentialverteilte Zufallszahl zurück.	–
<i>gammavariate(alpha,beta)</i>	Gibt eine gammaverteilte Zufallszahl zurück.	–
<i>gauss(mu,sigma)</i>	Gibt eine Gaußverteilte Zufallszahl zurück.	–
<i>lognormvariate(mu,sigma)</i>	Gibt eine log-normalverteilte Zufallszahl zurück.	–
<i>normalvariate(mu,sigma)</i>	Gibt eine normalverteilte Zufallszahl zurück. Diese Funktion ist äquivalent zu <i>gauss</i> .	–
<i>paretovariate(alpha)</i>	Gibt eine Pareto-verteilte Zufallszahl zurück.	–
<i>triangular(low,high,mode)</i>	Gibt eine dreiecksverteilte Zufallszahl zurück.	–
<i>vonmisesvariate(mu,kappa)</i>	Gibt eine von-Mises-verteilte Zufallszahl zurück.	–
<i>weibullvariate(alpha,beta)</i>	Gibt eine Weibull-verteilte Zufallszahl zurück.	–

Tabelle 13.4 Funktionen des Moduls random (Forts.)

Bevor Sie die Beispiele dieses Abschnitts ausprobieren können, müssen Sie das Modul `random` einbinden:

```
>>> import random
```

13.2.1 Steuerungsfunktionen

`seed([x], version=2)`

Die Funktion `seed` 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.

Hinweis

Mit Python 3.2 hat sich die Berechnung des Seeds aus `str`-, `bytes`- und `bytearray`-Instanzen verändert. Wenn Sie das alte Verhalten benötigen, können Sie für den Parameter *version* den Wert 1 übergeben.



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.

setstate(state)

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)
```

13.2.2 Funktionen für ganze Zahlen

randint(a, b)

Die Funktion *randint* erzeugt eine zufällige, ganze Zahl n , sodass gilt: $a \leq n \leq b$.

```
>>> random.randint(0, 10)
2
>>> random.randint(0, 10)
7
```

randrange([start,]stop[, step])

Die Funktion *randrange* gibt ein zufällig gewähltes Element dem Zahlenraum zurück, den ein Aufruf der Built-in Function *range* mit gleichen Parametern erzeugen würde.

```
>>> random.randrange(0, 50, 2)
40
```

Die Funktion *randrange* ist äquivalent zu *choice(range(start, stop, step))*, erzeugt aber intern kein *range*-Objekt.

13.2.3 Funktionen für Sequenzen

choice(seq)

Die Funktion *choice* 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
>>> random.choice(["A", "B", "C"])
'B'
```

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 dass mehrfach in *population* vorkommende Werte auch mehrfach gezogen werden können.

```
>>> 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 Funktion *range* verwendet werden:

```
>>> random.sample(range(10000000), 3)
[4571575, 2648561, 2009814]
```

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 verändert 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*. Durch Implementieren einer solchen Funktion ist es möglich, *shuffle* einen eigenen Zufallszahlengenerator vorzugeben.

```
>>> l = [1,2,3,"A","B"]
>>> random.shuffle(l)
>>> l
[1, 'B', 2, 'A', 3]
```

13.2.4 Alternative Generatoren

SystemRandom([seed])

Das Modul *random* enthält zusätzlich zu den oben erläuterten Funktionen eine Klasse namens *SystemRandom*, die es ermöglicht, den Zufallszahlengenerator des Betriebssystems statt des Python-eigenen zu verwenden. Diese Klasse existiert nicht auf allen, aber auf den gängigsten Betriebssystemen.

Beim Instanzieren 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
```

13.3 Präzise Dezimalzahlen – decimal

Sicherlich erinnern Sie sich noch an folgendes Beispiel, das zeigt, dass bei der Verwendung des eingebauten Datentyps *float* Rechenfehler auftreten:

```
>>> 1.1 + 2.2
3.3000000000000003
```

Das liegt daran, dass nicht jede Dezimalzahl durch das interne Speichermodell von `float` dargestellt werden kann, sondern nur mit einer gewissen Genauigkeit angenähert wird.³ Diese Rechenfehler werden jedoch in der Regel aus Gründen der Effizienz in Kauf genommen. In einigen Fällen, beispielsweise beim Rechnen mit Geldbeträgen, ist die Exaktheit des Ergebnisses jedoch wichtiger als die Rechenzeit. 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.

13.3.1 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 instanziiieren. Den Wert können Sie dem Konstruktor in Form eines Strings übergeben:

```
>>> Decimal("0.9")
Decimal("0.9")
>>> Decimal("1.33e7")
Decimal("1.33E+7")
```

³ Dabei handelt es sich nicht um ein Python-spezifisches Problem, sondern um eine grundsätzliche Einschränkung der Gleitkommadarstellung im Computer.

Dies ist die geläufigste Art, `Decimal` zu instanziiieren. 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 Fall eines Tupels bestimmt das erste Element 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 ist die Verschiebung des Dezimalpunktes in der im vorangegangenen Element angegebenen Zahl.

Hinweis

Seit Python 3.2 ist es möglich, dem Konstruktor der `Decimal`-Klasse eine Gleitkommazahl direkt zu übergeben:

```
>>> Decimal(0.7)
Decimal('0.6999999999999999555910790149937383830547332763671875')
```

Dabei sollten Sie stets im Hinterkopf behalten, dass sich dann die Ungenauigkeit von `float` bei der Initialisierung auf die `Decimal`-Instanz überträgt.

Sobald eine `Decimal`-Instanz erzeugt wurde, kann sie wie eine Instanz eines 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 gut 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 Nachkommanteil 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")
```



Decimal-Instanzen können untereinander oder mit Instanzen anderer numerischer Datentypen verglichen werden.

```
>>> Decimal("0.7") < Decimal("0.8")
True
>>> Decimal(0.7) == 0.7
True
>>> Decimal("0.7") == 0.7
False
```

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
```



Hinweis

Auch wenn Decimal-Instanzen an Funktionen des Moduls `math` übergeben werden können, geben diese Funktionen niemals eine Decimal-Instanz zurück. Sie laufen also Gefahr, durch den `float`-Rückgabewert an Genauigkeit zu verlieren.

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 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>	$d \cdot other + third^4$
<code>d.ln([context])</code>	$\log_e(d)$
<code>d.log10([context])</code>	$\log_{10}(d)$
<code>d.logb([context])</code>	$\log_b(d)$
<code>d.sqrt([context])</code>	$\sqrt{d}$

Tabelle 13.5 Mathematische Methoden des Datentyps `Decimal`

Die Verwendung dieser Methoden demonstriert das folgende Beispiel:

13

```
>>> 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")
```

13.3.2 Nichtnumerische Werte

Aus Abschnitt 7.3.5, »Gleitkommazahlen – float«, kennen Sie bereits die Werte `nan` und `inf` des Datentyps `float`, die immer dann auftreten, wenn eine Berechnung

4 Der Vorteil dieser Methode ist, dass sie die Berechnung »in einem Guss« durchführt, dass also nicht mit einem gerundeten Zwischenergebnis der Multiplikation weitergerechnet wird.

nicht möglich ist bzw. eine Zahl den Zahlenraum von `float` sprengt. 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> werden durchgeführt, ergeben allerdings immer wieder <code>NaN</code> .

Tabelle 13.6 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
```

13.3.3 Das Context-Objekt

Der Datentyp `Decimal` erlaubt es, Dezimalzahlen mit beliebiger Genauigkeit zu speichern. Die Genauigkeit, also 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 mit 28 vorbelegt.

```
>>> 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`.

13.4 Spezielle Generatoren – itertools

An dieser Stelle möchten wir Ihnen das Modul `itertools` der Standardbibliothek vorstellen, das eine Reihe von Generatorfunktionen enthält, die man im Program-

mieralltag 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«.

Die folgende Tabelle listet die im Modul `itertools` enthaltenen Generatoren auf und verweist jeweils auf eine detaillierte Erklärung.

Funktion	Beschreibung	Seite
<code>accumulate(iterable)</code>	Durchläuft die Partialsummen der Elemente aus <i>iterable</i> .	407
<code>chain(*iterables)</code>	Durchläuft die Verkettung der übergebenen iterierbaren Objekte.	407
<code>combinations(iterable,r)</code>	Durchläuft alle <i>r</i> -elementigen Kombinationen aus <i>iterable</i> .	408
<code>combinations_with_replacement(iterable,r)</code>	Durchläuft alle <i>r</i> -elementigen Kombinationen aus <i>iterable</i> (mit Zurücklegen).	408
<code>compress(data,selectors)</code>	Durchläuft die Elemente von <i>data</i> , für die das korrespondierende Element von <i>selectors</i> den Wert <code>True</code> ergibt.	409
<code>count(start,step)</code>	Zählt, beginnend mit <i>start</i> , Zahlen im Abstand von <i>step</i> auf.	409
<code>cycle(iterable)</code>	Durchläuft in einer Endlosschleife die Elemente von <i>iterable</i> .	410
<code>dropwhile(predicate,iterable)</code>	Durchläuft alle Elemente von <i>iterable</i> ab dem Element, für das <i>predicate</i> zum ersten Mal den Wert <code>False</code> ergibt.	410
<code>filterfalse(predicate,iterable)</code>	Durchläuft alle Elemente von <i>iterable</i> , für die <i>predicate</i> den Wert <code>False</code> ergibt.	410
<code>groupby(iterable,key)</code>	Durchläuft die Elemente von <i>iterable</i> , gruppiert nach der Schlüsselfunktion <i>key</i> .	411
<code>islice(iterable[,start], stop[,step])</code>	Ermöglicht das Slicing von iterierbaren Objekten.	412
<code>permutations(iterable,r)</code>	Durchläuft alle <i>r</i> -elementigen Permutationen aus <i>iterable</i> .	412

Tabelle 13.7 Funktionen des Moduls `itertools`

Funktion	Beschreibung	Seite
<i>product(*iterables,repeat)</i>	Durchläuft das kartesische Produkt der übergebenen iterierbaren Objekte.	412
<i>repeat(object[,times])</i>	Wiederholt das Objekt <i>times</i> Mal.	413
<i>starmap(function,iterable)</i>	Ruft die Funktion <i>function</i> mit den Elementen aus <i>iterable</i> als Parameter und durchläuft die Ergebnisse.	413
<i>takewhile(predicate,iterable)</i>	Durchläuft alle Elemente von <i>iterable</i> bis zu dem Element, für das <i>predicate</i> zum ersten Mal den Wert <code>False</code> ergibt.	414
<i>tee(iterable,n)</i>	Erzeugt <i>n</i> unabhängige Iteratoren über <i>iterable</i> .	414
<i>zip_longest(*iterables,fillvalue)</i>	Wie die Built-in Funktion <i>zip</i> , aber schneidet die iterierbaren Objekte nicht bei der Länge des kürzesten ab.	414

Tabelle 13.7 Funktionen des Moduls `itertools` (Forts.)

Zur Veranschaulichung werden in den folgenden Beispielen die von den Generatorfunktionen zurückgegebenen Iteratoren in Listen überführt und ausgegeben. Um die Beispiele nachvollziehen zu können, müssen Sie zuvor natürlich das Modul `itertools` importiert haben.

accumulate(iterable)

Die Funktion *accumulate* erzeugt einen Iterator, der die Partialsummen der Elemente von *iterable* durchläuft. Dies wird durch das folgende Beispiel veranschaulicht:

```
>>> list(itertools.accumulate([1,2,3,4]))
[1, 3, 6, 10]
```

Der erzeugte Iterator durchläuft die Elemente 1, 1+2, 1+2+3 und 1+2+3+4.

chain(*iterables)

Die Funktion *chain* (dt. »verketteten«) 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)

Die Funktion *combinations* 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 erzeugten 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* eine bessere Wahl.

combinations_with_replacement(iterable, r)

Die Funktion *combinations_with_replacement* durchläuft, wie *combinations*, alle *r*-elementigen Kombinationen aus *iterable*, allerdings mit Zurücklegen. Das bedeutet, dass ein Element aus *iterable* mehrfach in einer Kombination vorkommen darf.

```
>>> list(itertools.combinations_with_replacement(range(3), 2))
[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]
```

Wie bei *combinations* kommt es auch hier nicht auf die Reihenfolge der Elemente an.

compress(data, selectors)

Die Funktion *compress* erzeugt einen Iterator, der diejenigen Elemente des iterierbaren Objekts *data* durchläuft, deren korrespondierendes Element in *selectors* den Wert *True* hat. Das wird an folgendem Beispiel deutlich:

```
>>> list(itertools.compress("ABCDEFGH", [1,1,1,0,0,1,0,1]))
['A', 'B', 'C', 'F', 'H']
```

Die für *selectors* übergebene Liste gibt an, dass die ersten drei, sowie das sechste und achte Element von *data* durchlaufen werden sollen.

count(start=0, step=1)

Die Funktion *count* erzeugt einen Iterator, der die Werte $start + n \cdot step$ für alle $n \geq 0$, beginnend bei 0, durchläuft. Sowohl für *start* als auch für *step* können Gleitkommazahlen übergeben werden. 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* (S. 245). 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)
...
0
1
4
9
16
25

```

cycle(iterable)

Die Funktion *cycle* durchläuft alle Elemente des iterierbaren Objekts *iterable* und fängt danach wieder von vorn an. Der von *cycle* erzeugte Iterator läuft in einer Endlosschleife und muss daher von außen unterbrochen werden. 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* den Wert `True` zurückgegeben hat. Nachdem *predicate* zum ersten Mal den Wert `False` zurückgegeben hat, wird jedes nachfolgende Element von *iterable* durchlaufen, unabhängig davon, was *predicate* für diese Elemente zurückgibt. Man kann sich die Funktion *predicate* also als eine Art Startsignal vorstellen. 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)

Die Funktion *filterfalse* durchläuft alle Elemente von *iterable*, für die die Funktion *predicate* den Wert `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']
```

groupby(iterable, key=None)

Die Funktion *groupby* erzeugt einen Iterator, der die Elemente aus *iterable* gruppiert durchläuft. Die Gruppierung wird dabei anhand der für *key* übergebenen Schlüssel-funktion durchgeführt. Wenn der Parameter *key* nicht angegeben wird, werden die Elemente anhand ihres Wertes gruppiert.

Der von *groupby* erzeugte Iterator durchläuft Tupel, die den jeweiligen Gruppenschlüssel und einen Iterator über die Gruppenelemente enthalten. Das folgende Beispiel demonstriert die Funktionsweise von *groupby*:

```
>>> for l in list(itertools.groupby("AAABBBCCC")):
...     print(list(l))
...
['A', <itertools._grouper object at 0x7f4784b6c310>]
['B', <itertools._grouper object at 0x7f4784b6c5d0>]
['C', <itertools._grouper object at 0x7f4784b6c390>]

>>> [list(g) for k, g in itertools.groupby('AAABBBCCC')]
[['A', 'A', 'A'], ['B', 'B', 'B'], ['C', 'C', 'C']]
```

Mithilfe einer eigenen Schlüsselfunktion können die Elemente nach anderen Gesichtspunkten gruppiert werden. Im folgenden Beispiel wird eine Schlüsselfunktion eingesetzt, um eine Gruppierung nach der Wortlänge durchzuführen.

```
>>> def f(x):
...     return len(x)
...
>>> words = ["for", "while", "and", "or", "if", "elif", "else"]
>>> [list(g) for k, g in itertools.groupby(words, f)]
[['for'], ['while'], ['and'], ['or', 'if'], ['elif', 'else']]
```


Hier zeigt sich eine wichtige Anforderung an die Reihenfolge der Elemente in *iterable*. Obwohl die Wörter »for« und »and« gleich lang sind, wurden sie nicht zu einer Gruppe zusammengefasst. Damit das Gruppieren mit *groupby* funktioniert, müssen die in *iterable* enthaltenen Objekte im Hinblick auf die eingesetzte Schlüssel-funktion vorsortiert werden.

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])

Die Funktion *permutations* 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])

Die Funktion *product* erzeugt einen Iterator, der das 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 Objekts 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 veranschaulicht das folgende Beispiel:

```
>>> 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 es der Funktion mehrfach übergeben werden muss:

```
>>> 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])

Die Funktion *repeat* 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']
```

Wenn der Parameter *times* nicht angegeben wird, läuft der von *repeat* zurückgegebene Iterator in einer Endlosschleife.

starmap(function, iterable)

Die Funktion *starmap* arbeitet ähnlich wie die Built-in Function *map*. Die Funktion *function* wird für jede in *iterable* enthaltene Parameterliste aufgerufen. Der von *starmap* erzeugte Iterator durchläuft die Ergebnisse dieser Funktionsaufrufe.

```
>>> list(itertools.starmap(max, [(1,2), (4,4,3,6), [2,3,9]]))
[2, 6, 9]
```

Im Beispiel wurde die Funktion *starmap* gemeinsam mit der Built-in Function *max* eingesetzt, um die jeweils größten Elemente der Tupel zu durchlaufen.

`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 den Wert `True` zurückgibt. Sobald ein *predicate*-Aufruf den Wert `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.

`tee(iterable, n=2)`

Die Funktion *tee* erzeugt *n* voneinander unabhängige Iteratoren über die Elemente von *iterable*.

```
>>> list(itertools.tee([1,2,3,4]))
[<itertools.tee object at 0x190b128>,
 <itertools.tee object at 0x1987368>]
>>> [list(x) for x in itertools.tee([1,2,3,4])]
[[1, 2, 3, 4], [1, 2, 3, 4]]
```

Nach dem Aufruf von *tee* sollten nur noch die von *tee* zurückgegebenen Iteratoren verwendet werden und nicht mehr der als Parameter übergebene.

`zip_longest(*iterables, fillvalue=None)`

Die Funktion *zip_longest* arbeitet ähnlich wie die Built-in Function *zip*. Der Unterschied ist, dass bei *zip_longest* stets das längste der übergebenen iterierbaren Objekte ausschlaggebend ist und fehlende Elemente bei den anderen Objekten mit *fillvalue* aufgefüllt werden.

```
>>> list(zip("ABC", "abcde"))
[('A', 'a'), ('B', 'b'), ('C', 'c')]
>>> list(itertools.zip_longest("ABC", "abcde"))
[('A', 'a'), ('B', 'b'), ('C', 'c'), (None, 'd'), (None, 'e')]
>>> list(itertools.zip_longest("ABC", "abcde", fillvalue="-"))
[('A', 'a'), ('B', 'b'), ('C', 'c'), ('-', 'd'), ('-', 'e')]
```

Kapitel 14

Strings

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

Die in diesem Kapitel besprochenen Module stellen jeweils komfortable Funktionalität bereit, die im Zusammenhang mit Strings steht. Dabei handelt es sich um die in der folgenden Tabelle aufgelisteten Module.

Modul	Beschreibung	Seite
<i>re</i>	für die Arbeit mit regulären Ausdrücken	415
<i>gettext</i>	für die einfache Lokalisierung von Programmen	442
<i>hashlib</i>	Stellt diverse Arten von Hashfunktionen bereit.	446

Tabelle 14.1 String-Module der Standardbibliothek

14.1 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.

Beim Searching handelt es sich 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.



Hinweis

Sollten Sie sich mit regulären Ausdrücken bereits auskennen, sind Sie vielleicht gerade auf ein Problem aufmerksam geworden, denn der Backslash ist ein 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 Syntax regulärer Ausdrücke einführen. 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.

14.1.1 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 *Zeichenlitterale* genannt. Zeichenlitterale innerhalb regulärer Ausdrücke sind *case sensitive*, das heißt, dass der obige Ausdruck nicht auf den String "Python" passen würde.

In regulären Ausdrücken können eine 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.¹ 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 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 einfach hintereinander:

```
r"[A-Za-z]ython"
```

¹ Hier zeigt sich, dass ein Punkt innerhalb eines regulären Ausdrucks eine spezielle Bedeutung hat. Um das eigentliche Zeichen "." zu beschreiben, muss ihm im regulären Ausdruck ein Backslash vorangestellt werden: `r"P\..thon"` passt nur auf den String `"P..thon"`. Dies gilt analog für andere Sonderzeichen wie beispielsweise Klammern.

Dieser reguläre Ausdruck passt beispielsweise sowohl auf "Qython" als auch auf "qython", nicht aber auf "Sython" oder "3ython". Auch Ziffernbereiche können als Zeichenklasse verwendet werden:

```
r"[0-9]ython"
```

Als letzte Möglichkeit, die eine Zeichengruppe bietet, können Zeichen oder Zeichenbereiche ausgeschlossen werden. Dazu wird zu Beginn der Zeichengruppe ein Zirkumflex (^) geschrieben. So erlaubt der reguläre Ausdruck

```
r"^[^pP]ython"
```

jedes Zeichen, abgesehen von einem großen oder kleinen »P«. Demzufolge würden sowohl "Sython" als auch "wython" passen, während "Python" und "python" außen vor bleiben.

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 stehen allerdings vor einem Problem, wenn wir an einer bestimmten Stelle des Wortes eine gewisse Anzahl oder gar beliebig viele dieser Zeichen erlauben möchten. 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 14.2 Quantoren in regulären Ausdrücken

Die folgenden drei Beispiele zeigen einen regulären Ausdruck mit je einem Quantor.

- ▶ `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 beispielsweise auf die Wörter "Python" und "Pthon", 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 beispielsweise auf die Wörter "Python", "Pthon" und "PyyYYyython", 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 beispielsweise auf die Wörter "Python", "PYthon" und "PyyYYyython", 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 14.3 Quantoren in regulären Ausdrücken

Auch für diese Quantoren möchten wir das bisherige Beispiel abändern und untersuchen, was sie für Auswirkungen haben.

- ▶ `r"P[Yy]{2}thon"`
Dieser reguläre Ausdruck erwartet an der zweiten Stelle des Wortes exakt zwei jeweils große oder kleine »Y«. Damit passt der Ausdruck beispielsweise auf die Wörter "Python" oder "PYython", jedoch nicht auf "Pyython".
- ▶ `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 beispielsweise auf die Wörter "Python", "PYython" und "PyyYYyython", 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 beispielsweise auf die Wörter "Python", "Pthon" und "PYYthon", jedoch nicht auf "Pyython".
- ▶ `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 beispielsweise auf die Wörter "Python" oder "PYython", jedoch nicht auf "Pthon" oder "PYYython".

Vordefinierte Zeichenklassen

Damit Sie nicht bei jedem regulären Ausdruck das Rad neu erfinden 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 häufig benötigt und können deswegen durch einen speziellen Code abgekürzt werden. Jeder dieser Codes beginnt mit einem Backslash. Tabelle 14.4 listet alle vordefinierten Zeichenklassen mit ihren Bedeutungen auf.

Zeichenklasse	Bedeutung
<code>\d</code>	Passt auf alle Zeichen, die Ziffern des Dezimalsystems sind. Ist äquivalent zu <code>[0-9]</code> .
<code>\D</code>	Passt auf alle Zeichen, die nicht Ziffern des Dezimalsystems sind. Ist äquivalent zu <code>[^0-9]</code> .
<code>\s</code>	Passt auf alle Whitespace-Zeichen. Ist äquivalent zu <code>[\t\n\r\f\v]</code> .
<code>\S</code>	Passt auf alle Zeichen, die kein Whitespace sind. Ist äquivalent zu <code>^[^\t\n\r\f\v]</code> .

Tabelle 14.4 Vordefinierte Zeichenklassen in regulären Ausdrücken

Zeichenklasse	Bedeutung
<code>\w</code>	Passt auf alle alphanumerischen Zeichen und den Unterstrich. Ist äquivalent zu <code>[a-zA-Z0-9_]</code> .
<code>\W</code>	Passt auf alle Zeichen, die nicht alphanumerisch und kein Unterstrich sind. Ist äquivalent zu <code>[^a-zA-Z0-9_]</code> .

Tabelle 14.4 Vordefinierte Zeichenklassen in regulären Ausdrücken (Forts.)

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`.²

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 verlangt, Regeln aufstellen zu können, die über die bloße Zeichenebene hinausgehen. So ist es beispielsweise interessant, einen regulären Ausdruck zu erschaffen, der nur passt, wenn sich das Wort am Anfang oder Ende einer Textzeile befindet. Für solche und ähnliche Fälle gibt es einen bestimmten Satz an zusätzlichen Sonderzeichen, die genauso angewendet werden wie die vordefinierten Zeichenklassen.

² Eine Erklärung der Escape-Sequenzen finden Sie auf S. 144. Die Escape-Sequenz `\x` wird im Abschnitt »Zeichensätze und Sonderzeichen« (S. 165) erklärt.

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. ⁴
\$	Passt nur am Ende eines Strings. Wenn das MULTILINE-Flag gesetzt wurde, passt \$ auch direkt vor jedem Newline-Zeichen innerhalb des Strings.

Tabelle 14.5 Vordefinierte Zeichenklassen in regulären Ausdrücken

Im konkreten Beispiel passt also der reguläre Ausdruck

```
r"Python\Z"
```

nur bei dem String "Python", nicht jedoch bei "Python rockt".

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 Interesse. 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.

³ Näheres zu dem MULTILINE-Flag erfahren Sie im Abschnitt »Flags« (S. 431).

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, sodass 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 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. Die Gruppe muss aber mindestens einmal auftreten, der Ausdruck passt nicht auf den String `" ist gut"`.

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 14.1.2, »Verwendung des Moduls `re`« (S. 427).

- Es gibt Gruppen, die in einem regulären Ausdruck häufiger gebraucht werden. Um diese nicht jedes Mal erneut schreiben zu müssen, werden Gruppen, mit 1 beginnend, durchnummeriert und können dann anhand ihres Index referenziert werden. Eine solche Referenz besteht aus einem Backslash, gefolgt von dem Index der jeweiligen Gruppe, und passt auf den gleichen Teilstring, auf den die Gruppe gepasst hat. So passt der reguläre Ausdruck `r"(Python) \1"` auf `"Python Python"`.

Alternativen

Eine weitere Möglichkeit, die die Syntax regulärer Ausdrücke vorsieht, sind sogenannte *Alternativen*. Im Prinzip handelt es sich dabei um nichts anderes als um eine ODER-Verknüpfung zweier Zeichen oder Zeichengruppen, wie Sie sie bereits von dem Operator `or` her kennen. Diese Verknüpfung wird durch den senkrechten Strich `|`, auch *Pipe* genannt, durchgeführt.

```
r"P(ython|eter)"
```

Dieser reguläre Ausdruck passt sowohl auf den String `"Python"` als auch auf `"Peter"`. Durch die Gruppe kann später ausgelesen werden, welche der beiden Alternativen aufgetreten ist.

Extensions

Damit ist 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 nicht zwingend eine 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. Dabei bezeichnet jedes der Zeichen »a«, »i«, »L«, »m«, »s«, »u« und »x« ein bestimmtes Flag. 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` (S. 431) 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)"
```

Mithilfe dieses Konstrukts lässt sich ein regulärer Ausdruck strukturieren, ohne dass dabei ein Overhead durch die Bildung von Gruppen entsteht.

(?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)

Diese Extension 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"
```

(?=...)

Diese Extension passt nur dann, wenn der reguläre Ausdruck ... als Nächstes passt. Diese Extension greift also vor, ohne in der Auswertung des Ausdrucks tatsächlich voranzuschreiten.

Mit dem regulären Ausdruck `r"\w+(?= Meier)"` lässt sich beispielsweise nach den Vornamen aller im Text vorkommenden Meiers suchen. Die naheliegende Alternative `r"\w+ Meier"` würde nicht nur die Vornamen, sondern immer auch den Nachnamen Meier in das Ergebnis mit aufnehmen. Dieses Beispiel können Sie leicht auf die folgenden drei Extensions übertragen, weswegen wir dort auf ein Beispiel verzichten.

(?!...)

Diese Extension passt nur dann, wenn der reguläre Ausdruck ... als Nächstes nicht passt. Diese Extension ist das Gegenstück zu der vorherigen.

(?<=...)

Diese Extension 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 passt nur, wenn der reguläre Ausdruck ... zuvor nicht gepasst hat. Diese Extension ist damit das Gegenstück zu der vorherigen.

(?(id/name)yes-pattern|no-pattern)

Diese 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.

```
r"(?P<klammer>\()?Python(?:\kammer\))"
```

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.

14.1.2 Verwendung des Moduls `re`

Nachdem die Syntax der regulären Ausdrücke besprochen wurde, werden wir uns hier um ihre konkrete Verwendung in Python kümmern. Die folgende Tabelle listet die im Modul `re` enthaltenen Funktionen auf und beschreibt diese kurz.

Funktion	Beschreibung	Seite
<code>compile(pattern[,flags])</code>	Kompiliert den regulären Ausdruck <i>pattern</i> zu einem Regular-Expression-Objekt.	431
<code>escape(string)</code>	Ersetzt problematische Zeichen in <i>string</i> durch ihre Escape-Sequenzen.	428
<code>findall(pattern,string[,flags])</code>	Sucht nach allen Teilen von <i>string</i> , die auf den regulären Ausdruck <i>pattern</i> passen.	428
<code>finditer(pattern,string[,flags])</code>	Wie <i>findall</i> , gibt das Ergebnis aber als Iterator über Match-Objekte und nicht als Liste von Strings zurück.	–

Tabelle 14.6 Funktionen des Moduls `re`


```
>>> re.findall(r"P[Yy]thon", "Python oder PYthon und Python")
['Python', 'PYthon', 'Python']
```

Wenn *pattern* ein oder mehrere Gruppen enthält, werden diese anstelle der übereinstimmenden Teilstrings in die Ergebnisliste geschrieben.

```
>>> re.findall(r"P([Yy])thon", "Python oder PYthon und Python")
['y', 'Y', 'y']
>>> re.findall(r"P([Yy])th(.)n", "Python oder PYthon und Python")
[('y', 'o'), ('Y', 'o'), ('y', 'o')]
```

Bei mehreren Gruppen handelt es sich um eine Liste von Tupeln.

match(pattern, string, flags=0)

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>
```

search(pattern, string, flags=0)

Die Funktion *search* 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>
```

split(pattern, string, maxsplit=0, flags=0)

Der String *string* wird nach Übereinstimmungen mit dem regulären Ausdruck *pattern* durchsucht. Alle passenden Teilstrings werden als Trennzeichen angesehen, und die dazwischenliegenden Teile werden als Liste von Strings zurückgegeben.

```
>>> re.split(r"\s", "Python Python Python")
['Python', 'Python', 'Python']
```

Eventuell vorkommende Gruppen innerhalb des regulären Ausdrucks werden ebenfalls als Elemente dieser Liste zurückgegeben:

```
>>> re.split(r"(\s)", "Python, Python, Python")
['Python', ' ', 'Python', ' ', 'Python']
```

In diesem regulären Ausdruck werden alle von einem Whitespace gefolgten Kommata 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.

sub(pattern, repl, string, count=0, flags=0)

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. Im folgenden Beispiel wird für den Parameter *repl* ein Funktionsobjekt übergeben, um unanständige Wörter in einem Text zu zensieren:

```
>>> def f(m):
...     return "x" * len(m.group(0))
...
>>> re.sub(r"\b(\w*?sex\w*?)\b", f,
...       "Wirtschaftsexperten auf Arktisexpedition")
'xxxxxxxxxxxxxxxxxxxx auf xxxxxxxxxxxxxxxxxxxx'
```

Die Funktion *sub* sucht im angegebenen Text nach Wörtern, die das Teilwort »sex« enthalten. Diese Wörter werden dann mithilfe der Funktion *f* durch genauso viele »x« ersetzt, wie das entsprechende Wort lang ist.

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.

subn(pattern, repl, string[, count])

Die Funktion *subn* 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"([J]java)", "Python statt \g<1>", "Nimm doch Java")
('Nimm doch Python statt Java', 1)
```

compile(pattern[, flags])

Diese Funktion 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 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[yV]thon")
>>> c2 = re.compile(r"P[y]thon", re.I)
>>> c3 = re.compile(r"P[y]thon", re.I | re.S)
```

Die Angabe von Flags ist bei den meisten Funktionen des Moduls *re* über den Parameter *flags* möglich. Wir werden darauf in Zukunft nicht mehr eingehen.

Näheres zum RE-Objekt folgt im nächsten Abschnitt.

Flags

In den vorherigen Abschnitten 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
<i>re.A</i>	<i>re.ASCII</i>	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.
<i>re.I</i>	<i>re.IGNORECASE</i>	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.
<i>re.L</i>	<i>re.LOCALE</i>	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> .
<i>re.M</i>	<i>re.MULTILINE</i>	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.
<i>re.S</i>	<i>re.DOTALL</i>	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> .
<i>re.U</i>	<i>re.UNICODE</i>	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.
<i>re.X</i>	<i>re.VERBOSE</i>	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 14.7 Flags

Das Regular-Expression-Objekt

Ein Regular-Expression-Objekt, hier 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 zusammengefasst 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 häufig auftretenden Parameter `pos` und `endpos` dienen dazu, einen Bereich des ebenfalls anzugebenden Strings zu bestimmen, auf den sich eine Operation bezieht. Beispielsweise kann man auf diese Weise ein Matching auf einem Teilstring durchführen. Wenn `pos` und `endpos` nicht angegeben wurden, wird die jeweilige Operation auf dem ganzen String durchgeführt.

Methode bzw. Attribut	Beschreibung	Seite
Methoden		
<code>match(string[,pos[,endpos]])</code>	Führt ein Matching auf <i>string</i> durch. Ist äquivalent zu <i>re.match</i> .	434, 429
<code>search(string[,pos[,endpos]])</code>	Ist äquivalent zu <i>re.search</i> .	429
<code>split(string[,maxsplit])</code>	Ist äquivalent zu <i>re.split</i> .	429
<code>findall(string[,pos[,endpos]])</code>	Ist äquivalent zu <i>re.findall</i> .	428
<code>finditer(string[,pos[,endpos]])</code>	Ist äquivalent zu <i>re.finditer</i> .	–
<code>sub(repl,string[,count])</code>	Ist äquivalent zu <i>re.sub</i> .	430
<code>subn(repl,string[,count])</code>	Ist äquivalent zu <i>re.subn</i> .	431
Attribute		
<code>flags</code>	Enthält alle gesetzten Flags.	434
<code>groups</code>	Enthält die Anzahl der Gruppen im regulären Ausdruck.	–

Tabelle 14.8 Methoden und Attribute eines RE-Objekts

Methoden bzw. Attribut	Beschreibung	Seite
<i>groupindex</i>	Ein Dictionary, das die Namen benannter Gruppen auf ihre Indizes abbildet.	435
<i>pattern</i>	Enthält den regulären Ausdruck, der dem RE-Objekt zugrunde liegt.	–

Tabelle 14.8 Methoden und Attribute eines RE-Objekts (Forts.)

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]])

Diese Funktion ist 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.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 UND⁴ 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}
```

14

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 Methoden und Attribute eines Match-Objekts sind in der folgenden Tabelle zusammengefasst.

Methode bzw. Attribut	Beschreibung	Seite
Methoden		
<i>expand(template)</i>	Füllt einen Template-String mit Gruppeninhalten.	436
<i>group(*groups)</i>	Gibt die Teilstrings zurück, die auf die angegebenen Gruppen gepasst haben.	437
<i>groups([default])</i>	Gibt die Teilstrings zurück, die auf eine der Gruppen gepasst haben.	437

Tabelle 14.9 Methoden und Attribute eines Match-Objekts

⁴ Näheres zur Funktionsweise des bitweisen UND können Sie im Abschnitt »Bit-Operationen« (S. 99) nachlesen.

Methode bzw. Attribut	Beschreibung	Seite
<i>groupdict</i> (<i>[default]</i>)	Gibt ein Dictionary zurück, das die Namen benannter Gruppen auf die darauf passenden Teilstrings abbildet.	437
<i>start</i> (<i>[group]</i>)	Gibt den Startindex des Teilstrings zurück, der auf die Gruppe <i>group</i> gepasst hat.	437
<i>end</i> (<i>[group]</i>)	Gibt den Endindex des Teilstrings zurück, der auf die Gruppe <i>group</i> gepasst hat.	437
<i>span</i> (<i>[group]</i>)	Gibt das Tupel (<i>start</i> (<i>group</i>), <i>end</i> (<i>group</i>)) zurück.	–
Attribute		
<i>pos</i> , <i>endpos</i>	Start- bzw. Endindex des Teilstrings auf den sich das Matching bzw. Searching bezog, das dieses Match-Objekt ergab	–
<i>lastindex</i>	der Index der Gruppe, die bei der Auswertung als Letzte auf einen Teilstring gepasst hat, oder <i>None</i> , wenn keine Gruppe gepasst hat	–
<i>lastgroup</i>	der Name der symbolischen Gruppe, die bei der Auswertung als Letzte auf einen Teilstring gepasst hat, oder <i>None</i> , wenn keine Gruppe gepasst hat	–
<i>re</i>	der ursprüngliche reguläre Ausdruck als String	–
<i>string</i>	der String, auf den sich dieses Match-Objekt bezieht	–

Tabelle 14.9 Methoden und Attribute eines Match-Objekts (Forts.)

Die Beispiele in diesem Abschnitt verstehen sich in folgendem Kontext:

```
>>> import re
>>> m = re.match(r"(P[\\y\\Y])(th.n)", "Python")
```

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, dass Sie *template* wegen der Backslashes als Raw-String angeben sollten.

```
>>> 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.

```
>>> m.group(1)
'Py'
>>> m.group(1, 2)
('Py', 'thon')
```

Ein Index von 0 gibt den vollständigen passenden String zurück.

```
>>> m.group(0)
'Python'
```

14

m.groups([default])

Diese Methode 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.groups()
('Py', 'thon')
```

m.groupdict([default])

Diese Methode 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])

Die Methode *start* gibt den Start- bzw. Endindex des Teilstrings zurück, der auf die Gruppe *group* gepasst hat. Der optionale Parameter *group* ist mit 0 vorbelegt.

```
>>> m.start(2)
2
>>> m.end(2)
6
```

Wenn eine Gruppe mehrfach gepasst hat, zählt der Teilstring, auf den die Gruppe letztmalig passte.

14.1.3 Ein einfaches Beispielprogramm – Searching

Bisher wurde sowohl die Syntax regulärer Ausdrücke als auch deren Verwendung durch das Modul `re` der Standardbibliothek besprochen. Wir möchten an dieser Stelle noch zwei kleine Beispielprojekte vorstellen, bei denen die Anwendung regulärer Ausdrücke im Mittelpunkt steht. Zunächst erklären wir in diesem relativ einfach gehaltenen Beispiel das Searching und im nächsten, etwas komplexeren Beispiel das Matching. Mit Hilfe 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, sodass diese beiden Angaben später bequem ausgelesen werden können. Außerdem sollten Sie 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 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, wobei in jedem Iterationsschritt die aktuelle Übereinstimmung als Match-Objekt `m` verfügbar ist. Jetzt werden 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.

14.1.4 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 lässt sich dieses Problem mithilfe regulärer Ausdrücke elegant und mit verhältnismäßig wenig Code 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 Kategorienamen nicht vorkommt und somit das erste Auftreten eines Doppelpunktes in einer Zeile stets den Übergang zwischen Kategorie und Information markiert. 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.

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(":",1))
        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 ersten Doppelpunktes in die beiden Teile »Kategorie« 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 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 `regex`, das für jede Überschrift einen regulären Ausdruck bereitstellt, der verwendet werden kann, um die Information zu validieren:

```
regex = {
    "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},)"
}
```

14

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, regex):
    for key in daten:
        if key not in regex:
            return False
        m = re.match(regex[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 `regex` und zum anderen das Dictionary, das von der Funktion `leseDatei` 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` den Wert `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.

14.2 Lokalisierung von Programmen – gettext

Das Modul `gettext` der Standardbibliothek ist bei der *Internationalisierung* und *Lokalisierung* von Python-Programmen von Nutzen. Mit Internationalisierung wird der Vorgang bezeichnet, die Benutzerschnittstelle eines Programms so zu abstrahieren, dass sie leicht an andere sprachliche oder kulturelle Umgebungen angepasst werden kann. Als Lokalisierung 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*⁵ 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

⁵ <http://www.gnu.org/s/gettext>

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 vonstatten geht. 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, ist unter anderem Inhalt des nächsten Abschnitts.

14.2.1 Beispiel für die Verwendung von gettext

Internationalisierung 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. 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 `Enter` 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. Die Interaktion mit dem Benutzer geschieht ausschließlich auf Englisch, jeder String, der ausgegeben wird, wird aber vorher durch eine Funktion namens `_` geschickt.



Hinweis

Bei der `print`-Ausgabe am Ende des Beispielprogramms wird die Funktion `_` für einen mit einem Platzhalter behafteten String aufgerufen, bevor dieser Platzhalter durch dynamischen Inhalt ersetzt wird. 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 anzu-

fertigen, ist in Python ein Programm namens *pygettext.py*⁶ im Lieferumfang enthalten, das genau dies für Sie erledigt. Das Programm erstellt eine sogenannte *.po-Datei*. Das ist eine für Menschen lesbare Variante des *.mo*-Dateiformats. Diese *.po*-Datei wird dann von den Übersetzern in verschiedene Sprachen übersetzt. Dies kann von Hand geschehen oder durch Einsatz diverser Tools, die für diesen Zweck existieren. Die für unser Beispielprogramm erstellte *.po*-Datei sieht folgendermaßen aus:⁷

[...]

```
#: 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*,⁸ 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

Beim Aufruf der Funktion *gettext.translate* wird der Name des Verzeichnisses *Unterordner* angegeben. Er war in unserem Beispiel *locale*. Dieses Verzeichnis muss für jede vorhandene Sprache ein weiteres Verzeichnis enthalten, das seinerseits über ein

6 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.

7 Dabei sind nur die deutschsprachigen Inhalte der Datei manuell hineingeschrieben worden. Der Rest wurde mithilfe von *pygettext* automatisch erzeugt.

8 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*.

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

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 Programmieraufwand in mehrere Sprachen zu übersetzen und darüber hinaus zu einem späteren Zeitpunkt weitere Sprachen hinzuzufügen, ohne Änderungen am Programm selbst durchführen zu müssen.

14.3 Hash-Funktionen – hashlib

Das Modul `hashlib` der Standardbibliothek implementiert die gängigsten sogenannten *Hash-Funktionen*. Ganz allgemein sind das 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 vor, Sie würden eine Foren-Software entwickeln, die später für eine Community im Internet eingesetzt werden soll. Bevor ein Benutzer Beiträge im Forum verfassen darf, muss er sich mit seinem Benutzernamen und dem dazu passenden Passwort anmelden. Natürlich ist es im Sinne des Forenbetreibers und vor allem des Benutzers selbst, dass das Passwort nicht in falsche Hände gerät. Es stellt sich also die Frage, wie die Anmeldeprozedur möglichst sicher gestaltet werden kann.

Die intuitivste Möglichkeit wäre es, Benutzername und Passwort 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. Ein Angreifer, der über eine Sicherheitslücke in einem anderen Teil der Software Zugriff auf die Datenbank erlangt, 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 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.⁹

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

⁹ Da Hash-Funktionen deterministisch sind, ist es für den Angreifer weiterhin möglich, Passwörter auszuprobieren und die Hash-Werte mit den in der Datenbank gespeicherten zu vergleichen. Ein solcher Wörterbuchangriff wird mithilfe eines *Salts* erschwert. Das ist eine Zufallszahl, die an ein Passwort gehängt wird, bevor dessen Hash-Wert bestimmt wird.

können. Eine Hash-Funktion sollte also die Berechnung von Kollisionen so stark erschweren, dass sie nur unter extrem hohem Zeitaufwand zu bestimmen sind.

- Eine Hash-Funktion sollte möglichst willkürlich sein, sodass man nicht aufgrund eines ähnlichen Hash-Wertes darauf schließen kann, dass man in der Nähe des gesuchten Passworts ist. Sobald der Parameter der Hash-Funktion minimal verändert wird, sollte ein völlig verschiedener Hash-Wert berechnet werden.
- Zu guter Letzt sollte eine Hash-Funktion 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.

14.3.1 Verwendung des Moduls

Zunächst enthält das Modul `hashlib` eine Reihe von Klassen, die jeweils einen Hash-Algorithmus implementieren:

Klasse	Algorithmus	Beschreibung
<i>md5</i>	MD5	<i>Message-Digest Algorithm 5</i> Erzeugt aus einem beliebigen String einen 128-Bit-Hash-Wert. Beachten Sie, dass der MD5-Algorithmus bereits ansatzweise geknackt wurde.
<i>sha1</i>	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.

Tabelle 14.10 Unterstützte Hash-Funktionen

Klasse	Algorithmus	Beschreibung
<i>sha224</i>	SHA-224	<i>Secure Hash Algorithm 224</i> Erzeugt aus einem beliebigen String einen 224-Bit-Hash-Wert.
<i>sha256</i>	SHA-256	<i>Secure Hash Algorithm 256</i> Erzeugt aus einem beliebigen String einen 256-Bit-Hash-Wert.
<i>sha384</i>	SHA-384	<i>Secure Hash Algorithm 384</i> Erzeugt aus einem beliebigen String einen 384-Bit-Hash-Wert.
<i>sha512</i>	SHA-512	<i>Secure Hash Algorithm 512</i> Erzeugt aus einem beliebigen String einen 512-Bit-Hash-Wert.

Tabelle 14.10 Unterstützte Hash-Funktionen (Forts.)

Die Verwendung dieser Klassen ist identisch. Deshalb wird sie hier exemplarisch an der Klasse `md5` gezeigt.

Beim Instanziiieren 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'\\7*2\\xc9\\xaet\\x8aL\\x04\\x0e\\xba\\xdcQ\\xa8'
```

Durch Aufruf der Methode `hexdigest` wird der berechnete Hash-Wert als String zurückgegeben, der eine Folge von zweistelligen Hexadezimalzahlen enthält. Diese Hexadezimalzahlen repräsentieren jeweils ein Byte des Hash-Wertes. Der zurückgegebene String enthält ausschließlich druckbare Zeichen.

```
>>> m.hexdigest()
'5c372a32c9ae748a4c040ebadc51a829'
```

14.3.2 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«.

Kapitel 15

Datum und Zeit

»Zehn Minuten!« – Edmund Stoiber

In diesem Kapitel werden Sie die Python-Module kennen lernen, 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.

15

15.1 Elementare Zeitfunktionen – `time`

Bevor wir uns mit den Funktionen des `time`-Moduls beschäftigen, werden 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 sogenannte *Unix-Timestamps*. Ein Unix-Timestamp beschreibt einen Zeitpunkt durch die Anzahl der Sekunden, die seit dem 01.01.1970 um 00:00 Uhr¹ vergangen sind.

Beispielsweise markiert der Unix-Timestamp mit dem Wert 1320075757,0 den 31.10.2011 um 16:42 Uhr und 37 Sekunden, da seit dem Beginn der Unix-Epoche bis zu diesem Zeitpunkt genau 1.320.075.757,0 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 auf das, was die Uhren an diesem Standort anzeigen müssen, um richtig zu gehen. Als

¹ Mit dem 01.01.1970 um 00:00 Uhr ist der Beginn der sogenannten *Unix-Epoche* datiert. Sie wurde zur einheitlichen Beschreibung von Zeiten eingeführt.

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.² 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, sodass wir uns darum nicht zu kümmern brauchen.

Neben der schon angesprochenen Zeitdarstellung durch Unix-Timestamps gibt es eine weitere Darstellung von Zeiten durch den Datentyp *struct_time*. Die 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	Wertebereich
0	<i>tm_year</i>	Die Jahreszahl des Zeitstempels ³	1970–2038
1	<i>tm_mon</i>	Nummer des Monats	1–12
2	<i>tm_mday</i>	Nummer des Tags im Monat	1–31
3	<i>tm_hour</i>	Stunde der Uhrzeit des Zeitstempels	0–23
4	<i>tm_min</i>	Minute der Uhrzeit des Zeitstempels	0–59

Tabelle 15.1 Aufbau des Datentyps *struct_time*

2 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.

3 Diese Begrenzung kommt durch den Wertebereich für die Unix-Timestamps zustande, wenn man 32-Bit Integerzahlen für die Speicherung verwendet. Und ja, alle Programme, die auf Unix-Zeitstempel setzen und diese in 32-Bit Integervariablen abspeichern, werden im Jahr 2038 ein Problem bekommen ...

Index	Attributname	Bedeutung	Wertebereich
5	<i>tm_sec</i>	Sekunde der Uhrzeit des Zeitstempels ⁴	0–61
6	<i>tm_wday</i>	Nummer des Wochentages (0 entspricht Montag)	0–6
7	<i>tm_yday</i>	Nummer des Tages im Jahr	0–366
8	<i>tm_isdst</i>	Gibt an, ob der Zeitstempel durch die Sommerzeit angepasst wurde.	0 für »Nein«, 1 für »Ja«, -1 für »Unbekannt«

Tabelle 15.1 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.

15

Attribute

Das Modul `time` beinhaltet folgende Attribute:

Attribut	Beschreibung	Seite
<i>accept2dayear</i>	Dieses Attribut enthält einen Wahrheitswert, der angibt, ob Jahreszahlen mit nur zwei statt vier Ziffern angegeben werden dürfen.	454
<i>altzone</i>	Verschiebung der Lokalzeit inklusive Sommerzeit gegenüber der koordinierten Weltzeit in Sekunden	454
<i>daylight</i>	Gibt an, ob es in der lokalen Zeitzone eine Sommerzeit gibt.	454
<i>struct_time</i>	Referenz auf den Datentyp zur Speicherung von Zeiten	454
<i>timezone</i>	Verschiebung der Lokalzeit exklusive Sommerzeit gegenüber der koordinierten Weltzeit in Sekunden	454
<i>tzname</i>	Beschreibung der Zeitzone.	454

Tabelle 15.2 Attribute des Moduls `time`

⁴ 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.

accept2dayear

Dieses Attribut enthält einen Wahrheitswert, der angibt, ob Jahreszahlen mit nur zwei statt vier Ziffern angegeben werden dürfen, wie sie gegen Ende des 20. Jahrhunderts üblich waren.

**Hinweis**

Die Verwendung dieses Attributs wird seit Python 3.2 nicht mehr empfohlen. Anwendungen, die zweistellige Jahreszahlen benötigen, sollen sich selbst um die Umwandlung kümmern.

altzone

Dieses Attribut 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 `altzone` positiv; liegt die lokale Zeitzone westlich davon, ist er negativ.

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

daylight

Dieses Attribut 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 `daylight` den Wert 0. Die durch die Sommerzeit entstehende Verschiebung lässt sich mit `altzone` ermitteln.

struct_time

Dies ist eine Referenz auf den eingangs besprochenen Datentyp `struct_time`.

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

```
>>> t = time.struct_time((2011, 9, 18, 18, 24, 56, 0, 0, 0))
>>> t.tm_year
2011
```

timezone

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

tzname

Dieses Attribut 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

Einen Überblick über die Funktionen des Moduls `time` gibt die folgende Tabelle.

Funktion	Beschreibung	Seite
<i>asctime([t])</i>	Konvertiert die übergebene <code>struct_time</code> -Instanz in einen String.	456
<i>clock()</i>	Gibt die aktuelle Prozessorzeit zurück.	456
<i>ctime([secs])</i>	Konvertiert den übergebenen Unix-Zeitstempel in einen String. Falls kein Zeitstempel übergeben wurde, wird die aktuelle Systemzeit verwendet.	456
<i>gmtime([secs])</i>	Wandelt einen Unix-Timestamp in eine <code>struct_time</code> -Instanz um. Dabei wird die koordinierte Weltzeit zugrunde gelegt.	457
<i>localtime([secs])</i>	Wandelt einen Unix-Timestamp in eine <code>struct_time</code> -Instanz um. Dabei wird die Lokalzeit zugrunde gelegt.	457
<i>mktime(t)</i>	Wandelt eine <code>struct_time</code> -Instanz in einen Unix-Timestamp um. Dabei wird die Lokalzeit zugrunde gelegt.	457
<i>sleep(secs)</i>	Unterbricht den Programmablauf.	457
<i>strftime(format[, t])</i>	Wandelt eine <code>struct_time</code> -Instanz nach den übergebenen Regeln in einen String um.	457
<i>strptime(string[, format])</i>	Interpretiert einen String nach den übergebenen Regeln als Zeitangabe und liefert eine passende <code>struct_time</code> -Instanz.	459
<i>time()</i>	Gibt den aktuellen Unix-Zeitstempel zurück. Dabei wird die Koordinierte Weltzeit zugrunde gelegt.	459

Tabelle 15.3 Funktionen des Moduls `time`

asctime([t])

Diese Funktion wandelt eine `struct_time`-Instanz oder ein Tupel mit neun Elementen in einen 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 `asctime` einen String für die aktuelle Systemzeit zurück.

**Hinweis**

Die Funktion `asctime` liefert immer einen String zurück, der aus 24 Zeichen besteht und so formatiert ist, wie im Beispiel gezeigt.

Wenn Sie mehr Kontrolle über das Aussehen des resultierenden Strings wünschen, ist die Methode `strftime` auf Seite 457 besser geeignet.

clock()

Diese Funktion gibt die aktuelle Prozessorzeit zurück. Was dies konkret bedeutet, hängt von der verwendeten Plattform ab.

Unter Unix gibt `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 `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
```

ctime([secs])

Diese Funktion wandelt den als Parameter übergebenen Unix-Timestamp wie `asctime` in einen String um. Wird der optionale Parameter nicht übergeben oder hat er den Wert `None`, wird die aktuelle Systemzeit verwendet.

gmtime([secs])

Diese Funktion wandelt einen Unix-Timestamp in ein `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 benutzt, wie er von der Funktion `time` zurückgegeben wird.

```
>>> 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.

localtime([secs])

Genau wie `gmtime`, jedoch wandelt diese Funktion den übergebenen Timestamp in eine Angabe der lokalen Zeitzone um.

mktime(t)

Diese Funktion wandelt eine `struct_time`-Instanz in einen Unix-Timestamp der Lokalzeit um. Der Rückgabewert ist eine Gleitkommazahl.

Die Funktionen `localtime` und `mktime` sind jeweils Umkehrfunktionen voneinander:

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

sleep(secs)

Die Funktion `sleep` unterbricht die Programmausführung für die übergebene Zeitspanne. Der Parameter `secs` muss dabei eine Gleitkommazahl sein, die die Dauer der Unterbrechung in Sekunden angibt.

Wenn Sie ein Program mittels `sleep` unterbrechen, befindet es sich im Leerlauf und beansprucht den Prozessor nicht.

strftime(format[, t])

Diese Funktion wandelt die `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 wer-

den. 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 ⁵
%S	Sekunde als Zahl mit zwei Ziffern. Liegt immer im Bereich [00,61].
%U	Nummer der aktuellen Woche im Jahr, wobei der Sonntag als erster Tag der Woche betrachtet wird. Das 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].

Tabelle 15.4 Übersicht über alle Platzhalter der strftime-Funktion

5 von lat. »Ante Meridiem« (dt. »vor dem Mittag«) bzw. lat. »Post Meridiem« (»nach dem Mittag«)

Platzhalter	Bedeutung
%W	wie %U, nur dass statt des Sonntags der Montag als 0. 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 15.4 Übersicht über alle Platzhalter der strftime-Funktion (Forts.)

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

```
>>> time.strftime("%d.%m.%Y um %H:%M:%S Uhr")
'20.11.2011 um 12:50:41 Uhr'
```

strptime(string[, format])

Mit `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. Der Aufbau solcher Format-Strings ist der gleiche wie bei `strftime`.

```
>>> zeit_string = '19.09.2011 um 00:21:17 Uhr'
>>> time.strptime(zeit_string, "%d.%m.%Y um %H:%M:%S Uhr")
time.struct_time(tm_year=2011, tm_mon=9, tm_mday=19, tm_hour=0, tm_min=21,
                  tm_sec=17, tm_wday=0, tm_yday=262, tm_isdst=-1)
```

Wenn Sie den optionalen Parameter *format* nicht angeben, wird der Standardwert `"%a %b %d %H:%M:%S %Y"` verwendet. Dies entspricht dem Ausgabeformat von `ctime`.

time()

Dies 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.

15.2 Komfortable Datumsfunktionen – datetime

Das Modul `datetime` ist im Vergleich zum `time`-Modul abstrakter und durch seine eigenen Zeit- und Datumstypen auch angenehmer zu benutzen.

[+]

Hinweis

Aus Platzgründen gehen wir hier nicht auf die Möglichkeiten ein, die das Modul `datetime` für den Umgang mit verschiedenen Zeitzonen bietet. Näheres zu dieser Funktionalität können Sie bei Bedarf der Python-Dokumentation entnehmen.

An dieser Stelle sei nur so viel gesagt:

Es wird zwischen *naiven* und *bewussten* Zeitobjekten unterschieden. Ein bewusstes Zeitobjekt ist sich der Zeitzone, auf die es sich bezieht, bewusst, während ein naives Zeitobjekt keine Informationen zu seiner Zeitzone kennt.

15.2.1 Attribute und Datentypen (Übersicht)

Es gibt zwei Attribute, die das `datetime`-Modul definiert, um den Wertebereich für die Jahreszahlen zu definieren. Außerdem stellt das Modul eine Reihe von Datentypen zur Verfügung.

Attribut	Beschreibung	Seite
<i>MINYEAR</i>	Der minimal mögliche Wert für eine Jahreszahl. Der Wert ist in der Regel 1.	–
<i>MAXYEAR</i>	Der maximal mögliche Wert für eine Jahreszahl. Der Wert ist in der Regel 9999.	–

Tabelle 15.5 Die Attribute des Moduls `datetime`

<i>date</i>	ein Datentyp zum Speichern von Datumsangaben	461
<i>time</i>	Mit <code>datetime.time</code> werden Zeitpunkte an einem Tag gespeichert. Dabei wird idealisiert angenommen, dass jeder Tag $24 * 60 * 60$ Sekunden umfasst und dass es keine Schaltsekunden gibt.	466

Tabelle 15.6 Die Datentypen von `datetime`

<i>datetime</i>	Die Kombination aus <code>datetime.date</code> und <code>datetime.time</code> zum Speichern von ganzen Zeitpunkten, die sowohl ein Datum als auch eine Uhrzeit umfassen. Der Datentyp <code>datetime.datetime</code> ist der wichtigste des Moduls.	468
<i>timedelta</i>	Es ist möglich, Differenzen zwischen <code>datetime.date</code> - und auch <code>datetime.datetime</code> -Instanzen zu bilden. Die Ergebnisse solcher Subtraktionen sind <code>datetime.timedelta</code> -Objekte.	477

Tabelle 15.6 Die Datentypen von datetime (Forts.)

15.2.2 date

Hier werden wir die Attribute und Methoden des Datentyps `date` behandeln, der zum Speichern von Datumsangaben dient.

Die folgende Tabelle listet zunächst alle Attribute und Methoden für den Umgang mit `date`-Instanzen auf.

Name	Beschreibung	Seite
Konstrukteure		
<i>date(year, month, day)</i>	Erzeugt eine <code>date</code> -Instanz, die den übergebenen Tag beschreibt.	463
<i>date.today()</i>	Erzeugt eine <code>date</code> -Instanz, die den nach Systemzeit aktuellen Tag beschreibt.	463
<i>date.fromtimestamp(timestamp)</i>	Erzeugt eine <code>date</code> -Instanz, die den Tag des übergebenen Unix-Timestamps beschreibt.	463
Klassenattribute		
<i>min</i>	der früheste Tag, der durch eine <code>date</code> -Instanz beschrieben werden kann, <code>date(MINYEAR, 1, 1)</code>	464
<i>max</i>	der späteste Tag, der durch eine <code>date</code> -Instanz beschrieben werden kann, <code>date(MAXYEAR, 12, 31)</code>	464
<i>resolution</i>	die kleinstmögliche Differenz zwischen zwei nicht gleichen <code>date</code> -Instanzen, <code>timedelta(days=1)</code>	–

Tabelle 15.7 Member des Datentyps date

Name	Beschreibung	Seite
Operatoren		
–	Bildet die Differenz zwischen zwei <code>date</code> -Instanzen oder zwischen einer Instanz des Datentyps <code>date</code> und einer Instanz des Typs <code>timedelta</code> .	474
+	Bildet die Summe einer <code>date</code> -Instanz und einer <code>timedelta</code> -Instanz.	474
<, <=, >, >=	Vergleicht zwei <code>date</code> -Instanzen in Bezug auf ihre Position auf der Zeitachse.	474
Attribute		
<i>year</i>	Jahr der Datumsangabe. Dieses Attribut kann nur gelesen werden.	–
<i>month</i>	Monat der Datumsangabe. Dieses Attribut kann nur gelesen werden.	–
<i>day</i>	Tag der Datumsangabe. Dieses Attribut kann nur gelesen werden.	–
Methoden		
<i>ctime()</i>	Erzeugt einen String, der den von der <code>date</code> -Instanz beschriebenen Tag angibt.	466
<i>isocalendar()</i>	Gibt ein Tupel zurück, das das ISO-Jahr, den ISO-Monat und den ISO-Tag der <code>date</code> -Instanz enthält.	465
<i>isoformat()</i>	Wandelt die <code>date</code> -Instanz in einen String der Form "YYYY-MM-DD" um.	465
<i>isoweekday()</i>	Gibt den Tag innerhalb der Woche zurück. Dabei entspricht 1 dem Montag und 7 dem Sonntag.	465
<i>replace(year, month, day)</i>	Erzeugt eine neue <code>date</code> -Instanz aus einer bestehenden, indem einzelne Element ersetzt werden	464
<i>strftime(format)</i>	Wandelt die <code>date</code> -Instanz gemäß einer Formatbeschreibung in einen String um.	466

Tabelle 15.7 Member des Datentyps `date` (Forts.)

Name	Beschreibung	Seite
<i>timetuple()</i>	Erzeugt eine <code>time.struct_time</code> -Instanz, die dem Datum der <code>date</code> -Instanz entspricht (siehe S. 452).	464
<i>weekday()</i>	Gibt den Tag innerhalb der Woche zurück. Dabei entspricht 0 dem Montag und 6 dem Sonntag.	464

Tabelle 15.7 Member des Datentyps `date` (Forts.)**Konstruktoren der Klasse `datetime.date`**

Es gibt drei Konstruktoren für `date`-Instanzen:

`date(year, month, day)`

Dieser Konstruktor erzeugt eine neue Instanz des Datentyps `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)
```

`date.today()`

Dieser Konstruktor erzeugt eine neue `date`-Instanz, die den aktuellen Tag repräsentiert:

```
>>> datetime.date.today()
datetime.date(2011, 9, 19)
```

`date.fromtimestamp(timestamp)`

Hiermit wird ein neues `date`-Objekt erzeugt, das das Datum des übergebenen Unix-Timestamps speichert.

```
>>> datetime.date.fromtimestamp(0)
datetime.date(1970, 1, 1)
```

Im Beispiel wird der erste Tag der Unix-Epoche aus dem Unix-Timestamp 0 erzeugt.

Klassenattribute von `datetime.date`

min

Dies ist ein Klassenattribut, das den frühesten Tag enthält, der durch den `date`-Typ abgebildet werden kann. Wie das folgende Listing zeigt, ist dies der 1. Januar im Jahr 1:

```
>>> datetime.date.min
datetime.date(1, 1, 1)
```

max

Das Klassenattribut `max` speichert eine `date`-Instanz, die den spätesten Tag repräsentiert, der von `date` verwaltet werden kann: den 31.12. im Jahr 9999.

```
>>> datetime.date.max
datetime.date(9999, 12, 31)
```

Die Attribute und Methoden von `date`-Instanzen

Im Folgenden sei *d* eine `date`-Instanz.

d.replace(year, month, day)

Dies 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()

Diese Methode gibt eine `time.struct_time`-Instanz⁶ 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(2011, 7, 6)
>>> d.timetuple()
time.struct_time(tm_year=2011, tm_mon=7, tm_mday=6, tm_hour=0, tm_min=0,
                  tm_sec=0, tm_wday=2, tm_yday=187, tm_isdst=-1)
```

d.weekday()

Dies gibt den Wochentag als Zahl zurück, wobei Montag als 0 und Sonntag als 6 angegeben werden.

⁶ Siehe dazu Abschnitt 15.1, »Elementare Zeitfunktionen – time«, auf Seite 452.

d.isoweekday()

Diese Methode gibt den Wochentag als Zahl zurück, wobei Montag den Wert 1 und Sonntag den Wert 7 ergibt.

Siehe dazu auch `d.isocalendar()`.

d.isocalendar()

Hiermit wird ein Tupel zurückgegeben, 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 01.01.2011 ein Samstag, weshalb der 03.01.2011 der erste Tag im Jahr 2011 des ISO-Kalenders war:

```
>>> d = datetime.date(2011, 1, 3)
>>> d.isocalendar()
(2011, 1, 1)
```

d.isoformat()

Diese Methode 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'
```

Hinweis

Die Methode `isoformat` hat nichts mit dem ISO-Kalender zu tun, den die Methoden `isoweekday` und `isocalendar` verwenden.



d.ctime()

Diese Methode 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(2011, 10, 23)
>>> d.ctime()
'Sun Oct 23 00:00:00 2011'
```

d.strftime(format)

Diese Methode gibt den von *d* repräsentierten Tag formatiert aus, wobei der Parameter *format* die Beschreibung des gewünschten Ausgabeformats enthält.

Nähere Informationen können Sie in Abschnitt 15.1, »Elementare Zeitfunktionen – time«, unter `time.strftime` auf Seite 457 nachschlagen.

15.2.3 datetime.time

In diesem Abschnitt werden wir uns mit den Methoden und Attributen des Datentyps `datetime.time`, im Folgenden als *time* bezeichnet, beschäftigen.

Objekte des Typs *time* dienen dazu, Tageszeiten anhand von Stunde, Minute, Sekunde und auch Mikrosekunde zu verwalten.

In dem Attribut `tzinfo` können *time*-Instanzen Informationen zur lokalen Zeitzone speichern und ihre Werte damit an die Lokalzeit anpassen.

Zunächst gibt folgende Tabelle einen Überblick über die Attribute und Methoden des Datentyps *time*.

Name	Beschreibung	Seite
Konstruktoren		
<code>time([hour[, minute[, second[, microsecond[, tzinfo]]]])</code>	Erzeugt eine neue <i>time</i> -Instanz gemäß der übergebenen Daten.	468
Klassenattribute		
<code>min</code>	der früheste Zeitpunkt, der gespeichert werden kann, in der Regel <code>datetime.time(0, 0)</code>	–
<code>max</code>	der späteste darstellbare Zeitpunkt, in der Regel <code>datetime.time(23, 59, 59, 999999)</code>	–

Tabelle 15.8 Attribute und Methoden des Datentyps *time*

Name	Beschreibung	Seite
<i>resolution</i>	der minimale Unterschied zwischen zwei unterschiedlichen <code>datetime.time</code> -Objekten	–
Attribute		
<i>hour</i>	Stundenanteil des Zeitpunktes	–
<i>minute</i>	Minutenanteil des Zeitpunktes	–
<i>second</i>	Sekundenanteil des Zeitpunktes	–
<i>microsecond</i>	Mikrosekundenanteil (Mikrosekunde = Millionstelsekunde) des Zeitpunktes	–
<i>tzinfo</i>	Information zur lokalen Zeitzone. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.	–
Methoden		
<i>dst()</i>	Gibt Informationen über eventuell geltende Sommerzeit zurück. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.	–
<i>isoformat()</i>	Erzeugt einen String im Format "HH:MM:SS.mmmmm", der den Zeitpunkt beschreibt. Ist das Attribut <code>microseconds</code> 0, hat der String das Format "HH:MM:SS".	468
<i>replace([hour[, minute[, second[, microsecond[, tzinfo]]]])</i>	Erzeugt eine neue <code>time</code> -Instanz aus einer vorhandenen, indem die übergebenen Angaben ersetzt werden.	468
<i>strftime(format)</i>	Wandelt eine <code>time</code> -Instanz gemäß der übergebenen Formatbeschreibung in einen String um.	468
<i>tzname()</i>	Gibt den Namen der Zeitzone zurück. Näheres entnehmen Sie bitte der Python-Dokumentation.	–
<i>utcoffset()</i>	Gibt die Verschiebung der Lokalzeit relativ zur UTC in Sekunden an. Näheres entnehmen Sie bitte der Python-Dokumentation.	–

Tabelle 15.8 Attribute und Methoden des Datentyps `time` (Forts.)**Konstruktor von `datetime.time`**

Ein neues `time`-Objekt erzeugen Sie mit dem folgenden Konstruktor:

time([hour[, minute[, second[, microsecond[, tzinfo]]]])

Die vier ersten Parameter legen den Zeitpunkt fest und müssen folgende Bedingungen erfüllen, wobei nur Ganzzahlen zugelassen sind:

- ▶ $0 \leq \text{hour} < 24$
- ▶ $0 \leq \text{minute} < 60$
- ▶ $0 \leq \text{second} < 60$
- ▶ $0 \leq \text{microsecond} < 1000000$

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 funktioniert, können Sie der Python-Dokumentation entnehmen.

Methoden von time-Instanzen

Nachfolgend wird angenommen, dass *t* eine Instanz des Datentyps `time` ist.

t.replace([hour[, minute[, second[, microsecond[, tzinfo]]]])

Diese Methode funktioniert analog zur `date.replace`-Methode auf Seite 464.

t.isoformat()

Diese Methode 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)

Diese Methode erzeugt einen String, der den Zeitpunkt *t* nach der Formatbeschreibung in *format* enthält. Näheres dazu können Sie unter `time.strftime` auf Seite 457 nachlesen.

15.2.4 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.



Hinweis

Wir werden im Folgenden nur noch `datetime` anstelle von `datetime.datetime` schreiben. Lassen Sie sich nicht davon verwirren, dass hier der Datentyp genauso heißt wie das Modul.

Es folgt die Übersicht über alle Attribute und Methoden des Datentyps `datetime`.

Name	Beschreibung	Seite
Konstrukteure		
<code>datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]])</code>	Erzeugt eine <code>datetime</code> -Instanz mit den angegebenen Daten.	472
<code>combine(date, time)</code>	Baut eine <code>date</code> -Instanz und eine <code>time</code> -Instanz zu einer <code>datetime</code> -Instanz zusammen.	474
<code>fromtimestamp(timestamp[, tz])</code>	Erzeugt eine <code>datetime</code> -Instanz aus einem Unix-Timestamp.	473
<code>now([tz])</code>	Erzeugt eine <code>datetime</code> -Instanz, die den laut Systemzeit aktuellen Zeitpunkt speichert.	473
<code>strptime(date_string, format)</code>	Interpretiert einen String gemäß einer Formatbeschreibung als Zeitangabe und erzeugt eine entsprechende <code>datetime</code> -Instanz.	474
<code>today()</code>	Erzeugt eine <code>datetime</code> -Instanz, die den laut Systemzeit aktuellen Zeitpunkt speichert.	472
<code>utcfromtimestamp(timestamp)</code>	Erzeugt aus einem Unix-Timestamp eine <code>datetime</code> -Instanz, die die entsprechende koordinierte Weltzeit speichert. Eine Zeitverschiebung durch die lokale Systemzeit wird dabei berücksichtigt.	473
<code>utcnow()</code>	Erzeugt eine <code>datetime</code> -Instanz, die die aktuelle koordinierte Weltzeit speichert.	473

Tabelle 15.9 Attribute und Methoden des Datentyps `datetime`

Name	Beschreibung	Seite
Operationen		
+	Addiert eine <code>datetime</code> -Instanz und eine <code>datetime.timedelta</code> -Instanz.	474
–	Bildet die Differenz zwischen zwei <code>datetime</code> -Instanzen oder zwischen einer Instanz des Datentyps <code>datetime</code> und einer Instanz des Typs <code>timedelta</code> .	474
<, <=, >, >=	Vergleicht <code>datetime</code> -Instanzen bezüglich ihrer Lage auf der Zeitachse.	474
Klassenattribute		
<i>min</i>	die früheste darstellbare <code>datetime</code> -Instanz, <code>datetime(MINYEAR, 1, 1)</code>	–
<i>max</i>	die späteste darstellbare <code>datetime</code> -Instanz, <code>datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999)</code>	–
<i>resolution</i>	kleinstmögliche Differenz zweier verschiedener <code>datetime</code> -Instanzen, <code>datetime.timedelta(microseconds=1)</code>	–
Attribute		
<i>year</i>	Jahresanteil des Zeitpunktes	–
<i>month</i>	Monatsanteil des Zeitpunktes	–
<i>day</i>	Tagesanteil des Zeitpunktes	–
<i>hour</i>	Stundenanteil des Zeitpunktes	–
<i>minute</i>	Minutenanteil des Zeitpunktes	–
<i>second</i>	Sekundenanteil des Zeitpunktes	–
<i>microsecond</i>	Mikrosekundenanteil des Zeitpunktes	–
<i>tzinfo</i>	Information zur lokalen Zeitzone. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.	–

Tabelle 15.9 Attribute und Methoden des Datentyps `datetime` (Forts.)

Name	Beschreibung	Seite
Methoden		
<i>astimezone(tz)</i>	Erzeugt eine neue <code>datetime</code> -Instanz, die denselben Zeitpunkt wie die vorhandene Instanz in der Zeitzone <code>tz</code> beschreibt. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.	–
<i>ctime()</i>	Wandelt eine <code>datetime</code> -Instanz in einen String um.	476
<i>date()</i>	Gibt ein <code>datetime.date</code> -Objekt zurück, das dem Tag der <code>datetime</code> -Instanz entspricht.	475
<i>dst()</i>	Gibt Informationen über eventuell geltende Sommerzeit zurück. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.	–
<i>isocalendar()</i>	Gibt ein Tupel zurück, das den Tag im ISO-Kalender beschreibt (siehe S. 465).	–
<i>isoformat([sep])</i>	Wandelt die <code>datetime</code> -Instanz in einen String um gemäß dem Format <code>"YYYY-MM-DDTHH:MM:SS.mmmmm"</code>	476
<i>isoweekday()</i>	Gibt den Wochentag als Zahl zurück, wobei Montag den Wert 1 und Sonntag den Wert 7 ergibt.	–
<i>replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]])</i>	Erzeugt eine neue <code>datetime</code> -Instanz, die aus der vorhandenen dadurch hervorgeht, dass die übergebenen Werte ersetzt werden.	475
<i>strftime(format)</i>	Wandelt ein <code>datetime</code> -Objekt gemäß einer Formatbeschreibung in einen String um.	477
<i>time()</i>	Gibt ein <code>datetime.time</code> -Objekt zurück, das der Tageszeit der <code>datetime</code> -Instanz entspricht.	475
<i>timetuple()</i>	Erzeugt eine <code>time.struct_time</code> -Instanz, die denselben Zeitpunkt beschreibt wie die <code>datetime</code> -Instanz (siehe S. 452).	476

Tabelle 15.9 Attribute und Methoden des Datentyps `datetime` (Forts.)

Name	Beschreibung	Seite
<i>timetz()</i>	Wie <i>time</i> , aber es wird zusätzlich das <i>tzinfo</i> -Attribut kopiert.	–
<i>tzname()</i>	Gibt den Namen der Zeitzone zurück.	476
<i>utcoffset()</i>	Gibt die Verschiebung der Lokalzeit relativ zur koordinierten Weltzeit an. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.	–
<i>utctimetuple()</i>	Wie <i>timetuple</i> , wobei versucht wird, anhand von Informationen über die verwendete Zeitzone einen Zeitstempel in der koordinierten Weltzeit zu erzeugen. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.	–
<i>weekday()</i>	Gibt den Wochentag als Zahl zurück, wobei Montag den Wert 0 und Sonntag den Wert 6 ergibt.	–

Tabelle 15.9 Attribute und Methoden des Datentyps `datetime` (Forts.)

Konstrukturen von `datetime`

Es gibt acht Konstrukturen, um neue `datetime`-Instanzen zu erzeugen:

`datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]])`

Die Parameter haben die gleiche Bedeutung wie die gleichnamigen Elemente der Konstrukturen von `date` und `time` auf den Seiten 463 und 468, weshalb wir hier auf eine Wiederholung verzichten.

```
>>> bescherung = datetime.datetime(2011, 12, 24, 18, 30)
>>> bescherung
datetime.datetime(2011, 12, 24, 18, 30)
```

`today()`

Dieser Konstruktor erzeugt eine `datetime`-Instanz, die die aktuelle Lokalzeit speichert. Das `tzinfo`-Attribut wird dabei immer auf `None` gesetzt.

```
>>> datetime.datetime.today()
datetime.datetime(2011, 10, 20, 13, 10, 27, 21335)
```

**Hinweis**

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 `today` ein `datetime`-Objekt, das auch die Uhrzeit enthält.

`now([tz])`

Dies erzeugt eine `datetime`-Instanz mit dem aktuellen Datum und der aktuellen Zeit. Wird die Methode ohne Parameter aufgerufen, erzeugt sie das gleiche Ergebnis wie `today`.

Mit dem optionalen Parameter `tz` können Informationen zur Lokalzeit übergeben werden. Näheres dazu entnehmen Sie bitte der Python-Dokumentation.

`utcnow()`

Dieser Konstruktor gibt die aktuelle koordinierte Weltzeit (UTC) zurück, wobei das `tzinfo`-Attribut der `datetime`-Instanz den Wert `None` hat.

`fromtimestamp(timestamp[, tz])`

Dieser Konstruktor erzeugt eine `datetime`-Instanz, die den gleichen Zeitpunkt wie der für `timestamp` übergebene Unix-Zeitstempel repräsentiert.

Übergeben Sie für `tz` keinen Wert oder `None`, ist der Rückgabewert ein naives Zeitobjekt.

`utcfromtimestamp(timestamp)`

Dieser Konstruktor wandelt den übergebenen Unix-Timestamp in ein `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`-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.

combine(date, time)

Hiermit wird ein `datetime`-Objekt erzeugt, 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`-Objekt übergeben. In diesem Fall wird die in *date* enthaltene Uhrzeit ignoriert und nur das Datum betrachtet.

strptime(date_string, format)

Dieser Konstruktor interpretiert den String, der als Parameter *date_string* übergeben wurde, gemäß der Formatbeschreibung aus *format* als Zeitinformation und gibt ein entsprechendes `datetime`-Objekt zurück.

Für die Formatbeschreibung gelten die gleichen Regeln wie bei `time.strptime`.

Operatoren für `datetime.datetime` und `datetime.date`

Die Datentypen `datetime.datetime` und `datetime.date` überladen die Operatoren für die Subtraktion und Addition, sodass mit Zeitangaben gerechnet werden kann.

Dabei sind folgende Summen und Differenzen möglich, wobei *d1* und *d2* jeweils beide `datetime.datetime`-Instanzen bzw. `datetime.date`-Instanzen sind und *t* ein `datetime.timedelta`-Objekt referenziert:

Ausdruck	Hinweise
$d2 = d1 + t$	Der von <i>d2</i> beschriebene Zeitpunkt ergibt sich, indem in der Zeit von <i>d1</i> aus um die von <i>t</i> beschriebene Zeitspanne in die Zukunft oder die Vergangenheit gegangen wird, je nachdem, ob der Wert von <i>t</i> positiv oder negativ ist. Das entstehende Objekt <i>d2</i> übernimmt außerdem das <code>tzinfo</code> -Attribut von <i>d1</i> .
$d2 = d1 - t$	wie bei der Addition, außer dass nun bei positivem <i>t</i> in Richtung Vergangenheit und bei negativem <i>t</i> in Richtung Zukunft gegangen wird
$t = d1 - d2$	Das <code>datetime.timedelta</code> -Objekt <i>t</i> beschreibt den zeitlichen Abstand zwischen den Zeitpunkten <i>d1</i> und <i>d2</i> . Dabei wird <i>t</i> so gewählt, dass $d1 = d2 + t$ gilt. Diese Operation kann nur durchgeführt werden, wenn <i>d1</i> und <i>d2</i> 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 15.10 Rechnen mit `datetime.datetime` und `datetime.date`

Das folgende Beispiel zeigt die Differenzbildung von Zeitobjekten.

```
>>> datetime.date(1987, 11, 3) - datetime.date(1987, 7, 26)
datetime.timedelta(100)
>>> d1 = datetime.datetime(2012, 2, 15, 17, 0, 0)
>>> d2 = datetime.datetime(2012, 2, 15, 7, 0, 0)
>>> d1 - d2
datetime.timedelta(0, 36000)
```

In dem Beispiel liegen die beiden `datetime.date`-Instanzen 100 Tage und die beiden `datetime.datetime`-Instanzen 36.000 Sekunden, also 10 Stunden, auseinander.

Als nächstes wird zu einem Datum in Form einer `datetime.date`-Instanz ein `datetime.timedelta`-Objekt von 100 Tagen addiert:

```
>>> datetime.date(1987, 7, 26) + datetime.timedelta(100)
datetime.date(1987, 11, 3)
```

Außerdem können `datetime.date`-Instanzen bzw. `datetime.datetime`-Instanzen mit den Vergleichsoperatoren `<` und `>` sowie `≤` und `≥` verglichen werden. Dabei wird dasjenige Datum als »kleiner« betrachtet, das in der Zeit weiter in Richtung Vergangenheit liegt:

```
>>> datetime.date(1987, 7, 26) < datetime.date(1987, 11, 3)
True
```

Methoden von `datetime.datetime`-Instanzen

Im Folgenden wird angenommen, dass *d* eine Instanz des Datentyps `datetime` ist.

d.date()

Diese Methode gibt ein `datetime.date`-Objekt zurück, das die gleichen `year`-, `month`- und `day`-Attribute wie *d* hat.

d.time()

Diese Methode gibt ein `datetime.time`-Objekt zurück, das die gleichen `hour`-, `minute`-, `second`- und `microsecond`-Attribute wie *d* hat.

d.replace([year[, month[, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]]]]))

Diese Methode 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()

Diese Methode 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 ermittelt, indem intern *d.tzinfo.tzname(None)* aufgerufen wird.)

d.timetuple()

Diese Methode gibt ein *time.struct_time*-Objekt zurück, das den von *d* beschriebenen Zeitpunkt enthält.

d.isocalendar()

Hiermit wird ein Tupel mit drei Elementen zurückgegeben, das den von *d* beschriebenen Tag als Datum im ISO-Kalender ausdrückt.

Nähere Informationen dazu finden Sie unter der Methode *isocalendar* des Datentyps *datetime.date*.

d.isoformat()

Diese Methode 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« steht als Trennzeichen zwischen Datums- und Zeitangabe. In der Zeitangabe stehen die »H« für die Ziffern der Stunde, die »M« für die Ziffern der Minute und die »S« für die Ziffern der Sekunden.

Ist das *microseconds*-Attribut von *d* von 0 verschieden, werden die Mikrosekunden, durch einen Doppelpunkt abgetrennt, an das Ende des Strings geschrieben (in der Formatbeschreibung durch die »m« angedeutet). Ansonsten entfällt der Mikrosekundenteil inklusive Doppelpunkt.

```
>>> datetime.datetime(2001, 5, 1, 15, 14, 13, 100012).isoformat()
'2001-05-01T15:14:13.100012'
```

d.ctime()

Dies 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()

Hiermit wird einen String erzeugt, der den von *d* beschriebenen Zeitpunkt formatiert enthält.

Genauer es können Sie unter `time.strftime` auf Seite 457 nachlesen.

15.2.5 datetime.timedelta

Mit Instanzen des Datentyps `datetime.timedelta` können Zeitspannen beschrieben werden, wie sie beispielsweise entstehen, wenn Sie die Differenz zweier `datetime.datetime`-Instanzen bilden:

```
>>> d1 = datetime.datetime(2012, 1, 9, 12, 0, 0)
>>> d2 = datetime.datetime(2012, 2, 10, 20, 15, 0)
>>> delta1 = d2 - d1
>>> delta1
datetime.timedelta(32, 29700)
```

15

Im Beispiel liegen also 32 Tage und 29700 Sekunden zwischen den beiden Zeitpunkten.

Eine Zeitspanne hat ein Vorzeichen, das angibt, ob die Zeitspanne in Richtung Zukunft oder Vergangenheit zeigt. Im obigen Beispiel muss man 32 Tage und 29700 Sekunden vom Zeitpunkt *d1* aus in die Zukunft gehen, um bei *d2* zu landen. Dreht man die Differenz um, ändert sich das Vorzeichen:

```
>>> delta2 = d1 - d2
>>> delta2
datetime.timedelta(-33, 56700)
```

Diese Angabe ist so zu verstehen, dass man, ausgehend von *d2*, erst 33 Tage in Richtung Vergangenheit und dann wieder 56700 Sekunden in Richtung Zukunft gehen muss, um bei *d1* auszukommen.

Auch wenn es auf den ersten Blick nicht offensichtlich ist, beschreiben *delta1* und *delta2* jeweils die gleiche Dauer, einmal mit negativem und einmal mit positivem Vorzeichen. Dies wird klar, wenn man die Summe der beiden bildet:

```
>>> delta1 + delta2
datetime.timedelta(0)
```

Dieses Verhalten ist dadurch begründet, dass die Sekunden und Mikrosekundenangaben bei `datetime.timedelta`-Instanzen immer positiv sind. Einzig die Angabe des Tages kann ein negatives Vorzeichen haben.

Ein Tag hat $24 \cdot 60 \cdot 60 = 86400$ Sekunden. Möchte man also 32 Tage und 29700 Sekunden in die Vergangenheit gehen, kann man genauso gut erst 33 Tage Richtung Vergangenheit gehen und dann das, was man dadurch zu viel gegangen ist, wieder ausgleichen. Dieser Überschuss umfasst gerade $86400 - 29700 = 56700$ Sekunden.

Die Tabelle gibt eine Übersicht über die Fähigkeiten von `datetime.timedelta`, im Folgenden nur noch `timedelta`.

Name	Beschreibung	Seite
Konstruktor		
<code>timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]]])</code>	Erzeugt eine neue <code>timedelta</code> -Instanz gemäß den übergebenen Daten.	480
Klassenattribute		
<code>min</code>	die betragsmäßig größte negative <code>timedelta</code> -Instanz, <code>timedelta(-999999999)</code>	–
<code>max</code>	die betragsmäßig größte positive <code>timedelta</code> -Instanz, <code>timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)</code>	–
<code>resolution</code>	kleinstmöglicher zeitlicher Abstand zwischen zwei <code>timedelta</code> -Instanzen, <code>timedelta(microseconds=1)</code>	–
Attribute		
<code>days</code>	Anzahl der Tage, die die Zeitspanne umfasst	480
<code>seconds</code>	Anzahl der Sekunden, die die Zeitspanne abzüglich der Tage umfasst. Ist immer positiv.	480
<code>microseconds</code>	Anzahl der Mikrosekunden, die die Zeitspanne abzüglich der Tage und Sekunden umfasst. Ist immer positiv.	480

Tabelle 15.11 Attribute und Methoden von `datetime.timedelta`

Name	Beschreibung	Seite
Operationen (<i>t1,t2</i> timedelta-Instanzen, <i>i</i> int-Instanz, <i>f</i> float-Instanz)		
$t1 + t2$	Bildet die Summe zweier timedelta-Instanzen.	–
$t1 - t2$	Bildet die Differenz zweier timedelta-Instanzen.	–
$t1 * i$ $t1 * f$	Erzeugt eine timedelta-Instanz, die <i>i</i> bzw. <i>f</i> mal so lang ist wie <i>t1</i> . Ist das Vorzeichen von <i>i</i> bzw. <i>f</i> negativ, dreht sich außerdem die Richtung auf der Zeitachse um.	–
$t1 / t2$ $t1 / i$ $t1 / f$	Bildet den Quotienten aus zwei timedelta-Instanzen oder einer timedelta-Instanz und einer Ganz- oder Gleitkommazahl.	–
$t1 // t2$ $t1 // i$	wie der Operator /, außer dass zusätzlich abgerundet wird	–
$t1 \% t2$	Liefert den Divisionsrest bei der Division zweier timedelta-Instanzen.	–
$<, <=, >, >=$	Vergleicht zwei timedelta-Instanzen.	–
$q, r = \text{divmod}(t1, t2)$	Erzeugt ein Tupel aus $q = t1 // t2$ und $r = t1 \% t2$, also dem abgerundeten Quotienten und dem Divisionsrest.	–
$\text{abs}(t1)$	Erzeugt eine timedelta-Instanz, die die gleiche Dauer wie <i>t1</i> beschreibt, aber in Richtung der positiven Zeitachse zeigt.	–
Methoden		
$\text{total\_seconds}()$	Gibt die Gesamtzahl der Sekunden zurück, die die Zeitspanne umfasst.	–

Tabelle 15.11 Attribute und Methoden von datetime.timedelta (Forts.)

Nun werden einige Member von timedelta im Detail besprochen.

Konstruktor

`timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]])`

Der Konstruktor erzeugt eine neue `timedelta`-Instanz aus den übergebenen Daten. Dabei werden nicht angegebene Werte mit dem Standardwert 0 versehen.

```
>>> datetime.timedelta(10, 20, 3000)
datetime.timedelta(10, 20, 3000)
>>> datetime.timedelta(weeks=4, minutes=200)
datetime.timedelta(28, 12000)
```

Wichtig ist dabei, dass eine `timedelta`-Instanz intern nur die Anzahl der Tage, Sekunden und Mikrosekunden speichert. Alle anderen Angaben werden durch diese drei Angaben abgebildet, wobei folgendermaßen umgerechnet wird:

- ▶ Eine Millisekunde wird zu 1000 Mikrosekunden.
- ▶ Eine Minute wird zu 60 Sekunden.
- ▶ Eine Stunde wird zu 3600 Sekunden.
- ▶ Eine Woche wird zu 7 Tagen.

Es können neben ganzen Zahlen auch Gleitkommazahlen für alle Parameter übergeben werden. In diesem Fall werden alle Nachkommateile zusammengefasst und zur nächsten `timedelta`-Instanz gerundet.

```
>>> datetime.timedelta(days=0.5)
datetime.timedelta(0, 43200)
```

Attribute

Die Attribute `days`, `seconds` und `microseconds` werden als ganze Zahlen gespeichert. Dabei gelten folgende Einschränkungen:

- ▶ $-999999999 \leq \text{days} < 999999999$
- ▶ $0 \leq \text{seconds} < 3600 \cdot 24$
- ▶ $0 \leq \text{microseconds} < 1000000$

Es ist zu beachten, dass nur das `days`-Attribut ein negatives Vorzeichen haben kann, das angibt, ob die `timedelta`-Instanz in Richtung Zukunft oder Vergangenheit zeigt.

Kapitel 16

Schnittstelle zum Betriebssystem

»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«

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.

16.1 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.



Hinweis

Seit Python 3.0 wird streng zwischen Text und Daten durch die Datentypen `str` und `bytes` unterschieden, wie Sie in Abschnitt 7.5, »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.

16.1.1 Zugriff auf den eigenen Prozess und andere Prozesse

`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\username
```

```
>>> print(os.environ['HOME'])
/home/username
```

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.

`getpid()`

Der Python-Prozess, der das aktuell laufende Programm ausführt, hat eine eindeutige Identifikationsnummer. Sie lässt sich mit `os.getpid()` ermitteln:

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

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

system(cmd)

Mit `os.system` können Sie beliebige Kommandos des Betriebssystems ausführen, so als ob Sie es in einer echten Konsole tun würden. Beispielsweise lassen wir uns mit folgendem Beispiel einen neuen Ordner mit dem Namen `test_ordner` über das `mkdir`-Kommando anlegen:

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

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

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.

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

Mit der Funktion `os.popen` werden beliebige Befehle wie auf einer Kommandozeile des Betriebssystems ausgeführt. Die Funktion gibt ein *dateiähnliches Objekt*¹ zurück, mit dem Sie auf die Ausgabe des ausgeführten Programms zurückgreifen können. Der Parameter `mode` gibt wie bei der Built-in Function `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:²

```
>>> 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 8, »Dateien«, ab Seite 197 nachlesen.

1 Ein dateiähnliches Objekt bietet die gleiche Schnittstelle wie geöffnete Dateien an. Die Daten müssen jedoch nicht in einer Datei gelesen werden, sondern können beispielsweise auch im Arbeitsspeicher liegen oder über eine Netzwerkverbindung gelesen werden.

2 Der Parameter `/B` des `dir`-Kommandos sorgt dafür, dass nur eine Liste der Dateien und Verzeichnisse ohne Zusatzinformationen erzeugt wird.

16.1.2 Zugriff auf das Dateisystem

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

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

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

Name	Beschreibung	Seite
<code>access(path, mode)</code>	Prüft die Rechte, die das Programm auf den übergebenen Pfad <i>path</i> hat.	485
<code>chdir(path)</code>	Setzt das aktuelle Arbeitsverzeichnis auf den mit <i>path</i> übergebenen Pfad.	–
<code>getcwd()</code>	Gibt einen String zurück, der den Pfad des aktuellen Arbeitsverzeichnisses (<i>Current Working Directory</i>) enthält.	–
<code>getcwdb()</code>	Wie <code>getcwd</code> , gibt aber eine <code>bytes</code> -Instanz anstelle der <code>str</code> -Instanz zurück.	–
<code>chmod(path, mode)</code>	Ändert die Zugriffsrechte des Pfades <i>path</i> .	486
<code>listdir(path)</code>	Erzeugt eine Liste mit allen Dateien und Verzeichnissen, die sich unter <i>path</i> befinden.	486
<code>mkdir(path[, mode])</code>	Legt ein neues Verzeichnis an der Stelle <i>path</i> an.	486
<code>makedirs(path[, mode])</code>	Legt ein neues Verzeichnis an der Stelle <i>path</i> an. Falls erforderlich, werden auch übergeordnete Verzeichnisse mitangelegt.	487
<code>remove(path)</code>	Entfernt die mit <i>path</i> angegebene Datei aus dem Dateisystem.	487
<code>removedirs(path)</code>	Löscht eine ganze Ordnerstruktur. Dabei löscht es von der tiefsten bis zur höchsten Ebene nacheinander alle Ordner, sofern diese leer sind.	487
<code>rename(src, dst)</code>	Benennt die mit <i>src</i> angegebene Datei oder den Ordner in <i>dst</i> um.	488

Tabelle 16.1 Methoden für den Zugriff auf das Dateisystem

Name	Beschreibung	Seite
<i>renames(src, dst)</i>	Wie <i>rename</i> , legt aber bei Bedarf die Verzeichnisstruktur des Zielpfads an. Außerdem wird nach dem Benennungsvorgang versucht, den <i>src</i> -Pfad mittels <i>removedirs</i> von leeren Ordnern zu reinigen.	–
<i>rmdir(path)</i>	Entfernt das übergebene Verzeichnis aus dem Dateisystem oder wirft <i>os.error</i> , wenn das Verzeichnis nicht existiert. Dies funktioniert nur mit leeren Verzeichnissen.	–
<i>walk(top[, topdown=True[, onerror=None]])</i>	Durchläuft den Verzeichnisbaum unter <i>top</i> rekursiv.	488

Tabelle 16.1 Methoden für den Zugriff auf das Dateisystem (Forts.)

access(path, mode)

Mit `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 ODER zusammengefasst übergeben werden:

Konstante	Bedeutung
<code>F_OK</code>	Prüft, ob der Pfad überhaupt existiert.
<code>R_OK</code>	Prüft, ob der Pfad gelesen werden darf.
<code>W_OK</code>	Prüft, ob der Pfad geschrieben werden darf.
<code>X_OK</code>	Prüft, ob der Pfad ausführbar ist.

Tabelle 16.2 Wert für den `mode`-Parameter von `os.access`

Der Rückgabewert von `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:\\Python32\\python.exe", os.F_OK | os.X_OK)
True
```

Der Python-Interpreter unter der Adresse `C:\\Python32\\python.exe` existiert und ist natürlich ausführbar.

chmod(path, mode)

Diese Funktion 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 16.3 Zugriffsflags für `os.chmod`

Wenn Sie nun beispielsweise den nachstehenden `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.

listdir(path)

Diese Funktion 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`. Dabei werden auftretende Sonderzeichen mithilfe von UTF-8 kodiert.

makedirs(path[, mode])

Diese Funktion 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 `chmod`).

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

Beachten Sie, dass `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 `makedirs`.

makedirs(path[, mode])

Wie `mkdir`; aber diese Funktion erzeugt 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.

remove(path)

Diese Funktion entfernt die mit *path* angegebene Datei aus dem Dateisystem. Übergeben Sie statt eines Pfads zu einer Datei einen Pfad zu einem Ordner, wirft `remove` eine `os.error`-Exception.

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.

removedirs(path)

Diese Funktion löscht eine ganze Ordnerstruktur, wobei von der tiefsten bis zur höchsten Ebene nacheinander alle Ordner gelöscht werden, 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`.

Um verschachtelte, nicht leere Verzeichnisse zu löschen, können Sie die Funktion `shutil.rmtree` verwenden, die auf Seite 498 beschrieben wird.

rename(src, dst)

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



Hinweis

Auf Unix-Systemen wird eine bereits unter dem Pfad *dst* erreichbare Datei ohne Meldung überschrieben, wenn Sie `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.makedirs`.

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

Eine komfortable Möglichkeit, einen Verzeichnisbaum komplett zu durchlaufen, stellt die Funktion `walk` bereit. Der Parameter *top* gibt die Wurzel des zu durchlaufenen Teilbaums an. Die Iteration geht dabei so vonstatten, dass `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 der relative Pfad von *top* zu dem Unterordner, 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 16.1 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 `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.



Hinweis

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 Endlosschleifen entstehen können.



Hinweis

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.

16.2 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.

Die folgende Tabelle gibt einen Überblick über die wichtigsten Funktionen des Moduls `os.path`.

Name	Beschreibung	Seite
<i>abspath(path)</i>	Gibt zu einem relativen Pfad den dazugehörigen absoluten und normalisierten Pfad (siehe dazu <code>os.normpath</code>) zurück.	492
<i>basename(path)</i>	Gibt den sogenannten Basisnamen des Pfads zurück.	492
<i>commonprefix(list)</i>	Gibt einen möglichst langen String zurück, mit dem alle Elemente der als Parameter übergebenen Pfadliste <i>list</i> beginnen.	493
<i>dirname(path)</i>	Gibt den Pfad zu dem Verzeichnis zurück, in dem sich <i>path</i> befindet.	493
<i>exists(path)</i>	Gibt <code>True</code> zurück, wenn der Pfad <i>path</i> im Dateisystem existiert, sonst <code>False</code> .	–
<i>getatime(path)</i>	Gibt den Zeitpunkt des letzten Zugriffs auf <i>path</i> als Unix-Zeitstempel zurück.	493
<i>getmtime(path)</i>	Gibt den Zeitpunkt der letzten Änderung von <i>path</i> als Unix-Zeitstempel zurück.	494
<i>getsize(path)</i>	Gibt die Größe der unter <i>path</i> zu findenden Datei in Bytes zurück. Der Rückgabewert ist dabei immer eine <code>int</code> -Instanz.	–
<i>isabs(path)</i>	Der Rückgabewert ist <code>True</code> , wenn es sich bei <i>path</i> um eine absolute Pfadangabe handelt, sonst <code>False</code> .	–
<i>isfile(path)</i>	Gibt <code>True</code> zurück, wenn <i>path</i> auf eine Datei verweist, sonst <code>False</code> . Die Funktion folgt dabei gegebenenfalls symbolischen Links.	–
<i>isdir(path)</i>	Wenn der übergebene Pfad auf einen Ordner verweist, wird <code>True</code> zurückgegeben, ansonsten <code>False</code> .	–
<i>islink(path)</i>	Gibt <code>True</code> zurück, wenn unter <i>path</i> ein symbolischer Link zu finden ist, sonst <code>False</code> .	–
<i>join(path1[, path2[, ...]])</i>	Verkettet die übergebenen Pfadbausteine zu einem Gesamtpfad.	494
<i>normcase(path)</i>	Wandelt einen Unix-Pfad in einen Windows-Pfad um.	494

Tabelle 16.4 Die wichtigsten Funktionen des Moduls `os.path`

Name	Beschreibung	Seite
<i>realpath(path)</i>	Gibt einen zu <i>path</i> äquivalenten Pfad zurück, der keine Umwege über symbolische Links enthält.	–
<i>split(path)</i>	Spaltet <i>path</i> in Verzeichnis und Datei auf.	494
<i>splitdrive(path)</i>	Spaltet <i>path</i> in den Laufwerksbuchstaben und Pfad auf dem Laufwerk auf.	495
<i>splittext(path)</i>	Teilt den <i>path</i> in den Pfad zu der Datei und die Dateiendung. Beide Elemente werden in einem Tupel zurückgegeben.	495

Tabelle 16.4 Die wichtigsten Funktionen des Moduls `os.path` (Forts.)

abspath(path)

Diese Funktion 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.

basename(path)

Diese Funktion 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'
```



Hinweis

Diese Funktion unterscheidet sich von dem Unix-Kommando *basename* dadurch, dass sie einen leeren String zurückgibt, wenn der String mit einem Ordnertrennzeichen endet:

```
>>> os.path.basename(r"/usr/lib/compiz/")
''
```

Im Gegensatz dazu sieht die Ausgabe des gleichnamigen Unix-Kommandos so aus:

```
$ basename /usr/lib/compiz/
compiz
```

commonprefix(list)

Diese Funktion 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.

dirname(path)

Diese Funktion 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'
```

exists(path)

Diese Funktion gibt `True` zurück, wenn der angegebene Pfad auf eine existierende Datei oder ein vorhandenes Verzeichnis verweist, ansonsten `False`.

getatime(path)

Diese Funktion 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.

getmtime(path)

Diese Funktion gibt einen Unix-Zeitstempel zurück, der angibt, wann die Datei oder der Ordner unter *path* zum letzten Mal verändert wurden. 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.

join(path1[, path2[, ...]])

Diese Funktion fügt die übergebenen Pfadangaben zu einem einzigen Pfad zusammen, indem sie verkettet werden. Dabei wird das übliche Trennzeichen des Betriebssystems verwendet.

```
>>> 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'
```

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.

split(path)

Diese Funktion 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')
```

Hinweis

Wenn der Pfad mit einem Slash oder Backslash endet, ist das zweite Element des Tuples ein leerer String:

```
>>> os.path.split("/home/revelation/")
('/home/revelation', '')
```



splitdrive(path)

Diese Funktion 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')
```

Unter Betriebssystemen, die keine Laufwerksbuchstaben unterstützen, ist der erste String des Tuples ein leerer String:

```
>>> os.path.splitdrive("/usr/share/bin")
('', '/usr/share/bin')
```

splittext(path)

Diese Funktion teilt den *path* in den Pfad zu der Datei und die Dateiendung. Beide Elemente werden in einem Tupel zurückgegeben:

```
>>> os.path.splitext(r"C:\Windows\System32\notepad.exe")
('C:\\Windows\\System32\\Notepad', '.exe')
```

16.3 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. Dabei wird von den plattformspezifischen Programmen wie beispielsweise `copy` unter Windows oder `cp` unter Unix abstrahiert.

Mit Python 3.2 sind außerdem Funktionen zum Erzeugen und Entpacken von Archivdateien (wie ZIP- oder TAR-Archiven) hinzugekommen.

Folgende Funktionen werden von `shutil` implementiert, wobei die Parameter *src* und *dst* jeweils Strings sind, die den Pfad der Quell- bzw. der Zieldatei enthalten:

Name	Beschreibung	Seite
Verzeichnis- und Dateioperationen		
<i>copyfile(src, dst)</i>	Kopiert die Datei unter <i>src</i> nach <i>dst</i> . Wenn die Datei unter <i>dst</i> bereits existiert, wird sie ohne Rückfrage überschrieben. Dabei muss der Pfad <i>dst</i> schreibbar sein. Ansonsten wird eine <code>IOError</code> -Exception geworfen.	–
<i>copyfileobj(fsrc, fdst[, length])</i>	Kopiert den Inhalt des zum Lesen geöffneten Dateiobjekts <i>fsrc</i> in das zum Schreiben geöffnete <i>fdst</i> -Objekt.	497
<i>copymode(src, dst)</i>	Kopiert die Zugriffsrechte vom Pfad <i>src</i> auf den Pfad <i>dst</i> . Dabei bleiben der Inhalt von <i>dst</i> sowie der Besitzer und die Gruppe unangetastet. Beide Pfade müssen bereits im Dateisystem existieren.	–
<i>copystat(src, dst)</i>	Wie <i>copymode</i> , aber es werden zusätzlich die Zeiten für den letzten Zugriff in die letzte Modifikation kopiert.	–
<i>copy(src, dst)</i>	Kopiert die Datei unter dem Pfad <i>src</i> nach <i>dst</i> . Im Unterschied zu <i>copyfile</i> kann <i>dst</i> dabei auch auf ein Verzeichnis verweisen, in das die Datei kopiert werden soll.	498
<i>copy2(src, dst)</i>	Wie <i>copy</i> , aber es werden zusätzlich die Zeiten des letzten Zugriffs und der letzten Änderung kopiert.	–
<i>copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2, ignore_dangling_symlinks=False)</i>	Kopiert die gesamte Verzeichnisstruktur unter <i>src</i> nach <i>dst</i> .	498

Tabelle 16.5 Funktionen des Moduls `shutil`

Name	Beschreibung	Seite
<i>ignore_patterns(*patterns)</i>	Erzeugt eine Funktion, die bei der <i>copy-tree</i> -Funktion für den Parameter <i>ignore</i> übergeben werden kann, um bestimmte Dateien vom Kopieren auszuschließen.	–
<i>move(src, dst)</i>	Verschiebt die Datei oder den Ordner von <i>src</i> nach <i>dst</i> .	–
<i>rmtree(src[, ignore_errors[, onerror]])</i>	Löscht die gesamte Verzeichnisstruktur unter <i>src</i> .	498
Archivoperationen (Die Parameterlisten wurden gekürzt.)		
<i>make_archive(base_name, format[, root_dir])</i>	Erzeugt eine Archivdatei, in der Dateien im Verzeichnis <i>root_dir</i> enthalten sind, und gibt den Namen des Archivs zurück.	499
<i>get_archive_formats()</i>	Gibt eine Liste der verfügbaren Archivformate zum Erzeugen von Archiven zurück.	501
<i>get_unpack_formats()</i>	Gibt eine Liste von verfügbaren Archivformaten zum Entpacken von Archiven zurück.	501
<i>unpack_archive(filename[, extract_dir[, format]])</i>	Entpackt das Archiv unter <i>filename</i> in das Verzeichnis <i>extract_dir</i> .	501

Tabelle 16.5 Funktionen des Moduls `shutil` (Forts.)

Nachfolgend werden einige der Operationen für Verzeichnis und Dateien im Detail besprochen.

16.3.1 Verzeichnis- und Dateioperationen

`copyfileobj(fsrc, fdst[, length])`

Diese Operation 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.

copy(src, dst)

Diese Operation 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. Dies ist der wesentliche Unterschied zu der Funktion `copyfile`, die keinen Ordner als Ziel akzeptiert.

Die Zugriffsrechte werden dabei mitkopiert.

copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2, ignore_dangling_symlinks=False)

Diese Operation 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.

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. Falls *symlinks* den Wert `False` hat und der Parameter *ignore_dangling_symlinks* auf `True` gesetzt ist, werden Fehler ignoriert, die auftreten, falls ein symbolischer Link auf eine nicht existente Datei zeigt.

Der Parameter *ignore* erwartet eine Funktion, die bestimmte Dateien vom Kopieren ausschließt. Um eine solche Funktion zu erzeugen, ist die Funktion `ignore_patterns` gedacht. Das folgende Beispiel erzeugt eine Funktion, mit der alle auf `".txt"` endenden Dateinamen aussortiert werden. Außerdem werden solche Dateinamen ignoriert, die mit `"tmp"` beginnen.

```
>>> my_ignore_function = shutil.ignore_patterns("*.txt", "tmp*")
```

Für das Kopieren der Dateien wird die Funktion verwendet, die mit dem Parameter *copy_function* übergeben wird.

Die Funktion `copytree` greift intern auf die Funktion `copystat` zurück, um die Rechte der erzeugten Verzeichnisse und Dateien zu setzen.

rmtree(src[, ignore_errors[, onerror]])

Hiermit wird die gesamte Verzeichnisstruktur unter *src* gelöscht. 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

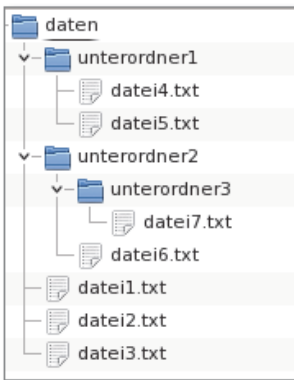
Hinweis

Exceptions, die von der Funktion `onerror` geworfen werden, werden nicht abgefangen.



16.3.2 Archivoperationen

Für die folgenden Tests gehen wir davon aus, dass sich im aktuellen Arbeitsverzeichnis ein Verzeichnis namens `daten` befindet, das wie in Abbildung 16.2 aufgebaut ist.



16

Abbildung 16.2 Beispieldaten für die Archivfunktionen

`make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]])`

Diese Funktion erzeugt ein neues Archiv, das die Dateien im Verzeichnis *root_dir* enthält. Wird *root_dir* nicht angegeben, werden die Dateien des aktuellen Arbeitsverzeichnisses gepackt. Mit dem Parameter *base_name* werden der Speicherort und der Name der Archivdatei festgelegt, wobei die Dateiendung *nicht* mitangegeben werden sollte.

Über den Parameter *format* wird das gewünschte Format der Archivdatei angegeben. Die verfügbaren Archivformate können Sie mit der Funktion `get_archive_formats` ermitteln (siehe folgender Abschnitt).

Ein typischer Aufruf von `make_archive` sieht folgendermaßen aus:


```
>>> shutil.make_archive("test", "zip", "daten")
'/home/user/test.zip'
```

Mithilfe des Unix-Programms `unzip` sehen wir, dass das Archiv alle Dateien im Verzeichnis `daten` enthält:

```
computer ~ $ unzip -v test.zip
Archive:  test.zip
  Length  Method      Size   Name
     2    Defl:N        4   datei2.txt
     2    Defl:N        4   datei3.txt
     2    Defl:N        4   datei1.txt
     2    Defl:N        4  unterordner1/datei5.txt
     2    Defl:N        4  unterordner1/datei4.txt
     2    Defl:N        4  unterordner2/datei6.txt
     2    Defl:N        4  unterordner2/unterordner3/datei7.txt
```

Möchte man nur die Dateien eines Unterverzeichnisses von *root_dir* inklusive des zugehörigen relativen Pfads packen, kann man dies mit dem Parameter *base_dir* erreichen.

Wir werden im folgenden Beispiel nur die Dateien im Unterverzeichnis `unterordner2` packen, wobei jedoch der relative Pfad innerhalb vom Verzeichnis `daten` erhalten bleibt:

```
>>> shutil.make_archive("test2", "zip", "daten", "unterordner2")
'/home/user/test2.zip'
```

Wieder überprüfen wir mithilfe von `unzip` den Inhalt des Archivs:

```
computer ~ $ unzip -v test2.zip
Archive:  test2.zip
  Length  Method      Size   Name
     2    Defl:N        4  unterordner2/datei6.txt
     2    Defl:N        4  unterordner2/unterordner3/datei7.txt
```

Wir sehen, dass alle Dateien und Ordner im Verzeichnis `unterordner2` gepackt worden sind, wobei der relative Pfad zum Verzeichnis `daten`, nämlich `unterordner2`, erhalten geblieben ist.

Für die technischen und selten gebrauchten Parameter *verbose*, *dry_run*, *owner*, *group* und *logger* verweisen wir Sie auf die Python-Dokumentation.

get_archive_formats()

Diese Funktion gibt eine Liste von zweielementigen Tupeln zurück, in denen die verfügbaren Formate zum Erstellen von Archiven beschrieben werden.

```
[('bztar', "bzip2'ed tar-file"),
 ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'),
 ('zip', 'ZIP file')]
```

Jedes Tupel in dieser Liste enthält zwei Strings: den Namen des Formats und eine kurze Beschreibung.

get_unpack_formats()

Diese Funktion arbeitet wie `get_archive_formats`, nur werden hier die verfügbaren Formate zum Entpacken von Archiven aufgelistet.

unpack_archive(filename[, extract_dir[, format]])

Diese Funktion entpackt das Archiv unter dem Pfad *filename* in das Zielverzeichnis *extract_dir*. Wird *extract_dir* nicht angegeben, werden die Daten in das aktuelle Arbeitsverzeichnis entpackt.

Mit dem Parameter *format* kann das Format des Archivs angegeben werden. Falls kein Wert für *format* übergeben wurde, versucht `unpack_archive`, das Format des Archivs anhand der Dateieindung zu ermitteln.

16.4 Zugriff auf die Laufzeitumgebung – sys

Das Modul `sys` der Standardbibliothek stellt vordefinierte Variablen 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
```

Die folgenden beiden Tabellen listen die wichtigsten Elemente des Moduls `sys` zusammen und verweisen jeweils auf eine detaillierte Erklärung einer Funktion oder Variable.

Vordefinierte Variable	Beschreibung	Seite
<i>argv</i>	eine Liste der Kommandozeilenparameter, die an das Python-Programm übergeben wurden	503
<i>byteorder</i>	Gibt die Byte-Order des Systems an.	504
<i>executable</i>	der Pfad zur ausführbaren Datei des Python-Interpreters	504
<i>hexversion</i>	die Versionsnummer des Python-Interpreters als Hexadezimalzahl	504
<i>last_traceback</i>	das Traceback-Objekt der zuletzt geworfenen Exception	507
<i>last_type</i>	der Typ der zuletzt geworfenen Exception	507
<i>last_value</i>	der Wert der zuletzt geworfenen Exception	507
<i>path</i>	die Liste der Verzeichnisse, aus denen Module eingebunden werden können	504
<i>platform</i>	Spezifiziert das verwendete Betriebssystem.	505
<i>stdin</i>	das Dateiobjekt, aus dem Eingaben gelesen werden	505
<i>stdout</i>	das Dateiobjekt, in das Ausgaben geschrieben werden	505
<i>stderr</i>	das Dateiobjekt, in das Fehlerausgaben geschrieben werden	505
<i>tracebacklimit</i>	maximale Länge der Aufrufhierarchie in Traceback-Objekten	507
<i>version</i>	die Versionsnummer des Python-Interpreters als String	506
<i>version_info</i>	die Versionsnummer des Python-Interpreters als Tupel	506

Tabelle 16.6 Vordefinierte Variablen des Moduls `sys`

Funktion	Beschreibung	Seite
<i>displayhook(value)</i>	Wird aufgerufen, wenn das Ergebnis eines Ausdrucks im interaktiven Modus ausgegeben werden soll.	507
<i>exc_info()</i>	Gibt Informationen über eine momentan abgefangene Exception zurück.	506
<i>excepthook(type, value, traceback)</i>	Wird aufgerufen, wenn eine unbehandelte Exception aufgetreten ist.	508
<i>exit([arg])</i>	Beendet das Programm.	509
<i>getrecursionlimit()</i>	Gibt die maximale Rekursionstiefe zurück.	510
<i>getrefcount(object)</i>	Gibt den Reference Count eines Objekts zurück.	509
<i>getwindowsversion()</i>	Spezifiziert die aktuell verwendete Windows-Version.	510
<i>setrecursionlimit(limit)</i>	Setzt die maximale Rekursionstiefe.	510

Tabelle 16.7 Funktionen des Moduls sys

16.4.1 Konstanten

Das Modul `sys` enthält eine Reihe von vordefinierten Variablen, die mitunter sehr nützliche Informationen bereitstellen. Die wichtigsten dieser Konstanten sollen im Folgenden erklärt werden.

argv

Die Liste `argv` enthält die Kommandozeilenparameter, mit denen das Python-Programm aufgerufen wurde. `argv[0]` ist der Name des Programms selbst. Im interaktiven Modus ist `argv` leer. Bei dem Programmaufruf

```
programm.py -bla 0 -blubb abc
```

referenziert `argv` die folgende Liste:

```
['programm.py', '-bla', '0', '-blubb', 'abc']
```

Verwenden Sie das Modul `argparse` (S. 512), wenn Sie Kommandozeilenparameter komfortabel verwalten möchten.

byteorder

Diese Konstante spezifiziert die Byte-Order³ 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.

executable

Dies ist ein String, der den vollen Pfad zur ausführbaren Datei des Python-Interpreters angibt.

```
>>> sys.executable
'C:\\Python25\\pythonw.exe'
```

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)
'0x30202f0'
```

In diesem Fall wurde Python 3.2.2 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.

path

Die Liste *path* enthält eine Reihe von Pfadangaben, die beim Einbinden eines Moduls der Reihe nach vom Interpreter durchsucht werden. Das zuerst gefundene Modul mit dem gesuchten Namen wird eingebunden.

Es steht dem Programmierer frei, die Liste so zu modifizieren, dass das Einbinden eines Moduls nach seinen Wünschen erfolgt.

```
>>> sys.path
['', 'C:\\WINDOWS\\system32\\python32.zip',
'C:\\Python32\\DLLs',
'C:\\Python32\\lib',
```

3 Die Byte-Order gibt an, in welcher Reihenfolge die einzelnen Bytes eines Wertes im Speicher abgelegt werden, der mehr als ein Byte belegt. Die Byte-Order ist relevant, wenn Binärdaten zwischen verschiedenen Plattformen ausgetauscht werden. Ein x86-PC ist ein Little-Endian-System.

```
'C:\\Python32\\lib\\plat-win',
'C:\\Python32',
'C:\\Python32\\lib\\site-packages']
```

platform

Dieser String enthält eine Kennung des zugrundeliegenden Betriebssystems. Die Kennungen der drei gängigsten Betriebssysteme kann der folgenden Tabelle entnommen werden.

System	Kennung
Linux	"linux2" bzw. "linux"
Mac OS X	"darwin"
Windows	"win32"

Tabelle 16.8 platform-Wert für verschiedene Betriebssysteme

Hinweis

Bis Python 3.2 ist der Wert von *platform* auch unter Linux 3.x "linux2". Ab Python 3.3 ist der Wert unter Linux "linux" unabhängig von der Linux-Version.

Aus diesem Grund ist es ratsam, mittels `sys.platform.startswith("linux")` zu prüfen, ob das Programm auf einem Linux-System ausgeführt wird.



stdin, stdout, stderr

Dies sind die Dateiobjekte, die für Ein- und Ausgaben des Interpreters verwendet werden. Dabei steht *stdin* für *Standard Input* und entspricht dem Dateiobjekt, aus dem die Benutzereingaben beim Aufruf von *input* gelesen werden. In das Dateiobjekt *stdout* (*Standard Output*) werden alle Ausgaben des Python-Programms geschrieben, während Ausgaben des Interpreters, beispielsweise Tracebacks, in *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 *stdin* stets ein vollwertiges Dateiobjekt sein muss, während für *stdout* und *stderr* eine Instanz reicht, die eine Methode *write* implementiert.

Die ursprünglichen Streams von *stdin*, *stdout* und *stderr* werden in `sys.__stdin__`, `sys.__stdout__` und `sys.__stderr__` gespeichert, sodass sie jederzeit wiederhergestellt werden können.

version

Dies ist ein String, der die Versionsnummer des Python-Interpreters und einige weitere Informationen, wie beispielsweise das Datum seiner Kompilierung und den verwendeten Compiler, enthält.

```
>>> print(sys.version)
3.2.2 (default, Sep  5 2011, 04:52:19)
[GCC 4.6.1 20110819 (prerelease)]
```

Beachten Sie, dass es bei *version* im Gegensatz zu *hexversion* nicht garantiert ist, dass die Versionsnummern mit den Operatoren `>` und `<` sinnvoll miteinander verglichen werden können.

version_info

Dies ist ein benanntes Tupel, das die einzelnen Komponenten der Versionsnummer des Interpreters enthält.

```
>>> sys.version_info
sys.version_info(major=3, minor=2, micro=2, releaselevel='final', serial=0)
```

Auf die einzelnen Elemente der Versionsnummer lässt sich über einen Index oder über den Komponentennamen zugreifen:

```
>>> sys.version_info[1]
2
>>> sys.version_info.minor
2
```

16.4.2 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 20.6 (S. 722).

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.

```
>>> try:
...     raise ValueError("Test")
... except ValueError:
...     print(sys.exc_info())
...
(<class 'ValueError'>, ValueError('Test',), <traceback object at
0x7fea775d3998>)
```

Die Informationen über die aktuell abgefangene Exception bleiben nicht erhalten, sondern sind nur innerhalb des `except`-Zweiges verwendbar. Falls Sie Informationen über die zuletzt geworfene Exception außerhalb eines `except`-Zweiges benötigen, sollten Sie `last_type`, `last_value` oder `last_traceback` verwenden.

last_type, last_value, last_traceback

Diese Referenzen erlauben es, Zugriff auf die zuletzt geworfene Exception zu erlangen. Die drei Informationen entsprechen denen, die von `exc_info` zurückgegeben werden.

Im Gegensatz zu den von `exc_info` zurückgegebenen Werten sind diese Referenzen auch außerhalb eines `except`-Zweiges gültig, da sie stets Informationen über die zuletzt geworfene Exception enthalten.

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.

16.4.3 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.

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 *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 *displayhook* nicht aufgerufen wird, wenn eine Ausgabe mittels `print` erfolgt:⁴

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

Das ursprüngliche Funktionsobjekt von *displayhook* können Sie über `sys.__displayhook__` erreichen und somit die ursprüngliche Funktionsweise wiederherstellen:

```
>>> sys.displayhook = sys.__displayhook__
```

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 lassen sich zum Beispiel Fehler protokollieren oder die Ausgabe eines Tracebacks verändern.

Die drei Parameter der Funktion entsprechen denen, die von *exc_info* zurückgegeben werden, und enthalten Informationen über die Exception.

⁴ 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.

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: "{}".format(value))
...
>>> sys.excepthook = f
>>> abc
gnahahaha: "name 'abc' is not defined"
```

Das ursprüngliche Funktionsobjekt von *excepthook* können Sie über *sys.__excepthook__* erreichen und somit die ursprüngliche Funktionsweise wiederherstellen.

16.4.4 Sonstige Funktionen

Neben den bereits besprochenen vordefinierten Variablen 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.

16

exit([arg])

Diese Funktion 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 *stderr* ausgegeben, bevor das Programm mit dem Exit Code 0 beendet wird.

getrefcount(object)

Diese Funktion 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, kann sie vom Garbage Collector entsorgt werden.

Beachten Sie, dass es dem Interpreter bei Instanzen unveränderlicher Datentypen freisteht, eine neue Instanz zu erzeugen oder eine bereits bestehende neu zu referen-

zieren. Aus diesem Grund kann es vorkommen, dass zum Beispiel Instanzen ganzer Zahlen einen hohen Reference Count haben.

getrecursionlimit(), setrecursionlimit(limit)

Mit diesen Funktionen wird die maximale Rekursionstiefe ausgelesen oder verändert. Die maximale Rekursionstiefe ist mit 1000 vorgelegt und bricht beispielsweise endlos rekursive Funktionsaufrufe ab, bevor diese zu einem Speicherüberlauf führen können.

getwindowsversion()

Diese Funktion erlaubt es, die Details über die Version des aktuell verwendeten Windows-Betriebssystems auszulesen. Die Funktion gibt ein benanntes 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 / Server 2003 / Vista / Server 2008 / 7
3	Windows CE

Tabelle 16.9 Windows-Plattformen

Das letzte Element des Tupels ist ein String, der weiterführende Informationen enthält.

```
>>> sys.getwindowsversion()
sys.getwindowsversion(major=5, minor=1, build=2600, platform=2,
service_pack='Service Pack 3')
```

Auf das benannte Tupel kann wie bei *version_info* sowohl über Indizes als auch über die Komponentennamen zugegriffen werden.

Um zwischen den einzelnen Windows-Versionen zu unterscheiden, die unter Plattform 2 zusammengefasst sind, kann die Versionsnummer (bestehend aus dem *major*- und dem *minor*-Feld des Tupels) herangezogen werden:

Name	Major	Minor
Windows 2000	5	0
Windows XP	5	1
Windows Server 2003	5	2
Windows Vista bzw. Windows Server 2008	6	0
Windows 7	6	1

Tabelle 16.10 Windows-Versionen

Die Funktion *getwindowsversion* ist unter anderen Betriebssystemen als Microsoft Windows nicht verfügbar.

16.5 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.

`platform.machine()`

Diese Funktion gibt die Prozessorarchitektur des PCs als String zurück. Bei aktuellen PCs ist dies `i686` (32-Bit Systeme) oder `x86_64` (64-Bit Systeme).

`platform.node()`

Diese Funktion gibt den Netzwerknamen des PCs als String zurück.

`platform.processor()`

Diese Funktion gibt einen String zurück, der den Typ und den Hersteller des Prozessors enthält.

`platform.system()`

Diese Funktion gibt einen String zurück, der den Namen des Betriebssystems, beispielsweise also »Linux« oder »Windows«, enthält.

16.6 Kommandozeilenparameter – argparse

Im Abschnitt über das Modul `sys` der Standardbibliothek wurde die globale Liste `sys.argv` besprochen, mithilfe derer Sie auf die Kommandozeilenparameter zugreifen können, die beim Aufruf eines Programms übergeben wurden. Dieser rudimentäre Zugang zu den Kommandozeilenparametern ist jedoch oft nicht ausreichend. Das Modul `argparse`, das in diesem Abschnitt besprochen wird, erlaubt Ihnen einen komfortableren Umgang mit Kommandozeilenparametern.

Bislang wurden in diesem Buch ausschließlich Konsolenprogramme behandelt, also Programme, die eine rein textbasierte Schnittstelle zum Benutzer haben. Solche Programme werden üblicherweise aus einer *Konsole*, auch *Shell* genannt, gestartet. Eine Konsole ist beispielsweise die Eingabeaufforderung unter Windows.

Unter Windows wird ein Python-Programm aus der Eingabeaufforderung heraus gestartet, indem in das Programmverzeichnis gewechselt und dann der Name der Programmdatei eingegeben wird. Hinter dem Namen können *Optionen* und *Argumente* übergeben werden:

- Ein Argument wird 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 kann beispielsweise folgendermaßen aussehen:

```
programm.py karl 1337 heinz
```

- Neben den Argumenten können Sie Optionen übergeben, welche mit Keyword Arguments vergleichbar sind. 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 kann 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 sind 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 wird die Verwendung des Moduls `argparse` anhand zweier Beispiele besprochen.

16.6.1 Taschenrechner – ein einfaches Beispiel

Das erste Beispiel ist ein einfaches Taschenrechnerprogramm, 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
```

Über die Option `-o` wird eine Rechenoperation festgelegt, die auf die beiden folgenden Argumente angewendet wird. Wenn die Option `-o` fehlt, sollen die Argumente addiert werden.

Zu Beginn des Programms muss die Klasse `ArgumentParser` des Moduls `argparse` eingebunden und instanziiert werden:

```
from argparse import ArgumentParser
parser = ArgumentParser()
```

Jetzt können durch die Methode `add_argument` der `ArgumentParser`-Instanz erlaubte Optionen hinzugefügt werden. In unserem Fall ist es nur eine:

```
parser.add_argument("-o", "--operation", dest="operation", default="plus")
```

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 sind die Optionen `-o` und `--operation` gleichbedeutend. Der dritte Parameter, ein Keyword Argument wohlgemerkt, gibt an, unter welchem Namen der Wert der Option später im Programm verfügbar gemacht werden soll. Über den letzten Parameter wird ein Standardwert für diese Option festgelegt, der verwendet wird, wenn die Option nicht angegeben ist.

Neben der Option `-o` müssen noch die Argumente für die beiden Operanden hinzugefügt werden. Dabei werden der Name und der zulässige Datentyp der Argumente angegeben.

```
parser.add_argument("op1", type=float)
parser.add_argument("op2", type=float)
```

Nachdem alle Optionen und Argumente hinzugefügt worden sind, wird die Methode `parse_args` aufgerufen, die die Kommandozeilenparameter ausliest und in der gewünschten Form aufbereitet.

```
args = parser.parse_args()
```

Als Nächstes legen wir ein Dictionary an, das alle möglichen Rechenoperationen als Schlüssel und die dazugehörige Berechnungsfunktion als jeweiligen Wert enthält. Die Schlüssel sind dieselben, die über die Option `-o` angegeben werden können, sodass wir anhand des bei der Option übergebenen Strings direkt auf die zu verwendende Berechnungsfunktion schließen können:

```
calc = {
    "plus" : lambda a, b: a + b,
    "minus" : lambda a, b: a - b,
    "mal" : lambda a, b: a * b,
    "geteilt" : lambda a, b: a / b
}
```

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 `args` 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`. Analog kann auf Argumente zugegriffen werden.

```
op = args.operation
if op in calc:
    print("Ergebnis:", calc[op](args.op1, args.op2))
else:
    parser.error("{} ist keine Operation".format(op))
```

Für Fehler, die aufgrund falscher oder fehlender Kommandozeilenparameter auftreten, eignet sich die Methode `error` der `ArgumentParser`-Instanz, die eine entsprechende Fehlermeldung ausgibt und das Programm beendet.

Bevor wir zu einem komplexeren Beispielprogramm übergehen, besprechen wir den Konstruktor der Klasse `ArgumentParser` sowie deren Methode `add_argument` im Detail.

ArgumentParser([description][, epilog][, prog][, usage][, add_help][, argument_default][, parents][, conflict_handler][, formatter_class])

Dem Konstruktor der Klasse `ArgumentParser` können eine Reihe von Schlüsselwortparametern übergeben werden, deren Bedeutung hier kurz erläutert wird. Viele der Parameter beziehen sich auf die dynamisch generierte Hilfeseite, die das Modul `argparse` zur Verfügung stellt. Ein Beispiel zur Verwendung dieser Hilfeseite finden Sie im nächsten Abschnitt.

Parameter	Bedeutung
<i>description</i>	ein String, der auf der Hilfeseite vor der Erklärung der Optionen und Argumente angezeigt wird
<i>epilog</i>	ein String, der auf der Hilfeseite nach der Erklärung der Optionen und Argumente angezeigt wird
<i>prog</i>	Der Programmname, wie er auf der Hilfeseite angezeigt wird. Standardmäßig wird hier der Name der ausgeführten Datei verwendet.
<i>usage</i>	Ein String, der die Verwendung des Programms mit seinen Kommandozeilenparametern zusammenfasst. Standardmäßig wird dieser String automatisch erzeugt.
<i>add_help</i>	Gibt an, ob eine Hilfeseite über die Optionen <code>-h</code> bzw. <code>--help</code> angeboten werden soll. Standardwert: <code>True</code>
<i>argument_default</i>	Setzt einen globalen Standardwert für nicht angegebene Optionen oder Argumente. Der Wert <code>argparse.SUPPRESS</code> bewirkt, dass nicht angegebene Optionen oder Argumente ignoriert werden. Standardwert: <code>None</code>
<i>parents</i>	eine Liste von <code>ArgumentParser</code> -Instanzen, deren Argumente und Optionen übernommen werden sollen
<i>conflict_handler</i>	Bei Verwendung von <i>parents</i> kann es zu Konflikten zwischen gleichnamigen Parametern aus verschiedenen <code>ArgumentParser</code> -Instanzen kommen. Wird für <i>conflict_handler</i> der Wert <code>"resolve"</code> übergeben, wird versucht, solche Konflikte aufzulösen. Standardmäßig wird eine <code>ArgumentError</code> -Exception geworfen.

Tabelle 16.11 Schlüsselwortparameter des Konstruktors `ArgumentParser`

Parameter	Bedeutung
<i>formatter_class</i>	Eine alternative Formatierungsklasse für die Hilfeseite. Mögliche Werte sind hier <code>argparse.RawDescriptionHelpFormatter</code> , <code>argparse.RawTextHelpFormatter</code> oder <code>argparse.ArgumentDefaultsHelpFormatter</code> . ⁵

Tabelle 16.11 Schlüsselwortparameter des Konstruktors `ArgumentParser` (Forts.)

`add_argument(*name_or_flags[, action][, nargs][, const][, default][, type][, choices][, required][, help][, metavar][, dest][, version])`

Die Funktion `add_argument` fügt ein Argument bzw. eine Option zur Programmschnittstelle hinzu. Dabei können eine Reihe von Parametern angegeben werden, die wir hier besprechen werden.

Parameter	Bedeutung
<i>*name_or_flags</i>	Bei einem Argument ein einzelner String, der den Argumentnamen angibt. Bei einer Option eine Reihe von Strings, die die alternativen Namen der Option angeben. Zwischen einem Argument und einer Option wird anhand eines führenden Bindestrichs unterschieden.
<i>action</i>	Spezifiziert, wie der programminterne Wert des Parameters aus dem vom Benutzer angegebenen Wert gebildet werden soll. Mögliche Werte sind: ▶ <code>"store"</code>: speichert den Wert unmodifiziert. ▶ <code>"store_const"</code>: speichert einen konstanten Wert. Dieser wird durch den Schlüsselwortparameter <code>const</code> festgelegt. ▶ <code>"store_true"</code>: wie <code>store_const</code> mit <code>const=True</code> ▶ <code>"store_false"</code>: wie <code>store_const</code> mit <code>const=False</code> ▶ <code>"append"</code>: speichert den Wert als Element einer Liste. Dies ist sinnvoll, wenn die Mehrfachangabe einer Option zulässig sein soll. ▶ <code>"append_const"</code>: wie <code>append</code> mit dem konstanten Wert, der durch den Schlüsselwortparameter <code>const</code> festgelegt wird.

Tabelle 16.12 Parameter der Methode `add_argument`

⁵ Die ersten beiden Klassen verhindern das Entfernen von Whitespaces im Beschreibungsteil bzw. der gesamten Hilfeseite. Die letzte Klasse fügt eine Beschreibung von Defaultwerten der Parameter zur Hilfeseite hinzu.

Parameter	Bedeutung
	▶ "version": gibt eine Versionsinformation aus und beendet das Programm. Der Schlüsselwortparameter <i>version</i> muss angegeben sein. ▶ Standardwert: "store"
<i>nargs</i>	Erlaubt es, zu steuern, wie oft ein Argument oder eine Option angegeben werden darf bzw. muss. Die übergebenen Werte werden dann als Elemente einer Liste programmintern verfügbar gemacht. Mögliche Werte für <i>nargs</i> sind: eine ganze Zahl N für genau N-maliges Vorkommen, "?" für ein oder kein Mal, "*" für beliebig oft und "+" für beliebig oft, aber mindestens einmal. Standardwert: 1, sofern nichts Gegenteiliges für <i>action</i> übergeben wird
<i>const</i>	Wird für einige der Aktionen benötigt, die über den Schlüsselwortparameter <i>action</i> festgelegt werden. Standardwert: None
<i>default</i>	Gibt den Defaultwert an, den eine Option annehmen soll, wenn kein entsprechender Wert vom Benutzer übergeben wurde. Standardwert: None.
<i>type</i>	Normalerweise werden die Benutzerangaben als Strings interpretiert. Mithilfe des <i>type</i>-Parameters werden die Angaben automatisch in einen anderen Datentyp überführt.
<i>choices</i>	Ein iterierbares Objekt, das die Werte für eine Option oder ein Argument enthält, aus denen der Benutzer auswählen kann.
<i>required</i>	Normalerweise sind Optionen stets optional. Mithilfe des Parameters <i>required</i> kann eine Option hinzugefügt werden, die vom Benutzer angegeben werden muss.
<i>help</i>	Ein String, der das Argument bzw. die Option beschreibt. Diese Beschreibung wird auf der automatisch generierten Hilfeseite angezeigt.
<i>metavar</i>	Auf der Hilfeseite werden standardmäßig die eigentlich programm-internen Namen der Argumente und Optionen verwendet, die für <i>dest</i> angegeben werden. Mithilfe des Parameters <i>metavar</i> lässt sich dieser Name ändern.
<i>dest</i>	Der programminterne Name eines Arguments bzw. einer Option.
<i>version</i>	Der Versionsstring, der bei <i>action</i>="version" ausgegeben wird.

Tabelle 16.12 Parameter der Methode `add_argument` (Forts.)

16.6.2 Komplexeres Beispiel

Im vorangegangenen Abschnitt wurde ein Minimalbeispiel für die Verwendung von `argparse` präsentiert. Dieses Beispielprogramm wird in diesem Abschnitt erweitert, um weitere Möglichkeiten vorzustellen, die das Modul `argparse` bietet. Hier sehen Sie zunächst den Quellcode des veränderten Beispielprogramms:

```
from argparse import ArgumentParser

parser = ArgumentParser(description = "Ein Taschenrechner")
parser.add_argument("-o", "--operation", dest="operation",
                    default="plus", help="Rechenoperation")
parser.add_argument("operanden", metavar="Operand", type=float,
                    nargs="+", help="Operanden")
parser.add_argument("-i", "--integer", dest="type",
                    action="store_const", const=int, default=float,
                    help="Ganzzahlige Berechnung")

args = parser.parse_args()

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
}

op = args.operation
if op in calc:
    res = 0 if op in ("plus", "minus") else 1
    for z in args.operanden:
        res = calc[op](res, args.type(z))
    print("Ergebnis:", res)
else:
    parser.error("{} ist keine gültige Operation".format(op))
```

Zunächst wird beim Hinzufügen neuer Argumente konsequent der Schlüsselwortparameter `help` übergeben. Hier kann eine Beschreibung des Arguments in Form eines Strings angegeben werden, die das Modul `argparse` dazu verwendet, eine dynamisch generierte Hilfeseite anzubieten. Diese Hilfeseite wird angezeigt, wenn das Programm mit der Option `-h` bzw. `--help` aufgerufen wird. Für das obige Beispiel sieht die Hilfeseite folgendermaßen aus:

```
usage: test3.py [-h] [-o OPERATION] [-i] Operand [Operand ...]
Ein Taschenrechner
positional arguments:
  Operand                Operanden
optional arguments:
  -h, --help              show this help message and exit
  -o OPERATION, --operation OPERATION
                          Rechenoperation
  -i, --integer            Ganzzahlige Berechnung
```

Anhand der Hilfeseite wird eine weitere Änderung im Beispielprogramm ersichtlich: Es werden nicht mehr zwei Argumente `op1` und `op2` erwartet, sondern eine Liste von beliebig vielen Argumenten (aber mindestens einem). Dies lässt sich erreichen, indem beim `add_argument`-Aufruf der Parameter `nargs` auf den Wert `"+"` gesetzt wird. Zusätzlich wird mittels `metavar` ein Name festgelegt, unter dem das Argument auf der Hilfeseite erscheint. Die Berechnung des Ergebnisses aus der Operandenliste `args.operanden` sollte selbsterklärend sein.⁶

Die dritte Änderung ist die Option `-i`, die das Taschenrechnerprogramm auf ganzzahlige Arithmetik umstellt. Dazu referenziert `args.type` entweder `int` oder `float`, je nachdem, ob `-i` angegeben wurde oder nicht. Dies wird über die Aktion `store_const` mit dem konstanten Wert `int` und dem Defaultwert `float` erreicht. Im Programm wird der über `args.type` zugängliche Datentyp verwendet, um die Operanden entsprechend zu konvertieren.

16

16.7 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.

⁶ Die Berechnung des Ergebnisses erfolgt von links nach rechts. Das ist insbesondere bei den nicht assoziativen Operationen `-` und `/` wichtig.

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, was im Regelfall unerwünscht ist:

```
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 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 intern verwalteten 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:⁷

```
>>> import copy
>>> s = [1, 2, 3]
>>> t = copy.copy(s)
>>> t
[1, 2, 3]
>>> t is s
False
```

⁷ Natürlich kann eine Liste auch per Slicing kopiert werden.

Das Modul `copy` erlaubt aber das Kopieren beliebiger Instanzen.

Das Beispiel zeigt, dass `t` zwar die gleichen Elemente wie `s` enthält, aber trotzdem nicht auf dieselbe Instanz wie `s` referenziert, sodass der Vergleich mit `is` negativ ausfällt.

Der Unterschied zwischen `copy.copy` und `copy.deepcopy` besteht darin, wie mit Referenzen umgegangen wird, die die zu kopierenden Instanzen enthalten. Die Funktion `copy.copy` erzeugt zwar eine neue Liste, aber die Referenzen innerhalb der Liste verweisen trotzdem auf dieselben Elemente. Mit `copy.deepcopy` hingegen wird die Instanz selbst kopiert und anschließend rekursiv auch alle von ihr referenzierten Instanzen.

Wir veranschaulichen diesen Unterschied anhand einer Liste, die eine weitere Liste enthält:

```
>>> liste = [1, [2, 3]]
>>> liste2 = copy.copy(liste)
>>> liste2.append(4)
>>> liste2
[1, [2, 3], 4]
>>> liste
[1, [2, 3]]
```

Wie erwartet, verändert sich beim Anhängen des neuen Elements 4 an `liste2` nicht die von `liste` referenzierte Instanz. Wenn wir aber die innere Liste `[2, 3]` verändern, betrifft dies sowohl `liste` als auch `liste2`:

```
>>> liste2[1].append(1337)
>>> liste2
[1, [2, 3, 1337], 4]
>>> liste
[1, [2, 3, 1337]]
```

Der `is`-Operator zeigt uns den Grund für dieses Verhalten: Bei `liste[1]` und `liste2[1]` handelt es sich um dieselbe Instanz:

```
>>> liste[1] is liste2[1]
True
```

Arbeiten wir stattdessen mit `copy.deepcopy`, wird die Liste inklusive aller enthaltenen Elemente kopiert:

```
>>> liste = [1, [2, 3]]
>>> liste2 = copy.deepcopy(liste)
>>> liste2[1].append(4)
```

```
>>> liste2
[1, [2, 3, 4]]
>>> liste
[1, [2, 3]]
>>> liste[1] is liste2[1]
False
```

Sowohl die Manipulation von `liste2[1]` als auch der `is`-Operator zeigen, dass es sich bei `liste2[1]` und `liste[1]` um verschiedene Instanzen handelt.

Zurück zum Eingangsbeispiel

Nun können wir unsere Beispielklasse `MeineKlasse` so anpassen, dass die Methode `getliste` eine Kopie der intern verwalteten Liste zurückgibt:

```
class MeineKlasse:
    def __init__(self):
        self.Liste = [1, 2, 3]

    def getliste(self):
        return copy.deepcopy(self.Liste)

    def zeigeliste(self):
        print(self.Liste)
```

Führen wir nun denselben Testcode wie oben mit der neuen Klasse aus, ist der unerwünschte Seiteneffekt verschwunden:

```
>>> instanz = MeineKlasse()
>>> liste = instanz.getliste()
>>> liste.append(1337)
>>> instanz.zeigeliste()
[1, 2, 3]
```

Wir verwenden hier `deepcopy`, damit das Attribut `Liste` auch dann vor Seiteneffekten geschützt ist, wenn sich veränderbare Elemente in der Liste befinden.



Hinweis

Es gibt 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 mehr Speicherplatz und Rechenzeit als eine einfache Zuweisung. Deshalb sollten Sie `copy` wirklich nur dann benutzen, wenn Sie tatsächlich eine echte Kopie brauchen.

16.8 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 Programmablauf 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 weitergereicht 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 = 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.



Hinweis

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.

Außerdem 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.

Kapitel 17

Parallele Programmierung

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

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, werden einige Begriffe eingeführt, und Sie müssen die prinzipielle Arbeitsweise moderner Betriebssysteme verstehen.

17.1 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 17.1 zeigt eine typische Arbeitssitzung, wobei jeder Kasten für ein laufendes Programm steht. Die Länge der Kästen entlang der Zeitachse zeigt an, wie lange der jeweilige Prozess läuft.

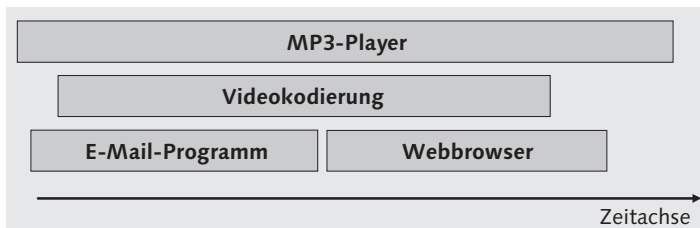


Abbildung 17.1 Mehrere Prozesse laufen gleichzeitig ab.

Faktisch kann ein Prozessor aber nur genau eine 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 vom Betriebssystem übernommen, das deshalb auch *Multitasking-System* (dt. »Mehrprozessbetriebssystem«) genannt wird.

Die korrekte Darstellung unseres anfänglichen Beispiels müsste also eher wie in Abbildung 17.2 gezeigt aussehen. Dabei symbolisiert jedes kleine Kästchen eine Zeitscheibe:

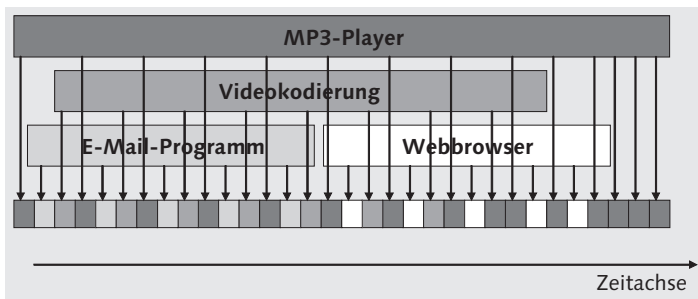


Abbildung 17.2 Die Prozesse wechseln sich ab und laufen nicht gleichzeitig.

Die Leichtgewichte unter den Prozessen – Threads

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 darf die Benutzeroberfläche während einer aufwendigen Berechnung nicht blockieren, sondern soll den aktuellen Status anzeigen, und der Benutzer muss die Möglichkeit haben, die Berechnung gegebenenfalls abbrechen zu

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 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 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 17.3 illustriert.

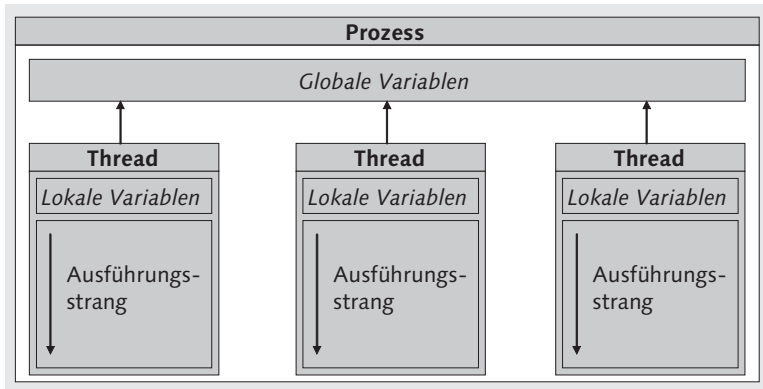


Abbildung 17.3 Ein Prozess mit drei Threads

- 1 Seit Python 3.0 gibt es das Modul `multiprocessing`, das die komfortable Nutzung mehrerer Prozesse und auch deren Synchronisation ermöglicht. Näheres dazu entnehmen Sie der Python-Dokumentation.
- 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.

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. Näheres zum Modul `multiprocessing` finden Sie in der Python-Dokumentation.

Nach dieser theoretischen Einführung wenden wir uns der Programmierung mit Threads in Python zu.

17.2 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.

17.3 Das Modul `_thread`

Das Modul `_thread` kann einzelne Funktionen in einem separaten Thread ausführen. Wir werden nun ein Beispielprogramm entwickeln, das diese Fähigkeit nutzt.

17.3.1 Ein Beispiel für eine rechenintensive Funktion – die Approximation von Pi

Um die Arbeit mit Threads zu demonstrieren, benötigen wir irgendeine Modellfunktion mit langer Laufzeit, die beim Aufruf unser Programm blockiert.

Wir werden dazu eine Funktion verwenden, die die Kreiszahl π mithilfe des Wallis'schen Produkts³ approximiert.

$$\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \dots = \frac{\pi}{2}$$

³ Das Produkt ist nach dem englischen Mathematiker John Wallis (1616–1703) benannt, der es im Jahre 1655 entdeckte.

Im Zähler stehen dabei immer gerade Zahlen, die sich bei jedem zweiten Faktor um 2 erhöhen. Der Nenner enthält nur ungerade Zahlen, die sich mit Ausnahme des ersten Faktors ebenfalls alle zwei Faktoren um 2 erhöhen.

Die Funktion `naehere_pi_an`, die als Parameter die Anzahl der zu berücksichtigenden Faktoren erhält, kann damit folgendermaßen definiert werden:

```
def naehere_pi_an(n):
    pi_halbe = 1
    zaehler, nenner = 2.0, 1.0

    for i in range(n):
        pi_halbe *= zaehler / nenner
        if i % 2:
            zaehler += 2
        else:
            nenner += 2

    print("Annäherung mit {} Faktoren: {:.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)
Annäherung 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.

17.3.2 Parallele Berechnung mithilfe von Threads

Um einen neuen Thread zu erzeugen, bietet das Modul `_thread` die Funktion `_thread.start_new_thread` an, 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.

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, (11111111,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (100000,))
_thread.start_new_thread(naehere_pi_an, (1234569,))
_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 unschön, weil sie Rechenleistung sinnlos vergeudet und das Programm mit `[Strg]+[C]` beendet werden muss. Wir werden erst bei dem Modul `threading` bessere Methoden kennen lernen, um einen Thread auf das Ende eines anderen warten zu lassen.

Das Interessante an diesem Programm ist die Reihenfolge der Ausgabe, die nicht mit der Reihenfolge der Aufrufe übereinstimmt:

```
Annaeherung mit 1337 Faktoren: 3.1427668611489281
Annaeherung mit 10000 Faktoren: 3.1414355935898644
Annaeherung mit 100000 Faktoren: 3.1415769458226377
Annaeherung mit 1234569 Faktoren: 3.1415939259321926
Annaeherung mit 11111111 Faktoren: 3.1415927949601699
```

Je größer das übergebene `n` war, desto länger musste auf die Ausgabe der dazugehörigen Annäherung von π gewartet werden, ganz egal, wann die Funktion gestartet wurde. Offensichtlich liefen alle Berechnungen parallel ab, wie wir es erwartet hatten.

Im letzten Beispiel hatte jeder Thread seine eigenen Variablen und musste keine Daten mit anderen Threads austauschen. Im nächsten Abschnitt werden wir uns mit dem Datenaustausch zwischen Threads beschäftigen.

17.3.3 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.⁴

Würde man beispielsweise unser vorhergehendes Beispiel um einen Zähler erweitern, der die Anzahl der zurzeit aktiven Threads enthält, damit das Programm nach dem Beenden aller Berechnungen von selbst terminiert, könnte man ganz naiv folgende Implementation vorschlagen:

```
import _thread

anzahl_threads = 0

def naehere_pi_an(n):
    global anzahl_threads
    anzahl_threads += 1

    # hier wurde der Berechnungscode zur Übersicht ausgelassen

    anzahl_threads -= 1

_thread.start_new_thread(naehere_pi_an, (10000000,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (99999999,))
_thread.start_new_thread(naehere_pi_an, (123456789,))
_thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while anzahl_threads > 0:
    pass
```

Dieses Programm hat zwei schwerwiegende Fehler: Erstens funktioniert es nicht 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.

⁴ Die eigentliche Kunst bei der Programmierung mit Threads ist es, diese Stolperfallen zu umgehen. Es ist oft schwierig, die Abläufe in parallelen Programmen zu überblicken, weswegen sich leicht Fehler einschleichen.

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 vorzeitig abgebrochen 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 verweist <code>anzahl_threads</code> auf den Wert 3.
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 17.1 Problemszenario beim gleichzeitigen Zugriff auf eine globale Variable

Im Beispiel wurde `anzahl_threads` also nur um eins 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 wird.

Wir können unser Beispielprogramm folgendermaßen um Critical Sections erweitern, wobei wir außerdem einen Schalter namens `thread_gestartet` einfügen, damit das Hauptprogramm mindestens so lange wartet, bis die Threads gestartet worden sind. Der Zugriff auf die Variablen `anzahl_threads` und `thread_gestartet` wird durch das Lock-Objekt `lock` gesichert:

```
import _thread

anzahl_threads = 0
thread_gestartet = False

lock = _thread.allocate_lock()

def naehere_pi_an(n):
    global anzahl_threads, thread_gestartet

    lock.acquire()
    anzahl_threads += 1
    thread_gestartet = True
    lock.release()

# hier wurde der Berechnungscode zur Übersicht ausgelassen
```

```
lock.acquire()
anzahl_threads -= 1
lock.release()

_thread.start_new_thread(naehere_pi_an, (100000,))
_thread.start_new_thread(naehere_pi_an, (10000,))
_thread.start_new_thread(naehere_pi_an, (11111111,))
_thread.start_new_thread(naehere_pi_an, (1234569,))
_thread.start_new_thread(naehere_pi_an, (), {"n" : 1337})

while not thread_gestartet:
    pass

while anzahl_threads > 0:
    pass
```

Am Anfang des Programms wird der Schalter `_thread_gestartet` auf `False` gesetzt, und 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.

Tabelle 17.2 Lösung des `anzahl_threads`-Problems mit einem Lock-Objekt

Zeitfenster	Thread A	Thread B
B wurde durch lock.acquire schlafen gelegt. A wird weiter ausgeführt.		
3	Den Wert um 1 erhöhen. Im Speicher existiert nun eine neue Instanz mit dem Wert 3. Die neue Instanz an die Referenz <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 <i>B</i> <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 17.2 Lösung des `anzahl_threads`-Problems mit einem Lock-Objekt (Forts.)

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 schwer reproduzierbare und lokalisierbare Fehler enthalten. Die Herausforderung beim Umgang mit Threads besteht deshalb darin, solche Probleme zu umgehen.



Hinweis

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.

Das folgende Ablaufprotokoll zeigt, wie ein Deadlock entstehen kann. Dabei gebe es zwei Threads *A* und *B*, und *M* und *L* seien Lock-Objekte.

Zeitfenster	Thread A	Thread B
1	das Lock-Objekt <i>L</i> mit <i>L.acquire()</i> sperren	<i>schläft</i>
Zeitfenster von A endet, und Thread B wird aktiviert.		
2	<i>schläft</i>	Mit <i>M.acquire()</i> wird das Lock-Objekt <i>M</i> gesperrt.
Zeitfenster von B endet, und Thread A wird aktiviert.		
3	<i>M.acquire</i> wird gerufen. Da <i>M</i> bereits gesperrt ist, wird A schlafen gelegt.	<i>schläft</i>
A wurde durch <i>M.acquire</i> schlafen gelegt. B wird weiter ausgeführt.		
4	<i>schläft</i>	Ruft <i>L.acquire</i> , woraufhin B schlafen gelegt wird, da <i>L</i> bereits gesperrt ist.
A wurde durch <i>M.aquire</i> und B durch <i>L.aquire</i> gesperrt.		
5	<i>schläft</i>	<i>schläft</i>

Tabelle 17.3 Beispielszenario eines Deadlocks.

Am Ende dieses Ablaufs befinden sich beide Threads im Schlafzustand und warten auf die Freigabe eines Lock-Objekts. Da aber der jeweils andere Thread das Lock-Objekt gesperrt hat, auf dessen Freigabe gewartet wird, werden die Threads nie aufgeweckt. Der Programmablauf hängt also in diesem Zustand fest – ein Deadlock hat sich eingestellt.

17.4 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 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 Primzahlen⁵ sind. Zu diesem Zweck definieren wir eine Klasse `PrimzahlThread`, die von `threading.Thread` erbt und als Parameter für den Konstruktor die zu überprüfende Zahl erwartet.

Die Klasse `threading.Thread` besitzt eine Methode namens `start`, die den Thread ausführt. Was genau ausgeführt werden soll, bestimmt die `run`-Methode, die wir mit unserer Primzahlberechnung überschreiben. Im ersten Schritt soll der Benutzer in einer Eingabeaufforderung Zahlen eingeben können, die dann überprüft werden. Ist die Überprüfung abgeschlossen, wird das Ergebnis auf dem Bildschirm ausgegeben. Das Programm inklusive der Klasse `PrimzahlThread` sieht dann folgendermaßen aus:⁶

```
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("> ")
```

5 Eine Primzahl ist eine natürliche Zahl, die genau zwei Teiler besitzt.

Die ersten sechs Primzahlen sind demnach 2, 3, 5, 7, 11 und 13.

6 Der verwendete Algorithmus für die Primzahlprüfung ist sehr primitiv und dient hier nur als Beispiel für irgendeine rechenintensive Funktion.

```

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 gestarteten Threads beendet worden sind, denn `join` unterbricht die Programmausführung so lange, bis der Thread, für den es aufgerufen wurde, terminiert wurde.

Diese Methode, auf das Ende aller Threads zu warten, ist wesentlich eleganter als die im letzten Abschnitt verwendete Endlosschleife, da mit `join` keine Rechenzeit verschwendet und das Programm automatisch beendet wird, sobald kein Thread mehr läuft.

Ein Programmlauf könnte dann so aussehen, wobei die teils verzögerten Ausgaben zeigen, dass tatsächlich parallel gerechnet wurde:

```

> 737373737373737
> 5672435793
5672435793 ist nicht prim, da 5672435793 = 3 * 1890811931
> 909091
909091 ist prim
> 100000000000037
> 5643257
5643257 ist nicht prim, da 5643257 = 23 * 245359
> 4567
4567 ist prim
100000000000037 ist prim
737373737373737 ist prim
> ende

```

17.4.1 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 besteht 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 100000000000037 auf ihre Primzahleigenschaft hin untersuchen wollen. Unglücklicherweise wurde der Thread, der die Überprüfung übernahm, genau dann fertig, als der Benutzer bereits die ersten beiden Ziffern, 56, der nächsten zu prüfenden Zahl, 56547, eingegeben hatte. Dies führte zu einer hässlichen »Zerstückelung« der Eingabe und sollte vermieden werden.

Wir werden zu diesem Zweck die Klasse `PrimzahlThread` mit einem statischen Attribut namens `Ergebnis` versehen, das in einem Dictionary die Ergebnisse der Berechnungen speichert. Dabei wird jeder zu prüfenden Zahl der Status bzw. das Ergebnis der Berechnung zugewiesen, wobei der Wert "in Arbeit" dafür steht, dass aktuell noch gerechnet wird, und der String "prim" anzeigt, dass es sich bei der Zahl um eine Primzahl handelt. Für Nicht-Primzahlen werden wir das gefundene Teilerprodukt in dem Dictionary speichern. Eine Momentaufnahme von `PrimzahlThread.Ergebnis` sieht dann folgendermaßen aus:

```
{
    737373737373737 : "in Arbeit",
    5672435793 : "3 * 1890811931",
    909091 : "prim",
    100000000000037 : "in Arbeit",
    5643257 : "23 * 245359"
}
```


In dem Beispiel befinden sich die Zahlen 7373737373737 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 soll 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
10000000000037 = in Arbeit
56547 = 56547 * 18849
-----
> status
----- Aktueller Status -----
5643257 = 5643257 * 245359
909091 = prim
737373737373737 = in Arbeit
10000000000037 = prim
56547 = 56547 * 18849
-----
> status
----- Aktueller Status -----
5643257 = 5643257 * 245359
909091 = prim
737373737373737 = prim
10000000000037 = prim
56547 = 56547 * 18849
-----
> ende

```

Mit dieser Version des Programms werden die angesprochenen Probleme zufriedenstellend beseitigt. Allerdings kann immer noch ein kleiner Schönheitsfehler auftreten: Wenn der Benutzer 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 übrig bleibt.

Um auch diese Unschönheit zu beseitigen, werden wir im nächsten Abschnitt eine Technik kennen lernen, mit der wir die Anzahl der Threads sinnvoll begrenzen können.

17.4.2 Worker-Threads und Queues

In unseren bisherigen Programmen haben wir immer für jede Aufgabe einen neuen Thread gestartet, sodass es theoretisch beliebig viele Threads geben konnte. Wie am Ende des letzten Abschnitts angemerkt wurde, kann dies zu Geschwindigkeitsproblemen führen, wenn sehr viele Threads gleichzeitig laufen.

Dies lässt sich an einem Beispiel veranschaulichen: Wären wir ein Unternehmen, das für seine Kunden Zahlen daraufhin untersucht, ob sie Primzahlen sind,⁷ 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. Abbildung 17.4 verdeutlicht dieses Szenario.

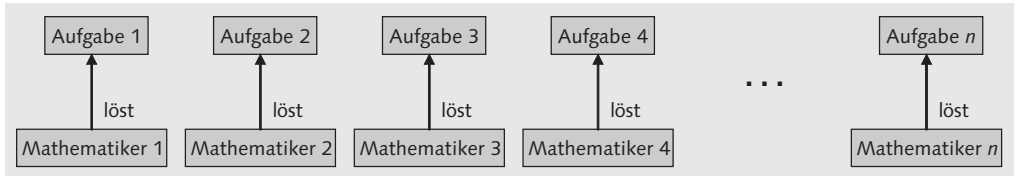


Abbildung 17.4 Für jede Aufgabe wird ein separater Thread gestartet.

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*, dt. »arbeiten«). Der Briefkasten wird *Queue* (dt. »Warteschlange«) genannt. Dieses Prinzip wird in Abbildung 17.5 veranschaulicht.

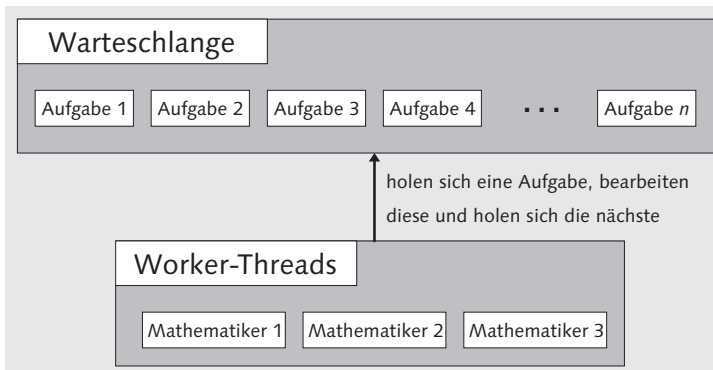


Abbildung 17.5 Eine Warteschlange wird von drei Threads bedient.

⁷ Ob dieses Geschäftsmodell besonders erfolgreich wäre, sei einmal dahingestellt.

Python hat ein eigenes Modul namens `queue`, um mit Warteschlangen zu arbeiten. Der Konstruktor von `queue` erwartet eine ganze Zahl als Parameter, die angibt, wie viele Elemente maximal in der Warteschlange stehen können. Ist der Parameter kleiner oder gleich 0, ist die Länge der Queue nicht begrenzt.

Queue-Instanzen verfügen im Wesentlichen über drei wichtige Methoden: `put`, `get` und `task_done`.

Mit der `put`-Methode 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 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 von allen Threads auf `Mathematiker.Briefkasten` zugegriffen wird, wundern Sie sich zu Recht: Normalerweise wäre es erforderlich, jedes Mal ein Lock-Objekt zu sperren und wieder zu entsperren. Allerdings nimmt uns das `queue`-Modul von Python diese lästige Arbeit ab, was die Arbeit mit Warteschlangen komfortabler macht.

Wir werden uns jetzt noch zwei Klassen zuwenden, die für sehr spezielle Zwecke im Zusammenhang mit Threads dienen.

17.4.3 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.

17.4.4 Barrieren definieren – `threading.Barrier`

Um mehrere Threads gegenseitig aufeinander warten zu lassen, dient die Klasse `threading.Barrier`.

Näheres zu dieser Klasse finden Sie in der Python-Dokumentation.

17.4.5 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 {0} Uhr".format(time.strftime("%H:%M:%S")))
>>> timer = threading.Timer(30, wecker, [time.strftime("%H:%M:%S")])
>>> timer.start()
```

(30 Sekunden später)

```
>>> RIIIIIIING!!!
Der Wecker wurde um 03:11:26 Uhr gestellt.
Es ist 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 zwei Sekunden rührt daher, dass zwischen dem Erstellen des *Timer*-Objekts und dem Aufrufen der *start*-Methode zwei Sekunden vergangen sind.

Nachdem die *start*-Methode aufgerufen wurde, kann der *Timer* außerdem mit der parameterlosen *cancel*-Methode wieder abgebrochen werden.

Kapitel 18

Datenspeicherung

»Gauß wusste alles.«
– Ulrich Kaiser

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.

18.1 Komprimierte Dateien lesen und schreiben – gzip

Mit dem Modul `gzip` der Standardbibliothek können Sie auf 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 Dateiojekt 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. Da diese Module dem eben vorgestellten Modul *gzip* sehr ähneln, verzichten wir an dieser Stelle auf eine ausführliche Beschreibung und verweisen auf die Python-Dokumentation.

Die folgende Tabelle gibt einen Überblick über alle Module, die komprimierte Daten verwalten:

Modul	Beschreibung
<i>zlib</i>	Eine Low-Level-Bibliothek, die direkten Zugriff auf die Funktionen der <i>zlib</i> ermöglicht. Mit ihr ist es unter anderem möglich, Strings zu komprimieren oder zu entpacken. Das Modul <i>gzip</i> greift intern auf das Modul <i>zlib</i> zurück.
<i>gzip</i>	Bietet komfortablen Zugriff auf Daten, die mit der <i>zlib</i> komprimiert wurden.
<i>bz2</i>	Bietet komfortablen Zugriff auf Daten, die mit dem <i>bzip2</i> -Algorithmus komprimiert wurden, und ermöglicht es, neue komprimierte Dateien zu erzeugen. Auch <i>bz2</i> implementiert ein Dateiobjekt, das genauso zu handhaben ist wie die Objekte, die die Built-in Function <i>open</i> zurückgibt. In der Regel ist die Kompression von <i>bzip2</i> der von <i>zlib</i> in puncto Kompressionsrate überlegen.
<i>zipfile</i>	Ermöglicht den Zugriff auf <i>ZIP</i> -Archive, wie sie beispielsweise von dem bekannten Programm <i>WinZip</i> erstellt werden. Auch die Manipulation und Erzeugung neuer Archive ist möglich. Das Modul <i>zipfile</i> ist sehr umfangreich und mächtig und in jedem Fall einen näheren Blick wert.

Tabelle 18.1 Übersicht über Pythons Kompressionsmodule

Modul	Beschreibung
<i>tarfile</i>	Implementiert Funktionen und Klassen, um die in der Unix-Welt weitverbreiteten tar-Archive zu lesen oder zu schreiben.

Tabelle 18.1 Übersicht über Pythons Kompressionsmodule (Forts.)

18.2 XML

Das Modul `xml` der Standardbibliothek erlaubt es, XML-Dateien einzulesen und zu schreiben. 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 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 wird eine kurze Einführung in XML gegeben. Dazu dient folgende einfache XML-Datei, die eine Möglichkeit aufzeigt, wie der Inhalt eines Python-Dictionaries dauerhaft abgespeichert werden kann:

```
<?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. Näheres zu Encodings erfahren Sie im Zusammenhang mit Strings auf S. 165.

Abgesehen von der XML-Deklaration besteht ein XML-Dokument aus sogenannten *Tags*. Ein Tag gibt es wie eine Klammer in einer öffnenden und einer schließenden Variante und stellt 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 die-

ser 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>
```

Zwischen einem öffnenden und dem entsprechenden schließenden Tag 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 *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 ein Schlüssel-Wert-Paar dar. Im Beispiel wird jedem `schluessel-` und `wert-`Tag ein Attribut `typ` verpasst, über das der Datentyp des Schlüssels bzw. des Wertes angegeben werden kann. Der Wert eines XML-Attributs ist stets in Anführungszeichen zu schreiben.

Zum Einlesen von XML-Dateien stellt Python, wie die meisten anderen Programmiersprachen oder XML-Bibliotheken auch, zwei *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 ist eine weitere Python-spezifische Herangehensweise an XML-Daten namens `ElementTree`.



Hinweis

Eine Besonderheit bei XML-Tags stellen *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. Ein XML-Parser behandelt ein körperloses Tag, als stünde `<tag attr="wert"></tag>` in der XML-Datei.

18.2.1 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 Baumstruktur aufbereitet. Jedes Tag wird durch eine Klasse repräsentiert, den *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 ist das sequentielle Einlesen der XML-Datei. Dieser Ansatz wird mit dem SAX-Parser verfolgt, der ab S. 565 besprochen wird.

Da die Datei zur Erzeugung des DOM stets vollständig eingelesen wird, ist die Verwendung von DOM für große Dateien 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>

```

Mit einem DOM-Parser werden die XML-Daten zu einem *Baum* aufbereitet. Ein Baum besteht aus einzelnen *Knoten*, die durch Klassen abgebildet werden. 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, verwandten Knoten. Auf diese Weise lässt sich der Baum vollständig durchlaufen und verarbeiten.

Die aus dem obigen Beispiel erzeugte Baumstruktur sieht folgendermaßen aus:

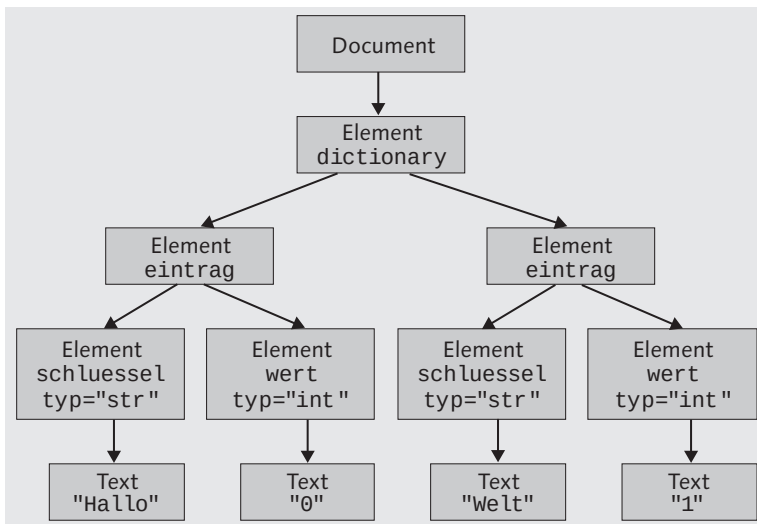


Abbildung 18.1 Vom DOM-Parser erzeugter Baum

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 das Tag `dictionary`. Diesem Knoten sind mehrere Instanzen der Klasse `Element` untergeordnet, die jeweils ein `eintrag`-Tag repräsentieren. Durch Attribute dieser Klasse können Informationen wie der Tag-Name, enthaltene XML-Attribute oder Ähnliches abgerufen werden.

In Abbildung 18.1 wurden aus Gründen der Übersichtlichkeit keine Geschwisterbeziehungen eingezeichnet.

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, also auf die Klassen `Document`, `Element` und `Text`, beschränken.

Beispiel

An dieser Stelle wird die Verwendung von DOM an einem einfachen Beispiel gezeigt. 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 wird der Quelltext des Beispielprogramms besprochen.


```
import xml.dom.minidom as dom

def knoten_auslesen(knoten):
    return eval("{}({})".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 einfache 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 das Ergebnis zurückgegeben.

Hinweis

Aus Gründen der Übersichtlichkeit wurden alle Konsistenzprüfungen weggelassen. 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 = {}
    with dom.parse(dateiname) as baum:
        if baum.firstChild.nodeName != "dictionary":
            return d
```

```

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

```

Die Aufgabe der Funktion `lade_dict` 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.² 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 wird über alle `eintrag`-Tags iteriert und das jeweilige Schlüssel-Wert-Paar ins Dictionary `d` eingefügt. 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 referenziert. 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 verworfen: Ihr `nodeName`-Wert ist `"#text"`. Zudem

² In diesem Beispiel haben wir die `with`-Anweisung verwendet.

Eine äquivalente Schreibweise ist: `baum = dom.parse(dateiname)`

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`- oder ein `wert`-Tag handelt, wird das Ergebnis des Funktionsaufrufs von `knoten_auslesen` dem Namen `schluessel` bzw. `wert` zugewiesen. Nachdem die innere Schleife durchlaufen ist, werden Schlüssel und Wert ins Dictionary `d` eingetragen.



Hinweis

Wir gehen in diesem Beispiel davon aus, dass die XML-Datei exakt unseren Ansprüchen entspricht. In einem wirklichen Programm sollten Sie grundsätzlich davon ausgehen, dass auch fehlerhafte Angaben vorkommen, und diese entsprechend behandeln. Auch der sorglose Umgang mit dem Attribut `typ` (direktes Übergeben nach `eval`) sollte in einem fertigen Programm so nicht vorkommen.

Dieses Beispiel bietet einen kurzen Überblick über die Verwendung des DOM-Baumes. 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. Die folgenden Tabellen liefern einen Überblick über die wichtigsten, von der Klasse `Node` bereitgestellten Attribute und Methoden und verweist, sofern nötig, auf eine detailliertere Erklärung. Zur Veranschaulichung der Verwandtschaftsbeziehungen von Knotenklassen untereinander folgt Abbildung 18.2 im Anschluss an die Tabellen.

Attribut	Beschreibung	Seite
<i>nodeType</i>	Gibt den Typ des Knotens an.	560
<i>parentNode</i>	Referenziert das Elternelement des Knotens oder <code>None</code> , wenn es dieses nicht gibt.	–
<i>previousSibling</i>	Referenziert das Geschwisterelement, das in der Rangfolge vor dem Knoten steht, oder <code>None</code> , wenn es kein solches Element gibt.	–

Tabelle 18.2 Attribute der Klasse `Node`

Attribut	Beschreibung	Seite
<i>nextSibling</i>	Referenziert das Geschwisterelement, das in der Rangfolge nach dem Knoten steht, oder <i>None</i> , wenn es kein solches Element gibt.	–
<i>firstChild</i>	Referenziert das erste Kindelement des Knotens oder <i>None</i> , wenn es keine untergeordneten Knoten gibt.	–
<i>lastChild</i>	Referenziert das letzte Kindelement des Knotens oder <i>None</i> , wenn es keine untergeordneten Knoten gibt.	–
<i>childNodes</i>	Referenziert eine Liste aller Kindelemente des Knotens.	–

Tabelle 18.2 Attribute der Klasse Node (Forts.)

Methoden	Beschreibung	Seite
<i>hasChildNodes()</i>	Gibt <i>True</i> zurück, wenn der Knoten über Kinder verfügt, andernfalls <i>False</i> .	–
<i>appendChild(newChild)</i>	Fügt die Node-Instanz <i>newChild</i> als Kindelement an das Ende der Liste aller Kinder des Knotens ein.	–
<i>insertBefore(newChild, refChild)</i>	Fügt die Node-Instanz <i>newChild</i> vor dem Kindelement <i>refChild</i> in die Liste der Kinder des Knotens ein.	–
<i>removeChild(oldChild)</i>	Löscht das Kindelement <i>oldChild</i> .	–
<i>replaceChild(newChild, oldChild)</i>	Ersetzt das Kindelement <i>oldChild</i> durch <i>newChild</i> .	–
<i>writexml(writer[, indent[, addindent[, new]]])</i>	Schreibt den Knoten und seine Kindelemente als XML in das geöffnete Dateiobjekt <i>writer</i> .	561

Tabelle 18.3 Methoden der Klasse Node

Methoden	Beschreibung	Seite
<code>toxml([encoding])</code>	Wie <code>writexml</code> , gibt die XML-Daten aber als String zurück. Wird <code>encoding</code> angegeben, so wird das Ergebnis als bytes-String unter Verwendung des angegebenen Encodings zurückgegeben.	–
<code>toprettyxml([indent[, newl[, encoding]]])</code>	Wie <code>toxml</code> , erlaubt aber eine individuelle Formatierung des Ergebnisses.	561

Tabelle 18.3 Methoden der Klasse Node (Forts.)

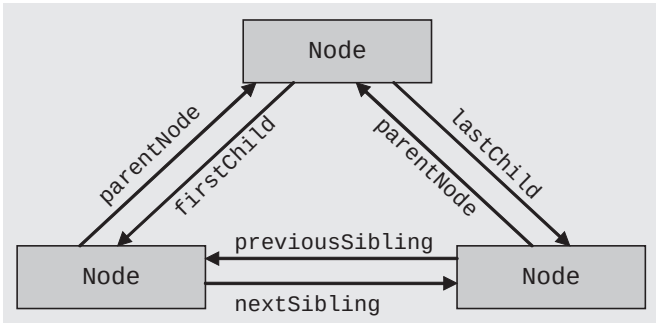


Abbildung 18.2 Verwandtschaftsbeziehungen dreier Knoten

nodeType

Dieses Attribut kennzeichnet den Typ des Knotens und 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 Document-Instanz.
<code>Node.ELEMENT_NODE</code>	Bei dem Knoten handelt es sich um eine Element-Instanz.
<code>Node.TEXT_NODE</code>	Bei dem Knoten handelt es sich um eine Text-Instanz.

Tabelle 18.4 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.

writexml(writer[, indent[, addindent[, newl]]], encoding)

Diese Methode schreibt den Knoten mitsamt all seinen Kindelementen als XML in das geöffnete Dateiojekt *writer*. Sie wird auch an die Klasse *Document* weitervererbt. 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.

toxml([encoding])

Diese Methode ist 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.

toprettyxml([indent[, newl]], encoding)

Diese Methode ist 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 werden die wichtigsten Methoden und Attribute der Klasse *Document* erläutert.

documentElement

Dieses Attribut referenziert die `Element`-Instanz des ersten Tags des XML-Dokuments. Ein wohlgeformtes XML-Dokument verfügt über genau ein Wurzel-Tag. Sollten mehrere solche *Toplevel-Tags* vorkommen, kann auf diese über ihre Geschwisterbeziehung zu `documentElement` zugegriffen werden.

createElement(tagName)

Diese Methode erzeugt einen neuen Elementknoten mit dem Tag-Namen *tagName*. Die Funktion gibt eine Instanz der Klasse `Element` zurück. Der Knoten wird zwar erzeugt, aber nicht automatisch in den Baum eingefügt. Dazu können beispielsweise die Methoden *insertBefore* oder *appendChild* der Klasse `Node` verwendet werden.

createTextNode(data)

Diese Methode 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.

getElementsByTagName(tagName)

Diese Methode 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`. In den folgenden Tabellen werden die wichtigsten Attribute und Methoden der Klasse `Element` erläutert.

Attribut	Beschreibung
<i>tagName</i>	Referenziert den Tag-Namen des Tags.

Tabelle 18.5 Attribute der Klasse `Element`

Methode	Beschreibung
<i>getElementsByTagName(tagName)</i>	Äquivalent der gleichnamigen Methode der Klasse <code>Document</code> . Diese Methode sucht nur nach untergeordneten Elementen mit dem Tag-Namen <i>tagName</i> .
<i>hasAttribute(name)</i>	Gibt <code>True</code> zurück, wenn das Tag ein Attribut mit dem Schlüssel <i>name</i> besitzt, andernfalls <code>False</code> .

Tabelle 18.6 Methoden der Klasse `Element`

Methode	Beschreibung
<i>getAttribute(name)</i>	Gibt den Wert des Attributs mit dem Schlüssel <i>name</i> zurück oder einen leeren String, falls kein solches Attribut existiert.
<i>getAttributeNode(attrname)</i>	Gibt den Attribut-Knoten des Attributs mit dem Schlüssel <i>attrname</i> zurück oder <i>None</i> , wenn kein solches Attribut existiert. Der Attribut-Knoten wird als <i>Attr</i> -Instanz zurückgegeben. ³
<i>removeAttribute(name)</i>	Löscht das Attribut mit dem Schlüssel <i>name</i> .
<i>removeAttributeNode(oldAttr)</i>	Löscht den Attribut-Knoten <i>oldAttr</i> aus dem DOM-Baum. Der Attribut-Knoten wird als <i>Attr</i> -Instanz übergeben. Wenn der Attribut-Knoten nicht im Baum vorhanden ist, wird eine <i>NotFoundErr</i> -Exception geworfen.
<i>setAttribute(name, value)</i>	Erzeugt ein neues Attribut mit dem Schlüssel <i>name</i> und dem Wert <i>value</i> oder überschreibt ein bereits bestehendes Attribut.
<i>setAttributeNode(newAttr)</i>	Fügt das durch die <i>Attr</i> -Instanz <i>newAttr</i> beschriebene Attribut hinzu.

Tabelle 18.6 Methoden der Klasse Element (Forts.)

Die Klasse Text

Die Klasse `Text` erbt von `Node` und fügt ein einziges Attribut hinzu:

data

Das Attribut *data* referenziert den String, den die `Text`-Instanz 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 kennen gelernt, die den Baum modifizieren. Der nächste logische Schritt ist es, den modifizierten Baum wieder als XML-Datei abzuspeichern.

³ Eine *Attr*-Instanz verfügt im Wesentlichen über die beiden Attribute *name* und *value*, die den Schlüssel bzw. den Wert des XML-Attributs angeben.

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 einlesen kann.

Das Schreiben der XML-Datei erfolgt durch eine Funktion `schreibe_dict`, der das zu schreibende Dictionary `d` und der Dateiname der Ausgabedatei als Parameter übergeben wird. Der Quelltext des Beispielprogramms sieht folgendermaßen aus:

```
import xml.dom.minidom as dom

def erstelle_eintrag(schluessel, wert):
    tag_eintrag = dom.Element("eintrag")
    tag_schluessel = dom.Element("schluessel")
    tag_wert = dom.Element("wert")

    tag_schluessel.setAttribute("typ", type(schluessel).__name__)
    tag_wert.setAttribute("typ", type(wert).__name__)

    text = dom.Text()
    text.data = str(schluessel)
    tag_schluessel.appendChild(text)

    text = dom.Text()
    text.data = str(wert)
    tag_wert.appendChild(text)

    tag_eintrag.appendChild(tag_schluessel)
    tag_eintrag.appendChild(tag_wert)
    return tag_eintrag
```

Auch hier wird, ähnlich wie beim vorherigen Beispiel, zuerst eine Hilfsfunktion angelegt, die einen Schlüssel und einen Wert übergeben bekommt und daraus eine Element-Instanz erzeugt, die das entsprechende `eintrag`-Tag repräsentiert. Die Funktion bedarf keiner weiteren Erläuterung. Einzig erwähnenswert ist folgender Ausdruck:

```
type(schluessel).__name__
```

Dieser Ausdruck ermittelt den Namen des Datentyps der von `schluessel` referenzieren Instanz. Das wäre beispielsweise `"int"` für ganze Zahlen oder `"str"` für Strings. Jetzt folgt die Hauptfunktion des Beispielprogramms:

```
def schreibe_dict(d, dateiname):
    baum = dom.Document()
    tag_dict = dom.Element("dictionary")
```

```

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.

Dann 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`-Tag 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 mit der Funktion `lade_dict` des vorherigen Beispiels zusammen. Das bedeutet, dass eine von `schreibe_dict` erzeugte XML-Datei von `lade_dict` wieder eingelesen werden kann.

18.2.2 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 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 (dt. »ereignisgesteuert«). Das bedeutet, dass der Programmierer beim Erstellen des Readers verschiedene *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 kann beispielsweise das Auffinden eines öffnenden Tags sein.

Der SAX-Reader stellt also nur die Infrastruktur zum Einlesen der XML-Datei bereit. Ob und in welcher Form die gelesenen Daten aufbereitet werden, entscheidet allein der Programmierer.

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 das folgende 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.

18

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 wird der Tag-Name als Parameter übergeben. Im Falle eines schließenden `eintrag-`Tags fügen wir das eingelesene Schlüssel-Wert-Paar, das aus `self.schluessel` und `self.wert` besteht, in das Dictionary `self.ergebnis` ein. Wenn ein schließendes `schluessel-` oder `wert-`Tag gefunden wurde, wird das Attribut `self.aktiv` wieder auf `None` gesetzt, sodass keine weiteren Zeichen mehr verarbeitet werden.

Die letzte Methode `characters` wird aufgerufen, wenn Zeichen eingelesen wurden, die nicht zu einem Tag gehören. Der SAX-Reader garantiert nicht, dass eine zusammenhängende Zeichenfolge auch in einem einzelnen Aufruf von `characters` resultiert. Je nachdem, welchen Namen das zuletzt eingelesene Tag hatte, werden die gelesenen Zeichen an `self.schluessel` oder `self.wert` angehängt.

Schlussendlich fehlt noch die Hauptfunktion `lade_dict` des Beispielprogramms, in der der SAX-Parser erzeugt und gestartet wird.

```
def lade_dict(dateiname):
    handler = DictHandler()
    parser = sax.make_parser()
    parser.setContentHandler(handler)
    parser.parse(dateiname)
    return handler.ergebnis
```

Im Funktionskörper wird die Klasse `DictHandler` instanziiert und durch die Funktion `make_parser` des Moduls `xml.sax` ein SAX-Parser erzeugt. Dann wird die Methode `setContentHandler` des Parsers aufgerufen, um die `DictHandler`-Instanz mit den enthaltenen Callback-Handleern 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. Die folgende Tabelle listet die wichtigsten Callback-Handler auf, die in einer von `ContentHandler` abgeleiteten Klasse überschrieben werden können.

Methoden	Beschreibung	Seite
<i>startDocument()</i>	Wird einmalig aufgerufen, wenn der SAX-Parser damit beginnt, ein XML-Dokument einzulesen.	–
<i>endDocument()</i>	Wird einmalig aufgerufen, wenn der SAX-Parser ein XML-Dokument vollständig eingelesen hat.	–
<i>startElement(name, attrs)</i>	Wird aufgerufen, wenn ein öffnendes Tag eingelesen wurde.	569
<i>endElement(name)</i>	Wird aufgerufen, wenn ein schließendes Tag mit dem Namen <i>name</i> eingelesen wurde.	–
<i>characters(content)</i>	Wird aufgerufen, wenn ein Textabschnitt eingelesen wurde.	569
<i>ignorableWhitespace(whitespace)</i>	Wird aufgerufen, wenn Whitespace-Zeichen eingelesen wurden.	569

Tabelle 18.7 Methoden der Klasse ContentHandler

startElement(name, attrs)

Diese Methode wird aufgerufen, wenn ein öffnendes Tag eingelesen wurde. Sie 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.

characters(content)

Diese Methode wird aufgerufen, wenn ein Textabschnitt eingelesen wurde. Über den Parameter *content* greifen Sie auf den gelesenen Text zu.

Es steht dem SAX-Parser frei, den gesamten Textabschnitt in einem Event zu verarbeiten oder auf mehrere Events aufzuteilen.

ignorableWhitespace(whitespace)

Diese Methode wird aufgerufen, wenn Whitespace-Zeichen eingelesen wurden. Auf die eingelesenen Zeichen kann über den Parameter *whitespaces* zugegriffen werden. Diese könnten von Bedeutung sein, sind jedoch in den meisten Fällen allein aus Gründen der Formatierung vorhanden und können ignoriert werden. Der SAX-Par-

ser kann auch hier eine Folge von mehreren Whitespace-Zeichen auf mehrere Events aufteilen.

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 gibt 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`.

18.2.3 `ElementTree`

Seit Python 2.5 ist im Modul `xml.etree.ElementTree` der Standardbibliothek der Datentyp `ElementTree` enthalten, der in Konkurrenz zu DOM steht. Der Datentyp `ElementTree` speichert ein XML-Dokument und stellt 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 besser 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])`

Diese Methode 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 `XMLParser`.

ElementTree.tostring(element[, encoding])

Diese Methode 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 wurde. Eine Instanz der Klasse `ElementTree` stellt die folgenden Methoden bereit:

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*.

getroot()

Diese Methode gibt die `Element`-Instanz des Wurzelements zurück.

write(file[, encoding[, xml_declaration]])

Diese Methode speichert den vollständigen Baum als XML-Datei *file* ab. Dabei können Sie für *file* sowohl einen Dateinamen als auch ein zum Schreiben geöffnetes Dateiobjekt übergeben. Über den optionalen Parameter *encoding* legen Sie das Encoding der geschriebenen Daten fest.

Der boolesche Parameter *xml_declaration* legt fest, ob eine XML-Deklaration ausgegeben werden soll. Standardmäßig wird nur dann eine Deklaration ausgegeben, wenn das verwendete Encoding nicht US-ASCII oder UTF-8 ist.

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. Darüber hinaus kann mittels *list(e)* eine Liste der Kindelemente der `Element`-Instanz *e* erzeugt werden.

Die folgenden Tabellen listen die wichtigsten Attribute und Methoden auf, die die Klasse `Element` bereitstellt.

Attribut	Beschreibung	Seite
<i>attrib</i>	Referenziert ein Dictionary, das alle im Element enthaltenen XML-Attribute als Schlüssel-Wert-Paare enthält.	–
<i>tag</i>	Referenziert den Tag-Namen des Elements.	–
<i>tail</i>	Referenziert den Text, der in der XML-Datei zwischen dem schließenden Tag des Elements und dem nächsten öffnenden oder schließenden Tag steht.	–
<i>text</i>	Referenziert den Text, der in der XML-Datei zwischen dem öffnenden Tag des Elements und dem nächsten öffnenden oder schließenden Tag steht.	–

Tabelle 18.8 Attribute der Klasse Element

Methode	Beschreibung	Seite
<i>append(subelement)</i>	Erweitert die Kinder des Elements um <i>subelement</i> .	–
<i>clear()</i>	Löscht alle Unterelemente und Attribute sowie den Text des Elements.	–
<i>extend(subelements)</i>	Wie <i>append</i> , erwartet aber ein iterierbares Objekt von <i>Element</i> -Instanzen.	–
<i>find(path)</i>	Sucht nach dem ersten direkten Kindelement, auf das das Pattern <i>path</i> passt.	573
<i>findall(path)</i>	Wie <i>find</i> , gibt aber eine Liste aller passenden <i>Element</i> -Instanzen zurück statt nur des zuerst gefundenen Elements.	–
<i>findtext(path[, default])</i>	Wie <i>find</i> , gibt aber den im gefundenen Element enthaltenen Text zurück.	574
<i>get(key[, default])</i>	Gibt den Wert des Attributs <i>key</i> des Elements zurück. Wenn kein Attribut mit dem Namen <i>key</i> vorhanden ist, wird <i>default</i> zurückgegeben. Der Parameter <i>default</i> ist mit <i>None</i> vorbelegt.	–

Tabelle 18.9 Methoden der Klasse Element

Methode	Beschreibung	Seite
<i>iter([tag])</i>	Wie die Methode <i>getiterator</i> der Klasse <code>ElementTree</code> , betrachtet aber nur den Unterbaum, dessen Wurzel das aktuelle Element ist.	–
<i>set(key, value)</i>	Legt ein neues Attribut mit dem Namen <i>key</i> und dem Wert <i>value</i> an. Ein bereits vorhandenes Attribut gleichen Namens wird überschrieben.	–

Tabelle 18.9 Methoden der Klasse `Element` (Forts.)

find(path)

Diese Methode gibt das erste direkte Kindelement des Elements mit dem Tag-Namen *path* zurück. Statt eines einzelnen Tag-Namens können Sie für *path*, wie der Name bereits andeutet, auch einen Pfad übergeben. Betrachten wir dazu die folgende XML-Datei:

```
<A>
  <B>
    <D>Hallo</D>
  </B>
  <C>
    <E>
      <F>Welt</F>
    </E>
  </C>
</A>
```

Diese XML-Datei laden wir in eine `ElementTree`-Instanz und führen dann auf dem Element A einige Suchoperationen aus.

```
>>> import xml.etree.ElementTree
>>> et = xml.etree.ElementTree.parse("test.xml")
>>> e = et.getroot()
```

Zunächst suchen wir mithilfe des Wildcard-Zeichens `*` nach einem beliebigen Tag-Namen. Das erste passende Element ist das Element B, welches zurückgegeben wird:

```
>>> e.find("*")
<Element 'B' at 0x7f15ce584f10>
```

Die Methode `find` durchsucht nur die direkten Kindelemente des Elements, für das sie aufgerufen wird. Daher führt eine Suche nach dem Tag-Namen E zu keinem Ergebnis:

```
>>> e.find("E")
>>>
```

Um nach Elementen zu suchen, die tiefer in der XML-Hierarchie liegen, muss für *path* ein Pfad übergeben werden. Dieser kann auch Wildcards enthalten.

```
>>> e.find("C/*/F")
<Element 'F' at 0x7f15ce58a050>
```

findtext(path[, default])

Wenn für *path* ein leerer String übergeben wird, gibt die Methode *findtext* den Text als String zurück, den das Element 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.

Beispiel

Als Beispiel für die Verwendung des Datentyps `ElementTree` wird hier das Beispielprogramm der vorherigen Abschnitte an diesen Datentyp angepasst. 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("{}('{}')".format(typ, element.text))

def lade_dict(dateiname):
    d = {}
```

```

baum = ElementTree.parse(dateiname)
tag_dict = baum.getroot()
for eintrag in tag_dict:
    tag_schlüssel = eintrag.find("schlüssel")
    tag_wert = eintrag.find("wert")
    d[lese_text(tag_schlüssel)] = lese_text(tag_wert)
return d

```

Zunächst wird die Funktion `lese_text` implementiert, die aus der `Element`-Instanz eines `schlüssel`- 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 Beispielsprogramms `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 `schlüssel` und `wert` gesucht und den Referenzen `tag_schlüssel` und `tag_wert` zugewiesen. Am Ende des Schleifenkörpers werden die `Element`-Instanzen der jeweiligen `schlüssel`- 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 und werden uns im nächsten Abschnitt um das Thema Datenbanken kümmern.

18.3 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 müssten Aufgaben wie das Lesen, Schreiben oder Aktualisieren von Daten, die in vielen Programmen gebraucht werden, immer wieder neu implementiert werden.

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.

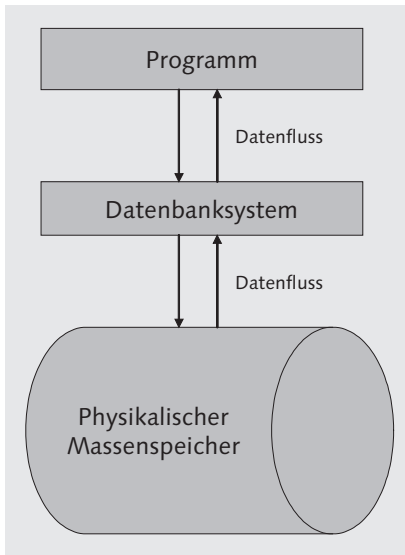


Abbildung 18.3 Die Datenbankschnittstelle

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.⁴ 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 den meisten relationalen Datenbanksystemen unterstützt. Dabei ist zu beachten, dass die Systeme häufig nur Teilmengen der Sprache implementieren und sie teilweise geringfügig abändern. Aus diesem Grund werden wir Sie hier in die SQL-Variante einführen, die von SQLite, der Standarddatenbank in Python, genutzt wird.

⁴ Das 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 in Form einer Tabelle.



Hinweis

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.

18.3.1 Beispieldatenbank

Bevor wir uns aber mit der Abfragesprache SQL selbst beschäftigen, werden wir eine kleine Beispieldatenbank erarbeiten und uns überlegen, welche Operationen man überhaupt ausführen kann. 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

Tabelle 18.10 Tabelle »Lager« für den Lagerbestand

Fachnummer	Seriennummer	Komponente	Lieferant	Reserviert
10	06198823	Netzteil Typ 3	FC	0
25	11198703	LED-Lüfter	FC	57
26	19880105	Festplatte 128 GB	LPE	12

Tabelle 18.10 Tabelle »Lager« für den Lagerbestand (Forts.)

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 18.11 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 18.12 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 möchten wir beispielsweise einfach nur eine Liste aller Kunden oder Lieferanten anfordern oder uns informieren, 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 Datensätze nach bestimmten Kriterien *ausgewählt* und an das aufrufende Benutzerprogramm zurückgegeben.

Nach dieser theoretischen Vorbereitung werden wir uns der Implementation des Beispiels in einer SQLite-Datenbank zuwenden.

18.3.2 Pythons eingebaute Datenbank – sqlite3

SQLite ist ein einfaches Datenbanksystem, das im Gegensatz zu anderen Systemen ohne separaten Datenbankserver auskommt und die gesamte Datenbank in einer einzigen Datei abspeichert. Die Bibliothek ist nur einige hundert Kilobyte groß, wodurch sich das System besonders für eingebettete Systeme eignet.

Trotzdem unterstützt SQLite einen Großteil der Sprache SQL und hat sich in der Praxis in puncto Skalierbarkeit und Geschwindigkeit bewährt.⁵

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 Dateierweiterung 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.

⁵ Beispielsweise setzen die Handy-Betriebssysteme Symbian und Android auf SQLite, und der populäre Browser Firefox verwaltet damit Lesezeichen und Cookies.

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 Cursors 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 unser erstes SQL-Statement an die Datenbank schicken, um unsere Tabellen anzulegen. Für das Anlegen der Tabelle »Lager« sieht das SQL-Statement folgendermaßen aus:

```
CREATE TABLE lager (
    fachnummer INTEGER, seriennummer INTEGER, komponente TEXT,
    lieferant TEXT, reserviert INTEGER
)
```



Hinweis

Alle großgeschriebenen Wörter sind Bestandteile der Sprache SQL. Allerdings unterscheidet SQL nicht zwischen Groß- und Kleinschreibung, weshalb wir auch alles hätten kleinschreiben können. Wegen der besseren Lesbarkeit werden wir SQL-Schlüsselwörter immer komplett groß- und von uns vergebene Namen durchgängig kleinschreiben.

Die Zeichenketten `INTEGER` und `TEXT` hinter den Spaltennamen geben den Datentyp an, der in den Spalten gespeichert werden soll. Sinnvollerweise werden die Spalten `fachnummer`, `seriennummer` und `reserviert` als Ganzzahlen und die Spalten `komponente` und `lieferant` als Zeichenketten definiert. SQLite kennt mehrere solcher Datentypen für die Spalten der Datenbank.

Standardmäßig werden Python-Datentypen beim Schreiben in eine SQLite-Datenbank automatisch in entsprechende SQLite-Datentypen umgewandelt. Die folgende Tabelle zeigt das Umwandlungsschema von Python- zu SQLite-Datentypen.

Python-Datentyp (Quellentyp)	SQLite-Datentyp (Zieltyp)
None	NULL
int	INTEGER
float	REAL
bytes (UTF8-Kodiert)	TEXT
str	TEXT
bytes	BLOB

Tabelle 18.13 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 auf Seite 592 im Abschnitt »Adapter und Konvertierer« 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 mit der `execute`-Methode 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:⁶

```
connection.commit()
```

In der Regel werden die Daten, die wir in die Datenbank einfügen wollen, nicht schon vor dem Programmablauf bekannt sein und deshalb auch nicht in Form von String-Konstanten im Quellcode stehen. Stattdessen werden es Benutzereingaben oder Berechnungsergebnisse sein, die wir dann als Python-Instanzen im Speicher haben. Auf den ersten Blick scheint für solche Fälle 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)'
```

⁶ 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. Um automatisch mehrere Anweisungen als einen atomaren Block zu kennzeichnen, können Sie das `connection`-Objekt als Transaktionsmanager zusammen mit der `with`-Anweisung verwenden. Mehr zur `with`-Anweisung finden Sie auf Seite 368.

Diese auf den ersten Blick 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, sodass der Versuch, sie auszuführen, zu einem Fehler führen und damit unser Programm zum Absturz bringen würde. Durch den außerdem enthaltenen Text `"Hier kann Schadcode stehen"` 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.⁷

Hinweis

Verwenden Sie deshalb niemals die String-Formatierung zur Übergabe von Parametern in SQL-Abfragen!



Um sichere Parameterübergaben durchzuführen, schreibt man in den Query-String an die Stelle, an der der Parameter stehen soll, ein Fragezeichen und übergibt der `execute`-Methode ein Tupel mit den entsprechenden Werten als zweiten Parameter:

```
werte = ("DR", "Danger Electronics",
        "666"); Hier kann Schadcode stehen")
sql = "INSERT INTO lieferanten VALUES (?, ?, ?)"
cursor.execute(sql, werte)
```

In diesem Fall kümmert sich SQLite darum, dass die übergebenen Werte korrekt umgewandelt werden und es zu keinen Sicherheitslücken durch böswillige Parameter kommen kann.

Analog zur String-Formatierung gibt es auch hier die Möglichkeit, den übergebenen Parametern Namen zu geben und statt der `tuple`-Instanz mit einem Dictionary zu arbeiten. Dazu schreiben Sie im Query-String statt des Fragezeichens einen Doppel-

⁷ Man nennt diese Form des Angriffs auf verwundbare Programme auch *SQL Injection*.

punkt, 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)
```



Hinweis

Im Gegensatz zu früheren Versionen von Python nimmt Ihnen Python 3 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.⁸

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)

kunden = ((12, "Heinz Elhurg",
            "Turnhallenstr. 1, 3763 Sporthausen"),
          (57, "Markus Altbirt",
```

⁸ In Python-Versionen vor 3.0 mussten Sie diesen String als eine `unicode`-Instanz übergeben und auch im Quellcode explizit als solche kennzeichnen.

```

        "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. Ein einfaches `SELECT`-Statement ist 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 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 mit `cursor.fetchall` immer alle Ergebnisse der Abfrage auf einmal 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` den Wert `None` zurück. Damit lassen sich auch große Datenmengen speichereffizient auslesen, auch wenn unser Beispiel mangels einer großen Datenbank nur drei Zeilen ermittelt:


```
>>> cursor.execute("SELECT * FROM kunden")
>>> row = cursor.fetchone()
>>> while row:
    print(row)
    row = cursor.fetchone()
(12, '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 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 (Quellentyp)	Python-Datentyp (Zieltyp)
NULL	None
INTEGER	int
REAL	float
TEXT	str
BLOB	bytes

Tabelle 18.14 Typumwandlung beim Lesen von SQLite-Datenbanken

Im Wesentlichen wirft diese Tabelle 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 Daten beim Auslesen aus der Datenbank unseren Vorstellungen entsprechend anzupassen?

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 Function `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`-Objekts 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 Datensätze. 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 Althert'},
 {'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. Das Attribut `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:⁹

```
>>> 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 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
```

Hinweis

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. Mehr finden Sie auf Seite 239.



18

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':
```

⁹ Die anderen sechs Einträge existieren nur aus Kompatibilitätsgründen zum Python DB API und sind immer mit dem Wert `None` belegt.

```
'Markus Albert'},
{'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` 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 Funktion unserem Beispiel vorziehen und nur dann eine eigene `row_factory` implementieren, wenn Sie etwas ganz anderes erreichen möchten. Beispielsweise könnten Sie eine `row_factory`-Funktion schreiben, die einen Datensatz in ein Objekt einer selbstgeschriebenen Klasse übersetzt.

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 unbeschränkten Länge eignen sich die `TEXT`-Spalten am besten, um beliebige Daten aufzunehmen, weshalb wir uns im Folgenden auf sie beschränken werden.

Bei der String-Kodierung haben wir `str`-Instanzen mittels ihrer `encode`-Methode in gleichwertige `bytes`-Instanzen umgeformt und die ursprünglichen Unicode-Daten mithilfe der `decode`-Methode wiederherstellen können. Analog dazu betrachten wir nun Operationen, um *beliebige* Datentypen erst in Strings zu transformieren und anschließend die Ursprungsdaten wieder aus dem String zu extrahieren. Dabei geht es uns darum, die generierten Strings in einer Datenbank zu speichern und später wieder auszulesen.

Das Umwandeln von beliebigen Datentypen in einen String wird *Adaption* genannt, und die Rückgewinnung der Daten aus diesem String heißt *Konvertierung*. Abbildung 18.4 veranschaulicht diesen Zusammenhang am Beispiel der Klasse `Kreis`, die als Attribute die Koordinaten des Kreismittelpunktes M_x und M_y sowie die Länge des Radius R besitzt:

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
```

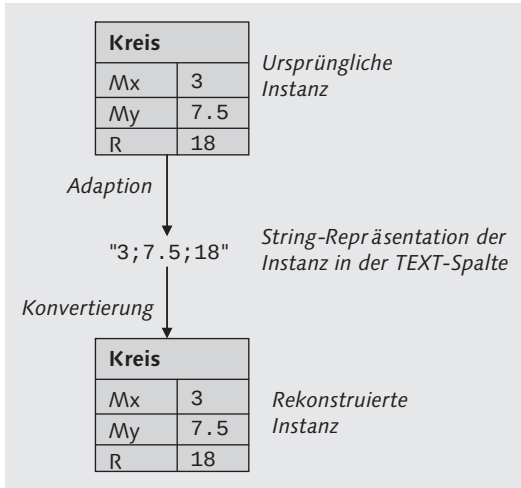


Abbildung 18.4 Schema der Adaption und Konvertierung

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))
```

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(bytesstring):
        mx, my, r = bytesstring.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 unpraktisch 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.

18.4 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, sodass es später wieder durch das sogenannte *Deserialisieren* rekonstruiert werden kann.

Besonders für die dauerhafte Speicherung von Daten in Dateien ist `pickle` 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 Dictionarys (dict), solange alle ihre Elemente auch von pickle serialisiert werden können
- ▶ globale Funktionen
- ▶ Built-in Functions
- ▶ globale Klassen
- ▶ Klasseninstanzen, deren Attribute serialisiert werden können

Bei Klassen und Funktionen 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. Das Protokoll 0 ist abwärtskompatibel mit älteren Python-Versionen.
1	Dieses Protokoll erzeugt einen Binärstring, der die Daten im Vergleich zur ASCII-Variante platzsparender speichert. Auch das Protokoll 1 ist abwärtskompatibel mit älteren Python-Versionen.
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 18.15 Die pickle-Protokolle

Das Modul `pickle` bietet seine Funktionalität über zwei Schnittstellen an: eine 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 Schnittstelle über `dump` und `load`

`pickle.dump(obj, file[, protocol])`

Diese Funktion schreibt die Serialisierung von *obj* in das Dateiojekt *file*. Das übergebene Dateiojekt 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 Dateiojekt im binären Schreibmodus geöffnet worden sein.

```
>>> f = open("pickle-test.dat", "bw")
>>> pickle.dump([1, 2, 3], f)
```

Für *file* können Sie neben echten Dateiojekten jedes Objekt übergeben, das eine `write`-Methode mit einem String-Parameter implementiert, zum Beispiel `StringIO`-Instanzen.

`pickle.load(file)`

Diese Funktion lädt, ausgehend von der aktuellen Leseposition des Dateiojekts *file*, das nächste serialisierte Objekt. 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])`

Diese Funktion 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\x03]q\x00(K\x01K\x02K\x03e.'
```

pickle.loads(string)

Diese Funktion 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]
```



Hinweis

Während ersteres vollständig in Python geschrieben und daher auf allen Plattformen verfügbar war, handelte es sich bei `cPickle` um eine optimierte Implementation in der Sprache C, die nicht auf allen Plattformen benutzt werden konnte, dafür aber höhere Geschwindigkeit bot.

Ab Python 3 brauchen Sie sich nicht mehr darum zu kümmern, ob das optimierte Modul verfügbar ist oder nicht, da das in diesem Kapitel beschriebene Modul `pickle` automatisch auf eine vorhandene C-Implementation zurückgreift oder anderenfalls die Standardimplementation verwendet.

In Python-Versionen vor Python 3.0 gab es zwei Module, mit denen Daten im `pickle`-Dateiformat gespeichert werden konnten: `pickle` und `cPickle`.

18.5 Das Tabellenformat CSV – csv

Ein weit 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
Heinrich,Huhn,19.07.1980,Berlin,Braun
Rudolf,Geier,19.09.1990,Dortmund,Braun
Haken,Habicht,14.04.1959,Hamburg,Dunkelblond
Edith,Falke,13.09.1987,Köln,Schwarz
Rüdiger,Amsel,25.03.1988,München,Hellrot
```

Die erste Zeile enthält die jeweiligen Spaltenköpfe, und alle folgenden Zeilen enthalten die eigentlichen Datensätze.

Leider existiert kein Standard für CSV-Dateien, sodass sich beispielsweise das Trennzeichen von Programm zu Programm unterscheiden kann. Dieser Umstand erschwert es, CSV-Dateien von verschiedenen Quellen zu lesen, da immer auf das besondere Format der exportierenden Anwendung eingegangen werden muss.

Um trotzdem mit CSV-Dateien der verschiedensten Formate umgehen zu können, stellt Python das Modul `csv` zur Verfügung. Das `csv`-Modul implementiert `reader`-

und `writer`-Klassen, die den Lese- bzw. Schreibzugriff auf CSV-Daten kapseln. Mithilfe sogenannter *Dialekte* kann dabei das Format der Datei angegeben werden. Standardmäßig gibt es vordefinierte Dialekte für die CSV-Dateien, die von Microsoft Excel generiert werden. Außerdem stellt das Modul eine Klasse namens `Sniffer` (dt. »Schnüffler«) bereit, die den Dialekt einer Datei erraten kann.

Eine Liste aller definierten Dialekte erhalten Sie mit `csv.list_dialects`:

```
>>> import csv
>>> csv.list_dialects()
['excel-tab', 'excel']
```

reader-Objekte – Daten aus einer CSV-Datei lesen

Mithilfe von `reader`-Objekten können CSV-Dateien gelesen werden. Der Konstruktor sieht dabei folgendermaßen aus:

`csv.reader(csvfile[, dialect[, **fmtparam]])`

Der Parameter `csvfile` muss eine Referenz auf ein für den Lesezugriff geöffnetes Dateiobjekt sein, aus dem die Daten gelesen werden sollen.

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 auf Seite 602 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=";")
```

Eine Übersicht über Dialekte und mögliche Werte für `fmtparam` finden Sie auf Seite 602 und in der Tabelle auf Seite 603.

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' ]
[ 'Heinrich', 'Huhn', '19.07.1980', 'Berlin', 'Braun' ]
[ 'Rudolf', 'Geier', '19.09.1990', 'Dortmund', 'Braun' ]
[ 'Haken', 'Habicht', '14.04.1959', 'Hamburg', 'Dunkelblond' ]
[ 'Edith', 'Falke', '13.09.1987', 'Köln', 'Schwarz' ]
[ 'Rüdiger', 'Amsel', '25.03.1988', 'München', '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 beispielhaft ausgeben:

```
>>> reader = csv.DictReader(open("namen.csv"))
>>> for row in reader:
    print(row)
{'nachname': 'Huhn', 'geburtsdatum': '19.07.1980', 'wohntort': 'Berlin',
'vorname': 'Heinrich', 'haarfarbe': 'Braun'}
{'nachname': 'Geier', 'geburtsdatum': '19.09.1990', 'wohntort': 'Dortmund',
'vorname': 'Rudolf', 'haarfarbe': 'Braun'}
```

18

writer-Objekte – Daten in CSV-Dateien schreiben

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.writer(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", "w"))
>>> 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 erste Zeile mit `writerow` ist notwendig, um die Spaltenköpfe zu schreiben, da dies nicht automatisch geschieht.

Dialect-Objekte – Eigene Dialekte verwenden

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
<i>delimiter</i>	Trennzeichen zwischen den Spaltenwerten. Der Standardwert ist das Komma ", ".
<i>quotechar</i>	Ein Zeichen, um Felder zu umschließen, die besondere Zeichen wie das Trennzeichen oder den Zeilenumbruch enthalten. Der Standardwert sind die doppelten Anführungszeichen '" '.
<i>doublequote</i>	Ein boolescher Wert, der angibt, wie das für <i>quotechar</i> angegebene Zeichen innerhalb von Feldern selbst maskiert werden soll. Hat <i>doublequote</i> den Wert <code>True</code> , so wird <i>quotechar</i> zweimal hintereinander eingefügt. Ist der Wert von <i>doublequote</i> <code>False</code> , wird stattdessen das für <i>escapechar</i> angegebene Zeichen vor <i>quotechar</i> geschrieben. Standardmäßig hat <i>doublequote</i> den Wert <code>True</code> .
<i>escapechar</i>	Ein Zeichen, das benutzt wird, um das Trennzeichen innerhalb von Spaltenwerten zu maskieren, sofern <i>quoting</i> den Wert <code>QUOTE_NONE</code> hat. Bei einem <i>doublequote</i> -Wert von <code>False</code> wird <i>escapechar</i> außerdem für die Maskierung <i>quotechar</i> verwendet. Standardmäßig ist die Maskierung deaktiviert, und <i>escapechar</i> hat den Wert <code>None</code> .
<i>lineterminator</i>	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 <i>lineterminator</i> -Einstellung unbeeinflusst und verwenden immer <code>"\r"</code> , <code>"\n"</code> oder die Kombination aus beiden als Zeilentrennzeichen.
<i>quoting</i>	Gibt an, ob und wann Spaltenwerte mit <i>quotechar</i> umschlossen werden sollen. Gültige Werte sind: <code>QUOTE_ALL</code> : Alle Spaltenwerte werden umschlossen.

Tabelle 18.16 Schlüsselwortparameter für `register_dialect`

Name	Bedeutung
<i>quoting</i>	QUOTE_MINIMAL: Nur die Felder mit speziellen Zeichen wie Zeilenvorschüben oder dem Trennzeichen für Spaltenwerte werden umschlossen. QUOTE_NONNUMERIC: Beim Schreiben werden alle nicht-numerischen Felder von <i>quotechar</i> umschlossen. Beim Lesen werden alle nicht umschlossenen Felder automatisch nach <i>float</i> konvertiert. QUOTE_NONE: Keine Umschließung mit <i>quotechar</i> wird vorgenommen. Standardmäßig ist <i>quoting</i> auf <i>QUOTE_MINIMAL</i> eingestellt.
<i>skipinitialspace</i>	Ein boolescher Wert, der angibt, wie mit führenden Whitespaces in einem Spaltenwert verfahren werden soll. Eine Einstellung auf <i>True</i> bewirkt, dass alle führenden Whitespaces ignoriert werden; bei einem Wert von <i>False</i> wird der komplette Spalteninhalt gelesen und zurückgegeben. Der Standardwert ist <i>False</i>.

Tabelle 18.16 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.

18.6 Temporäre Dateien – `tempfile`

Wenn Ihre Programme 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. Allerdings eignen sich temporäre Dateien nicht dazu, Daten dauerhaft zu speichern.

Für den komfortablen Umgang mit temporären Dateien stellt Python das Modul `tempfile` bereit.

Die wichtigste Funktion dieses Moduls ist `TemporaryFile`, die ein geöffnetes Dateiojekt 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 Dateiojekt 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 Dateinamen 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 Funktion `open` (nachzulesen in Kapitel 8 »Dateien«, ab Seite 197). 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` 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 im obigen 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.TemporaryDirectory([suffix="", prefix='tmp', dir=None])`

Mithilfe von `tempfile.TemporaryDirectory` ist es 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.TemporaryDirectory` ist analog zu `tempfile.TemporaryFile` zu verwenden.

Als Rückgabewert erhalten Sie den absoluten Pfadnamen des temporären Ordners:

```
>>> tempfile.mkdtemp()
'/tmp/tmpFvqxTh'
```



Hinweis

Die Methode `tempfile.TemporaryDirectory` wurde mit Python 3.2 eingeführt. In früheren Versionen können Sie stattdessen die schnittstellenidentische Methode `tempfile.mkdtemp` verwenden.

Kapitel 19

Netzwerkcommunication

»Alle reden von Kommunikation, aber die wenigsten haben sich etwas mitzuteilen.«
 – Hans Magnus Enzensberger

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 Netzwerkcommunication in mehrere sogenannte *Protokollebenen* (engl. *layer*) aufteilen. Abbildung 19.1 zeigt eine stark vereinfachte Version des *OSI-Schichtenmodells*¹, das die Hierarchie der verschiedenen Protokollebenen veranschaulicht.

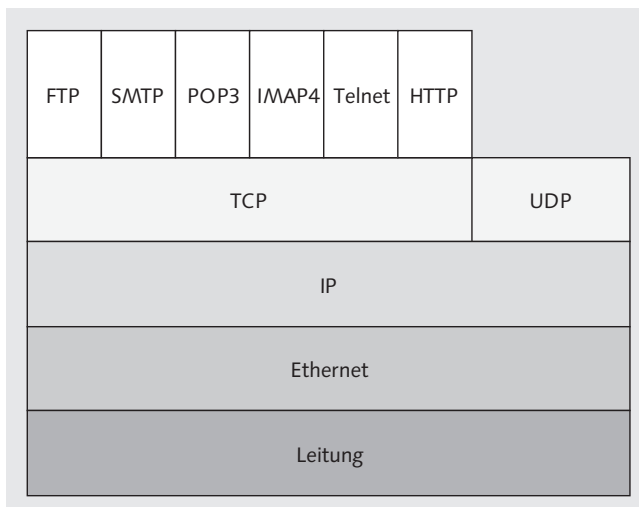


Abbildung 19.1 Netzwerkprotokolle

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 übermit-

¹ Das OSI-Modell wurde 1983 von der Internationalen Organisation für Normung (ISO) standardisiert und spezifiziert auch, was die Protokolle der einzelnen Schichten zu leisten haben.

telt 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 interessant. In diesem Buch werden wir folgende Protokolle behandeln:

Protokoll	Beschreibung	Modul	Abschnitt	Seite
UDP	grundlegendes verbindungsloses Netzwerkprotokoll	<i>socket</i>	19.1.2	612
TCP	grundlegendes verbindungsorientiertes Netzwerkprotokoll	<i>socket</i>	19.1.3	614
HTTP	Übertragen von Textdateien, beispielsweise Webseiten	<i>urllib</i>	19.2	634
FTP	Dateiübertragung	<i>ftplib</i>	19.3	646
SMTP	Versenden von E-Mails	<i>smtplib</i>	19.4.1	655
POP3	Abholen von E-Mails	<i>poplib</i>	19.4.2	660
IMAP4	Abholen und Verwalten von E-Mails	<i>imaplib</i>	19.4.3	665
Telnet	Terminalemulation	<i>telnetlib</i>	19.5	676

Tabelle 19.1 Netzwerkprotokolle

Es gibt auch abstrakte Protokolle, die auf UDP aufbauen, beispielsweise NFS (*Network File System*). Wir werden in diesem Buch aber ausschließlich auf TCP basierende Protokolle behandeln, da diese die am häufigsten verwendeten sind.

Wir werden im ersten Unterabschnitt zunächst eine grundlegende Einführung in das systemnahe Modul *socket* bringen. Es lohnt sich, einen Blick in dieses Modul zu riskieren, denn es bietet viele Möglichkeiten der Netzwerkprogrammierung, die bei den anderen, abstrakteren Modulen verloren gehen. Außerdem lernen Sie den Komfort, den die abstrakten Schnittstellen bieten, erst wirklich zu schätzen, wenn Sie das *socket*-Modul kennen gelernt 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.

19.1 Socket API

Das Modul `socket` der Standardbibliothek bietet grundlegende Funktionalität zur Netzwerkkommunikation. Es bildet dabei die standardisierte *Socket API* ab, die so oder in 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 können 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 können. 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.

19.1.1 Client-Server-Systeme

Die beiden Kommunikationspartner einer Netzwerkkommunikation haben in der Regel verschiedene Aufgaben. So existiert zum einen ein *Server* (dt. »Diener«), der bestimmte Dienstleistungen anbietet, und zum anderen ein *Client* (dt. »Kunde«), der diese Dienstleistungen in Anspruch nimmt.

Ein Server ist unter einer bekannten Adresse im Netzwerk erreichbar und operiert passiv, das heißt, er wartet auf eingehende Verbindungen. Sobald eine Verbindungsanfrage eines Clients eintrifft, wird, sofern der Server die Anfrage akzeptiert, ein neuer Socket erzeugt, über den die Kommunikation mit diesem speziellen Client läuft. Wir werden uns zunächst mit sogenannten *seriellen Servern* befassen, das sind Server, bei denen die Kommunikation mit dem vorherigen Client abgeschlossen sein muss, bevor eine neue Verbindung akzeptiert werden kann. Dem gegenüber stehen die Konzepte der *parallelen Server* und der *multiplexenden Server*, auf die wir auch noch zu sprechen kommen werden.

Der Client stellt den aktiven Kommunikationspartner dar. Das heißt, er sendet eine Verbindungsanfrage an den Server und nimmt dann aktiv dessen Dienstleistungen in Anspruch.

Die Stadien, in denen sich ein serieller Server und ein Client vor, während und nach der Kommunikation befinden, verdeutlicht das Flussdiagramm in Abbildung 19.2. Sie können es als eine Art Bauplan für einen seriellen Server und den dazugehörigen 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. Ein Kommunikationssocket ist immer nur für einen verbundenen Client zuständig.

Sobald die Kommunikation beendet ist, wird das Kommunikationsobjekt geschlossen und eventuell eine weitere Verbindung eingegangen. Verbindungsanfragen, die nicht sofort akzeptiert werden, sind keineswegs verloren, sondern werden gepuffert. Sie befinden sich in der sogenannten *Queue* und können nacheinander abgearbeitet werden. Zum Schluss wird auch der Verbindungssocket geschlossen.

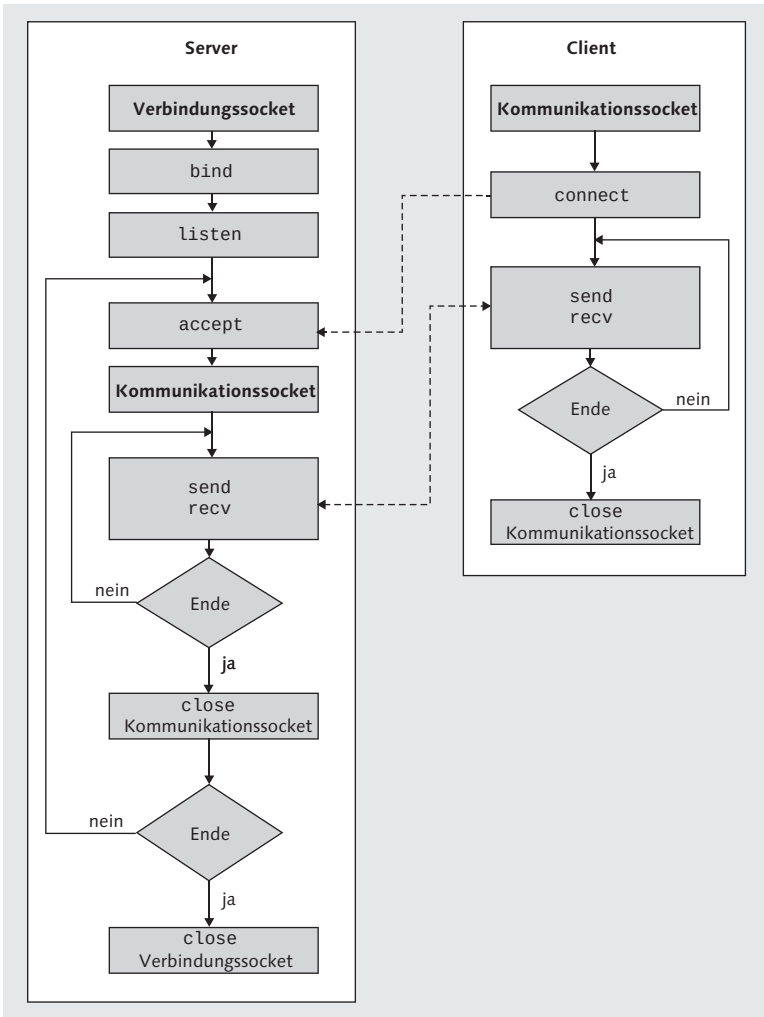


Abbildung 19.2 Das Client-Server-Modell

Die Struktur des Clients ist vergleichsweise einfach. So gibt es beispielsweise nur einen Kommunikationssocket, über den mithilfe der Methode `connect` eine Verbindungsanfrage an einen bestimmten Server gesendet werden kann. Danach erfolgt, ähnlich wie beim Server, die tatsächliche Kommunikation über Methoden wie `send` oder `recv`. Nach dem Ende der Kommunikation wird der Verbindungssocket geschlossen.

Grundsätzlich kann für die Datenübertragung zwischen Server und Client aus zwei verfügbaren Netzwerkprotokollen gewählt werden: UDP und TCP. In den folgenden beiden Abschnitten sollen kleine Beispielservers 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.

19.1.2 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 Beispielpjekt 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(nachricht.encode(), (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 kodiert 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 19.1.1, »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("{} {} {}".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 auch über 127.0.0.1 oder localhost.

Nachdem der Socket an eine Adresse gebunden wurde, 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.

19.1.3 TCP

TCP (Transmission Control Protocol) ist kein Konkurrenzprodukt zu UDP, sondern füllt mit seinen Möglichkeiten die Lücken auf, die UDP offen lässt. So ist TCP vor allem verbindungsorientiert und zuverlässig. Verbindungsorientiert bedeutet, dass nicht, wie bei UDP, einfach Datenpakete an bestimmte IP-Adressen geschickt werden, sondern dass zuvor eine Verbindung aufgebaut wird und auf Basis dieser Verbindung weitere Operationen durchgeführt werden. Zuverlässig bedeutet, dass es mit TCP nicht, wie bei UDP, vorkommen kann, dass Pakete verloren gehen, fehlerhaft oder in falscher Reihenfolge ankommen. Solche Vorkommnisse korrigiert das TCP-Protokoll intern, indem es beispielsweise unvollständige oder fehlerhafte Pakete neu anfordert.

Aus diesem Grund ist TCP zumeist die erste Wahl, wenn es um eine Netzwerkschnittstelle geht. Bedenken Sie aber unbedingt, dass jedes Paket, das neu angefordert werden muss, Zeit kostet und die Latenz der Verbindung somit steigen kann. Außerdem sind fehlerhafte Übertragungen in einem LAN äußerst selten, weswegen Sie gerade dort die Performance von UDP und die Verbindungsqualität von TCP gegeneinander abwägen sollten.

Im Folgenden wird 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("[{}] {}".format(addr[0], data.decode()))
    nachricht = input("Antwort: ")
    komm.send(nachricht.encode())
finally:
    s.close()

```

Bei der Erzeugung des Verbindungssockets unterscheidet sich TCP von UDP nur in den zu übergebenden Werten. In diesem Fall wird `AF_INET` für das IPv4-Protokoll und `SOCK_STREAM` für die Verwendung von TCP übergeben. Damit ist allerdings nur der Socket in seiner Rohform instanziiert. Auch bei TCP muss der Socket an eine IP-Adresse und einen Port gebunden werden. Beachten Sie, dass `bind` ein Adressobjekt als Parameter erwartet – die Angaben von IP-Adresse und Port also noch in ein Tupel gefasst sind. Auch hier werden wieder alle IP-Adressen des Servers genutzt.

Danach wird der Server durch Aufruf der Methode `listen` in den passiven Modus geschaltet und instruiert, nach Verbindungsanfragen zu horchen. Beachten Sie, dass diese Methode noch keine Verbindung herstellt. Der übergebene Parameter bestimmt die maximale Anzahl von zu puffernden Verbindungsversuchen und sollte mindestens 1 sein.

In der darauffolgenden Endlosschleife wartet die aufgerufene Methode `accept` des Verbindungssockets 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(nachricht.encode())
        antwort = s.recv(1024)
        print("{} [{}] {}".format(ip, antwort.decode()))
finally:
    s.close()

```

Auf der Clientseite wird der instanziierte Socket `s` durch Aufruf der Methode `connect` mit dem Verbindungspartner verbunden. Die Methode `connect` verschickt genau die Verbindungsanfrage, die beim Server durch `accept` akzeptiert werden kann. Wenn die Verbindung abgelehnt wurde, wird eine Exception geworfen.

Die nachfolgende Endlosschleife funktioniert ähnlich wie die des Servers, mit dem Unterschied, dass zuerst eine Nachricht eingegeben und abgeschickt und danach auf eine Antwort des Servers gewartet wird. Damit wären Client und Server in einen Rhythmus gebracht, bei dem der Server immer dann auf eine Nachricht wartet, wenn beim Client eine eingegeben wird und umgekehrt.

Genau dieser Rhythmus ist aber auch der größte Knackpunkt des Beispielprojekts, denn es ist für einen der Kommunikationspartner schlicht unmöglich, zwei Nachrichten direkt hintereinander abzusetzen. Für den praktischen Einsatz hätte das Programm also allenfalls Unterhaltungswert. Das Ziel war es auch nicht, eine möglichst perfekte Chat-Applikation zu schreiben, sondern eine einfache und kurze Beispielimplementierung einer Client-Server-Kommunikation über TCP zu erstellen.

Betrachten Sie es also als Herausforderung, Client und Server, beispielsweise durch Threads,² 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.

19.1.4 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

² Siehe Kapitel 17 (ab Seite 525).

warten, bis die von ihnen angestoßene Operation durchgeführt wurde. So blockiert ein Aufruf der Methode `recv` eines Sockets so lange das komplette Programm, 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. Durch mehrmaligen Aufruf von `connect` lässt sich feststellen, ob die Operation immer noch durchgeführt wird. Wenn `connect` aufgerufen wird und die Verbindungsanfrage noch läuft, wird eine `socket.error`-Exception mit der Fehlermeldung »Operation now in progress« geworfen.

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 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` versetzt den Socket wieder in den ursprünglichen blockierenden Modus.

Socket-Operationen werden im Falle des blockierenden Modus auch *synchrone Operationen* und im Falle des nicht-blockierenden Modus *asynchrone Operationen* genannt.

Es ist durchaus möglich, auch während des Betriebs zwischen dem blockierenden und dem nicht-blockierenden Modus eines Sockets umzuschalten. So können Sie beispielsweise die Methode `connect` blockierend und anschließend die Methode `read` nicht-blockierend verwenden.

19.1.5 Verwendung des Moduls

Nach der anwendungsorientierten Einführung in das `socket`-Modul besprechen wir in diesem Abschnitt die wichtigsten Funktionen des Moduls noch einmal im Detail.

Funktion	Beschreibung	Seite
<code>create_connection(address[, timeout[, source_address]])</code>	Verbindet zu der angegebenen Adresse und gibt das verwendete Socket-Objekt zurück.	619
<code>getfqdn([name])</code>	Gibt den vollständigen Domainnamen ³ der Domain <i>name</i> zurück. Wenn <i>name</i> weggelassen wird, wird der FQDN des lokalen Hosts zurückgegeben.	–
<code>gethostbyaddr(ip_address)</code>	wie <code>gethostbyname_ex</code> , nur für eine gegebene IP-Adresse statt für einen gegebenen Hostnamen	–
<code>gethostbyname(hostname)</code>	Gibt die IPv4-Adresse des Hosts <i>hostname</i> als String zurück. Inkompatibel zu IPv6	–
<code>gethostbyname_ex(hostname)</code>	Gibt die IPv4-Adressen und alternativen Hostnamen des Hosts <i>hostname</i> als String zurück. Inkompatibel zu IPv6	620
<code>gethostname()</code>	Gibt den Hostnamen des Systems als String zurück.	–

Tabelle 19.2 Funktionen des `socket`-Moduls

³ FQDN: Fully Qualified Domain Name

Funktion	Beschreibung	Seite
<i>getservbyname(service[,protocol])</i>	Gibt den Port für den Service <i>servicename</i> mit dem Netzwerkprotokoll <i>protocolname</i> zurück.	620
<i>getservbyport(port[, protocol])</i>	Das Gegenstück zu <i>getservbyname</i> : Gibt den Namen des für einen Port registrierten Services zurück.	–
<i>socket([family[, type[, proto]]])</i>	Erzeugt einen neuen Socket.	620
<i>socketpair([family[, type[, proto]]])</i>	Erzeugt ein Paar von miteinander verbundenen Sockets. Die Parameter <i>family</i> , <i>type</i> und <i>proto</i> haben die gleiche Bedeutung wie bei <i>socket</i> .	–
<i>getdefaulttimeout()</i>	Gibt den Timeout-Wert für Socket-Operationen zurück.	621
<i>setdefaulttimeout(timeout)</i>	Setzt den Timeout-Wert für Socket-Operationen.	621

Tabelle 19.2 Funktionen des socket-Moduls (Forts.)

create_connection(address[, timeout[, source_address]])

Diese Funktion verbindet über TCP zu der über das Adressobjekt *address* identifizierten Gegenstelle und gibt das zum Verbindungsaufbau verwendete Socket-Objekt zurück. Für den Parameter *timeout* kann ein Timeout-Wert übergeben werden, der beim Verbindungsaufbau berücksichtigt wird.

Wenn für den Parameter *source_address* ein Adressobjekt übergeben wird, wird das Socketobjekt vor dem Verbindungsaufbau an diese Adresse gebunden. Der Aufruf:

```
import socket
s = socket.create_connection((ip1,port1), timeout, (ip2, port2))
```

ist damit äquivalent zu:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(timeout)
s.bind((ip2, port2))
s.connect((ip1, port1))
```


Die Funktion `create_connection` kann im Zusammenhang mit der `with`-Anweisung verwendet werden:

```
with socket.create_connection(("ip", port)) as s:
    s.send(b"Hallo Welt")
```

gethostbyname_ex(hostname)

Die Funktion `gethostbyname_ex` gibt für einen gegebenen Hostnamen *hostname* ein Tupel zurück, das den primären Hostnamen, eine Liste der alternativen Hostnamen und eine Liste der IPv4-Adressen, unter denen der Server erreichbar ist.

```
>>> socket.gethostbyname_ex("www.google.de")
('www.l.google.com',
 ['www.google.de', 'www.google.com'],
 ['74.125.39.99', '74.125.39.104', '74.125.39.105', '74.125.39.103',
 '74.125.39.106', '74.125.39.147'])
```

getservbyname(service[, protocol])

Diese Funktion gibt den Port für den Service *service* mit dem Netzwerkprotokoll *protocol* zurück. Bekannte Services sind beispielsweise "http" oder "ftp" mit den Portnummern 80 bzw. 21. Der Parameter *protocol* sollte entweder "tcp" oder "udp" sein.

```
>>> socket.getservbyname("http", "tcp")
80
```

socket([family[, type[, proto]]])

Diese Funktion 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.



Hinweis

IPv6 (Internet Protocol version 6) ist der Nachfolger des weit verbreiteten IPv4-Protokolls, dessen Adressraum inzwischen nahezu erschöpft ist.

Python bietet eine grundlegende IPv6-Unterstützung, für die es in der Regel ausreicht, bei der Socket-Instanziierung den Wert `AF_INET6` statt `AF_INET` zu übergeben. Einige Funktionen des `socket`-Moduls sind inkompatibel zu IPv6, was bei der Besprechung der jeweiligen Funktion erwähnt wird.

IPv6 muss vom Betriebssystem unterstützt werden. Ob dies der Fall ist, können Sie anhand der booleschen Variablen `socket.has_ipv6` ablesen.

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 Manpages⁴ unter dem Stichwort »socket«.

getdefaulttimeout()
setdefaulttimeout(timeout)

Diese Funktion 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.

Ein Timeout-Wert von `None` bedeutet, dass Socket-Operationen unbegrenzt warten dürfen.

19.1.6 Die Socket-Klasse

Nachdem durch die Funktion *socket* eine neue Instanz der Klasse `Socket` erzeugt wurde, stellt diese 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 19.1.4, »Blockierende und nicht-blockierende Sockets«.

Methode	Beschreibung	Spezifisch	Seite
<i>accept()</i>	Wartet auf eine eingehende Verbindungsanfrage und akzeptiert diese.	TCP	623
<i>bind(address)</i>	Bindet den Socket an die Adresse <i>address</i> .	–	623
<i>close()</i>	Schließt den Socket. Das bedeutet, dass keine Daten mehr über ihn gesendet oder empfangen werden können.	–	–

Tabelle 19.3 Methoden der Socket-Klasse

⁴ <http://linux.die.net/man/>

Methode	Beschreibung	Spezifisch	Seite
<i>connect(address)</i>	Verbindet zu einem Server mit der Adresse <i>address</i> .	TCP	623
<i>connect_ex(address)</i>	Wie <i>connect</i> , wirft aber im nicht-blockierenden Modus keine Exception.	TCP	623
<i>getpeername()</i>	Gibt das Adressobjekt des verbundenen Sockets zurück. Das Adressobjekt ist ein Tupel aus IP-Adresse und Portnummer.	TCP	–
<i>getsockname()</i>	Gibt das Adressobjekt des Sockets selbst zurück.	–	–
<i>listen(backlog)</i>	Lässt den Socket auf Verbindungsanfragen achten.	TCP	624
<i>recv(bufsize[, flags])</i>	Liest beim Socket eingegangene Daten.	TCP	624
<i>recv_into(buffer[, nbytes[, flags]])</i>	Wie <i>recv</i> , schreibt die gelesenen Daten aber in <i>buffer</i> , statt sie als bytes-String zurückzugeben. Für <i>buffer</i> kann beispielsweise eine bytearray-Instanz übergeben werden.	TCP	–
<i>recvfrom(bufsize[, flags])</i>	Liest beim Socket eingegangene Daten.	UDP	624
<i>recvfrom_into(buffer[, nbytes[, flags]])</i>	Wie <i>recvfrom</i> , schreibt die gelesenen Daten aber in <i>buffer</i> , statt sie als bytes-String zurückzugeben. Für <i>buffer</i> kann beispielsweise eine bytearray-Instanz übergeben werden.	UDP	–
<i>send(string[, flags])</i>	Sendet den String <i>string</i> an den verbundenen Socket.	TCP	624
<i>sendall(string[, flags])</i>	Sendet den String <i>string</i> an den verbundenen Socket.	TCP	625
<i>sendto(string[, flags], address)</i>	Sendet den String <i>string</i> an einen Socket mit der Adresse <i>address</i> . Der Parameter <i>address</i> muss ein Adressobjekt sein.	UDP	–

Tabelle 19.3 Methoden der Socket-Klasse (Forts.)

Methode	Beschreibung	Spezifisch	Seite
<i>setblocking(flag)</i>	Versetzt den Socket in den blockierenden bzw. nicht-blockierenden Modus.	–	625
<i>settimeout(value)</i>	Setzt einen Timeout-Wert für den Socket.	–	625
<i>gettimeout()</i>	Gibt den Timeout-Wert des Sockets zurück.	–	625

Tabelle 19.3 Methoden der Socket-Klasse (Forts.)

accept()

Diese Methode 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.

bind(address)

Diese Methode 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 und auf Verbindungsanfragen warten oder, im Falle von UDP, direkt Datenpakete empfangen.

connect(address)

Diese Methode 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.

connect_ex(address)

Diese Methode 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 Rückgabe-

wert 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.

listen(backlog)

Diese Methode 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 wurde, 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.

recv(bufsize[, flags])

Diese Methode 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.⁵ Dasselbe gilt für den gleichnamigen Parameter der folgenden Methoden.

recvfrom(bufsize[, flags])

Diese Methode 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.

send(string[, flags])

Diese Methode 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.

⁵ Eine nähere Erläuterung des Parameters *flags* finden Sie zum Beispiel in den Linux Manpages (<http://linux.die.net/man/>) unter dem jeweiligen Methodennamen.

sendall(string[, flags])

Diese Methode 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.

setblocking(flag)

Wenn für *flag* der Wert `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.

settimeout(value)**gettimeout()**

Diese Methode 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.

19.1.7 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>ntohl(x)</code>	Konvertiert eine 32-Bit-Zahl von der Netzwerk- in die Host-Byte-Order.
<code>ntohs(x)</code>	Konvertiert eine 16-Bit-Zahl von der Netzwerk- in die Host-Byte-Order.
<code>htonl(x)</code>	Konvertiert eine 32-Bit-Zahl von der Host- in die Netzwerk-Byte-Order.
<code>htons(x)</code>	Konvertiert eine 16-Bit-Zahl von der Host- in die Netzwerk-Byte-Order.

Tabelle 19.4 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.



Hinweis

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 Netzerkwendungen schreiben, die Binärdaten ausschließlich zwischen x86er-PCs austauschen, brauchen Sie sich über die Byte-Order keine Gedanken zu machen. Zudem können ASCII-Zeichen, wie gesagt, problemlos auch zwischen Systemen mit verschiedener Byte-Order ausgetauscht werden, sodass auch in diesem Fall keine explizite Konvertierung nötig ist.

19.1.8 Multiplexende Server – select

Ein Server ist in den meisten Fällen nicht dazu gedacht, immer nur einen Client zu bedienen, wie es in den bisherigen Beispielen vereinfacht angenommen wurde. In der Regel muss ein Server eine ganze Reihe von verbundenen Clients verwalten, die sich in verschiedenen Phasen der Kommunikation befinden. Es stellt sich die Frage, wie so etwas sinnvoll in einem Prozess, also ohne den Einsatz von Threads, durchgeführt werden kann.

Selbstverständlich könnte man alle verwendeten Sockets in den nicht-blockierenden Modus schalten und die Verwaltung selbst in die Hand nehmen. Das ist aber nur auf den ersten Blick eine Lösung, denn der blockierende Modus besitzt einen unschätzbaren Vorteil: Ein blockierender Socket veranlasst, dass das Programm bei einer Netzwerkoperation so lange schlafen gelegt wird, bis die Operation durchgeführt werden kann. Auf diese Weise kann die Prozessorauslastung reduziert werden.

Im Gegensatz dazu müssten wir beim Einsatz von nicht-blockierenden Sockets in einer Schleife ständig über alle verbundenen Sockets iterieren und prüfen, ob sich etwas getan hat, also ob beispielsweise Daten zum Auslesen bereitstehen. Dieser Ansatz, auch *Busy Waiting* genannt, ermöglicht uns zwar das quasi-parallele Auslesen mehrerer Sockets, das Programm lastet den Prozessor aber wesentlich mehr aus, da es über den gesamten Zeitraum aktiv ist.

Das Modul `select` ermöglicht es, im gleichen Prozess mehrere blockierende Sockets zu verwalten, sodass die Vorteile blockierender Sockets erhalten bleiben. Ein Server, der `select` verwendet, wird *multiplexender Server* genannt. Im Modul ist im Wesentlichen die Funktion `select` enthalten, die im Folgenden besprochen werden soll.

`select(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. Dieselbe Socket-Instanz darf beim Aufruf von *select* durchaus in mehreren der übergebenen Listen vorkommen.

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("[{}] {}".format(ip, nachricht.decode()))
        else:
            print("+++ Verbindung zu {} "
                  "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.

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` 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(nachricht.encode())
finally:
    s.close()
```

Dabei handelt es sich um reine Socket-Programmierung, wie wir sie bereits in den vorherigen Abschnitten behandelt haben. Der Client bemerkt, abgesehen von eventuell auftretenden Latenzen, nicht, ob er von einem seriellen oder einem multiplexenden Server bedient wird.

19.1.9 socketserver

Sie können sich vorstellen, dass die Implementierung eines komplexeren Servers unter Verwendung des Moduls `socket` schnell unübersichtlich und kompliziert werden 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("{} Verbindung hergestellt".format(addr))
```

```

while True:
    s = self.request.recv(1024)
    if s:
        print("{} {}".format(addr, s.decode()))
    else:
        print("{} 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 instanziiert:

```

server = socketserver.ThreadingTCPServer(("", 50000),
                                         ChatRequestHandler)

server.serve_forever()

```

19

Um den tatsächlichen Server zu erstellen, erzeugen wir 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, alle von nun an eingehenden Verbindungsanfragen zu akzeptieren.

Hinweis

Der Programmierer trägt selbst Verantwortung für eventuell von mehreren Threads gemeinsam genutzte Ressourcen. Diese müssen gegebenenfalls durch Critical Sections abgesichert werden.

Näheres zur parallelen Programmierung finden Sie in Kapitel 17 (S. 525).



Neben der Klasse `ThreadingTCPServer` können auch andere Server-Klassen instanziiert werden, je nachdem, wie sich der Server verhalten soll. Die Schnittstelle der Konstrukturen ist immer dieselbe.

socketserver.TCPServer**socketserver.UDPServer**

Dies ist ein einfacher TCP- bzw. UDP-Server. Beachten Sie, dass diese Server immer nur eine Verbindung gleichzeitig eingehen können. Aus diesem Grund ist die Klasse `TCPServer` für unser Beispielprogramm nicht einsetzbar.

socketserver.ThreadingTCPServer**socketserver.ThreadingUDPServer**

Diese Klassen implementieren einen TCP- bzw. UDP-Server, der jede Anfrage eines Clients in einem eigenen Thread behandelt, sodass der Server mit mehreren Clients gleichzeitig in Kontakt sein kann. Damit ist die Klasse `ThreadingTCPServer` ideal für unser obiges Beispiel.

socketserver.ForkingTCPServer**socketserver.ForkingUDPServer**

Diese Klassen implementieren einen TCP- bzw. UDP-Server, der jede Anfrage eines Clients in einem eigenen Prozess behandelt, sodass der Server mit mehreren Clients gleichzeitig in Kontakt sein kann. Die Methode `handle` des Request Handlers wird in einem eigenen Prozess ausgeführt, kann also nicht auf Instanzen des Hauptprozesses zugreifen.

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* auf eingehenden Verbindungsanfragen wartet.

s.socket

Dieses Attribut referenziert die von dem Server verwendete `Socket`-Instanz, mit der er auf eingehende Verbindungsanfragen wartet.

s.fileno()

Diese Methode gibt den Dateideskriptor des Serversockets zurück.

s.handle_request()

Diese Methode instruiert den Server, genau eine Verbindungsanfrage zu akzeptieren und zu behandeln.

s.serve_forever([poll_intervall])

Diese Methode 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()

Die Methode *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* ein Tupel, das die vom Client gesendeten Daten und den Kommunikationssocket enthält, der für die Antwort verwendet werden kann.

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.

19.2 URLs

Eine *URL* (für *Uniform Resource Locator*) spezifiziert eine Ressource, beispielsweise im Internet, über ihren Ort und das zum Zugriff zu verwendende Protokoll.

Das Paket `urllib` bietet eine komfortable Schnittstelle zum Umgang mit Ressourcen im Internet. Dazu enthält `urllib` die folgenden Module:

Modul	Beschreibung	Seite
<code>urllib.request</code>	Enthält Funktionen und Klassen zum Zugriff auf eine Ressource im Internet.	635
<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.	640
<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 ⁶ einer Website interpretiert.	–

Tabelle 19.5 Module des Pakets `urllib`

In den folgenden Abschnitten sollen die Module `request` und `parse` des Pakets `urllib` erläutert werden

19.2.1 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

⁶ Programme, die automatisiert das Internet durchforsten, lesen in der Regel zunächst eine Datei namens *robots.txt* im Hauptverzeichnis des Webservers. Dort kann festgelegt werden, welche Teile der Website durchsucht werden dürfen.

übergeben wird. Außerdem können auf dem resultierenden Dateiobjekt aus naheliegendem Grund 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], *, [cafile][capath])`

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 Dateiobjekt ist ein dateiähnliches Objekt (engl. *file-like object*), da es nur eine Untermenge der Funktionalität eines echten Dateiobjekts bereitstellt. Die folgende Tabelle zeigt die verfügbaren Methoden des zurückgegebenen dateiähnlichen Objekts mit einer kurzen Beschreibung. Die Parameter `timeout`, `cafile` und `capath` werden im Anschluss besprochen.

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.

Tabelle 19.6 Methoden des dateiähnlichen Objekts

Methode	Beschreibung
<i>readlines([sizehint])</i>	Liest die Ressource zeilenweise aus und gibt sie in Form einer Liste von Strings zurück. Wird <i>sizehint</i> angegeben, so werden Zeilen nur so lange eingelesen, bis die Gesamtgröße der gelesenen Zeilen <i>sizehint</i> überschreitet.
<i>fileno()</i>	Gibt den Dateideskriptor der geöffneten Ressource als ganze Zahl zurück.
<i>close()</i>	Schließt das geöffnete Objekt. Nach Aufruf dieser Methode sind keine weiteren Operationen mehr möglich.
<i>info()</i>	Gibt ein dictionary-ähnliches Objekt zurück, das Metainformationen der heruntergeladenen Seite enthält. Im Anschluss an diese Tabelle werden wir uns eingehend mit der von <i>info</i> zurückgegebenen Instanz beschäftigen.
<i>geturl()</i>	Gibt einen String mit der URL der Ressource zurück.

Tabelle 19.6 Methoden des dateiähnlichen Objekts (Forts.)

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', 'Vary', '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 Dictionarys lassen sich alle enthaltenen Schlüssel anzeigen. Welche Informationen enthalten sind, hängt vom verwendeten Protokoll ab. Beim HTTP-Protokoll enthält das dictionary-ähnliche Objekt alle vom Server gesendeten Informationen. So können Sie beispielsweise über die Schlüssel "Content-Length" und "Server" die Größe der heruntergeladenen Datei in Byte bzw. den Identifikationsstring der Serversoftware auslesen.⁷

⁷ Bei dem in diesem Fall verwendeten Server *Zope* handelt es sich um einen Webserver für Python.

```
>>> d["Content-Length"]
'27395'
>>> d["Server"]
'Zope/(Zope 2.7.6-final, python 2.3.5, linux2) ZServer/1.1'
```

Wenn das verwendete Protokoll `http` ist, dient der optionale Parameter *data* dazu, POST-Parameter⁸ 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` (S. 640).

Neben POST existiert eine weitere Methode zur Parameterübergabe an eine Website: 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 verstanden wird. Wenn dieser Parameter nicht übergeben wird, wird ein Standardwert des Betriebssystems als Timeout verwendet.

Die optionalen reinen Schlüsselwortparameter *cafile* und *capath* erlauben es, Zertifikate bereitzustellen, über die sich die Funktion `urlopen` bei der Gegenstelle authentifiziert. Für *cafile* kann ein Pfad zu einer Datei angegeben werden, die ein Zertifikat im PEM-Format enthält. Über den Parameter *capath* lässt sich ein Verzeichnis festlegen, in dem die Zertifikat-Dateien liegen.

`urllib.request.urlretrieve(url[, filename[, reporthook[, data]]])`

Diese Funktion 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

⁸ Das HTTP-Protokoll kennt zwei Arten der Argumentübertragung: Bei *POST* werden die Daten für den Benutzer unsichtbar im HTTP-Body mitgesendet, während sie bei *GET* in die URL hineinkodiert werden.

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 übergeben Sie drei Parameter: die Anzahl der bisher übertragenen Blöcke, die Größe eines Blocks in Byte und die Gesamtgröße der Ressource in Byte. Mithilfe des dritten Parameters lässt sich also eine Statusanzeige des Downloads realisieren:

```
>>> def f(blocks, blocksize, size):
...     print("Status: {}".format(int(blocksize*blocks*100/size)))
...
>>> res = urllib.request.urlretrieve(url, "datei.html", f)
Status: 0%
Status: 14%
Status: 29%
Status: 44%
Status: 59%
Status: 74%
Status: 88%
Status: 103%
>>> res
('datei.html', <http.client.HTTPMessage object at 0xd8cc50>)
```

Dass die Statusanzeige zuletzt 103% anzeigt, liegt daran, dass das letzte Paket nicht die volle Größe hat.

Der vierte Parameter, *data*, entspricht dem Parameter *data* der Funktion *urlopen* und wird auch so verwendet.

urllib.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 zuordnen. Das Dictionary `{"http": "http://proxy.beispiel.de:8080/"}` würde beispielsweise den Server `proxy.beispiel.de` als http-Proxy einrichten.

Zusätzlich können Sie bei der Instanziierung 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 Benutzername 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.

19.2.2 Verarbeiten einer URL – urllib.parse

Das Modul `urllib.parse` enthält Funktionen, die es ermöglichen, eine URL in ihre Bestandteile zu zerlegen oder diese Bestandteile wieder zu einer gültigen URL zusammenzufügen. Die folgende Tabelle fasst den Inhalt des Moduls `urllib.parse` zusammen und verweist, wenn nötig, auf eine detaillierte Erklärung.

Funktion	Beschreibung	Seite
<code>urlparse(urlstring[, scheme[, allow_fragments]])</code>	Zerlegt die URL <i>urlstring</i> in ihre Bestandteile.	641
<code>parse_qs(qs[, keep_blank_values[, strict_parsing[, encoding[, errors]]]])</code>	Zerlegt den Query-String <i>qs</i> einer URL in seine Bestandteile. Das Ergebnis wird als Dictionary zurückgegeben.	643
<code>parse_qsl(qsl[, keep_blank_values[, strict_parsing[, encoding[, errors]]]])</code>	Zerlegt den Query-String <i>qs</i> einer URL in seine Bestandteile. Das Ergebnis wird als Liste von Schlüssel-Wert-Paaren zurückgegeben.	643
<code>urlunparse(parts)</code>	Das Gegenstück zu <code>urlparse</code> : Erzeugt aus einem Tupel mit den Bestandteilen einer URL einen URL-String.	644
<code>urlsplit(urlstring[, scheme[, allow_fragments]])</code>	Wie <code>urlparse</code> , berücksichtigt aber nicht den Parameter Teil einer URL.	644
<code>urlunsplit(parts)</code>	das Gegenstück zu <code>urlsplit</code>	–
<code>urljoin(base, url[, allow_fragments])</code>	Kombiniert die Basis-URL <i>base</i> und die relative URL <i>url</i> zu einer absoluten Pfadangabe.	644
<code>urldefrag(url)</code>	Spaltet den Anker von der URL <i>url</i> ab, sofern dieser vorhanden ist.	645
<code>quote(string[, safe[, encoding[, errors]]])</code>	Ersetzt Sonderzeichen im String <i>string</i> durch Escape-Sequenzen, wie sie in URLs erlaubt sind.	645
<code>quote_plus(string[, safe[, encoding[, errors]]])</code>	Funktioniert wie <code>quote</code> mit dem Unterschied, dass ein Leerzeichen in <i>string</i> durch ein + ersetzt wird. Dies ist insbesondere im Zusammenhang mit HTML-Formulardaten interessant.	–

Tabelle 19.7 Funktion des Moduls `urllib.parse`

Funktion	Beschreibung	Seite
<code>quote_from_bytes(bytes[, safe])</code>	Funktioniert wie <code>quote</code> mit dem Unterschied, dass der betrachtete Text als <code>bytes</code> -Instanz und nicht als String übergeben wird.	–
<code>unquote(string[, encoding[, errors]])</code>	das Gegenstück zu <code>quote</code>	645
<code>unquote_plus(string[, encoding[, errors]])</code>	das Gegenstück zu <code>quote_plus</code>	–
<code>unquote_to_bytes(string)</code>	das Gegenstück zu <code>quote_from_bytes</code>	–
<code>urlencode(query[, doseq[, safe[, encoding[, errors]]])</code>	Erzeugt aus dem Dictionary <code>query</code> einen Query-String, der in einer URL verwendet werden kann.	646

Tabelle 19.7 Funktion des Moduls `urllib.parse` (Forts.)

Um die Beispiele ausführen zu können, muss zuvor das Modul `urllib.parse` eingebunden worden sein:

```
>>> import urllib.parse
```

`urllib.parse.urlparse(urlstring[, 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 Teilen⁹ bestehen:

```
scheme://netloc/path;params?query#fragment
```

Der `netloc`-Bereich der URL wird außerdem in vier weitere Bereiche unterteilt:

```
username:password@host:port
```

Bis auf die Host-Angabe im `netloc`-Bereich sind alle Angaben optional und können 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. Sie können über Attri-

⁹ Der `params`-Teil einer URL wird nur sehr selten verwendet.

bute auch auf die vier Unterbereiche des `netloc`-Teils zugreifen, die nicht über einen Index erreichbar sind.

Die folgende Tabelle listet alle Attribute des Rückgabewertes der Funktion `urlparse` auf und erläutert sie jeweils mit einem kurzen Satz. Zusätzlich ist der entsprechende Index angegeben, sofern sich das entsprechende Attribut auch über einen Index ansprechen lässt. Die Attributnamen entsprechen den Namen der Bereiche, wie sie in den obigen URL-Beispielen verwendet wurden.

Attribut	Index	Beschreibung
<i>scheme</i>	0	das Protokoll der URL, beispielsweise <code>http</code> oder <code>file</code>
<i>netloc</i>	1	Die <i>Network Location</i> besteht üblicherweise aus einem Domainnamen mit Subdomain und TLD, beispielsweise <code>www.galileo-press.de</code> . Optional können auch Benutzername, Passwort und Portnummer in <i>netloc</i> enthalten sein.
<i>path</i>	2	eine Pfadangabe, die einen Unterordner der Network Location kennzeichnet
<i>params</i>	3	Parameter für das letzte Element des Pfades
<i>query</i>	4	Über den <i>Query-String</i> können zusätzliche Informationen an ein serverseitiges Script übertragen werden.
<i>fragment</i>	5	Das Fragment, auch <i>Anker</i> genannt. Ein geläufiges Beispiel für einen Anker ist eine Sprungmarke innerhalb einer HTML-Datei.
<i>username</i>	–	der in der URL angegebene Benutzername, sofern vorhanden
<i>password</i>	–	das in der URL angegebene Passwort, sofern vorhanden.
<i>hostname</i>	–	der Domainname der URL, beispielsweise <code>www.galileo-press.de</code>
<i>port</i>	–	die in der URL angegebene Portnummer, sofern vorhanden

Tabelle 19.8 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, referen-

ziert 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'
```

19

`urllib.parse.parse_qs(qs[, keep_blank_values[, strict_parsing[, encoding[, errors]]])`,
`urllib.parse.parse_qs(qs[, keep_blank_values[, strict_parsing[, encoding[, errors]]])`

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 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&hallo=blubb&xyz=12"
>>> teile = urllib.parse.urlparse(url)
>>> urllib.parse.parse_qs(teile.query)
{'hallo': ['welt', 'blubb'], 'xyz': ['12']}
>>> urllib.parse.parse_qsl(teile.query)
[('hallo', 'welt'), ('hallo', 'blubb'), ('xyz', '12')]
```


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.

Über die optionalen Parameter `encoding` und `errors` lässt sich das Umkodieren von Escape-Sequenzen steuern.

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 den Rückgabewert von `urlparse`, übergeben.

```
>>> url = ("http", "beispiel.de", "/pfad/datei.py", "", "", "")
>>> urllib.parse.urlunparse(url)
'http://beispiel.de/pfad/datei.py'
```



Hinweis

Der Ausdruck

```
urllib.parse.urlunparse(urllib.parse.urlparse(url)) == url
```

ergibt nicht immer `True`, da überflüssige Angaben, wie beispielsweise ein leeres Fragment am Ende einer URL, beim Aufruf von `urlparse` verloren gehen.

urllib.parse.urlsplit(urlstring[, 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.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'
```

Sie sehen, dass *urljoin* die beiden übergebenen Pfade nicht einfach aneinanderhängt, sondern 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]]])

Die Funktion *quote* 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.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'
```

urllib.parse.urlencode(query[, doseq[, safe[, encoding[, errors]]])

Die Funktion *urlencode* 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 Parameternamen der Schlüssel der jeweiligen Sequenz im Dictionary verwendet:

```
>>> urllib.urlencode({"abc" : [1,2,3], "def" : "ghi"}, True)
'abc=1&abc=2&abc=3&def=ghi'
>>> urllib.urlencode({"abc" : [1,2,3], "def" : "ghi"}, False)
'abc=%5B1%2C+2%2C+3%5D&def=ghi'
```

Wenn für *doseq* *False* übergeben wird, werden eventuell vorhandene Sequenzen als Text in den Parameterstring eingetragen.

Die Parameter *safe*, *encoding* und *errors* werden an die Funktion *quote_plus* weitergereicht, sofern es sich bei den Elementen von *query* um Strings handelt.

19.3 FTP – ftplib

Das Modul *ftplib* ermöglicht 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 verbreitet. So erfolgen beispielsweise Dateiübertragungen auf einen Webserver üblicherweise via FTP.

Das Protokoll FTP ist sehr einfach aufgebaut und besteht aus einer Reihe 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* 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 LS	Überträgt eine Liste aller im Arbeitsverzeichnis enthaltenen Dateien und Ordner.
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 19.9 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 Instanziiieren 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* und das Passwort *passwd* 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.

Über den Parameter *acct* (für *accounting information*) können weitere Informationen an den FTP-Server gesendet werden, was aber in der Regel nicht benötigt wird.

Die Klasse FTP

Die folgende Tabelle listet die wichtigsten Methoden der Klasse `FTP` auf und erklärt diese kurz.

Methode	Beschreibung	Seite
<code>connect(host[, port[, timeout]])</code>	Verbindet zum FTP-Server <i>host</i> .	650
<code>getwelcome()</code>	Gibt die Willkommensnachricht des verbundenen FTP-Servers als String zurück.	–
<code>login([user[, passwd[, acct]]])</code>	Loggt sich auf dem verbundenen FTP-Server ein.	650
<code>abort()</code>	Unterbricht einen laufenden Datentransfer. Je nach Server kann eine solche Unterbrechung nicht zu jedem Zeitpunkt durchgeführt werden.	–
<code>sendcmd(cmd)</code>	Sendet das Kommando <i>cmd</i> an den verbundenen FTP-Server.	650

Tabelle 19.10 Methoden der Klasse `FTP`

Methode	Beschreibung	Seite
<i>voidcmd(cmd)</i>	Wie <i>sendcmd</i> , jedoch ohne Rückgabewert. Im Fehlerfall wird eine <i>error_reply</i> -Exception geworfen.	–
<i>retrbinary(cmd, callback[, maxblocksize[, rest]])</i>	Leitet einen Dateitransfer im Binärmodus ein.	651
<i>retrlines(cmd[, callback])</i>	Leitet einen Dateitransfer im ASCII-Modus ein.	652
<i>set_pasv(boolean)</i>	Versetzt die FTP-Instanz in den aktiven bzw. passiven Modus.	652
<i>storbinary(cmd, file[, blocksize[, callback[, rest]]])</i>	Leitet einen binären Datei-Upload ein.	653
<i>storlines(cmd, file[, callback])</i>	Leitet einen Datei-Upload im ASCII-Modus ein.	653
<i>nlst([argument])</i>	Gibt den Inhalt des aktuellen Arbeitsverzeichnisses auf dem FTP-Server als Liste von Strings zurück.	653
<i>dir([argument[, f]])</i>	Gibt den Inhalt des aktuellen Arbeitsverzeichnisses auf dem FTP-Server als String zurück.	654
<i>rename(fromname, toname)</i>	Benennt die Datei <i>fromname</i> auf dem FTP-Server in <i>toname</i> um.	–
<i>delete(filename)</i>	Löscht die Datei <i>filename</i> auf dem FTP-Server.	–
<i>cwd(pathname)</i>	Ändert das aktuelle Arbeitsverzeichnis auf dem FTP-Server in <i>pathname</i> .	–
<i>mkd(pathname)</i>	Erzeugt das Verzeichnis <i>pathname</i> auf dem FTP-Server. Der Pfad zum neu erstellten Verzeichnis wird zurückgegeben.	–
<i>pwd()</i>	Gibt den Pfad des aktuellen Arbeitsverzeichnisses auf dem FTP-Server zurück.	–

Tabelle 19.10 Methoden der Klasse FTP (Forts.)

Methoden	Beschreibung	Seite
<i>rmd(dirname)</i>	Löscht das Verzeichnis <i>dirname</i> auf dem FTP-Server. Das Verzeichnis muss vorhanden und leer sein, damit es gelöscht werden kann.	–
<i>size(filename)</i>	Ermittelt die Dateigröße der Datei <i>filename</i> auf dem FTP-Server.	654
<i>quit()</i>	Beendet die Verbindung zum FTP-Server.	654
<i>close()</i>	Beendet die Verbindung zum FTP-Server.	654

Tabelle 19.10 Methoden der Klasse FTP (Forts.)

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")
```

connect(host[, port[, timeout]])

Diese Methode verbindet zu dem FTP-Server *host* unter Verwendung des Ports *port*. Diese Methode sollte nicht aufgerufen werden, wenn bei der Instanziierung der Klasse `FTP` bereits die Adresse des FTP-Servers übergeben wurde. Der optionale Parameter *timeout* hat dieselbe Bedeutung wie beim Konstruktor der Klasse `FTP`.

login([user[, passwd[, acct]]])

Diese Methode loggt sich auf dem verbundenen FTP-Server ein. Beachten Sie, dass die Methode `connect` 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.

sendcmd(cmd)

Die Methode `sendcmd` schickt den Kommandostring *cmd* an den verbundenen FTP-Server und gibt dessen Antwort ebenfalls als String zurück.

```
>>> ftp.sendcmd("PWD")
'257 "/" is the current directory'
```

retrbinary(cmd, callback[, maxblocksize[, rest]])

Diese Methode 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 Datei-transfer 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 `bytes`-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 Dateiinhalt senden soll. Dies ist zum Beispiel nützlich, um abgebrochene Downloads wiederaufzunehmen, ohne dabei Teile der Datei doppelt herunterladen zu müssen.

Zur Verwendung von *retrbinary* nun folgendes Beispiel:

```
class Downloader:
    def __init__(self):
        self.data = bytes()
    def __call__(self, data):
        self.data += data

bild = Downloader()
ftp.retrbinary("RETR bild.jpg", bild)
```

Das Beispielprogramm lädt die Bilddatei *bild.jpg* aus dem aktuellen Arbeitsverzeichnis des FTP-Servers herunter und speichert die Binärdaten im `bytes`-String `bild.data`. Um zum Speichern der Daten nicht auf eine globale Referenz zurückgreifen zu müssen, haben wir eine Klasse `Downloader` erstellt, die sich mithilfe der Magic Method `__call__` wie eine Funktion aufrufen lässt.

Alternativ kann auch ein `LIST`-Kommando abgesetzt werden. Der Verzeichnisinhalt wird vom Server ebenfalls über den Datenkanal gesendet.

```
>>> 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
[...]
```

retrlines(command[, callback])

Diese Methode 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 ist 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.

```
class Downloader:
    def __init__(self):
        self.lines = []

    def __call__(self, line):
        self.lines.append(line)

text = Downloader()
ftp.retrlines("RETR text.txt", text)
print("\n".join(text.lines))
```

Dieses Beispielprogramm lädt die Textdatei *text.txt* zeilenweise herunter und fügt die heruntergeladenen Zeilen im String *text* wieder zu einem Gesamttext zusammen. Dabei bedienen wir uns wieder der aufrufbaren Klasse `Downloader`, um die Zwischenergebnisse zu speichern.

set_pasv(boolean)

Übergeben Sie für *boolean* den Wert `False`, wird die FTP-Instanz in den sogenannten aktiven Zustand versetzt. Ein Wert von `True` versetzt sie zurück in den passiven Zustand. Im aktiven Zustand muss der Client für den Server erreichbar sein, darf sich also nicht hinter einer Firewall oder einem Router befinden.

storbinary(command, file[, blocksize[, callback[, rest]]])

Diese Methode 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 Dateiobjekt 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.

Der letzte Parameter *rest* hat die gleiche Bedeutung wie bei *retrbinary*.

Das folgende Beispielprogramm führt einen binären Datei-Upload durch:

```
f = open("bild.jpg", "rb")
ftp.storbinary("STOR hallo.jpg", f)
f.close()
```

Die Datei heißt im lokalen Arbeitsverzeichnis *bild.jpg*, wird jedoch unter dem Namen *hallo.jpg* hochgeladen.

storlines(command, file[, callback])

Diese Methode 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 Dateiobjekt wie bei *storbinary* auch im Binärmodus geöffnet haben müssen.

nlst([argument])

Diese Methode gibt eine Liste mit dem Inhalt des aktuellen Arbeitsverzeichnisses auf dem FTP-Server zurück. Über den optionalen Parameter *argument* 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.

dir([argument[, f]])

Diese Methode 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 *argument* 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 ..
[...]
```

rmd(dirname)

Diese Methode löscht das Verzeichnis *dirname* auf dem FTP-Server. Das Verzeichnis *dirname* muss vorhanden und leer sein, damit es erfolgreich gelöscht werden kann. Im Fehlerfall wird eine `ftplib.error_perm`-Exception geworfen.

```
>>> ftp.rmd("ordner")
'250 RMD command successful'
```

size(filename)

Diese Methode 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`. Das dieser Methode zugrundeliegende FTP-Kommando `SIZE` ist nicht standardisiert und wird somit nicht von allen FTP-Servern unterstützt.

quit()

Diese Methode 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*.

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.



Hinweis

Die FTP-Klasse lässt sich mit dem `with`-Statement verwenden:

```
with ftplib.FTP("ftp.server.de") as f:
    f.login("Benutzername", "Passwort")
    print(f.dir("."))
```

19.4 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 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.

19.4.1 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, menschenlesbares 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 schnell nicht mehr tragbar, und so wurde das SMTP-Protokoll um den *ESMTP-Standard* (*Extended SMTP*) erweitert.

Ähnlich wie im Abschnitt 19.3, »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 beigefügt.
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 19.11 SMTP-Befehle

Wie schon die `ftplib` enthält das Modul `smtplib` im Wesentlichen nur eine Klasse namens `SMTP`. Über diese Klasse läuft, nachdem sie instanziiert wurde, alle weitere Kommunikation mit dem Server. Der Konstruktor der Klasse `SMTP` hat folgende Schnittstelle:

```
SMTP([host[, port[, local_hostname[, timeout]]]])
```

Hiermit wird eine Instanz der Klasse `SMTP` erzeugt. 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.

Methode	Beschreibung	Seite
<i>connect([host[, port]])</i>	Verbindet zum SMTP-Server <i>host</i> .	657
<i>login(user, password)</i>	Authentifiziert sich beim verbundenen SMTP-Server mit Benutzername und Passwort.	657
<i>sendmail(from_addr, to_addrs, msg[, mail_options[, rctp_options]])</i>	Sendet eine E-Mail über den verbundenen SMTP-Server	658
<i>send_message(msg[, from_addr[, to_addrs[, mail_options[, rctp_options]]]])</i>	Sendet die <code>email.Message.Message</code> -Instanz ¹⁰ <i>msg</i> als E-Mail über den verbundenen SMTP-Server. Die übrigen Parameter haben dieselbe Bedeutung wie bei <i>sendmail</i> .	–
<i>quit()</i>	Beendet die Verbindung zum SMTP-Server.	–

Tabelle 19.12 Methoden der Klasse SMTP

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]])

Diese Methode verbindet zum SMTP-Server *host* mit der Portnummer *port*. Sie 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')
```

¹⁰ Siehe Abschnitt 19.4.4 (S. 671).

Im Fehlerfall wird eine der folgenden Exceptions geworfen:

Exception	Beschreibung
<i>SMTPHeloError</i>	Der SMTP-Server hat nicht oder nicht richtig auf das Begrüßungskommando <code>HELO</code> geantwortet.
<i>SMTPAuthenticationError</i>	Die angegebene Benutzername-Passwort-Kombination wurde vom SMTP-Server nicht akzeptiert.
<i>SMTPError</i>	Es wurde keine Möglichkeit gefunden, eine Authentifizierung bei diesem SMTP-Server durchzuführen.

Tabelle 19.13 Mögliche Exceptions beim Login

`s.sendmail(from_addr, to_addrs, msg[, mail_options[, rcpt_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 im String stehen.

Als dritten Parameter, *msg*, übergeben Sie den Text der E-Mail. Hier werden auch weitere Angaben wie beispielsweise der Betreff der E-Mail definiert. Wie so etwas genau aussieht und welche Möglichkeiten Python bietet, diesen Header komfortabel zu erzeugen, erfahren Sie in Abschnitt 19.4.4 (S. 671).

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* eine der folgenden Exceptions:

Exception	Beschreibung
<i>SMTPHeloError</i>	Der SMTP-Server hat nicht oder nicht richtig auf das Begrüßungskommando <code>HELO</code> geantwortet.

Tabelle 19.14 Mögliche Exceptions beim Login

Exception	Beschreibung
<i>SMTPRecipientsRefused</i>	Alle Empfänger wurden vom SMTP-Server zurückgewiesen. Das heißt, dass die E-Mail an niemanden verschickt wurde. Als Attribut <code>recipients</code> enthält die Exception ein Dictionary, wie es von <i>sendmail</i> im Erfolgsfall zurückgegeben wird.
<i>SMTPSenderRefused</i>	Der angegebene Absender wurde vom SMTP-Server zurückgewiesen.
<i>SMTPDataError</i>	Der Server hat mit einem unerwarteten Fehler geantwortet.

Tabelle 19.14 Mögliche Exceptions beim Login (Forts.)

Hinweis

Der Text einer E-Mail darf nur aus ASCII-Zeichen bestehen. Um auch andere Zeichen und insbesondere auch Binärdaten verschicken zu können, bedient man sich der sogenannten MIME-Kodierung, die wir im Abschnitt 19.4.4, »Erstellen komplexer E-Mails – email« (S. 671), behandeln werden.

Über die optionalen Parameter *mail_options* und *rcpt_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, während die für *rcpt_options* übergebenen Optionen dem Kommando RCPT TO angehängt werden.

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@lpe-media.de>",
...     "Dies ist der Text")
{}
```



```
>>> smtp.sendmail(  
...     "Peter Kaiser <p@penguin-p.de>",  
...     ["je@lpe-media.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.

19.4.2 POP3 – poplib

Nachdem anhand der `smtpplib` 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` im nächsten Abschnitt besprochen wird. 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	Widerruft alle anstehenden Löschvorgänge. ¹¹
QUIT	Beendet die POP3-Sitzung.

Tabelle 19.15 POP3-Befehle

11 Löschvorgänge werden gepuffert und erst am Ende der Sitzung ausgeführt.

Wie bereits beim Modul `smtpLib` ist im Modul `popLib` im Wesentlichen die Klasse `POP3` enthalten, die instanziiert werden muss, bevor Operationen auf einem POP3-Server durchgeführt werden können. Die Schnittstelle des Konstruktors sieht folgendermaßen aus:

```
POP3(host[, port[, timeout]])
```

Hiermit wird eine Instanz der Klasse `POP3` erzeugt. 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.

Methode	Beschreibung	Seite
<i>user(username)</i>	Übermittelt den Benutzernamen an den POP3-Server. Zur Authentifizierung muss noch das Passwort mittels <i>pass_</i> gesendet werden.	–
<i>pass_(password)</i>	Übermittelt das Passwort an den POP3-Server.	662
<i>stat()</i>	Gibt den Status des Posteingangs zurück.	662
<i>list([which])</i>	Gibt eine Liste der im Posteingang liegenden E-Mails zurück.	662
<i>retr(which)</i>	Gibt den Inhalt der E-Mail mit der laufenden Nummer <i>which</i> zurück.	663
<i>dele(which)</i>	Löscht die E-Mail mit der laufenden Nummer <i>which</i> .	664
<i>rset()</i>	Verwirft alle anstehenden Löschvorgänge.	–
<i>quit()</i>	Beendet die Verbindung zum POP3-Server.	–

Tabelle 19.16 Methoden der Klasse `POP3`

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:

```
>>> import poplib
>>> pop = poplib.POP3("pop.beispiel.de")
```

Für die meisten Beispiele muss diese Instanz außerdem mit einem POP3-Server verbunden und bei diesem authentifiziert sein.

pop.pass_(password)

Diese Methode 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.



Hinweis

Der bevorzugte Name `pass` für diese Methode ist in Python bereits mit einem Schlüsselwort belegt. In solchen Fällen wird an den belegten Namen häufig ein Unterstrich angehängt.

pop.stat()

Diese Methode 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 in Byte.

```
>>> 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])

Diese Methode 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 Num-

mer 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 komplette Länge der Serverantwort berechnen zu können, existiert das dritte Element des Tupels (`datlen`). Dieses referenziert die Länge des Datenbereichs der Antwort des Servers. Damit entspräche `len(antwort) + datlen` der Gesamtgröße des vom Server tatsächlich gesendeten Antwortstrings.

Über den optionalen Parameter *which* kann die laufende Nummer einer E-Mail angegeben werden, über die nähere Informationen zurückgegeben werden sollen. In diesem Fall gibt die Methode einen bytes-String des Formats `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)

Diese Methode 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.¹² 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.

¹² Im Antwortstring ist von »623 octets« die Rede. Mit Octets (dt. »Achtergruppen«) sind Bytes gemeint.

pop.dele(which)

Diese Methode 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.'
```

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. Die Indizierung der E-Mails im Posteingang beginnt mit 1.

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 Tuples, enthält eine Liste mit allen Zeilen des Mail-Inhalts. Diese Liste wird in der Schleife durchlaufen, und es wird jeweils die aktuelle Zeile ausgegeben.

Aus Gründen der Übersichtlichkeit wurde im Beispielprogramm auf jegliche Fehlerbehandlung verzichtet. In einem fertigen Programm müssten Sie auf jeden Fall prüfen, ob die Verbindung zum Server hergestellt werden konnte und ob die Authentifizierung erfolgreich war.

19.4.3 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-Café 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 ein Verzeichnis vorstellen, das E-Mails enthalten kann, wie ein Ordner Dateien enthält. Die Mailbox-Struktur des verwendeten Beispielservers sieht folgendermaßen aus:

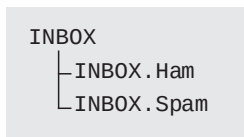


Abbildung 19.3 Mailbox-Struktur des Beispielservers

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:

```
IMAP4([host[, port]])
```

Hiermit wird eine Instanz der Klasse `IMAP4` erzeugt. Optional kann direkt nach der Instanziierung automatisch eine Verbindung zu einem IMAP4-Server mit dem Hostnamen *host* unter Verwendung des Ports *port* aufgebaut werden. Wenn der Parameter *port* nicht angegeben wurde, wird der IMAP4-Standardport 143 verwendet.

Die Klasse IMAP4

Nachdem eine Instanz der Klasse `IMAP4` erzeugt wurde, stellt diese verschiedene Methoden bereit, um mit dem verbundenen Server zu kommunizieren. Jede Methode, die ein IMAP4-Kommando repräsentiert, gibt ein Tupel der folgenden Form zurück:

(Status, [Daten, ...])

Dabei steht im resultierenden Tupel für *Status* entweder "OK" oder "NO", je nachdem, ob die Operation erfolgreich verlaufen oder fehlgeschlagen ist. Das zweite Element des Tupels ist eine Liste, die die Daten enthält, die der Server als Antwort geschickt hat. Diese Daten können entweder 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.

Methode	Beschreibung	Seite
<i>login(user, password)</i>	Sendet Benutzernamen und Passwort an den verbundenen IMAP4-Server.	–
<i>logout()</i>	Beendet die Verbindung zum IMAP4-Server.	–
<i>select([mailbox[, readonly]])</i>	Wählt eine Mailbox aus, um weitere Operationen auf dieser durchführen zu können.	667
<i>close()</i>	Schließt die momentan ausgewählte Mailbox.	–
<i>list([directory[, pattern]])</i>	Gibt die Namen aller Mailboxen zurück, die sich im Ordner <i>directory</i> befinden und auf <i>pattern</i> passen.	667
<i>fetch(message_set, message_parts)</i>	Lädt Teile der E-Mails vom Server herunter.	668
<i>create(mailbox)</i>	Erstellt eine neue Mailbox namens <i>mailbox</i> .	–
<i>delete(mailbox)</i>	Löscht die Mailbox <i>mailbox</i> .	–
<i>rename(old_mailbox, new_mailbox)</i>	Benennt die Mailbox <i>old_mailbox</i> in <i>new_mailbox</i> um.	–

Tabelle 19.17 Methoden der Klasse IMAP4

Methode	Beschreibung	Seite
<i>copy(message_set, new_mailbox)</i>	Kopiert die E-Mails <i>message_set</i> in die Mailbox <i>new_mailbox</i> . Der Parameter <i>message_set</i> ist wie bei <i>fetch</i> zu verstehen.	–
<i>search(charset, criterion[, ...])</i>	Sucht innerhalb der ausgewählten Mailbox nach E-Mails, die auf bestimmte Kriterien passen.	669
<i>store(message_set, command, flag_list)</i>	Verändert die Eigenschaften der E-Mails <i>message_set</i> .	669

Tabelle 19.17 Methoden der Klasse IMAP4 (Forts.)

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.select([mailbox[, readonly]])`

Diese Methode 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* der Wert `True` übergeben wird, ist die gewählte Mailbox bei diesem Zugriff schreibgeschützt und kann somit nicht verändert werden. Die Methode *select* gibt die Anzahl der E-Mails zurück, die sich in der gewählten Mailbox befinden.

```
>>> im.select("INBOX")
('OK', [b'2'])
```

Es wird keine Exception geworfen, wenn die gewünschte Mailbox nicht existiert. Stattdessen muss der Fehler anhand des Rückgabewertes ausgemacht werden:

```
>>> im.select("INBOX.NichtExistent")
('NO', [b'Mailbox does not exist, or must be subscribed to.'])
```

`im.list([directory[, pattern]])`

Diese Methode 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)

Diese Methode 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.search(charset, criterion[, ...])

Diese Methode 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''])
```

im.store(message_set, command, flag_list)

Diese Methode modifiziert die Eigenschaften (Flags) der E-Mails *message_set*. Mögliche Werte für den Parameter *command* sind "+FLAGS" für das Hinzufügen eines Flags, "-FLAGS" für das Entfernen eines Flags und "FLAGS" für das Ersetzen der bisherigen Flagliste.

Für *flag_list* wird eine Liste von Strings erwartet, die die zu setzenden bzw. zu entfernenden Einstellungen enthält. Mögliche Werte sind hier "\Answered", "\Flagged", "\Deleted", "\Seen" und "\Draft". Insbesondere lassen sich mit dem Flag "\Deleted" Nachrichten vom Server löschen.

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("Benutzername", "Passwort")

print("Vorhandene Mailboxen:")
for mb in im.list()[1]:
    name = mb.split(b' "."')[1]
    print(" - {}".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("{}\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.

Auch bei diesem Beispielprogramm wurde keine Fehlerbehandlung durchgeführt. In einem fertigen Programm sollten sowohl die Verbindungsanfrage als auch das Login und insbesondere die Benutzereingabe überprüft werden.

19.4.4 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 herunterladen. 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 solche Zeichen oder Binärdaten verschicken zu können, wurde der sogenannte MIME-Standard (*Multipurpose Internet Mail Extension*) entwickelt, der Sonderzeichen und Binärdaten so kodiert, dass sie als eine Folge reiner ASCII-Zeichen versandt werden können. Durch eine solche Form der Kodierung steigt allerdings die Größe der zu übermittelnden Daten. Zudem definiert der MIME-Standard verschiedene Dateitypen und legt eine Syntax fest, mit der Dateianhänge einem bestimmten Dateityp zugeordnet werden, sodass die Dateien beim Empfänger leichter verarbeitet werden können.

Das `email`-Paket ist sehr mächtig, weswegen hier nur ein Teil seiner Funktionalität besprochen werden kann. Zunächst werden wir uns darum kümmern, wie eine simple ASCII-Mail mittels `email` erstellt werden kann. Darauf aufbauend, werden wir zu komplexeren MIME-kodierten Mails übergehen.

Erstellen einer einfachen E-Mail

Die Klasse `Message` des Moduls `email.message` dient als Basisklasse für eine neue E-Mail. 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())

```

Die Ausgabe des Beispielsprogramms, 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.

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.¹³

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 Beispielsprogramm 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()

```

¹³ Mithilfe des Moduls `smtplib` können der erzeugte E-Mail-String oder die `Message`-Instanz direkt verschickt werden.

```

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 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

iVBORwOKGgoAAAANSUhEUGAAASwAAD8CAIAAABCvg65AAACXBIWXMAACcQAAAnE
B3RJTUUH1wkMERU1+MujjgAAIABJREFUeNrsfXecJWWV9nPet+rezt2ThzDkqCBLB
[...]

--=====0094312333==--

Sie sehen, dass sowohl der Text als auch das Bild in ähnlicher Form kodiert wurden. Die Aufbereitung der beiden Sektionen zum Textteil der E-Mail und zu einem Bild im Anhang erledigt Ihr Mail-Programm. Das mime-Paket bietet auch eine entsprechende Funktionalität an, auf die wir noch zu sprechen kommen werden.

Hier wurden nur `MIMEText` und `MIMEImage` besprochen. Im Folgenden sind alle verfügbaren MIME-Datentypen aufgelistet:

- ▶ `email.mime.application.MIMEApplication` für ausführbare Programme
- ▶ `email.mime.audio.MIMEAudio` für Sounddateien
- ▶ `email.mime.image.MIMEImage` für Grafikdateien
- ▶ `email.mime.message.MIMEMessage` für Message-Instanzen
- ▶ `email.mime.image.MIMEText` für reinen Text

Beim Instanzieren 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 Basisklasse verfügen.

Internationale Zeichensätze

Bisher wurde besprochen, wie der MIME-Standard dazu verwendet werden kann, Binärdaten im Anhang einer E-Mail zu versenden. Beim Text der E-Mail waren wir aber bislang auf die Zeichen des 7-Bit-ASCII-Standards beschränkt. Die Frage ist, wie ein spezielles Encoding innerhalb einer E-Mail verwendet werden kann. Auch dies ermöglicht der MIME-Standard. Das folgende Beispielprogramm 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 Instanziierung 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 Textes 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 `Message`-Instanz zurück, wie die darauffolgende `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 Dateiojekt übergeben müssen.

19.5 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. Der Vorteil des Telnet-Protokolls liegt in seiner Einfachheit, so ist keine große Infrastruktur notwendig, um es einzusetzen. Auf der anderen Seite bietet es aber keine Möglichkeit zur Verschlüsselung der übertragenen Daten und wurde deshalb 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]]])
```

Hiermit wird eine Instanz der Klasse `Telnet` erzeugt. Optional übergeben Sie bereits hier den Hostnamen und den Port des Rechners, 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])

Diese Methode 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()

Diese Methode 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[, timeout]])

Diese Methode 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()

Diese Methode schließt die `Telnet`-Verbindung zum entfernten Rechner.

t.write(buffer)

Diese Methode 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 = "pop.beispiel.de"
port = 110
user = "benutzername"
passwd = "password"

```

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.

Der Server sendet auf `LIST` zwei Antwortstrings. Von diesen interessiert uns nur der zweite, 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.

19.6 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 Programmierers aufgerufen werden, als gehörten sie zum lokalen Programm. Das Übertragen der Funktionsaufrufe und insbesondere der Parameter und des Rückgabewertes wird vollständig von der XML-RPC-Bibliothek übernommen, sodass der Programmierer die Funktionen tatsächlich nur aufzurufen braucht.

Neben XML-RPC existieren weitere mehr oder weniger standardisierte Verfahren zum entfernten Funktionsaufruf. Da aber XML-RPC auf zwei bereits bestehenden Standards, nämlich XML und HTTP, basiert und keine völlig neuen binären Protokolle einführt, ist es vergleichsweise einfach umzusetzen und daher in vielen Programmiersprachen verfügbar.

Da XML-RPC unabhängig von einer bestimmten Programmiersprache entwickelt wurde, ist es 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 können.

19.6.1 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 zugeteilten 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` vorbelegt.

Der vierte Parameter, `allow_none`, ermöglicht es, sofern hier `True` übergeben wird, `None` in XML-RPC-Funktionen zu verwenden. Normalerweise verursacht die Verwendung von `None` eine Exception, da kein solcher Datentyp im XML-RPC-Standard vorgesehen ist. Weil 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 Instanziierung 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 Instanziierung des XML-RPC-Servers folgender Aufruf des Konstruktors:

```
>>> from xmlrpc.server import SimpleXMLRPCServer
>>> srv = SimpleXMLRPCServer(("", 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, alle eingehenden Anfragen zu beantworten. Näheres zum Modul `socketserver` erfahren Sie in Abschnitt 19.1.9 (S. 630).

Methode	Beschreibung	Seite
<code>register_function(function[, name])</code>	Registriert das Funktionsobjekt <i>function</i> für einen RPC-Aufruf.	682
<code>register_instance(instance[, allow_dotted_names])</code>	Registriert die Instanz <i>instance</i> für den entfernten Zugriff.	682
<code>register_introspection_functions()</code>	Registriert einige spezielle Funktionen, die dem Client Informationen über die Schnittstelle liefern.	683
<code>register_multicall_functions()</code>	Erlaubt es dem Client, mehrere Methodenaufrufe zu bündeln.	683

Tabelle 19.18 Methoden der Klasse SimpleXMLRPCServer

Im Folgenden sei *s* eine Instanz der Klasse SimpleXMLRPCServer.

s.register_function(function[, name])

Diese Methode 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 Bindestrich oder ein Punkt.

s.register_instance(instance[, allow_dotted_names])

Diese Methode 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* die dabei angegebenen Parameter.¹⁴

¹⁴ Wenn die Parameter entpackt und an eine Methode weitergereicht werden sollen, kann folgender Methodenaufruf verwendet werden:
`self.methode(*params)`

Eine konkrete Implementierung der Methode `_dispatch`, die die tatsächliche Methode der registrierten Instanz mit dem Namen `method` aufruft und die Parameter übergibt, kann folgendermaßen aussehen:

```
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.

Hinweis

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()`

Diese Methode registriert die Funktionen `system.listMethods`, `system.methodHelp` und `system.methodSignature` für den entfernten Zugriff. Diese Funktionen ermöglichen es einem verbundenen Client, eine Liste aller verfügbaren Funktionen und Informationen zu einzelnen dieser Funktionen zu bekommen.

Näheres zur Verwendung der Funktionen `system.listMethods`, `system.methodHelp` und `system.methodSignature` erfahren Sie in Abschnitt 19.6.2, »Der Client« (S. 685).

`s.register_multicall_functions()`

Diese Methode registriert die Funktion `system.multicall` für den entfernten Zugriff. Durch Aufruf der Funktion `system.multicall` kann der Client mehrere Methodenaufrufe bündeln, um so Traffic zu sparen. Auch die Rückgabewerte der Methodenaufrufe werden gebündelt zurückgegeben.

Näheres zur Verwendung der Funktion `system.multicall` erläutert Abschnitt 19.6.2, »Der Client« (S. 685).

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.¹⁵

```
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. Dann werden die soeben erstellten Funktionen registriert. Schlussendlich wird der Server durch Aufruf der Methode `serve_forever` gestartet und ist nun bereit, eingehende Verbindungsanfragen und Methodenaufrufe entgegenzunehmen und zu bearbeiten.

Der hier vorgestellte Server ist natürlich nur eine Hälfte des Beispielprogramms. Im nächsten Abschnitt werden wir besprechen, wie ein XML-RPC-Client auszusehen hat, und schließlich werden wir am Ende des folgenden Abschnitts einen Client entwickeln, der mit diesem Server kommunizieren kann.

¹⁵ Dieses Szenario ist durchaus sinnvoll, wenn man sich vorstellt, der Server laufe auf einem Rechner, der für diese mathematischen Operationen besonders geeignet ist. Clients könnten diese Berechnungen dann an den Server delegieren.

19.6.2 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]]]])
```

Hiermit wird eine Instanz der Klasse `ServerProxy` erzeugt, die mit dem XML-RPC-Server verbunden ist, den die URI¹⁶ *uri* beschreibt.

An zweiter Stelle kann wie bei der Klasse `SimpleXMLRPCServer` ein 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 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 (S. 460) 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 Instanziierung 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 Aufruf der Methoden *register_introspection_functions* der `SimpleXMLRPCServer`-Instanz.

Im Folgenden sei *s* eine Instanz der Klasse `ServerProxy`.

¹⁶ Ein URI (für *Uniform Resource Identifier*) ist die Verallgemeinerung einer URL.

s.system.listMethods()

Diese Methode gibt die Namen aller beim XML-RPC-Server registrierten entfernten Funktionen in Form einer Liste von Strings zurück. Die Systemmethoden *listMethods*, *methodSignature* und *methodHelp* sind nicht in dieser Liste enthalten.

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.¹⁷ 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)

Diese Methode gibt den Docstring der entfernten Funktion *name* zurück, wenn ein solcher existiert. Wenn kein Docstring gefunden werden konnte, wird ein leerer String zurückgegeben.

Beispiel

Damit ist die Verwendung einer *ServerProxy*-Instanz beschrieben. 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))
```

¹⁷ Dies wird auch *Funktionsüberladung* genannt.

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.

19.6.3 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 genauso einen, wie wir ihn in Abschnitt 19.6.1, »Der Server« (S. 680), 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 werden, als wären es lokale Methoden der `MultiCall`-Instanz. Beachten Sie aber, dass diese Methodenaufrufe keinen sofortigen entfernten Funktionsaufruf zur Folge haben und somit auch zu dieser Zeit keinen Wert zurückgeben.

Im Beispiel werden `fak` und `quad` jeweils zehnmal mit einer fortlaufenden ganzen Zahl aufgerufen.

Durch Aufruf der `MultiCall`-Instanz (`mc()`) werden alle gepufferten entfernten Funktionsaufrufe zusammen an den Server geschickt. Als Ergebnis wird ein Iterator zurückgegeben, der über alle Rückgabewerte in der Reihenfolge des jeweiligen Funktionsaufrufes iteriert. Im Beispielprogramm nutzen wir den Iterator dazu, die Ergebnisse 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.

19.6.4 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, sodass beispielsweise Server und Client auch dann 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 Instanziierung der Server- bzw. Clientklasse explizit angegeben wurde. Das bedeutet natürlich nur, dass Instanzen dieser Datentypen nicht über die XML-RPC-Schnittstelle versendet werden können. 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. Es ist natürlich dennoch möglich, eine komplexe Zahl über eine XML-RPC-Schnittstelle zu schicken, indem Sie Real- und Imaginärteil getrennt jeweils als 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	<i>bool</i>	–
ganze Zahlen	<i>int</i>	–

Tabelle 19.19 Erlaubte Datentypen bei XML-RPC

XML-RPC	Python	Anmerkungen
Gleitkommazahlen	<i>float</i>	–
Strings	<i>str</i>	–
Arrays	<i>list</i>	In der Liste dürfen als Elemente nur XML-RPC-konforme Instanzen verwendet werden.
Strukturen	<i>dict</i>	Alle Schlüssel müssen Strings sein. Als Werte dürfen nur XML-RPC-konforme Instanzen verwendet werden.
Datum/Zeit	<i>DateTime</i>	Der spezielle Datentyp <code>xmlrpc.client.DateTime</code> wird verwendet. ¹⁸
Binärdaten	<i>Binary</i>	Der spezielle Datentyp <code>xmlrpc.client.Binary</code> wird verwendet.

Tabelle 19.19 Erlaubte Datentypen bei XML-RPC (Forts.)

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 und die jeweils referenzierten Instanzen als Werte eingetragen werden. Dies geschieht automatisch. Beachten Sie jedoch, dass das auf der Gegenseite ankommende Dictionary nicht automatisch wieder in eine Instanz der ursprünglichen Klasse umgewandelt wird.

Die letzten beiden Datentypen, die in der Tabelle aufgelistet sind, sind 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 bekannten Datentypen `datetime.date`, `datetime.time` oder `datetime.datetime` verwendet werden.

¹⁸ Dabei handelt es sich nicht um den Datentyp `datetime` aus dem Modul `datetime` der Standardbibliothek.

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 Instanziierung 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 Instanziierung 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.

Kapitel 20

Debugging

»Intelligente Fehler zu machen, ist eine große Kunst.«
– Federico Fellini

Das *Debugging* bezeichnet das Aufspüren und Beseitigen von Fehlern, sogenannten *Bugs*, in einem Programm. Üblicherweise steht dem Programmierer dabei ein *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 oder die Analyse von Programmausgaben.

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.

20

20.1 Der Debugger

Im Lieferumfang von Python ist ein Programm zum Debuggen von Python-Code enthalten, der sogenannte *PDB* (*Python Debugger*). Dieser Debugger läuft in einem Konsolenfenster und ist damit weder übersichtlich noch intuitiv. Aus diesem Grund haben wir uns dagegen entschieden, den PDB an dieser Stelle zu besprechen. Sollten Sie dennoch Interesse an diesem Debugger haben, beispielsweise gerade wegen seiner kommandozeilenbasierenden Benutzerschnittstelle, so finden Sie nähere Informationen dazu in der Python-Dokumentation.

Viele moderne Entwicklungsumgebungen für Python beinhalten einen umfangreichen, integrierten Debugger mit grafischer Benutzeroberfläche, der die Fehlersuche in einem Python-Programm recht komfortabel gestaltet. Auch IDLE bietet einen rudimentären grafischen Debugger:

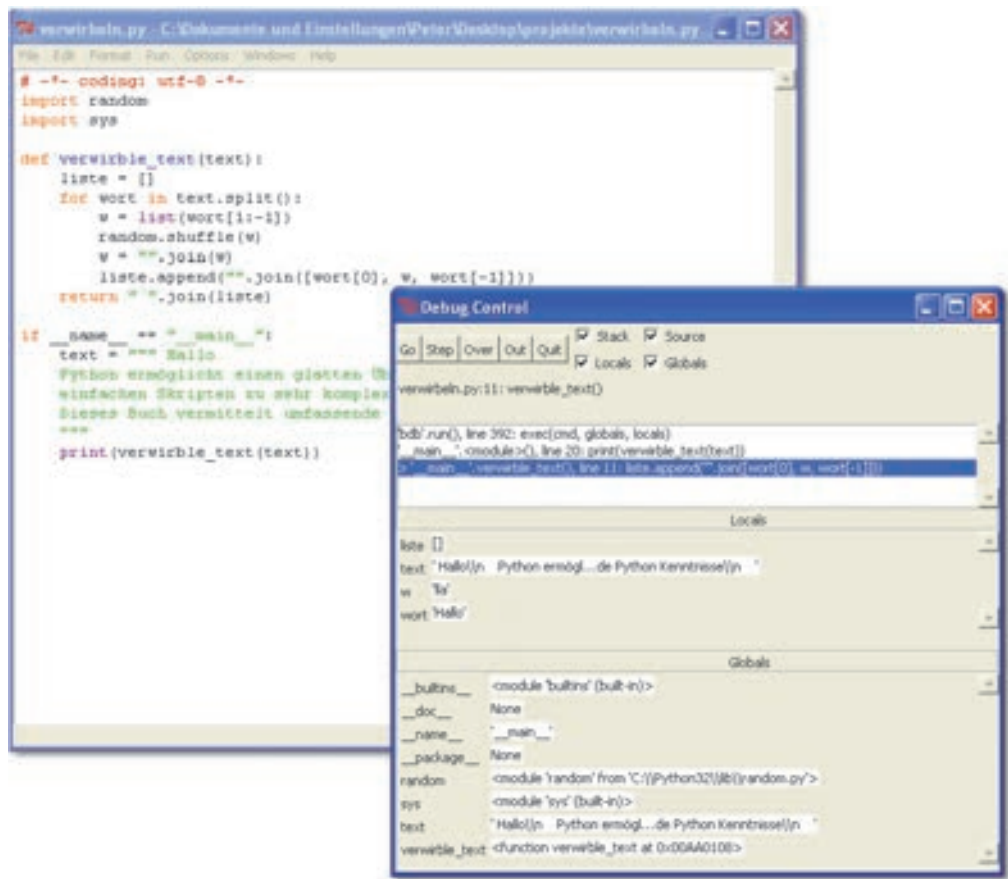


Abbildung 20.1 Der grafische Debugger von IDLE

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, sodass 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, sodass wir allgemein darauf eingehen möchten, welche Funktionen ein grafischer Debugger in der Regel anbietet.

Das grundsätzliche Prinzip eines Debuggers ist es, dem Programmierer das schrittweise Ausführen eines Programms zu ermöglichen, um sich somit von Zeile zu Zeile

ein genaues Bild davon zu machen, welche Änderungen sich ergeben haben und wie sich diese im Laufe des Programms auswirken. Eine *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 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 Werkzeuge 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 eine 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, leichter aufspüren.
- ▶ Zusätzlich zu den lokalen und globalen Referenzen ist der sogenannte *Stack* von Interesse. In diesem wird die momentane Funktionshierarchie aufgelistet, sodass sich genau verfolgen lässt, welche Funktion welche Unterfunktion aufgerufen hat.

- ▶ Gerade in Bezug auf die Programmiersprache Python bieten einige Debugger eine interaktive Shell, die sich im Kontext des angehaltenen Programms befindet und es dem Programmierer erlaubt, komfortabel Referenzen zu verändern, um somit in den Programmfluss einzugreifen.
- ▶ Ein sogenannter *Post-Mortem-Debugger* kann in Anlehnung an den vorherigen Punkt betrachtet werden. In einem solchen Modus hält der Debugger das Programm erst an, wenn eine nicht abgefangene Exception aufgetreten ist. Im angehaltenen Zustand verfügt der Programmierer wieder über eine Shell sowie über die genannten Hilfsmittel, um dem Fehler auf die Spur zu kommen. Diese Form des Debuggens wird »post mortem« genannt, da sie erst nach dem Auftreten des tatsächlichen Fehlers, also nach dem »Tod« des Programms, aktiviert wird.

Mithilfe dieser Einführung in die Techniken des Debuggens und mit ein wenig Experimentierfreude 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.

20.2 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
- ▶ Die folgende Tabelle listet die wichtigsten im Modul `inspect` enthaltenen Funktionen auf und erläutert diese kurz. Eine detailliertere Erklärung finden Sie jeweils auf der angegebenen Seite.

Funktion	Beschreibung	Seite
Datentypen, Attribute und Methoden		
<i>getmembers(object[, predicate])</i>	Gibt alle Attribute und Methoden der Instanz <i>object</i> in Form einer Liste von Tupeln zurück.	696
<i>isbuiltin(object)</i>	True, wenn <i>object</i> eine Built-in Function ist, andernfalls False	–
<i>isclass(object)</i>	True, wenn <i>object</i> eine Klasse ist, andernfalls False	–
<i>isfunction(object)</i>	True, wenn <i>object</i> eine Klasse ist, andernfalls False	–
<i>ismethod(object)</i>	True, wenn <i>object</i> eine Klasse ist, andernfalls False	–
<i>ismodule(object)</i>	True, wenn <i>object</i> eine Klasse ist, andernfalls False	–
<i>isroutine(object)</i>	True, wenn <i>object</i> eine Klasse ist, andernfalls False	–
<i>istraceback(object)</i>	True, wenn <i>object</i> eine Klasse ist, andernfalls False	–
Quellcode		
<i>getfile(object)</i>	Gibt den Namen der Datei zurück, in der <i>object</i> definiert wurde.	697
<i>getmodule(object)</i>	Gibt das Modul zurück, in dem <i>object</i> definiert wurde.	698
<i>getsourcefile(object)</i>	Gibt den Namen der Quellcode-Datei zurück, in der <i>object</i> definiert wurde.	698
<i>getsourcelines(object)</i>	Gibt den Quellcode der Definition von <i>object</i> als Liste von Codezeilen zurück.	698
<i>getsource(object)</i>	Gibt den Quellcode der Definition von <i>object</i> als String zurück.	699

Tabelle 20.1 Funktionen des Moduls inspect

Funktion	Beschreibung	Seite
Klassen und Funktionen		
<i>getclasstree(classes[, unique])</i>	Gibt die Vererbungshierarchie der übergebenen Klassen in Form eines Baums zurück.	699
<i>getmro(cls)</i>	Gibt die Basisklassen der Klasse <i>cls</i> zurück.	701
<i>getfullargspec(func)</i>	Gibt die Funktionsschnittstelle von <i>func</i> zurück.	701
<i>getargvalues(frame)</i>	Gibt die einer Funktion übergebenen Parameterwerte zurück.	703

Tabelle 20.1 Funktionen des Moduls inspect (Forts.)

Bevor Sie die Beispiele dieser Abschnitte ausführen können, müssen Sie das Modul `inspect` einbinden:

```
>>> import inspect
```

20.2.1 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.

getmembers(object[, predicate])

Diese Funktion 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 jedes Member der Instanz *object* aufgerufen und bekommt diesen als einzigen Parameter übergeben. Es werden alle Member in die Ergebnisliste aufgenommen, für die die Filterfunktion den Wert `True` zurückgegeben hat.

```
>>> inspect.getmembers(klasse(), inspect.ismethod)
[('__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 die Funktion *ismethod* übergeben, die genau dann `True` zurückgibt, wenn es sich bei dem Member um eine Methode handelt. Dementsprechend klein ist die Ergebnisliste.

20.2.2 Quellcode

In diesem Abschnitt werden die wichtigsten Funktionen des Moduls `inspect` behandelt, die sich direkt auf den Quellcode beispielsweise einer Funktion beziehen.

20

`getfile(object)`

Die Funktion *getfile* 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:\\Python32\\lib\\inspect.py'
```

In diesem Beispiel wurde die Funktion *getfile* verwendet, um herauszufinden, in welcher Quelldatei sie selbst definiert ist.

getmodule(object)

Diese Funktion gibt die Modulinstanz des Moduls zurück, in dem das Objekt *object* definiert wurde.

```
>>> inspect.getmodule(inspect.getmodule)
<module 'inspect' from 'C:\Python32\lib\inspect.py'>
```

In diesem Beispiel wurde *getmodule* dazu verwendet, den Pfad des Moduls herauszufinden, in dem die Funktion selbst definiert ist.

getsourcefile(object)

Diese Funktion 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:\\Python32\\lib\\inspect.py'
```

In diesem Fall wurde *getsourcefile* dazu verwendet, die Quellcodedatei herauszufinden, in der die Funktion selbst definiert ist.

getsourcelines(object)

Diese Funktion 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 Abschnitt.

Die Funktion wirft eine *IOError*-Exception, wenn der Quellcode zum Objekt *object* nicht geladen werden konnte.

```
>>> inspect.getsourcelines(inspect.getsourcelines)
(['def getsourcelines(object):\n', [...]
 '    lines, lnum = findsource(object)\n', '\n',
 '    if ismodule(object): return lines, 0\n',
 '    else: return getblock(lines[lnum:]), lnum+1\n'], 678)
```

In diesem Fall wurde die Funktion *getsourcelines* dazu verwendet, die Quellcodezeilen ihrer eigenen Definition zurückzugeben. Das Auslassungszeichen deutet keine rekursive Liste an, sondern steht für den hier nicht abgedruckten Docstring.

getsource(object)

Diese Funktion 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.

20.2.3 Klassen und Funktionen

Dieser Abschnitt behandelt die wichtigsten Funktionen des Moduls *inspect*, die sich auf Klassen und Funktionen beziehen.

getclasstree(classes[, unique])

Diese Funktion gibt die Vererbungshierarchie der übergebenen Klassen in Form eines Baums¹ zurück. Um dies zu demonstrieren, bauen wir zunächst eine Klassenhierarchie auf:

```
>>> class a:
...     pass
...
>>> class b(a):
...     pass
...
>>> class c(a):
...     pass
...
>>> class d(b):
...     pass
...
>>> class e(b):
...     pass
...
```

20

¹ Ein Baum ist eine, ausgehend von einer Wurzel, verzweigende, zyklensfreie Hierarchie. Im Zusammenhang mit *getclasstree* wird ein Baum durch verschachtelte Listen abgebildet.

Abbildung 20.2 veranschaulicht die Hierarchie.

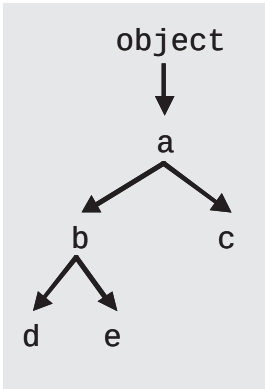


Abbildung 20.2 Die Klassenhierarchie

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'>,))
  ]
]
```

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 Basis-klasse `object` von `a`.

Wenn für den optionalen Parameter *unique* der Wert `True` übergeben wird, taucht jede Klasse auch im Falle von Mehrfachvererbung nur ein einziges Mal im Klassenbaum auf.

getmro(cls)

Diese Funktion 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.

getfullargspec(func)

Die Funktion 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. Diese Attribute können wie bei einem Tupel auch über einen Index angesprochen werden.

Index	Attribut	Beschreibung
0	args	eine Liste, die die Namen aller Positionsparameter enthält
1	varargs	der Name, über den zusätzliche Positionsparameter funktionsintern angesprochen werden
2	varkw	der Name des Parameters, über den zusätzliche Schlüsselwortparameter funktionsintern angesprochen werden
3	defaults	ein Tupel, das die Defaultwerte der Positionsparameter enthält, sofern diese vorhanden sind
4	kwonlyargs	eine Liste, die die Namen aller reinen Schlüsselwortparameter enthält ²
5	kwonlydefaults	ein Tupel, das die Defaultwerte der reinen Schlüsselwortparameter enthält
6	annotations	ein Dictionary, das die in der Funktionsschnittstelle definierten Function Annotations enthält

Tabelle 20.2 Attribute einer inspect.ArgSpec-Instanz

Zur näheren Erläuterung dient folgendes Beispiel einer komplexen Funktionsschnittstelle:

```
>>> def funktion(a : "Hallo", b=1, c=2, *, d=3, e=4, **args):
...     pass
...
>>> inspect.getfullargspec(funktion)
FullArgSpec(args=['a', 'b', 'c'], varargs=None, varkw='args',
defaults=(1, 2), kwonlyargs=['d', 'e'], kwonlydefaults={'e': 4, 'd': 3},
annotations={'a': 'Hallo'})
```

Die Funktionsschnittstelle besteht aus den positionsbezogenen Parametern a, b und c sowie den reinen Schlüsselwortparametern d und e. Zusätzlich können weitere Schlüsselwortparameter angegeben werden, welche intern über den Namen args zugänglich sind. Eine Möglichkeit, beliebige Positionsparameter zu übergeben, gibt es nicht, das entsprechende Element des FullArgSpec-Tupels ist None. Schlussendlich wurde für den Parameter a eine Annotation festgelegt, die sich im annotations-Dictionary des FullArgSpec-Tupels wiederfindet.

2 Näheres zu reinen Schlüsselwortparametern erfahren Sie in Abschnitt 9.2.4 (S. 219).

getargvalues(frame)

Diese Funktion 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	<code>args</code>	eine Liste der für die Positionsparameter übergebenen Werte
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>locals</code>	ein Dictionary, das die Namen und Werte aller lokalen Variablen des übergebenen Frame-Objekts <i>frame</i> enthält

Tabelle 20.3 Attribute einer `inspect.ArgInfo`-Instanz

Das sogenannte *Frame-Objekt* repräsentiert einen Codeblock und kann mithilfe der Funktion `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})
```

20

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.

20.3 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.

Funktion	Beschreibung	Seite
<code>pprint(object[, stream[, indent[, width[, depth]]]])</code>	Gibt die Instanz <i>object</i> , formatiert auf dem Stream <i>stream</i> aus.	704
<code>pformat(object[, indent[, width[, depth]]])</code>	wie <code>pprint</code> , mit dem Unterschied, dass die formatierte Repräsentation von <i>object</i> als String zurückgegeben statt in einen Stream geschrieben wird	–
<code>isreadable(object)</code>	Gibt <code>True</code> zurück, wenn die formatierte Ausgabe der Instanz <i>object</i> lesbar ist.	705
<code>isrecursive(object)</code>	Gibt <code>True</code> zurück, wenn die Instanz <i>object</i> rekursiv ist.	706

Tabelle 20.4 Funktionen des Moduls `pprint`

Um die Beispiele ausführen zu können, muss zuvor das Modul `pprint` eingebunden werden:

```
>>> import pprint
```

`pprint(object[, stream[, indent[, width[, depth]]]])`

Die Funktion `pprint` 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*, *width* 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:\\WINDOWS\\system32\\python32.zip',
```

```
'C:\\Python32\\DLLs',
'C:\\Python32\\lib',
'C:\\Python32',
'C:\\Python32\\lib\\site-packages']
```

Zum Vergleich geben wir `sys.path` noch einmal unformatiert mit `print` aus:

```
>>> print(sys.path)
['', 'C:\\WINDOWS\\system32\\python32.zip', 'C:\\Python32\\DLLs',
'C:\\Python32\\lib', 'C:\\Python32', 'C:\\Python32\\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:\\WINDOWS\\system32\\python32.zip',
           'C:\\Python32\\DLLs',
           'C:\\Python32\\lib',
           'C:\\Python32',
           'C:\\Python32\\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. Die Schlüssel-Wert-Paare eines Dictionarys werden bei der Ausgabe mit `pprint` nach den Schlüsseln sortiert.

isreadable(object)

Diese Funktion 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 den Wert `True` zurück:

```
>>> pprint.isreadable(sys.path)
True
>>> a = []
>>> a.append(a)
>>> pprint.isreadable(a)
False
```

Im zweiten Teil des Beispiels wurde `False` zurückgegeben, weil es sich um eine rekursive Liste handelt, die sich selbst als Element enthält. Eine solche Liste kann aus naheliegenden Gründen nicht vollständig ausgegeben werden. Auch Instanzen selbst erstellter Klassen können nicht lesbar repräsentiert werden. Letzteres liegt daran, dass die Ausgabe dann den Code zum Erstellen der Klasse sowie der Instanz enthalten müsste, was keinen Sinn ergibt.

isrecursive(object)

Diese Funktion 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 des Moduls `pprint` erläutert, und wir werden uns im Folgenden mit dem Modul `logging` beschäftigen.

20.4 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 Zeitstempel 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")
```

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
NOTSET	keine Dringlichkeitsstufe
DEBUG	eine Meldung, die nur für den Programmierer zur Fehlersuche interessant ist
INFO	eine Informationsmeldung über den Programmstatus
WARNING	eine Warnmeldung, die auf einen möglichen Fehler hinweist
ERROR	eine Fehlermeldung, nach der das Programm weiterarbeiten kann
CRITICAL	eine Meldung über einen kritischen Fehler, der das sofortige Beenden des Programms oder der aktuell durchgeführten Operation zur Folge hat

Tabelle 20.5 Vordefinierte Dringlichkeitsstufen

Aus Gründen des Komforts existiert zu jeder Dringlichkeitsstufe eine eigene Funktion. So sind die beiden Funktionsaufrufe von *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 Debug- 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
```

Die folgende Tabelle listet die wichtigsten Schlüsselwortparameter auf, die der Funktion *basicConfig* übergeben werden können.

Parameter	Beschreibung
<i>datefmt</i>	Spezifiziert das Datumsformat. Näheres dazu erfahren Sie in Abschnitt 20.4.1.
<i>filemode</i>	Gibt den Modus ³ an, in dem die Logdatei geöffnet werden soll. Defaultwert: "a"
<i>filename</i>	Gibt den Dateinamen der Logdatei an.

Tabelle 20.6 Schlüsselwortparameter der Funktion *basicConfig*

³ Die verschiedenen Modi, in denen Dateien geöffnet werden können, können Sie in Abschnitt 8.4 (S. 204) nachlesen.

Parameter	Beschreibung
<i>format</i>	Spezifiziert das Meldungsformat. Näheres dazu erfahren Sie in Abschnitt 20.4.1.
<i>level</i>	Legt die Mindestdringlichkeit für Meldungen fest, damit diese in der Logdatei erscheinen.
<i>stream</i>	Gibt einen Stream an, in den die Logmeldungen geschrieben werden sollen. Wenn die Parameter <i>stream</i> und <i>filename</i> gemeinsam angegeben werden, wird <i>stream</i> ignoriert.
<i>style</i>	Bestimmt die Formatierungssyntax für die Meldung. Der voreingestellte Wert "%" bedingt die alte %-Syntax aus Python 2, während ein Wert von "{" die neue Syntax zur Stringformatierung ⁴ erzwingt.

Tabelle 20.6 Schlüsselwortparameter der Funktion `basicConfig` (Forts.)

20.4.1 Das Meldungsformat anpassen

Zu Beginn von Abschnitt 20.4 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 eine Möglichkeit, das Format der geloggtten Meldung anzupassen. Dazu übergeben Sie beim Funktionsaufruf von `basicConfig` den Schlüsselwortparameter `format`:

```
logging.basicConfig(
    filename="programm.log",
    level = logging.DEBUG,
    style = "{",
    format = "{asctime} [{levelname:8}] {message}")
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. Dabei stehen die Bezeichner `asctime` für den Timestamp, `levelname` für die Dringlichkeitsstufe und `message` für die Meldung. Die von diesem Beispiel generierten Meldungen sehen folgendermaßen aus:

⁴ Näheres zur Stringformatierung können Sie in Abschnitt 7.5.4 (ab S.156) nachlesen.

```
2012-04-28 03:28:55,811 [ERROR   ] Ein Fehler ist aufgetreten
2012-04-28 03:29:00,690 [INFO    ] Dies ist eine Information
2012-04-28 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. Je nach Kontext, in dem die Meldung erzeugt wird, haben einige der Bezeichner keine Bedeutung.

Bezeichner	Beschreibung
asctime	Zeitpunkt der Meldung. Das Datums- und Zeitformat kann beim Funktionsaufruf von <i>basicConfig</i> über den Parameter <i>datefmt</i> angegeben werden. Näheres dazu folgt im Anschluss an diese Tabelle.
filename	der Dateiname der Programmdatei, in der die Meldung abgesetzt wurde
funcName	der Name der Funktion, in der die Meldung abgesetzt wurde
levelname	die Dringlichkeitsstufe der Meldung
lineno	die Quellcodezeile, in der die Meldung abgesetzt wurde
message	der Text der Meldung
module	Der Name des Moduls, in dem die Meldung abgesetzt wurde. Der Modulname entspricht dem Dateinamen ohne Dateiendung.
pathname	der Pfad zur Programmdatei, in der die Meldung abgesetzt wurde
process	die ID des Prozesses, in dem die Meldung abgesetzt wurde
thread	die ID des Threads, in dem die Meldung abgesetzt wurde

Tabelle 20.7 Bezeichner im Formatstring

Was uns an diesen Meldungen noch stört, ist das Format des Zeitstempels. Zum einen wird das amerikanische Datumsformat verwendet, und zum anderen ist eine Auflösung bis auf die Millisekunde für unsere Zwecke etwas 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,
    style = "{",
```

```
format = "{asctime} [{levelname:8}] {message}",
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 15.1, »Elementare Zeitfunktionen – time« (S. 451), eingeführt. Die von diesem Beispiel erzeugte Meldung sieht folgendermaßen aus:

```
28.04.2012 03:38:49 ERROR: Ein Fehler ist aufgetreten
```

20.4.2 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} {levelname}: {message}",
                        "%d.%m.%Y %H:%M:%S", style="{")
handler.setFormatter(frm)

logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.critical("Ein wirklich kritischer Fehler")
logger.warning("Und eine Warnung hinterher")
logger.info("Dies hingegen ist nur eine Info")
```

Zunächst wird der Handler, in diesem Fall ein `StreamHandler`, instanziiert. Im nächsten Schritt wird eine Instanz der Klasse `Formatter` erzeugt. Diese Klasse kapselt die

Formatierungsanweisungen, die wir in den vorherigen Beispielen beim Aufruf der Funktion *basicConfig* übergeben haben. Mithilfe der Methode *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 *getLogger*. Danach wird über *addHandler* der Handler hinzugefügt und über *setLevel* die gewünschte Dringlichkeitsstufe eingestellt.

Die Meldungen werden im Folgenden nicht über Funktionen des Moduls *logging*, sondern über die Methoden *critical*, *warning* und *info* der Logger-Instanz *logger* abgesetzt. Das Beispielprogramm gibt folgenden Text auf dem Bildschirm aus:

```
28.04.2012 17:21:46 CRITICAL: Ein wirklich kritischer Fehler
28.04.2012 17:21:46 WARNING: Und eine Warnung hinterher
28.04.2012 17:21:46 INFO: Dies hingegen ist nur eine Info
```

Im Folgenden sollen die wichtigsten zusätzlichen Handler-Klassen beschrieben werden, die im Paket *logging* bzw. *logging.handlers* 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. Der Handler *FileHandler* kann auch implizit durch Angabe der Schlüsselwortparameter *filename* und *filemode* beim Aufruf der Funktion *basicConfig* verwendet werden.

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*.

`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.

20.5 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 Ihnen 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.

20.5.1 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 neuen Definitionssprache verfasst, sondern können direkt aus einer Sitzung im interaktiven Modus in den Docstring kopiert werden.

Docstrings sind auch bzw. hauptsächlich für die Dokumentation beispielsweise einer Funktion gedacht. Aus diesem Grund sollten Sie die Testfälle im Docstring möglichst einfach und lehrreich halten, sodass der resultierende Docstring auch in Dokumentationen Ihres Programms verwendet werden kann.

Das folgende Beispiel 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, sodass 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 ist zu 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 keiner garantierten Reihenfolge, sodass Sie innerhalb eines Doctests nie ein Dictionary als Ergebnis ausgeben sollten. Des Weiteren gibt es Informationen, die vom Interpreter oder anderen Gegebenheiten abhängen; beispielsweise entspricht die Identität einer Instanz intern ihrer Speicheradresse und wird sich deswegen natürlich beim Neustart des Programms ändern.

Eine weitere Besonderheit, auf die Sie achten müssen, ist, dass eine Leerzeile in der erwarteten Ausgabe einer Funktion durch den String `<BLANKLINE>` gekennzeichnet werden muss, da eine Leerzeile als Trennung zwischen Testfällen und Dokumentation fungiert:

```
def f(a, b):
    """
    >>> f(3, 4)
    7
    <BLANKLINE>
    12
    """
    print(a + b)
    print()
    print(a * b)
```



Hinweis

Mit den Python-Versionen 3.1 und 2.7 wurde die Darstellung von Gleitkommazahlen verändert. Dies kann dazu führen, dass Doctests, die für ältere Python-Versionen geschrieben wurden, fehlschlagen, wenn sie in einem aktuellen Interpreter ausgeführt werden.

Flags

Um einen Testfall genau an Ihre Bedürfnisse anzupassen, können Sie sogenannte *Flags* vorgeben. 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 20.8 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.

20.5.2 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 mit `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.assertEqual(fak.fak(5), 120)
        self.assertEqual(fak.fak(10), 3628800)
        self.assertEqual(fak.fak(20), 2432902008176640000)

    def testAusnahmen(self):
        self.assertRaises(ValueError, fak.fak, -1)

if __name__ == "__main__":
    unittest.main()
```

Es wurde eine Klasse namens `MeinTest` erzeugt, welche 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 `assertEqual` bzw. `assertRaises` 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.assertEqual(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 in der ersten Tabelle vorgestellten Methoden können in einer von `TestCase` abgeleiteten Klasse implementiert werden.

Methode	Bedeutung
<code>setUp()</code>	Wird vor jedem Aufruf einer der Testmethoden gerufen und kann somit für den Test benötigten Initialisierungscode enthalten. Eine in der Methode <code>setUp</code> geworfene Exception wird als Fehler in den Testbericht eingetragen und der Test abgebrochen.
<code>tearDown()</code>	Wird nach jedem Aufruf einer der Testmethoden gerufen und kann somit abschließenden Code enthalten. Eine in der Methode <code>tearDown</code> geworfene Exception wird als Fehler in den Testbericht eingetragen.

Tabelle 20.9 Methoden der Klasse `TestCase`

Jetzt folgt eine Zusammenfassung der Methoden, die die Klasse `TestCase` zur Durchführung der Tests bereitstellt. Zwei dieser Funktionen, nämlich `assertEqual` und `assertRaises` kennen Sie bereits aus dem vorangegangenen Beispiel.

Die Methoden verfügen alle über den optionalen Parameter `msg`, für den eine Fehlerbeschreibung angegeben werden kann, die im Falle eines fehlschlagenden Tests ausgegeben wird.

Methode	Testet auf
<code>assertEqual(first, second[, msg])</code>	<code>first == second</code>
<code>assertNotEqual(first, second[, msg])</code>	<code>first != second</code>
<code>assertTrue(expr[, msg])</code>	<code>bool(expr) is True</code>
<code>assertFalse(expr[, msg])</code>	<code>bool(expr) is False</code>

Tabelle 20.10 Methoden der Klasse `TestCase`

Methode	Testet auf
<code>assertIs(first, second[, msg])</code>	first is second
<code>assertIsNot(first, second[, msg])</code>	first is not second
<code>assertIsNone(expr[, msg])</code>	expr is None
<code>assertIsNotNone(expr[, msg])</code>	expr is not None
<code>assertIn(first, second[, msg])</code>	first in second
<code>assertNotIn(first, second[, msg])</code>	first not in second
<code>assertIsInstance(obj, cls[, msg])</code>	<code>isinstance(obj, cls)</code>
<code>assertNotIsInstance(obj, cls[, msg])</code>	not <code>isinstance(obj, cls)</code>
<code>assertRaises(exception, callable, *args, **kwargs)</code>	<code>callable(*args, **kwargs)</code> wirft exception-Exception.
<code>assertRaisesRegex(exception, regex, callable, *args, **kwargs)</code>	Wie <code>assertRaises</code> , die Fehlermeldung der Exception muss aber zusätzlich noch auf den regulären Ausdruck <i>regex</i> passen. Dieser kann als String oder in Form eines RE-Objekts ⁵ übergeben werden.
<code>assertWarns(warning, callable, *args, **kwargs)</code>	wie <code>assertRaises</code> , nur für Warnungen
<code>assertWarnsRegex(warning, regex, callable, *args, **kwargs)</code>	wie <code>assertRaisesRegex</code> , nur für Warnungen
<code>assertAlmostEqual(first, second[, places[, msg[, delta]]])</code>	<code>round(first - second, places) == 0</code> bzw. <code>abs(first - second) < delta</code>
<code>assertNotAlmostEqual(first, second[, places[, msg[, delta]]])</code>	<code>round(first - second, places) != 0</code> bzw. <code>abs(first - second) >= delta</code>
<code>assertGreater(first, second[, msg])</code>	first > second
<code>assertGreaterEqual(first, second[, msg])</code>	first >= second

Tabelle 20.10 Methoden der Klasse TestCase (Forts.)

⁵ Siehe Abschnitt 14.1 (S. 432).

Methode	Testet auf
<code>assertLess(first, second[, msg])</code>	<code>first < second</code>
<code>assertLessEqual(first, second[, msg])</code>	<code>first <= second</code>
<code>assertRegex(text, regex[, msg])</code>	Der reguläre Ausdruck <i>regex</i> passt auf den String <i>text</i> . Der reguläre Ausdruck kann als String oder RE-Objekt übergeben werden.
<code>assertNotRegex(text, regex[, msg])</code>	Der reguläre Ausdruck <i>regex</i> passt nicht auf den String <i>text</i> . Der reguläre Ausdruck kann als String oder RE-Objekt übergeben werden.
<code>assertCountEqual(first, second[, msg])</code>	Die Sequenzen <i>first</i> und <i>second</i> enthalten die gleichen Elemente, unabhängig von deren Reihenfolge.

Tabelle 20.10 Methoden der Klasse TestCase (Forts.)

Die Methoden *assertRaises*, *assertRaisesRegex*, *assertWarns* und *assertWarnsReges* können auch ohne die Parameter *callable*, **args* und ***kwargs* aufgerufen werden. In diesem Fall geben sie ein Kontextobjekt zurück, das mit einer *with*-Anweisung verwendet werden kann:

```
with self.assertRaises(TypeError):  
    Code
```

Der Vorteil dieser Schreibweise ist, dass der zu testende Code nicht extra in eine Funktion gekapselt werden muss.

20.6 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.

Funktion	Bedeutung	Seite
<i>print_tb</i> (<i>traceback</i> [, <i>limit</i> [, <i>file</i>]])	Gibt den Stacktrace des Traceback-Objekts <i>traceback</i> aus.	724
<i>print_exception</i> (<i>type</i> , <i>value</i> , <i>traceback</i> [, <i>limit</i> [, <i>file</i> [, <i>chain</i>]]])	Gibt einen Traceback aus.	724
<i>print_exc</i> ([[<i>limit</i> [, <i>file</i> [, <i>chain</i>]]]])	Gibt den Traceback der aktuell gefangenen Exception aus.	725
<i>format_exc</i> ([[<i>limit</i> [, <i>chain</i>]]])	Wie <i>print_exc</i> , gibt den Traceback aber als String zurück.	–
<i>print_last</i> ([[<i>limit</i> [, <i>file</i> [, <i>chain</i>]]]])	Wie <i>print_exception</i> , jedoch immer für die zuletzt gefangene Exception. Diese Funktion kann auch außerhalb eines <code>except</code> -Zweiges gerufen werden.	–
<i>print_stack</i> ([[<i>f</i> [, <i>limit</i> [, <i>file</i>]]]])	Gibt einen Stacktrace des Kontextes aus, in dem <i>print_stack</i> aufgerufen wird. Die Parameter <i>limit</i> und <i>file</i> verstehen sich wie bei <i>print_exception</i> . Über <i>f</i> kann ein Frame-Objekt übergeben werden, das statt des aktuellen Kontextes verwendet wird.	–
<i>extract_tb</i> (<i>traceback</i> [, <i>limit</i>])	Gibt den Stacktrace des Traceback-Objekts <i>traceback</i> in Form einer Liste zurück.	725
<i>extract_stack</i> ([[<i>f</i> [, <i>limit</i>]]]])	Wie <i>print_stack</i> , gibt das Ergebnis aber als Liste von Tupeln zurück, statt es auszugeben.	–
<i>format_list</i> (<i>lst</i>)	Konvertiert eine Liste <i>lst</i> von Tupeln, wie sie von <i>extract_tb</i> oder <i>extract_stack</i> zurückgegeben wird, in eine Liste von Strings.	–
<i>format_exception</i> (<i>type</i> , <i>value</i> , <i>tb</i> [, <i>limit</i> [, <i>chain</i>]]])	Formatiert eine Exception in eine Liste von Strings.	726
<i>format_exception_only</i> (<i>type</i> , <i>value</i>)	wie <i>format_exception</i> , jedoch ohne Berücksichtigung des Traceback-Objekts	–
<i>format_tb</i> (<i>tb</i> [, <i>limit</i>])	äquivalent zu <code>format_list(extract_tb(tb, limit))</code> .	–

Tabelle 20.11 Funktionen des Moduls traceback

Alle Beispiele werden 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 wird ein `Traceback`-Objekt `tb` erzeugt, das den Callstack einer `TypeError`-Exception beschreibt, die zuvor aus einer verschachtelten Funktionshierarchie heraus geworfen wurde.

print_tb(traceback[, limit[, file]])

Diese Funktion 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
```

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 Parameter *type* und *value* müssen `Exception`-Typ und `Exception`-Wert übergeben werden. Die restlichen Parameter haben dieselbe Bedeutung wie bei `print_tb`.

```
>>> 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* den Wert `False` übergeben, werden die Attribute `__cause__` und `__context__` einer Exception nicht mit in die Traceback-Ausgabe aufgenommen. Der Parameter *chain* ist mit `True` vorbelegt.

print_exc([limit[, file[, chain]]])

Diese Funktion arbeitet wie *print_exception*, jedoch immer für die aktuell abgefangene Exception. Die Parameter *limit*, *file* und *chain* haben dieselbe Bedeutung wie bei *print_exception*. Diese Funktion kann nur innerhalb eines `except`-Zweiges aufgerufen werden.

```
>>> try:
...     raise TypeError
... except TypeError:
...     traceback.print_exc()
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError
```

extract_tb(traceback[, limit])

Diese Funktion 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)]
```

format_list(list)

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']
```

format_exception(type, value, tb[, limit[, chain]])

Diese Funktion 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']
```

20.7 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 sind das Thema der nächsten Abschnitte.

20.7.1 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` instanziiert werden. Der Konstruktor der Klasse `Timer` hat folgende Schnittstelle:

```
Timer([stmt[, setup[, timer]])
```

Hiermit wird eine Instanz der Klasse `Timer` erzeugt. 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 wurde, 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 möglichst unabhängig 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]])

Diese Methode 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 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 bedeutet, 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 annäherungsweise deckungsgleich mit denen der vorherigen Tests:

```
Iterativ: 0.162111997604
Rekursiv: 0.272562026978
```

Die absoluten Zahlenwerte hängen stark vom verwendeten System ab. Auf einem schnelleren Computer sind sie dementsprechend kleiner.

20.7.2 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.



Hinweis

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. Der Profiler `cProfile` ist möglicherweise nicht für alle Python-Interpreter verfügbar. Das reine Python-Pendant `profile` hingegen kann überall verwendet werden.

Verwendung des Moduls

Im Modul `cProfile` sind zwei wichtige Funktionen enthalten, die im Folgenden besprochen werden.

`run(command[, filename[, sort]])`

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.

Der Parameter *sort* steuert, wonach die Ergebnistabelle sortiert werden soll. Die folgenden Werte sind möglich:

Wert	Sortierung nach
"stdname"	Programmdatei und Funktionsname. Dies ist der Defaultwert.
"calls"	Anzahl der Aufrufe der Funktion
"time"	Gesamtzeit, die in der Funktion verbracht wurde
"cumulative"	Gesamtzeit, die in der Funktion und ihren Unterfunktionen verbracht wurde.

Tabelle 20.12 Mögliche Werte für den Parameter *sort*

`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 wird eine Laufzeitanalyse für ein kleines Beispielprogramm erstellt. 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 diese 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. Die Codezeile `programm()` des Beispielprogramms ist jetzt überflüssig. 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 ist man gut damit beraten, alle Ressourcen in die Optimierung der Funktion `calc3` zu stecken, da diese mit 1.000.000 Aufrufen und 73 % Laufzeitanteil stark dominiert.

20.7.3 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 laufzeitechnisch 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` der Wert `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. Diese Zeilen wurden natürlich nur nicht ausgeführt, solange die Überwachung des Programms aktiv war.



Hinweis

Es mag zunächst verwirrend erscheinen, dass die Zeile

```
for i in range(100):
```

insgesamt 101-mal ausgeführt wird, obwohl die Schleife von 0 bis 99 zählt. Das liegt daran, dass der Kontrollfluss nach Abarbeitung des 100. Schleifendurchlaufs noch einmal zum Schleifenkopf zurückkehrt, um die Abbruchbedingung zu prüfen. Nach diesem 101. Ausführen des Schleifenkopfes verlässt der Kontrollfluss die Schleife.

Analog erklären sich die Werte 10100 und 1010000 in den Zeilen 7 und 9 der Überdeckungsanalyse.

Teil IV

Weiterführende Themen

Herzlich willkommen zum vierten Teil dieses Buches. Zu diesem Zeitpunkt sollten 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 die Distribution und Optimierung von Python-Programmen, die Programmierung grafischer Benutzeroberflächen und die Interoperabilität zwischen Python und C. Am Ende des vierten Teils finden Sie noch ein Kapitel zur Migration von Python 2 nach Python 3.

Die in diesem Teil enthaltenen Kapitel sind in sich geschlossen und können unabhängig voneinander nach Interesse gelesen werden.

Kapitel 21

Distribution von Python-Projekten

»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

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 das Modul `distutils` enthalten, mit dem sich fertige Distributionen Ihres Programms oder Moduls erstellen lassen.

21.1 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 kann durch das Paket `distutils` automatisiert und standardisiert 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 herunterzula-

den, zu entpacken und das Installationsscript zu starten. Der Vorteil einer Quellcodedistribution ist ihre Plattformunabhängigkeit. Es muss 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.

21.1.1 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. Während ein Modul aus nur einer Programmdatei besteht, ist ein Paket ein Ordner, der mehrere Untermodule oder -pakete enthalten kann. Ein Paket erkennt man an der Programmdatei `__init__.py` im Paketverzeichnis. Die Unterscheidung der Begriffe

¹ Eigentlich handelt es sich dabei nicht um einen Arbeitsschritt zum Erstellen einer Distribution; dennoch ist es einleuchtenderweise eine unverzichtbare Voraussetzung.

Sie können auch mehrere Module und/oder Pakete in eine gemeinsame Distribution verpacken. Näheres dazu erfahren Sie im Laufe dieses Abschnitts.

»Modul« und »Paket« wird beim Erstellen des Installationsscripts noch eine Rolle spielen.

An dieser Stelle soll das Beispielm modul 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 Beispielm oduls 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 Reiehnfogle die Bchustebaen in Woeretrn vokrmomen.

Es ist nur withcig, dsas der ertse und lettze Bchusatbe an der ricthgien Stele 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 die Liste der erzeugten Wörter `liste` angehängt. Am Schluss wird diese Wortliste zu einem Text zusammengefügt und zurückgegeben.

Die Funktion geht der Einfachheit halber von einem absolut gutartigen String aus. 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.

21.1.2 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 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. Die folgende Tabelle listet die wichtigsten möglichen Argumente auf und klärt kurz ihre Bedeutung.

Wenn nichts anderes angegeben ist, handelt es sich bei den jeweiligen Parametern um Strings.

Parametername	Beschreibung
<i>name</i>	der Name der Distribution
<i>version</i>	die Versionsnummer der Distribution
<i>description</i>	eine kurze Beschreibung der Distribution
<i>long_description</i>	eine ausführliche Beschreibung der Distribution
<i>author</i>	der Name des Autors
<i>author_email</i>	die E-Mail-Adresse des Autors
<i>maintainer</i>	der Name des Paketverwalters (Maintainer), sofern dies nicht der Autor selbst ist
<i>maintainer_email</i>	die E-Mail-Adresse des Paketverwalters
<i>url</i>	die URL einer Homepage mit weiteren Informationen zur Distribution
<i>download_url</i>	die URL, unter der die Distribution direkt heruntergeladen werden kann
<i>packages</i>	eine Liste von Strings, die die Namen aller Pakete enthält, die in der Distribution enthalten sein sollen
<i>package_dir</i>	Ein Dictionary, über das Pakete in Unterverzeichnissen in die Distribution aufgenommen werden können. Näheres zur Verwendung von <i>package_dir</i> folgt weiter unten.
<i>package_data</i>	Ein Dictionary, über das Dateien, die zu einem Paket gehören, mit in die Distribution aufgenommen werden können. Näheres zur Verwendung von <i>package_data</i> finden Sie weiter unten.
<i>py_modules</i>	eine Liste von Strings, die die Namen aller Python-Module enthält, die in der Distribution enthalten sein sollen
<i>scripts</i>	eine Liste von Strings, die die Namen aller Scriptdateien enthält, die in der Distribution enthalten sein sollen
<i>data_files</i>	Eine Liste von Tupeln, über die zusätzliche Dateien in die Distribution mit aufgenommen werden können. Näheres zur Verwendung von <i>data_files</i> finden Sie weiter unten.

Tabelle 21.1 Mögliche Schlüsselwortparameter für setup

Parametername	Beschreibung
<i>ext_modules</i>	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«.
<i>script_name</i>	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.
<i>license</i>	ein String, der die Lizenz angibt, unter der die Distribution veröffentlicht wird

Tabelle 21.1 Mögliche Schlüsselwortparameter für `setup` (Forts.)

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. Sie können durchaus sowohl Pakete über *packages* als auch einzelne Module über *py_modules* in die Distribution aufnehmen.

Oftmals existiert im Hauptordner 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. Diese Angabe kann auch für einzelne Pakete getätigt werden. 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` 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. Das `distutils`-Paket installiert Scriptdateien in ein Verzeichnis, in dem sie systemweit ausführbar sind.²

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 Dictionaries übergeben werden:

² Unter Windows ist das der Unterordner *Scripts* der Python-Installation, unter Linux das Verzeichnis `/usr/bin`.

```

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 Pakets `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 Installations-scripts. Hier können Sie durchaus auch absolute Pfade, beispielsweise für eine systemweite Konfigurationsdatei, angeben. Über `data_files` angegebene Dateien werden, sofern kein absoluter Pfad angegeben wurde, unter Windows in das Python-Stammverzeichnis und unter Linux in das Verzeichnis `/usr` installiert, wobei die Ordnerstruktur jeweils beibehalten wird.



Hinweis

Sie sollten Ordner innerhalb eines Pfades immer durch einen einfachen Slash (/) voneinander trennen. Das Paket `distutils` kümmert sich dann um die korrekte »Übersetzung« des Pfades in das Format des jeweiligen Betriebssystems.

21.1.3 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,gztar
```

Mögliche Werte sind dabei `zip` für ein zip-Archiv (**.zip*), `gztar` für ein gz-komprimiertes tar-Archiv (**.tar.gz*), `bztar` für ein bz2-komprimiertes tar-Archiv (**.tar.bz2*), `ztar` für ein Z-komprimiertes tar-Archiv (**.tar.Z*) sowie `tar` für ein unkomprimiertes tar-Archiv. Wenn die Option `--formats` nicht angegeben wurde, wird unter Windows ein zip-Archiv und unter Unix-Systemen ein gz-komprimiertes tar-Archiv erstellt.

In das Archiv werden alle Dateien aufgenommen, die im Installationsscript eingetragen wurden. Zusätzlich wird eine Datei namens *README* oder *README.txt* automatisch in das Archiv mit aufgenommen, sofern eine solche im selben Ordner wie das Installationsscript existiert.

Das resultierende Archiv, die Quellcodedistribution, kann jetzt veröffentlicht und verbreitet werden. Der Benutzer, der diese Distribution herunterlädt, kann Ihr Modul bzw. Ihr Paket so installieren, wie in Abschnitt 21.1.5, »Distributionen installieren«, 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 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.



21.1.4 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-Distributionen³ 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
<code>bdist_msi</code>	Erzeugt einen Windows MSI-Installer, der dazu da ist, ein Modul bzw. Paket auf einem Windows-System zu installieren. Es wird die Datei <i>Projektname-Version.win32.msi</i> im Unterordner <i>dist</i> erzeugt.
<code>bdist_rpm</code>	Erzeugt ein RPM-Paket für RPM-basierende Linux-Distributionen wie beispielsweise Fedora Core oder openSUSE. Es wird die Datei <i>Projektname-Version.src.rpm</i> im Unterordner <i>dist</i> erzeugt.
<code>bdist_wininst</code>	Erzeugt einen Windows Installer, der dazu da ist, ein Modul bzw. Paket auf einem Windows-System zu installieren. Es wird die Datei <i>Projektname-Version.win32.exe</i> im Unterordner <i>dist</i> erzeugt.

Tabelle 21.2 Mögliche Argumente des Installationsscripts

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
```

³ Dies sind unter anderem Red Hat, Fedora, SUSE und Mandriva.

**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.

21.1.5 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 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.

**Hinweis**

Die Distribution wird in das Standardverzeichnis für Python-Drittanbieterbibliotheken des Systems installiert. Wenn Sie dies nicht wünschen, können Sie über das Argument *--prefix* ein Zielverzeichnis vorgeben:

```
python setup.py install --prefix="Pfad/Zum/Zielverzeichnis"
```

21.2 Erstellen von EXE-Dateien – cx_Freeze

Mit dem `distutils`-Paket lassen sich Distributionen aus Python-Projekten erzeugen, die dann auf dem Zielsystem im Kontext einer existierenden Python-Umgebung installiert werden können. Besonders unter Windows ist es manchmal wünschenswert, ein Programm als einfache ausführbare Datei auszuliefern, die ohne weitere Voraussetzungen auch auf Systemen läuft, auf denen keine Python-Umgebung installiert ist. Eine solche Distribution kann mit dem Drittanbietermodul `cx_Freeze` erstellt werden, das Sie unter <http://www.cx-freeze.sourceforge.net> herunterladen können.

In diesem Abschnitt verwenden wir das Modul `cx_Freeze`, um aus einem kleinen Beispielprogramm eine ausführbare Datei zu schnüren. Hier sehen Sie zunächst den Quelltext des Beispielprogramms:

```
import sys

if len(sys.argv) > 2:
    print("Ergebnis: {}".format(int(sys.argv[1]) + int(sys.argv[2])))
```

Es handelt sich dabei um ein einfaches Programm, das zwei als Argument übergebene Zahlen addiert und das Ergebnis ausgibt.

Es gibt zwei Möglichkeiten, das Modul `cx_Freeze` zu verwenden: einmal über das Skript `cxfreeze`,⁴ dem der Pfad zur zu bearbeitenden Programmdatei als Argument übergeben werden muss, oder alternativ in Kombination mit `distutils`. Dazu schreiben wir im Kontext des obigen Beispielprogramms die folgende Programmdatei `setup.py`:

```
from cx_Freeze import setup, Executable

setup(
    [...]
    executables = [Executable("calc.py")]
)
```

Statt der Auslassungszeichen stehen die üblichen Parameter der Funktion `setup` im Quellcode, über die beispielsweise der Name des Programms oder die E-Mail-Adresse des Autors angegeben werden können. Um mit `cx_Freeze` eine ausführbare Datei zu erstellen, muss der Schlüsselwortparameter `executables` angegeben werden. Für die-

4 Unter Windows und Mac OS finden Sie `cxfreeze` im Unterverzeichnis *Scripts* der Python-Installation. Unter Linux wird `cxfreeze` üblicherweise in das globale Systemverzeichnis für ausführbare Dateien installiert.

sen wird eine Liste von `Executable`-Instanzen übergeben, die jeweils eine zu erzeugende EXE-Datei repräsentieren.

Nachdem das Installationsscript fertig ist, kann die ausführbare Datei erzeugt werden. Dazu muss das Installationsscript *setup.py* mit dem Argument `build` aufgerufen werden:

```
setup.py build
```

Der Rest geschieht automatisch. Nachdem sich das Installationsscript beendet hat, finden Sie im Programmverzeichnis den Unterordner *dist*, der die fertige Distribution Ihres Python-Programms enthält. Diese beinhaltet nicht nur die ausführbare Datei selbst, in diesem Fall *calc.exe*, sondern noch weitere für das Programm benötigte Dateien. So sind beispielsweise der Python-Interpreter in der DLL *python32.dll* und die benötigten Teile der Standardbibliothek im Archiv *library.zip* ausgelagert.

Kapitel 22

Optimierung

»Premature optimization is the root of all evil.«
– Donald E. Knuth

Als Programmierer sollten Sie mit der Zeit einen 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 mit der Optimierung der Laufzeit eines Python-Programms beschäftigen. Abgesehen von der Laufzeit können Programme 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, sodass 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.

22.1 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 mehr auf das Laufzeitverhalten des optimierten Programms 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. Es ist dann aber beispielsweise für die Built-in Funktion `help` nicht mehr möglich, eine Hilfeseite zu Elementen Ihres Moduls zu generieren, weil keine Docstrings mehr vorhanden sind.

22.2 Mutable vs. immutable

Bei der Einführung der Basisdatentypen zu Beginn dieses Buches haben wir 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.

Im folgenden 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 Wertes 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.

Hinweis

Das hier gezeigte Beispiel hat in Python-Versionen vor 2.5 auch für Strings funktioniert. Inzwischen greifen hier Optimierungen des Python-Interpreters, die verhindern, dass in jedem Schleifendurchlauf ein neuer String erzeugt wird.



22.3 Schleifen

Betrachten wir noch einmal das zweite Beispiel aus dem vorherigen Abschnitt:

```
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)]
```




Hinweis

Ein Aufruf der Built-in Function `map` ist ähnlich effizient wie eine List Comprehension.

22.4 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()
```

22.5 C

Ein in Python geschriebener Programmteil ist aufgrund des zwischengeschalteten Interpreters in der Regel langsamer als ein vergleichbares C-Programm. Aus diesem Grund sollten Sie an einer laufzeitkritischen Stelle so oft wie möglich auf Algorithmen zurückgreifen, die in C implementiert wurden. So lohnt es sich beispielsweise immer, eine Built-in Function einzusetzen, anstatt den entsprechenden Algorithmus selbst zu implementieren.

In Kapitel 24, »Wissenschaftliches Rechnen« (S. 881), wird behandelt, wie Sie eigene Module oder Programmteile in C schreiben.

¹ Das Wort »inlinen« kommt vom Schlüsselwort `inline` aus C++, das Funktionsaufrufe durch den Programmcode der jeweiligen Funktion ersetzt.

22.6 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
wurzeln = [math.sqrt(i) for i in range(100)]
```

Wesentlich effizienter ist es jedoch, die Funktion `sqrt` des Moduls `math` direkt zu referenzieren und über diese Referenz anzusprechen:

```
import math
s = math.sqrt
wurzeln = [s(i) for i in range(100)]
```

Die Schleife der zweiten Variante kann um ca. 20 % schneller ausgeführt werden als die Schleife der ersten Variante.

22.7 Exceptions

Im folgenden Beispiel möchten wir in einer sehr frequentierten Schleife mit einem sich ständig ändernden Index `i` auf eine Liste `liste` zugreifen, können uns 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
```

Bei einer Liste mit 100 Einträgen ist die untere Variante der Funktion `f`, unabhängig davon, wie häufig eine Exception geworfen werden muss, um ca. 35 % schneller als die erste. Je kleiner die Liste, desto effizienter ist das Durchsuchen im Vergleich zum Behandeln von Exceptions. In unserem Test war die obere Variante von `f` effizienter, sobald die Liste weniger als 30 Elemente hatte.

22.8 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 "{} {} {} {}".format(a,b,c,d)
```

Der Funktionsaufruf

```
f("Hallo", "du", "schöne", "Welt")
```

kann um ca. 18 % schneller ausgeführt werden als der Funktionsaufruf

```
f(a="Hallo", b="du", c="schöne", d="Welt")
```

Beachten Sie, dass dies kein Grund sein sollte, allgemein auf Schlüsselwortparameter zu verzichten, allenfalls in extrem laufzeitkritischen Bereichen.

Kapitel 23

Grafische Benutzeroberflächen

»What you see is all you get«
– Brian Kernighan

Nachdem wir uns bisher ausschließlich mit Konsolenanwendungen beschäftigt haben, also mit Programmen, die über eine rein textbasierte Schnittstelle zum Benutzer verfügen, 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 *Fenstern* (engl. *windows*). Innerhalb dieser Fenster lassen sich beliebige *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 eine Schaltfläche (engl. *button*) oder ein Textfeld ein Steuerelement.

Sowohl die Terminologie als auch die Implementierung einer grafischen Oberfläche hängt sehr stark davon ab, welche Bibliothek, auch *Toolkit* genannt, verwendet wird. Aus diesem Grund werden wir zunächst allgemein verschiedene Toolkits auflisten, die mit Python verwendet werden können, und erst im zweiten Abschnitt zur eigentlichen Programmierung einer grafischen Benutzeroberfläche kommen. Dort behandeln wir zunächst ausführlich das Modul *tkinter*, das das Tk-Toolkit verwendet und in der Standardbibliothek enthalten ist. Danach präsentieren wir einen projektorientierten Einstieg in das umfangreichere und zeitgemäßere Qt-Framework unter Verwendung von PyQt.

23.1 Toolkits

Unter einem Toolkit versteht man 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 werden die wichtigsten Python-Bindings für GUI-Toolkits aufgelistet und kurz erläutert.

23.1.1 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.

23.1.2 PyGObject

Website: <http://live.gnome.org/PyGObject>

Toolkit: Gtk

Das Toolkit Gtk (GIMP Toolkit) wurde ursprünglich für das Grafikprogramm GIMP entwickelt und zählt heute neben Qt zu den am meisten verbreiteten plattformübergreifenden Toolkits. Sowohl das Toolkit selbst als auch die Python-Bindings `PyGtk` stehen unter der *GNU Lesser General Public License* und können frei heruntergeladen und verwendet werden.

Das Gtk-Toolkit ist die Grundlage der freien Desktop-Umgebung GNOME und erfreut sich daher, gerade unter Linux-Anwendern, einer hohen Beliebtheit. Obwohl es eigentlich für C geschrieben wurde, ist Gtk von Grund auf objektorientiert und kann somit gut mit Python verwendet werden.

23.1.3 PyQt

Website: <http://www.riverbankcomputing.co.uk>

Toolkit: Qt

Bei Qt handelt es sich um ein umfassendes Framework, das von der norwegischen Firma *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 GPL, veröffentlichen, stehen Qt und PyQt unter der GPL. Für den kommerziellen Gebrauch muss eine Lizenzgebühr entrichtet werden. Qt wird für viele kommerzielle Projekte verwendet, darunter zum Beispiel Google Earth, Skype oder Opera.

23.1.4 PySide

Website: <http://www.pyside.org/>

Toolkit: Qt

PySide ist ein von Nokia geführtes Projekt zur Entwicklung von Python Bindings für das Qt-Framework. PySide ist unter den Bestimmungen der LGPL erhältlich und damit sowohl für nicht-kommerzielle als auch für kommerzielle Zwecke kostenlos zu verwenden.

PyQt und PySide sind zu einem hohen Grade kompatibel, sodass es häufig ausreicht, die `import`-Anweisungen von PyQt4 auf PySide umzuschreiben, um ein PyQt-Programm unter PySide lauffähig zu machen.¹ Diese API-Kompatibilität ist aber nicht vollständig und auch für die Zukunft nicht garantiert.

23.1.5 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 erschienen ist. 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-Projekts ist es, das Look-and-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.

Zum Zeitpunkt der Drucklegung dieses Buchs existierte keine mit Python 3 kompatible Version von wxPython.

¹ Um die Beispielprogramme in diesem Buch unter PySide lauffähig zu machen, muss zusätzlich die Klasse `PySide.QtUiTools.QUiLoader` zum Einlesen der UI-Datei verwendet werden.

23.2 Einführung in tkinter

Nachdem wir die verschiedenen GUI-Toolkits vorgestellt haben, für die Python-Bindings existieren, möchten wir uns endlich der Programmierung grafischer Benutzeroberflächen widmen. Dazu ist in der Standardbibliothek das Modul `tkinter` enthalten, über das sich grafische Oberflächen mit dem Tk-Toolkit programmieren lassen. Das Modul `tkinter` ist die einzige Möglichkeit, ohne die Installation von Dritt-anbieterbibliotheken eine grafische Benutzeroberfläche in Python zu schreiben. Aus diesem Grund lohnt es sich, einen Blick auf `tkinter` und seine Möglichkeiten zu werfen.

23.2.1 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 23.1 Die erste grafische Oberfläche

Jetzt möchten wir den Quelltext besprechen, der dem Programm zugrunde liegt. Wir werden dabei recht oberflächlich bleiben. In späteren Kapiteln 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 den Ausgangspunkt für ein eigenes Widget bzw. einen eigenen Dialog dar. Wir benutzen das `Frame`-Widget als Container für die Steuerelemente unseres Dialogs.

Im Konstruktor von `MyApp` werden 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 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` kann also beispielsweise festgelegt werden, 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 instanziiert 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 instanziiieren 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.

23.2.2 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 vom Programm heraus verändern.

Es ist klar, dass Steuerelementvariablen nicht zwangsläufig Strings sein müssen, denken Sie an ein Steuerelement zur Eingabe einer ganzen Zahl. Die Basisdatentypen von Python können nicht als Datentypen für Steuerelementvariablen verwendet werden,² hier muss auf spezielle Datentypen des `tkinter`-Moduls zurückgegriffen werden. 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 23.1 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]]])

Hiermit wird eine Steuerelementvariable erzeugt und initialisiert. Über das Argument *master* kann ein Masterwidget angegeben werden. Dies ist insbesondere dann sinnvoll, wenn Sie mehrere Tk-Instanzen gleichzeitig verwenden. Sollte nur eine Tk-Instanz existieren, wird diese automatisch als Masterwidget 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

2 Das liegt daran, dass das mit einer Steuerelementvariablen verbundene Widget benachrichtigt werden muss, wenn sich der Wert der Variablen ändert. Dies ist mit den Basisdatentypen nicht möglich.

Name "PY_VAR n " verwendet, wobei n eine laufende Nummer ist. Dieser Parameter ist sehr speziell und sollte in der Regel nicht benötigt werden.

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 Exception 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.



Hinweis

Eine interessante Eigenschaft von Steuerelementvariablen ist, dass sie von mehreren Steuerelementen gleichzeitig verwendet werden können. Auf diese Weise können einfache Zusammenhänge zwischen den Inhalten der Steuerelemente hergestellt werden.

So lässt sich im einführenden Beispielprogramm die Steuerelementvariable `self.name` neben dem Eingabefeld `self.nameEntry` auch der Schaltfläche `self.rev` zuordnen:

```
self.rev["textvariable"] = self.name
```

Auf diese Weise wird der im Eingabefeld eingegebene Name unmittelbar als Beschriftung der Schaltfläche übernommen.

23.2.3 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 beispielsweise 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 Vatelement verfügt. Außerdem darf jedes Steuerelement beliebig viele Kindelemente enthalten. In der Tk-Terminologie wird ein Vatelement *Master* und ein Kindelement *Slave* genannt. Die Hierarchie der Steuerelemente in unserem Beispielprogramm zu Beginn dieses Kapitels soll durch die folgende Grafik veranschaulicht werden:

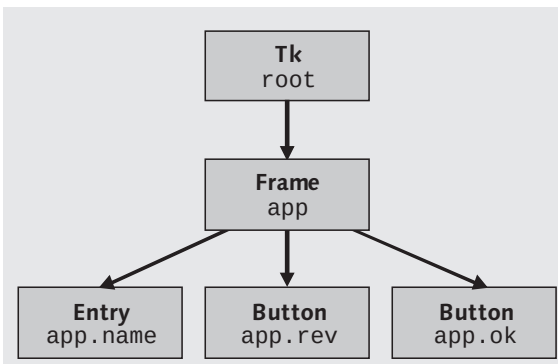


Abbildung 23.2 Steuerelementhierarchie im Beispielprogramm

Die Hierarchie der Steuerelemente ist wichtig, um die Arbeitsweise des Packers zu verstehen. Dieser ordnet nämlich die Kindelemente innerhalb ihres Vatelements an und dann das Vatelement mitsamt Kindelemente in dessen Vatelement. 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 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 funktioniert.

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 23.3 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 23.4 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 hergeschoben.

Als nächstes werden die Buttons in den verbleibenden Arbeitsbereich eingefügt. Diese 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 23.5 Arbeitsweise des Packers, Schritt 3

Soviel zur allgemeinen Arbeitsweise des Packers. Im Folgenden möchten wir die Layoutanweisungen, die dem Packer in Form von Schlüsselwortparametern beim Aufruf der `pack`-Methode erteilt werden können, genauer unter die Lupe nehmen.

Parameter	Mögliche Werte	Bedeutung
<i>after</i>	Widget	Das Steuerelement soll nach dem angegebenen Widget gepackt werden.
<i>anchor</i>	"n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	Wenn der dem Widget zugeteilte Bereich größer ist als das Widget, kann über <i>anchor</i> die Ausrichtung des Widgets innerhalb dieses Bereichs festgelegt werden. Bei den möglichen Werten handelt es sich um die Himmelsrichtungen sowie »zentriert«.
<i>before</i>	Widget	Das Steuerelement soll vor dem angegebenen Widget gepackt werden.
<i>expand</i>	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.
<i>fill</i>	"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.
<i>in</i>	Widget	Fügt das Steuerelement in das angegebene Master-Widget ein.
<i>ipadx</i>	int	Pixel-Größe für das horizontale innere Padding
<i>ipady</i>	int	Pixel-Größe für das vertikale innere Padding
<i>padx</i>	int	Pixel-Größe für das horizontale äußere Padding
<i>pady</i>	int	Pixel-Größe für das vertikale äußere Padding

Tabelle 23.2 Layoutanweisungen

Parameter	Mögliche Werte	Bedeutung
<i>side</i>	"left", "right", "top", "bottom"	Die Seite des Arbeitsbereichs, an der das Widget eingefügt wird. Ein auf der linken bzw. rechten Seite platziertes Widget beansprucht die gesamte Höhe und ein oben bzw. unten platziertes Widget die gesamte Breite des Arbeitsbereichs. Differenziertere Layouts erreichen Sie mithilfe eines Frame-Widgets.

Tabelle 23.2 Layoutanweisungen (Forts.)

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 aneinander gelegt werden. Mithilfe des sogenannten *Paddings* (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. Die folgende Abbildung zeigt unseren Dialog, bei dem jedes Steuerelement ein äußeres Padding von fünf Pixeln verpasst bekommen hat:



Abbildung 23.6 Padding im Einsatz

Beim *inneren Padding* handelt es sich um eine Abstandsangabe innerhalb eines Widgets. In diesem Widget enthaltene Slave-Widgets³ respektieren dieses Padding und werden entsprechend weiter innen angeordnet. Diese Form des Paddings ist beson-

³ Slave-Widget ist die Tkinter-Terminologie für ein untergeordnetes Steuerelement. Bei der abgebildeten Oberfläche ist das Eingabefeld beispielsweise ein Slave des übergeordneten Frame-Widgets.

ders für Frame-Widgets interessant. Das innere Padding wird analog zum äußeren Padding über die Schlüsselwortparameter `ipadx` und `ipady` festgelegt.

Verhalten bei Vergrößerung des Master-Widgets

Die folgende Grafik demonstriert die Auswirkungen der Layoutanweisungen `fill` und `expand` konkret an einem Beispiel:

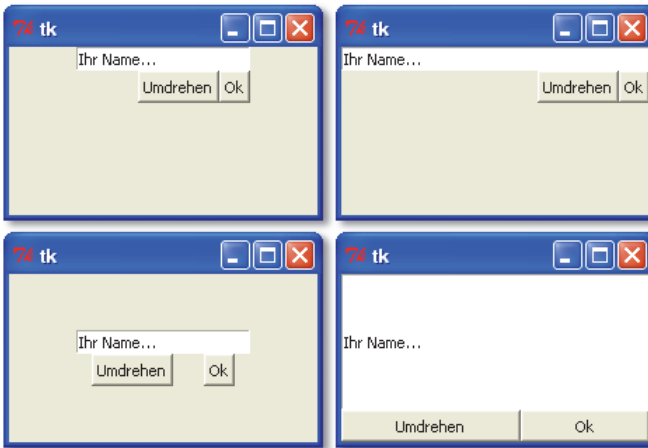


Abbildung 23.7 Die Layoutanweisungen `fill` und `expand`

Das obere linke Bild zeigt das Verhalten des Dialogs ohne weitere Layoutanweisungen beim Vergrößern des Fensters. Rechts daneben wurde die Layoutanweisung `fill="both"` verwendet, unten links `expand=True` und unten rechts schließlich die Kombination aus `expand` und `fill`. Die Layoutanweisungen wurden stets an Steuerelemente übergeben.

23.2.4 Events

Beim Schreiben einer Tk-Anwendung wird nach dem Erstellen und Instanzieren 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, das sogenannte *Event-Objekt* erwartet. Diese Funktion wird *Eventhandler* 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 `[Shift]`-Taste und einen normalen Mausklick gesondert zu betrachten. Der *Type*-Eintrag kennzeichnet den Eventtyp, und über den *Detail*-Eintrag kann eine nähere Spezifikation erfolgen, beispielsweise kann hier angegeben werden, 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 `[Shift]`-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 `[Strg]`-Taste. Die wichtigsten Modifier sind in der folgenden Tabelle aufgelistet und erläutert:

Attribut	Bedeutung
Alt	Die <code>[Alt]</code> -Taste muss beim Auftreten des Events gedrückt sein.
Control	Die <code>[Strg]</code> -Taste muss beim Auftreten des Events gedrückt sein.
Lock	Die Feststelltaste muss beim Auftreten des Events aktiviert sein.

Tabelle 23.3 Modifier

Attribut	Bedeutung
Shift	Die <code>Shift</code> -Taste muss beim Auftreten des Events gedrückt sein.
Buttonx	Die Maustaste <code>x</code> muss beim Auftreten des Events gedrückt sein. Die Zahl <code>x</code> spezifiziert dabei, welche Taste gemeint ist: 1 steht für die linke, 2 für die mittlere ⁴ und 3 für die rechte Maustaste. Weitere Maustasten werden entsprechend fortlaufend nummeriert. Alternative Schreibweise: <code>Bx</code>
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 23.3 Modifier (Forts.)

Nachdem wir uns um die Modifier gekümmert haben, werden wir nun den *Type*-Eintrag der Event-Spezifikation besprechen. Diese 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 Eventtyp zu bestimmen.

Event	Typ	Bedeutung
KeyPress	2	Der Benutzer hat eine Taste heruntergedrückt. Über den <i>Detail</i> -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.

Tabelle 23.4 Events

⁴ Das Scrollrad zählt als mittlere Maustaste. Ist weder ein Scrollrad noch eine echte mittlere Maustaste verfügbar, zählt ein gleichzeitiger Klick mit der linken und der rechten Maustaste als Klicken der mittleren Maustaste.

Event	Typ	Bedeutung
		Beachten Sie, dass das Widget den Eingabefokus haben muss, um dieses Event empfangen zu können. Alternative Schreibweisen: ▶ <i>Key</i> statt <i>KeyPress</i> ▶ Der <i>Type</i>-Eintrag <i>Key</i> oder <i>KeyPress</i> kann 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 ausgelöst wird. Die Event-Spezifikation <code>"abc"</code> löst also genau dann ein <i>KeyPress</i>-Event aus, wenn der Benutzer die Zeichenfolge <code>"abc"</code> eingetippt hat.
KeyRelease	3	Der Benutzer hat eine Taste losgelassen. Hier können die gleichen <i>Detail</i>-Angaben verwendet werden wie bei <i>KeyPress</i>. Beachten Sie aber, dass die alternativen Schreibweisen, die bei <i>KeyPress</i> existieren, für <i>KeyRelease</i> nicht gültig sind.
ButtonPress	4	Der Benutzer hat eine Maustaste über dem Widget gedrückt.⁵ Im <i>Detail</i>-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 <i>Detail</i>-Eintrag weggelassen, so ist es egal, mit welcher Maustaste der Benutzer geklickt hat. Alternative Schreibweisen: <i>Button</i> oder schlicht <code><n></code>⁶ (wobei <i>n</i> die Nummer der Maustaste ist)
ButtonRelease	5	Der Benutzer hat eine Maustaste losgelassen. Auch hier kann über den <i>Detail</i>-Eintrag in der Event-Spezifikation angegeben werden, welche Maustaste gemeint ist.

Tabelle 23.4 Events (Forts.)

5 Das Widget, über dem die Maustaste gedrückt wurde, bekommt sozusagen ein »Abonnement« auf Mausevents. 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.

6 In diesem Fall lautet die vollständige Event-Spezifikation `"<n>`".

Event	Typ	Bedeutung
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 Mousrad bewegt. Beachten Sie, dass dieses Event unabhängig von der Position des Mauszeigers immer das Widget empfängt, das den Eingabefokus besitzt.

Tabelle 23.4 Events (Forts.)

Nachdem ein Event ausgelöst wird, wird der damit verbundene Eventhandler aufgerufen. Dieser Handler bekommt dann ein *Event-Objekt*, genauer 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
<i>char</i>	Enthält das eingegebene Zeichen als String. Ist dieses Zeichen nicht lesbar, referenziert <i>char</i> einen leeren String. Gültig für: <code>KeyPress</code> , <code>KeyRelease</code>

Tabelle 23.5 Attribute des Event-Objekts

Attribut	Bedeutung
<i>delta</i>	Gibt an, wie weit der Benutzer das Mousrad gedreht hat. Dabei hat <i>delta</i> bei einer Bewegung nach vorne ein positives und bei einer Bewegung nach hinten ein negatives Vorzeichen. Gültig für: <code>MouseWheel</code>
<i>focus</i>	Gibt an, ob das Widget, über dem sich die Maus befindet, den Eingabefokus hat oder nicht. Gültig für: <code>Enter</code> , <code>Leave</code>
<i>height</i>	Enthält die neue Höhe des Widgets in Pixel. Gültig für: <code>Configure</code> , <code>Expose</code>
<i>keycode</i>	Enthält den Keycode der gedrückten Taste. Dies ist ein eindeutiger, ganzzahliger Identifier einer Taste. Gültig für: <code>KeyPress</code> , <code>KeyRelease</code>
<i>keysym</i>	Enthält den symbolischen Namen der gedrückten Taste. Gültig für: <code>KeyPress</code> , <code>KeyRelease</code>
<i>time</i>	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 Eventtypen
<i>type</i>	Enthält den ganzzahligen Identifier des Eventtypen. Gültig für alle Eventtypen
<i>widget</i>	Referenziert das Widget, in dem das Event aufgetreten ist. Gültig für alle Eventtypen
<i>width</i>	Enthält die neue Breite des Widgets in Pixel. Gültig bei: <code>Configure</code> , <code>Expose</code>
<i>x</i>	Enthält die X-Koordinate des Mauszeigers in Pixel. Diese Angabe versteht sich relativ zur linken oberen Ecke des Widgets. Gültig für alle Eventtypen
<i>x_root</i>	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>

Tabelle 23.5 Attribute des Event-Objekts (Forts.)

Attribut	Bedeutung
y	Enthält die Y-Koordinate des Mauszeigers. Diese Angabe versteht sich relativ zur linken oberen Ecke des Widgets. Gültig für alle Eventtypen
y_root	Enthält die Y-Koordinate des Mauszeigers, relativ zur linken oberen Ecke des Bildschirms. Gültig für: ButtonPress, ButtonRelease, KeyPress, KeyRelease, Motion

Tabelle 23.5 Attribute des Event-Objekts (Forts.)

Damit ist die Einführung in die wichtigsten Event-Typen abgehandelt. An dieser Stelle möchten wir ein Beispielprogramm bringen, das die Verwendung von Events veranschaulicht. Die grafische Oberfläche des Programms beinhaltet 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 " \
        "{},{}".format(event.x, event.y)

def eventMouseWheel(self, event):
    if event.delta < 0:
        self.label["text"] = "Bitte bewegen Sie das Mausehrad\"
            " 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 Eventhandler zu registrieren. In dieser Methode werden drei Events berücksichtigt: das Eingeben der Zeichenfolge »Entenhausen«, das Klicken mit der linken Maustaste und das Bewegen des Mausehrrads. 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 Eventhandler-Methoden `eventEntenhausen`, `eventMouseClicked` und `eventMouseWheel`. In der Methode `eventMouseWheel` wird ausgenutzt, dass das Vorzeichen von `event.delta` angibt, in welche Richtung das Mausehrad bewegt wurde.

Wir haben die Verwendung des Label-Widgets noch nicht im Detail besprochen, doch es ist ersichtlich, dass man über den Schlüssel `text` den Text verändern kann, den es anzeigt. Eine ausführlichere Besprechung dieses Widgets finden Sie im nächsten Kapitel.

Schlussendlich fehlt 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:

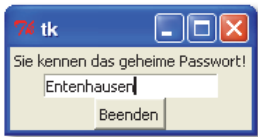


Abbildung 23.8 Beispielprogramm mit Events

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.

23.2.5 Die Steuerelemente

Nachdem wir uns mit den Grundlagen von Tk beschäftigt haben, werden wir jetzt besprechen, 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 in diesem Kapitel besprochen werden, und gibt die Seite an, auf der die jeweiligen Erklärungen zu finden sind.

Widget	Bedeutung	Seite
<i>Button</i>	eine Schaltfläche	783
<i>Canvas</i>	ein Steuerelement für Zeichnungen und Grafiken	806
<i>Checkbutton</i>	ein Steuerelement, das entweder aktiviert oder deaktiviert sein kann	786
<i>Entry</i>	ein einzeliges Eingabefeld	789
<i>Label</i>	ein Steuerelement für Beschriftungen	790
<i>LabelFrame</i>	ein Steuerelement für beschriftete Rahmen	791
<i>Listbox</i>	eine Liste von Einträgen	792
<i>Menu</i>	ein Kontextmenü	796
<i>Menubutton</i>	eine Schaltfläche, die bei Betätigung ein Kontextmenü anzeigt	799
<i>OptionMenu</i>	eine Schaltfläche, die bei Betätigung eine Auswahlliste anzeigt	800

Tabelle 23.6 Erklärte Widgets

Widget	Bedeutung	Seite
<i>Radiobutton</i>	Ein Steuerelement, das entweder aktiviert oder deaktiviert sein kann. Innerhalb einer Gruppe darf nur genau ein Radiobutton aktiviert sein.	788
<i>Scrollbar</i>	eine Leiste, die das Scrollen übergroßer Widgets ermöglicht	801
<i>Spinbox</i>	ein Steuerelement zum Einstellen eines Zahlenwertes	802
<i>Text</i>	ein mehrzeiliges Eingabefeld	804
<i>Widget</i>	die Basisklasse eines Steuerelements	780

Tabelle 23.6 Erklärte Widgets (Forts.)

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 man Informationen über ein Widget, beispielsweise seine Position, herausfinden kann. In der folgenden Tabelle werden die wichtigsten *winfo*-Methoden erläutert. Dabei sei *w* ein Widget.

Schnittstelle	Bedeutung
<i>w.winfo_children()</i>	Gibt eine Liste zurück, die alle Unterwidgets von <i>w</i> enthält. Auch Unterwidgets von Unterwidgets von <i>w</i> sind in der Liste enthalten.
<i>w.winfo_class()</i>	Gibt den Namen der Widgetklasse von <i>w</i> als String zurück. Beispielsweise wird "Frame" zurückgegeben, wenn es sich bei <i>w</i> um ein Frame-Widget handelt.
<i>w.winfo_geometry()</i>	Gibt die Position und die Dimension des Widgets <i>w</i> in Form eines Strings zurück. Beachten Sie, dass die Position relativ zum übergeordneten Widget zu sehen ist. Der zurückgegebene String hat die Form " <i>BxH+X+Y</i> ", wobei <i>B</i> die Breite, <i>H</i> die Höhe, <i>X</i> die X-Position und <i>Y</i> die Y-Position des Widgets ist.
<i>w.winfo_height()</i>	Gibt die Höhe des Widgets <i>w</i> in Pixel zurück.
<i>w.winfo_pointerx()</i>	Gibt die X-Koordinate des Mauszeigers relativ zur oberen linken Ecke des Bildschirms zurück.

Tabelle 23.7 winfo-Methoden

Schnittstelle	Bedeutung
<i>w.winfo_pointerxy()</i>	Gibt ein Tupel, bestehend aus den X- und Y-Koordinaten des Mauszeigers, zurück.
<i>w.winfo_pointery()</i>	Gibt die Y-Koordinate des Mauszeigers relativ zur oberen linken Ecke des Bildschirms zurück.
<i>w.winfo_reqheight()</i>	Gibt die für das Widget <i>w</i> angeforderte Höhe zurück. Diese Methode ist beispielsweise direkt nach dem Instanzieren eines Widgets sinnvoll, da das Widget zu diesem Zeitpunkt möglicherweise noch nicht gepackt wurde und somit keine realen Dimensionen hat.
<i>w.winfo_reqwidth()</i>	Gibt die für das Widget <i>w</i> angeforderte Breite zurück.
<i>w.winfo_rootx()</i>	Gibt die X-Position des Widgets <i>w</i> relativ zur oberen linken Ecke des Bildschirms zurück.
<i>w.winfo_rooty()</i>	Gibt die Y-Position des Widgets <i>w</i> relativ zur oberen linken Ecke des Bildschirms zurück.
<i>w.winfo_screenheight()</i>	Gibt die Höhe des Bildschirms in Pixel zurück.
<i>w.winfo_screenwidth()</i>	Gibt die Breite des Bildschirms in Pixel zurück.
<i>w.winfo_width()</i>	Gibt die Breite des Widgets <i>w</i> in Pixel zurück.
<i>w.winfo_x()</i>	Gibt die X-Koordinate des Widgets <i>w</i> relativ zum übergeordneten Widget zurück.
<i>w.winfo_y()</i>	Gibt die Y-Koordinate des Widgets <i>w</i> relativ zum übergeordneten Widget zurück.

Tabelle 23.7 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
<i>w.bind(event, func[, add])</i>	Verknüpft ein Event mit einer Handlerfunktion. Besprochen auf Seite 772.

Tabelle 23.8 Methoden eines Widgets

Schnittstelle	Bedeutung
<i>w.bind_all(event, func[, add])</i>	wie <i>bind</i> , mit dem Unterschied aber, dass das Event <i>event</i> in allen Widgets der Applikation mit der Handlerfunktion <i>func</i> verbunden wird
<i>w.unbind(event[, funcid])</i>	Löst alle existierenden Bindungen an das Event <i>event</i> auf. Über den Parameter <i>funcid</i> kann eine bestimmte Verbindung gezielt aufgelöst werden. Jede mit einem Event verbundene Handlerfunktion bekommt dazu eine ID zugeteilt, welche beim Aufruf der Methode <i>bind</i> zurückgegeben wird.
<i>w.unbind_all(event)</i>	wie <i>unbind</i> , jedoch stets für alle Verbindungen an das Event <i>event</i> und alle Widgets der Applikation
<i>w.cget(key)</i>	Gibt den Wert der Option <i>key</i> von <i>w</i> zurück. Die für das Widget <i>w</i> gültigen Optionen lassen sich über die Methode <i>keys</i> herausfinden.
<i>w.configure([cnf][, **kw])</i>	Setzt eine oder mehrere Optionen des Widgets <i>w</i> . Die Optionen können dabei als Dictionary über den Parameter <i>cnf</i> oder in Form von Schlüsselwortparametern übergeben werden. Näheres zu Optionen finden Sie im Laufe dieses Kapitels. Alternative Schreibweise: <i>config</i>
<i>w.destroy()</i>	Zerstört das Widget <i>w</i> und alle ihm untergeordneten Widgets.
<i>w.focus_set()</i>	Gibt <i>w</i> den Eingabefokus. Wenn der Eingabefokus gerade nicht bei der Applikation liegt, wird er an <i>w</i> gegeben, sobald die Applikation ihn bekommt. Alternative Schreibweise: <i>focus</i>
<i>w.info()</i>	Gibt ein Dictionary zurück, das die Ausrichtungsanweisungen an den Packer für <i>w</i> enthält.
<i>w.keys()</i>	Gibt eine Liste aller für <i>w</i> gültigen Optionen zurück.
<i>w.mainloop()</i>	Startet die Hauptschleife des Widgets <i>w</i> . Von nun an übernimmt das Tk-Toolkit den Programmfluss. Die Methode <i>mainloop</i> blockiert so lange, bis die Anwendung beendet wurde.

Tabelle 23.8 Methoden eines Widgets (Forts.)

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 23.8 Methoden eines Widgets (Forts.)

Nach dieser etwas trockenen Vorstellung der Basisfunktionalität eines Widgets möchten wir 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 und entspricht einer Schaltfläche an der grafischen Oberfläche. Sie haben bereits im Beispielpogramm gesehen, wie man einen Button verwendet. Wichtig ist dabei, nach dem Instanziiieren 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.

Option	Wertebereich	Bedeutung
<i>activebackground</i>	"#RRGGBB" ⁷	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.
<i>activeforeground</i>	"#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.
<i>anchor</i>	"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.
<i>background</i>	"#RRGGBB"	Legt die Hintergrundfarbe des Buttons im nicht aktiven Zustand fest. Alternative Schreibweise: <i>bg</i>
<i>borderwidth</i>	int	Legt die Dicke des Steuerelementrands in Pixeln fest. Alternative Schreibweise: <i>bd</i>
<i>command</i>	Funktions- objekt	Setzt die Handlerfunktion, die gerufen wird, wenn der Benutzer den Button gedrückt hat. Die Handlerfunktion darf keine Parameter erwarten.
<i>foreground</i>	"#RRGGBB"	Legt die Vordergrundfarbe des Buttons im inaktiven Zustand fest. Alternative Schreibweise: <i>fg</i>
<i>height</i>	int	Legt die Höhe des Buttons fest. Die Höhe wird dabei in Textzeilen gemessen.

Tabelle 23.9 Optionen eines Buttons

⁷ 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 (kein Anteil dieser Farbe) bis FF (voller Anteil dieser Farbe). Die RGB-Farbwerte #000000 und #FFFFFF entsprechen schwarz und weiß.

Option	Wertebereich	Bedeutung
<i>justify</i>	"left", "center", "right"	Legt bei einem Button mit mehrzeiliger Beschriftung die Ausrichtung der Textzeilen zueinander fest.
<i>overrelief</i>	"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.
<i>padx</i>	int	Legt das Padding in X-Richtung fest.
<i>pady</i>	int	Legt das Padding in Y-Richtung fest.
<i>relief</i>	"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.
<i>state</i>	"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.
<i>takefocus</i>	bool	Legt fest, ob der Button den Fokus bekommen soll, wenn der Benutzer mit der Tabulator-Taste die Widgets des Dialogs durchschaltet.
<i>text</i>	str	Setzt die Beschriftung des Buttons.

Tabelle 23.9 Optionen eines Buttons (Forts.)

Option	Wertebereich	Bedeutung
<i>textvariable</i>	StringVar	Setzt eine Steuerelementvariable für die Beschriftung des Buttons. Die Verwendung dieser Variable funktioniert analog zum Entry-Widget, für das sie bereits im einführenden Beispielprogramm demonstriert wurde.
<i>width</i>	int	Legt die Breite des Buttons fest. Die Breite wird dabei in Zeichen angegeben.

Tabelle 23.9 Optionen eines Buttons (Forts.)



Hinweis

Die obige Liste ist unvollständig. Neben den aufgeführten Optionen existieren noch einige weitere, die allerdings zu speziell sind und den Rahmen dieses Buchs 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. Damit ist der Button, das erste und eines der elementarsten Widgets, besprochen, und wir wenden uns weiteren Widgets zu.



Abbildung 23.9 Darstellungsarten eines Buttons

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 23.10 Zwei Checkbuttons im Einsatz

Erstellung und Verwenden 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"] = checked
```

```
check.pack()
```

Nach der Instanziierung 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 Handlerfunktion angegeben werden, die vom Tk-Framework gerufen wird, wenn der Benutzer den Status des Checkbuttons wechselt. Innerhalb dieser Handlerfunktion 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
<i>indicatoron</i>	bool	Gibt an, ob die kleine Box vor der Beschriftung angezeigt werden soll oder nicht. Wenn diese Option auf <i>False</i> 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.
<i>offvalue</i>	beliebig	Legt den Wert fest, den die Steuerelementvariable annehmen soll, wenn der Checkbutton deaktiviert ist. Die Steuerelementvariable muss einen passenden Typ haben, um diesen Wert speichern zu können.

Tabelle 23.10 Spezifische Optionen eines Checkbuttons

Option	Wertebereich	Bedeutung
<i>onvalue</i>	beliebig	Legt den Wert fest, den die Steuerelementvariable annehmen soll, wenn der Checkbutton aktiviert ist. Die Steuerelementvariable muss einen passenden Typ haben, um diesen Wert speichern zu können.
<i>variable</i>	BooleanVar (üblicherweise)	Setzt eine Steuerelementvariable, über die der Status des Checkbuttons gelesen und verändert werden kann. Beachten Sie, dass der Typ der Steuerelementvariable zu dem für die Werte von <i>onvalue</i> und <i>offvalue</i> verwendeten Datentyp passen muss.

Tabelle 23.10 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 man eine Gruppe von Radiobuttons definieren kann, innerhalb derer immer genau einer der Radiobuttons aktiviert ist. Dabei bilden die Radiobuttons eine Gruppe, die sich dieselbe Steuerelementvariable teilen. 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. Statt die Radiobutton-Instanz nach dem Instanzieren wie ein Dictionary zu verwenden, können die Optionen auch dem Konstruktor als Schlüsselwortparameter übergeben werden. In diesem Fall werden die Optionen *text* für die Beschriftung, *value* für den Wert, den die Steuerelementvariable annimmt, wenn der Radiobutton ausgewählt wird, *variable* für die Steuerelementvariable und *command* für die Handlerfunktion gesetzt. Analog zum Checkbutton wird diese Handlerfunktion immer dann gerufen, wenn der Benutzer seine Auswahl verändert.

Die grafische Oberfläche des obigen Beispielprogramms sehen Sie in Abbildung 23.11.

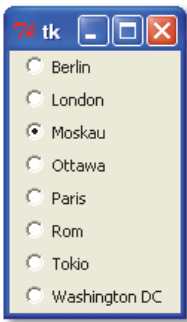


Abbildung 23.11 Eine Menge Radiobuttons

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 instanziiert. Dann wird die Steuerelementvariable auf einen Wert gesetzt

und mit dem Eingabefeld verbunden. Schlussendlich wird das Eingabefeld dem Packer übergeben. Die grafische Oberfläche dieses Beispielprogramms sieht so aus:



Abbildung 23.12 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
<i>show</i>	str	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.
<i>textvariable</i>	Variable	Setzt eine Steuerelementvariable, über die auf den im Eingabefeld angezeigten Text lesend und schreibend zugegriffen werden kann.

Tabelle 23.11 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 exemplarisch die Verwendung des Label-Widgets:

```
label = tkinter.Label(master)
label["text"] = "Hallo Welt"
label.pack()
```

Nach dem Instanziiieren 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 folgender Abbildung zu sehen:



Abbildung 23.13 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 *active-background*, *activeforeground*, *anchor*, *background*, *bd*, *bg*, *borderwidth*, *command*, *fg*, *foreground*, *height*, *justify*, *overrelief*, *padx*, *pady*, *relief*, *state*, *takefocus*, *text*, *text-variable* 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 soll im folgenden Code demonstriert werden:

```
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 auf die Instanziierung 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 folgendermaßen aus:



Abbildung 23.14 Ein LabelFrame-Widget

Neben der bereits angesprochenen Option *text* für die Beschriftung des Rahmens bietet ein LabelFrame-Widget zwei weitere Optionen an, die in der folgenden Tabelle zusammengefasst werden:

Option	Wertebereich	Bedeutung
<i>labelanchor</i>	"n", "ne", "e", "se", "s", "sw", "w", "nw"	Legt die Position und die Ausrichtung der Beschriftung auf dem Rahmen fest.
<i>labelwidget</i>	<i>Widget</i>	Statt einer schlichten Beschriftung kann über diese Option auch ein beliebiges Widget auf dem Rahmen positioniert werden.

Tabelle 23.12 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* wird durch das folgende Beispielprogramm demonstriert:

```
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 wird eine Instanz der Klasse *tkinter.Listbox* erzeugt und gepackt. 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 23.15 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:

Methoden	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 <i>first</i> bzw. alle Einträge mit einem Index zwischen <i>first</i> und <i>last</i> .
<code>get(first[, last])</code>	Gibt den Eintrag mit dem Index <i>first</i> bzw. eine Liste mit den Einträgen, deren Indizes zwischen <i>first</i> und <i>last</i> liegen, zurück.
<code>insert(index, *elements)</code>	Fügt die Elemente <i>elements</i> an der Position <i>index</i> in die Listbox ein. Die bestehenden Einträge der Listbox werden dabei nicht überschrieben, sondern nach hinten gerückt.
<code>itemcget(index, option)</code>	Gibt den Wert der Option <i>option</i> des Eintrags mit dem Index <i>index</i> zurück.
<code>itemconfig(index, **option)</code>	Ermöglicht das Setzen von Optionen des Eintrags mit dem Index <i>index</i> . Die Optionen müssen als Schlüsselwortparameter angegeben werden.
<code>selection_clear(first[, last])</code>	Hebt eine eventuelle Auswahl des Eintrags mit dem Index <i>first</i> bzw. der Einträge mit einem Index zwischen <i>first</i> und <i>last</i> auf. Alternative Schreibweise: <code>select_clear</code>
<code>selection_includes(index)</code>	Gibt <code>True</code> zurück, wenn der Eintrag mit dem Index <i>index</i> in der Listbox ausgewählt ist.
<code>selection_set(first[, last])</code>	Fügt das Element mit dem Index <i>first</i> bzw. die Elemente mit Indizes zwischen <i>first</i> und <i>last</i> zur aktuellen Auswahl hinzu.
<code>size()</code>	Gibt die Anzahl der Einträge der Listbox zurück.

Tabelle 23.13 Methoden einer Listbox

Zu guter Letzt besprechen wir noch die wichtigsten Optionen, die im Zusammenhang mit einer Listbox verwendet werden können:

Option	Wertebereich	Bedeutung
<i>selectmode</i>	"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. Die Werte "multiple" und "extended" erlauben es, mehrere Einträge auszuwählen.
<i>xscrollcommand</i>	<i>Funktionsobjekt</i>	Möglichkeit zur Anbindung einer horizontalen Scrollbar. Näheres dazu erfahren Sie im Abschnitt über die Scrollbar.
<i>yscrollcommand</i>	<i>Funktionsobjekt</i>	Möglichkeit zur Anbindung einer vertikalen Scrollbar. Näheres dazu erfahren Sie im Abschnitt über die Scrollbar.

Tabelle 23.14 Spezifische Optionen eines Listbox-Widgets

Das vorangegangene Beispielprogramm zur Listbox war sehr statisch. Der Benutzer konnte zwar einen Eintrag der Listbox auswählen, doch passiert ist daraufhin nichts. Das folgende Beispielprogramm zeigt, wie man auf eine Änderung der Benutzerauswahl reagieren kann. 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 Handlermethode mit dem sogenannten *virtuellen Event* <<ListBoxSelect>>. Ein virtuelles Event ist ein spezielles Event, das nur für mit einem bestimmten Widgettyp verwendet werden kann.⁸ 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 Handlermethode `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 instanziiert 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 der folgenden Abbildung dargestellt:



Abbildung 23.16 Eine Listbox mit Benutzerinteraktion

⁸ 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 Schaltfläche, über die der Benutzer eine Liste weiterer Kommandos erreichen kann. Üblicherweise findet man zum Beispiel im Menü »Datei« die Kommandos »Speichern« und »Speichern unter«. Das folgende Beispielprogramm zeigt exemplarisch, wie man einen tkinter-Dialog mit einem Menü ausstatten kann:

```
import tkinter

class MyApp(tkinter.Frame):
    def __init__(self, master):
        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, welches 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 Desktopumgebungen unpassend 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 Handlerfunktion (*command*), welche analog zur Handlerfunktion 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 Handlermethode 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` instanziiert 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 folgendermaßen aus:

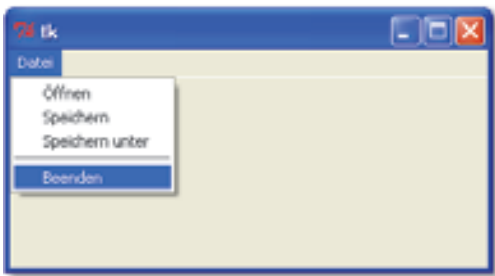


Abbildung 23.17 Eine Menüleiste

Dieses simple Beispiel lässt sich noch etwas erweitern, denn neben den Methoden `add_command` zum Hinzufügen eines einfachen Menüpunktes und `add_separator` zum Hinzufügen einer Trennlinie verfügt eine Menu-Instanz noch über die Metho-

den `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 so aussehen:

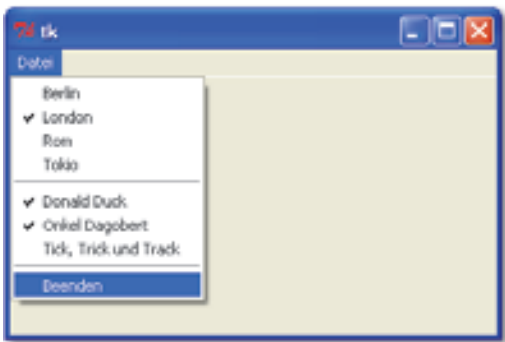


Abbildung 23.18 Eine komplexere Menüleiste

Die folgende Tabelle listet 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 Handlerfunktion.
<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.

Tabelle 23.15 Methoden einer `tkinter.Menu`-Instanz

Methode	Bedeutung
<i>add_separator([cnf][, **kw])</i>	Fügt eine Trennlinie ins Menü ein.
<i>delete(index1[, index2])</i>	Löscht den Menüeintrag an der Stelle <i>index1</i> bzw. alle Menüeinträge zwischen den Stellen <i>index1</i> und <i>index2</i> . Die Menüeinträge sind, beginnend bei 0, entsprechend der Anzeigereihenfolge durchnummeriert.
<i>entrycget(index, option)</i>	Gibt den Wert der Option <i>option</i> des Menüeintrags an der Stelle <i>index</i> zurück.
<i>entryconfig(index[, cnf][, **kw])</i>	Ruft die <i>configure</i> -Methode des Menüeintrags an der Stelle <i>index</i> mit den übergebenen Schlüsselwortparametern auf. Alternative Schreibweise: <i>entryconfigure</i>
<i>post(x, y)</i>	Zeigt das Menü an der Position an, die durch <i>x</i> und <i>y</i> bestimmt wird. Auf diese Weise lässt sich zum Beispiel ein Kontextmenü realisieren. Die Koordinaten <i>x</i> und <i>y</i> verstehen sich relativ zur oberen linken Ecke des Bildschirms.
<i>unpost()</i>	Versteckt das Menü wieder.

Tabelle 23.15 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 wird eine Instanz der Klasse `tkinter.Menubutton` erzeugt. Die Beschriftung der Menubutton-Schaltfläche wird dabei über die Option `text` festgelegt. Danach

erstellen wir das Menü, das beim Betätigen des Menubuttons angezeigt werden kann. Dabei handelt es sich um eine `tkinter.Menu`-Instanz, die 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 sieht wie folgt aus:

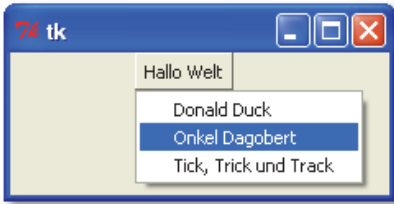


Abbildung 23.19 Ein Menubutton im Einsatz

Die Beschriftung des Menubuttons ändert sich unabhängig von dem im Menü ausgewählten Eintrag nicht.

Wie im Beispielprogramm 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. 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, instanziiieren 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 folgendermaßen aus:



Abbildung 23.20 Ein OptionMenu-Widget im Einsatz

Um auf eine Auswahl des Benutzers reagieren zu können, existiert wieder die `Option command`. Über diese Option kann eine Handlerfunktion angegeben werden, die immer dann vom Tk-Framework gerufen wird, wenn der Benutzer eine Auswahl getroffen hat. Die Handlerfunktion muss einen Parameter, den Text des aktuell ausgewählten Elements, erwarten.

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 Widgettypen, 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(50)])
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 Mauseisen 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 der folgenden Abbildung dargestellt:

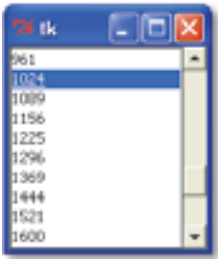


Abbildung 23.21 Eine Listbox mit angebundener Scrollbar

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.

Die hier demonstrierte Möglichkeit, eine Scrollbar anzubinden, funktioniert nicht nur bei Listbox-, 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 dieser 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 Schalt-

flächen 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 instanziiert, 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. Die Option *from* muss, wenn sie in Form eines Schlüsselwortparameters bei der Instanziierung von `tkinter.Spinbox` oder einem Aufruf der *configure*-Methode übergeben wird, als *from_* geschrieben werden. Dies ist wichtig, da *from* in Python ein Schlüsselwort ist.

Die in dem obigen Beispiel erzeugte Spinbox ist in der folgenden Grafik abgebildet:



Abbildung 23.22 Eine Spinbox im Einsatz

Statt 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. Die Reihenfolge, in der die Zahlen in der Spinbox erscheinen, ist durch die Reihenfolge der Werte im Tupel gegeben. Es muss kein Zusammenhang zur tatsächlichen Ordnungsrelation der Zahlen gegeben sein.

Wenn die Werte, die die Spinbox annehmen kann, konkret angegeben werden, können dort auch andere Datentypen als `int` verwendet werden:

```
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
<i>format</i>	"% <i>pad1</i> . <i>pad2</i> f"	Legt die Formatierung der Ausgabe fest. Dabei bezeichnen <i>pad1</i> die Anzahl der für den Vorkommaanteil reservierten Ziffern und <i>pad2</i> die Zahl der darzustellenden Nachkommastellen.
<i>from</i>	float	untere Grenze für den in der Spinbox dargestellten Wert
<i>increment</i>	float	Legt den Wert fest, um den der Wert bei einem Klick auf die kleinen Buttons der Spinbox inkrementiert bzw. dekrementiert werden soll.
<i>to</i>	float	obere Grenze für den in der Spinbox dargestellten Wert
<i>values</i>	tuple, list	Legt die möglichen Werte der Spinbox konkret fest.
<i>xscrollcommand</i>	Funktionsobjekt	Möglichkeit zur Anbindung einer horizontalen Scrollbar. Näheres dazu erfahren Sie im Abschnitt über die Scrollbar.

Tabelle 23.16 Spezifische Optionen eines Spinbox-Widgets

Text

Bisher haben wir nur eine Möglichkeit kennen gelernt, mithilfe des Entry-Widgets einzeilige 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, formatierten 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 instanziiert und gepackt. Danach definieren wir sogenannte Tags, die es uns später erlauben, den darzustellenden Text zu formatieren. In diesem Fall definieren wir das Tag `o` für orangefarbenen 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 orange und die dritte unterstrichen angezeigt werden. Die folgende Abbildung zeigt, wie der Text dargestellt wird:

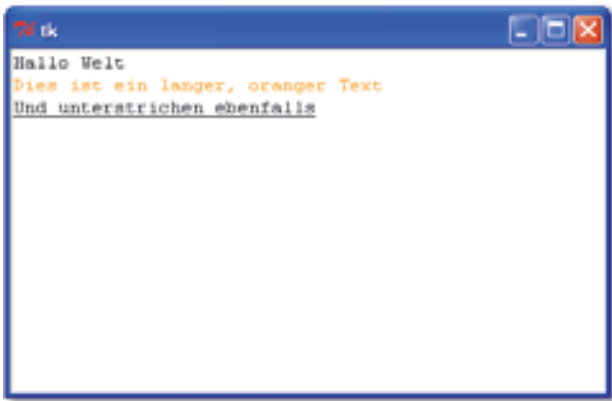


Abbildung 23.23 Das Text-Widget

Standardmäßig ist es dem Benutzer erlaubt, den im Text-Widget dargestellten Text zu verändern. Das folgende Beispiel zeigt, wie man auf eine Eingabe des Benutzers im Text-Widget reagiert und den eingegebenen Text ausliest:

```

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 man den im Widget dargestellten Text auslesen kann. Es handelt sich dabei um den reinen Text, jegliche Formatierungsanweisungen gehen beim Auslesen mittels *get* verloren.

Im Beispielprogramm wurde ein Eventhandler 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 verwenden, würde unser Eventhandler aufgerufen, bevor das vom Benutzer eingegebene Zeichen ins Text-Widget eingetragen wurde.

Im Eventhandler *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* bezeichnet selbstverständlich 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.

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. In Abschnitt 23.2.7 (S. 818) besprechen wir das Modul *scrolledtext*, das ein Text-Widget bereitstellt, das bereits mit Scrollbars ausgestattet ist.

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. Die positive Y-Achse dieses Koordinatensystems zeigt nach unten. Das Koordinatenpaar (50, 100) bezeichnet also den Punkt, der 50 Pixel rechts und 100 Pixel unterhalb der oberen linken Ecke des Canvas-Widgets liegt.

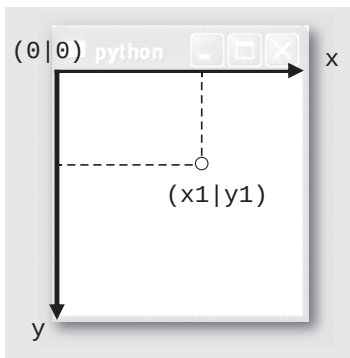


Abbildung 23.24 Das TkInter-Koordinatensystem

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 den *create*-Methoden Optionen in Form von Schlüsselwortparametern übergeben werden, die das Aussehen der gezeichneten Form spezifizieren. In

diesem Fall werden die Optionen *fill* für die Füllfarbe und *width* für die Stiftdicke gesetzt.

Die von diesem Code erzeugte Zeichnung sieht folgendermaßen aus:

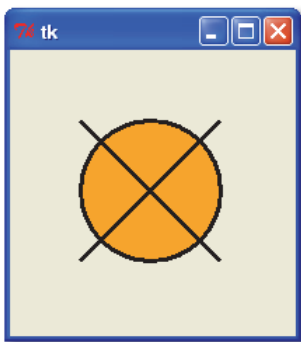


Abbildung 23.25 Eine Zeichnung in einem Canvas-Widget

Im Folgenden möchten wir die *create*-Methoden, die das Canvas-Widget bereitstellt besprechen und gegebenenfalls ein Beispiel präsentieren.

Methode	Bedeutung	Seite
<i>create_arc</i> (<i>x1</i> , <i>y1</i> , <i>x2</i> , <i>y2</i> , **kw)	Zeichnet ein Teilstück einer Ellipse.	808
<i>create_image</i> (<i>x</i> , <i>y</i> , **kw)	Zeichnet eine externe Grafik in das Canvas-Widget.	810
<i>create_line</i> (*coords , **kw)	Zeichnet einen Linienzug.	811
<i>create_oval</i> (<i>x1</i> , <i>y1</i> , <i>x2</i> , <i>y2</i> , **kw)	Zeichnet eine Ellipse.	812
<i>create_polygon</i> (*coords , **kw)	Zeichnet ein Polygon.	812
<i>create_rectangle</i> (<i>x1</i> , <i>y1</i> , <i>x2</i> , <i>y2</i> , **kw)	Zeichnet ein Rechteck.	813
<i>create_text</i> (<i>x</i> , <i>y</i> , **kw)	Schreibt einen Text.	814
<i>create_window</i> (<i>x</i> , <i>y</i> , **kw)	Platziert ein Steuerelement im Canvas-Widget.	815

Tabelle 23.17 Methoden des Canvas-Widgets

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 folgende Grafik zeigt die Resultate dreier Aufrufe von *create_arc*, bei denen lediglich die *style*-Option verändert wurde.



Abbildung 23.26 Von links nach rechts: pieslice, chord und arc

Um festzulegen, welches Teilstück der Ellipse gezeichnet werden soll, sind die Optionen *start* und *extent* notwendig. Über die *start*-Option kann der Winkel bestimmt werden, ab dem das zu zeichnende Teilstück beginnt. Die *extent*-Option ist ebenfalls ein Winkel und bezeichnet die Ausdehnung des Teilstücks. Die Winkel werden in Grad gemessen und gegen den Uhrzeigersinn abgetragen. Die Optionen *start* und *extent* sind mit 0.0 bzw. 90.0 vorbelegt.

Die folgende Tabelle listet die wichtigsten Optionen auf, die der *create_arc*-Methode übergeben werden können:

Option	Wertebereich	Bedeutung
<i>dash</i>	list, tuple	Legt ein Muster für die Umrandung des Teilstücks fest. Ein Wert von (4,1,6,2) bedeutet beispielsweise, dass nach 4 Pixeln Linie eine Lücke von einem Pixel und nach 6 darauf folgenden Pixeln eine Lücke von 2 Pixeln folgt. Danach beginnt das Muster von vorne.
<i>extent</i>	float	Der Ausdehnungswinkel des Teilstücks. Die Option ist mit 90.0 vorbelegt.
<i>fill</i>	"#RRGGBB"	die Füllfarbe des Teilstücks
<i>outline</i>	"#RRGGBB"	die Farbe der Umrandung
<i>start</i>	float	der Winkel, ab dem das Teilstück beginnt

Tabelle 23.18 Optionen der *create_arc* Methode

Option	Wertebereich	Bedeutung
<i>style</i>	"pieslice", "chord", "arc"	Der Typ des Teilstücks. In Abbildung 23.26 sind die drei möglichen Typen dargestellt.
<i>width</i>	float	die Breite der Umrandung

Tabelle 23.18 Optionen der `create_arc` Methode (Forts.)

Neben den hier vorgestellten Optionen existieren noch 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, wird anhand des folgenden Beispielprogramms erklärt:

```
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.⁹ 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.

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).

⁹ Wenn Sie weitere Grafikformate benötigen, sollten Sie sich mit der *Python Imaging Library* (PIL) auseinander setzen, die eine zu `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 von obigem Beispielprogramm erzeugte grafische Oberfläche sieht folgendermaßen aus:



Abbildung 23.27 Eine Grafik im Canvas-Widget

`create_line(*coords, **kw)`

Die Methode `create_line` zeichnet einen Linienzug ins Canvas-Widget. Dazu werden 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 sieht folgendermaßen aus:



Abbildung 23.28 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, welche den oberen linken und den unteren rechten Eckpunkt des die Ellipse umfassenden Rechtecks angeben. Die folgende Abbildung zeigt eine mittels `create_oval` gezeichnete Ellipse samt umfassenden Rechteck und Lage der beiden Eckpunkte:¹⁰

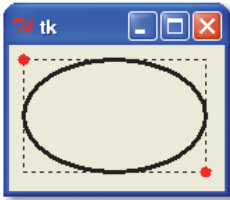


Abbildung 23.29 Die beiden Eckpunkte beschreiben die Ellipse

Der Methode `create_oval` können dieselben Optionen übergeben werden 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")
```

¹⁰ Das umfassende Rechteck und die Eckpunkte werden nicht von `create_oval` gezeichnet, sondern wurden zur Veranschaulichung hinzugefügt.

Das resultierende Polynom sieht folgendermaßen aus:



Abbildung 23.30 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, nichtkonvexe Polygone¹¹ zu zeichnen und zu füllen, wie die folgende Grafik zeigt:



Abbildung 23.31 Ein nichtkonvexes Polynom

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.

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.

¹¹ Ein nichtkonvexes Polygon ist ein Polygon, das zwei Punkte enthält, deren Verbindungslinie nicht vollständig innerhalb des Polygons verläuft. Nichtkonvexe Polygone werden häufig als konkav bezeichnet, was streng genommen falsch ist: Ein konkaves Polygon ist ein Polygon, dessen Komplement konvex ist.

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 23.2.7 (S. 818) im Abschnitt »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, jeweils mit einer anderen Schriftart, auf. Der zu schreibende Text wird über die Option `text` und der Schrifttyp über die Option `font` angegeben. Das Beispielprogramm erzeugt folgende grafische Oberfläche:



Abbildung 23.32 Text in einem Canvas-Widget

In der folgenden Tabelle möchten wir eine Übersicht über die wichtigsten Optionen geben, die im Zusammenhang mit der `create_text`-Methode verwendet werden können.

Option	Wertebereich	Bedeutung
<i>anchor</i>	"n", "ne", "e", "se", "s", "sw", "w", "nw", "center"	Gibt an, in welcher Relation die bei <i>create_text</i> angegebene Position zu dem zu schreibenden Text stehen soll. Standardmäßig ist diese Option mit "center" vorbelegt, das heißt, die Mitte des Textes liegt an der angegebenen Position.
<i>fill</i>	"#RRGGBB"	die Schriftfarbe
<i>font</i>	tuple	die Schriftart
<i>justify</i>	"left", "right", "center"	Gibt bei mehrzeiligen Texten an, wie die einzelnen Zeilen zueinander ausgerichtet werden sollen. Standardmäßig werden sie linksbündig (left) ausgerichtet.
<i>text</i>	str	der darzustellende Text
<i>width</i>	int	Gibt an, wie breit eine Textzeile maximal sein darf. Längere Zeilen werden umbrochen. Dieser Wert wird in Pixeln angegeben.

Tabelle 23.19 Optionen der *create_text* Methode**create_window(x, y, **kw)**

Die Methode *create_window* ermöglicht es, Steuerelemente innerhalb eines Canvas-Widgets zu platzieren. 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 instanziiieren 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 selbstverständlich sein und erzeugt die folgende GUI:



Abbildung 23.33 Ein in einem 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 man nachträglich auf das erstellte Element zugreifen kann.

Verändern von Zeichnungen

Es gibt einige Methoden, mit deren Hilfe sich ein einmal gezeichnetes Element modifizieren lässt:

Methode	Bedeutung
<i>delete(index)</i>	Löscht das Element mit dem Index <i>index</i> aus dem Canvas-Widget.
<i>itemconfig(index, **options)</i>	Ermöglicht das Anpassen der Optionen des Elements mit dem Index <i>index</i> .
<i>move(index, dx, dy)</i>	Verschiebt das Element mit dem Index <i>index</i> um <i>dx</i> Pixel nach rechts und um <i>dy</i> Pixel nach unten.

Tabelle 23.20 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 Kapitel einen umfassenden Überblick über die Möglichkeiten bekommen, die die Steuerelemente des Tk-Frameworks bieten.

23.2.6 Die Klasse Tk

Die Klasse Tk, die in jeder tkinter-Anwendung instanziiert wird, repräsentiert ein sogenanntes Toplevel-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*¹² 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
<i>wm_geometry([geom])</i>	Setzt Position und Größe des Fensters. Der Parameter <i>geom</i> muss ein String der Form " <i>widthx-height+x+y</i> " sein, wobei <i>width</i> und <i>height</i> der gewünschten Breite und Höhe entsprechen. Die Angaben <i>x</i> und <i>y</i> bestimmen die Position des Fensters relativ zur linken oberen Ecke des Bildschirms. Wenn <i>geom</i> nicht übergeben wird, werden die aktuelle Dimension und Position des Fensters in Form des oben genannten Strings zurückgegeben.
<i>wm_iconify()</i>	Minimiert das Fenster in die Taskleiste des Betriebssystems.
<i>wm_deiconify()</i>	Holt das minimierte Fenster wieder hervor. Wenn das Fenster nicht minimiert war, passiert nichts.
<i>wm_resizable([width[, height]])</i>	Legt fest, ob der Benutzer die Fenstergröße in horizontaler (<i>width</i>) oder vertikaler (<i>height</i>) Richtung verändern darf. Wenn die Parameter <i>width</i> und <i>height</i> nicht angegeben werden, wird das aktuelle Verhalten in Form eines zweielementigen Tupels zurückgegeben.
<i>wm_title([title])</i>	Setzt den Fenstertitel auf <i>title</i> . Wenn <i>title</i> nicht angegeben wurde, wird der aktuelle Fenstertitel zurückgegeben.

Tabelle 23.21 Methoden zur Kommunikation mit dem Window-Manager

12 Ein Window-Manager ist die Komponente eines grafischen Betriebssystems, die sich um die Anordnung und Darstellung der Fensterrahmen kümmert (nicht um deren Inhalt).

Die Tk- und die Widget-Klasse haben eine gemeinsame Basisklasse, weswegen die in Abschnitt 23.2.5 im Abschnitt über die Widget-Klasse eingeführten *winfo*-Methoden auch bei jeder Tk-Instanz verfügbar sind.

23.2.7 Weitere Module

Zum Schluss möchten wir noch einen Überblick über weitere, im `tkinter`-Paket enthaltenen Module geben, da diese einige durchaus interessante Bereiche abdecken und in mancher Hinsicht die Arbeit mit dem Tk-Framework erleichtern.

Im Folgenden werden die Module `scrolledtext`, `filedialog`, `font`, `messagebox` und `simplifiedialog` besprochen.

`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 Instanzieren 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, sodass 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, folgendermaßen aus:

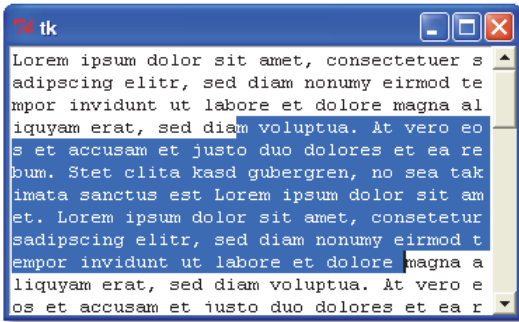


Abbildung 23.34 Das ScrolledText-Widget

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 kann man 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.filedialog

Bei der Programmierung grafischer Benutzeroberflächen 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 keinen 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 Desktopumgebung zu nutzen.

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 hineinzuladen oder Inhalte dorthin zu speichern. Dateidialoge werden ständig benötigt.

Das Modul `filedialog` des Pakets `tkinter` stellt vorgefertigte Dateidialoge 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.

Funktion	Standarddialog für	Seite
<i>askdirectory(**options)</i>	die Auswahl eines Verzeichnisses	820
<i>askopenfilename(**options)</i>	die Auswahl einer Datei	820
<i>asksaveasfilename(**options)</i>	die Auswahl eines Speicherorts für eine Datei	821

Tabelle 23.22 Standarddialoge im Modul `tkinter.filedialog`

`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 einige Optionen übergeben werden, die das Verhalten des Dialogs steuern. Die wichtigsten dieser Optionen möchten wir in der folgenden Tabelle vorstellen:

Option	Wertebereich	Bedeutung
<i>initialdir</i>	str	Das Verzeichnis, das im Dialog voreingestellt ist. Standardmäßig ist das Arbeitsverzeichnis des Programms voreingestellt.
<i>mustexist</i>	bool	Gibt an, ob das vom Benutzer ausgewählte Verzeichnis existieren muss oder nicht.
<i>title</i>	str	der Dialogtitel

Tabelle 23.23 Optionen der `askdirectory`-Funktion

`askopenfilename(**options)`

Diese Funktion 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 der Funktion `askopenfilename` in Form von Schlüsselwortparametern übergeben werden können:

Option	Wertebereich	Bedeutung
<i>defaultextension</i>	str	ein String, der an den eingegebenen Dateinamen angehängt wird, wenn er keine Dateierweiterung hat

Tabelle 23.24 Optionen der `askopenfilename`-Funktion

Option	Wertebereich	Bedeutung
<i>filetypes</i>	list, tuple	Ein Tupel der Form (("Textdateien", ".txt"), ("Alle Dateien", "*")). Der Benutzer kann die hier angegebenen Dateiendungen aus einer Liste im Dialog auswählen. Es werden dann nur noch die Dateien des aktuellen Verzeichnisses angezeigt, die dem ausgewählten Typ entsprechen.
<i>initialdir</i>	str	Das Verzeichnis, das im Dialog voreingestellt ist. Standardmäßig ist das Arbeitsverzeichnis des Programms voreingestellt.
<i>initialfile</i>	str	Gibt an, welche Datei im Dialog voreingestellt sein soll. Der Pfad zu dieser Datei muss entweder relativ zu <i>initialdir</i> oder absolut sein.
<i>multiple</i>	bool	Gibt an, ob es dem Benutzer erlaubt sein soll, mehrere Dateien auszuwählen. In diesem Fall wird von <i>askopenfilename</i> ein Tupel mit den Pfaden aller ausgewählten Dateien zurückgegeben.
<i>title</i>	str	Gibt den Dialogtitel an.

Tabelle 23.24 Optionen der *askopenfilename*-Funktion (Forts.)

Analog zur Funktion *askopenfilename* existiert die Funktion *askopenfile* im Modul *filedialog*. Diese Funktion verhält sich wie *askopenfilename* mit dem Unterschied, dass die ausgewählte Datei geöffnet und das geöffnete Dateiobjekt zurückgegeben wird. Dazu kann der Funktion *askopenfile* optional ein Positionsparameter übergeben werden, 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 *defaulttextension*, *filetypes*, *initialdir*, *initialfile* und *title* können auch bei *asksaveasfilename* verwendet werden und haben dieselbe Bedeutung wie bei *askopenfilename*. Zudem existiert auch die Funktion *asksaveasfile*, die die vom Benutzer ausgewählte Datei öffnet und das resultierende Dateiobjekt 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 23.2.5 im Abschnitt über das Canvas-Widget haben Sie gesehen, wie Sie einen beliebigen Text in das Canvas-Widget zeichnen können. Dabei konnte man 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 man beispielsweise im Zusammenhang mit der *font*-Option bei einem Aufruf der Methode *create_text* des Canvas-Widgets verwenden kann. Der Funktion *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 Messageboxen anzuzeigen. Eine *Messagebox* ist ein kleines Fenster mit einer Botschaft an den Benutzer. Eine *Messagebox* kann dazu verwendet werden, den Benutzer über einen Fehler zu informieren oder ihm eine Frage zu stellen. Es gibt mehrere Typen von Messageboxen, 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 Botschaft gestellte Frage beantworten kann.

Im Folgenden werden wir die im Modul *messagebox* enthaltenen Funktionen besprechen. Alle besprochenen Funktionen verfügen über dieselbe Schnittstelle, weswegen wir nur bei der ersten Funktion darauf eingehen werden.

Funktion	Bedeutung	Seite
<code>askokcancel([title[, message]][, **options])</code>	Messagebox mit den Schaltflächen »Ok« und »Abbrechen«	823
<code>askquestion([title[, message]][, **options])</code>	Messagebox mit den Schaltflächen »Ja« und »Nein«. Je nach Wahl des Benutzers gibt die Funktion "yes" bzw. "no" zurück.	–
<code>askretrycancel([title[, message]][, **options])</code>	Messagebox mit den Schaltflächen »Wiederholen« und »Abbrechen«. Je nach Wahl des Benutzers gibt die Funktion <code>True</code> (»Wiederholen«) bzw. <code>False</code> (»Abbrechen«) zurück.	–
<code>askyesno([title[, message]][, **options])</code>	Wie <code>askquestion</code> , mit dem Unterschied, dass je nach Wahl des Benutzers <code>True</code> (»Ja«) bzw. <code>False</code> (»Nein«) zurückgegeben wird.	–
<code>askyesnocancel([title[, message]][, **options])</code>	Messagebox mit den Schaltflächen »Ja«, »Nein« und »Abbrechen«. Je nach Wahl des Benutzers gibt die Funktion <code>True</code> (»Ja«), <code>False</code> (»Nein«) oder <code>None</code> (»Abbrechen«) zurück.	–
<code>showerror([title[, message]][, **options])</code>	Messagebox mit einem Fehlersymbol und der Schaltfläche »Ok«	–
<code>showinfo([title[, message]][, **options])</code>	Messagebox mit einem Informationssymbol und der Schaltfläche »Ok«	–
<code>showwarning([title[, message]][, **options])</code>	Messagebox mit einem Warnungssymbol und der Schaltfläche »Ok«	–

Tabelle 23.25 Standarddialoge im Modul `tkinter.filedialog`

`askokcancel([title[, message]][, **options])`

Die Funktion `askokcancel` zeigt eine Messagebox mit dem Fenstertitel `title` und der Botschaft `message` an. Die Messagebox kann über die Buttons »Ok« und »Abbrechen« geschlossen werden. Welchen der Buttons der Benutzer betätigt hat, kann über den Rückgabewert der Funktion erfahren werden. Die Funktion gibt `True` zurück, wenn der Benutzer die Messagebox über den Button »Ok« geschlossen hat, und `False` bei »Abbrechen«.

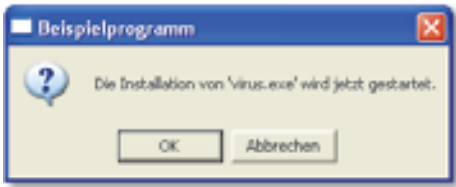


Abbildung 23.35 Eine mit askokcancel erzeugte MessageBox

Der Funktion *askokcancel* können einige Optionen in Form von Schlüsselwortparametern übergeben werden. Die folgende Tabelle listet die wichtigsten dieser Optionen auf:

Option	Wertebereich	Bedeutung
<i>default</i>	str	Legt den sogenannten Default-Button der Messagebox fest. Dieser Button wird grafisch hervorgehoben und kann über die Leertaste betätigt werden. Im Falle der <i>askokcancel</i> -Funktion sind die Werte "ok" und "cancel" für die default-Option möglich.
<i>detail</i>	str	Für die <i>detail</i> -Option kann ein Text angegeben werden, der nähere Angaben zur Botschaft der Messagebox enthält. Dieser Text wird unter die Botschaft geschrieben. Auf einigen Systemen wird die Botschaft grafisch vom <i>detail</i> -Text getrennt.
<i>icon</i>	"error", "info", "question", "warning"	Legt das Icon fest, das auf der linken Seite der Message Box angezeigt wird.

Tabelle 23.26 Optionen der askokcancel-Funktion

Diese Optionen sind auch für die folgenden Funktionen gültig.

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 der Funktion der

gewünschte Dialogtitel (*title*) und ein erklärender Text für den Benutzer (*prompt*) angegeben werden.

Zusätzlich können einige Optionen in Form von Schlüsselwortparametern übergeben werden. Die wichtigsten dieser Optionen möchten wir in der folgenden Tabelle vorstellen.

Option	Wertebereich	Bedeutung
<i>initialvalue</i>	float	Gibt den Wert an, der im Dialog voreingestellt sein soll.
<i>maxvalue</i>	float	Gibt den größten Wert an, den der Benutzer eingeben darf.
<i>minvalue</i>	float	Gibt den kleinsten Wert an, den der Benutzer eingeben darf.

Tabelle 23.27 Optionen der askfloat-Funktion

Der von der *askfloat*-Funktion erzeugte Dialog ist in der folgenden Abbildung zu sehen.

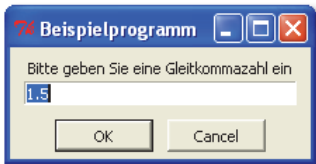


Abbildung 23.36 Ein mit askfloat erzeugter Dialog

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* den Wert *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 Rückgabetyt identisch.

askstring(title, prompt, **kw)

Die Funktion *askstring* erzeugt einen Dialog, der den Benutzer zur Eingabe eines Strings auffordert.

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.

23.3 Einführung in PyQt

Im vorangegangenen Abschnitt wurde das Modul `tkinter` besprochen, mit dessen Hilfe sich Tk-GUIs unter Python schreiben lassen. Der große Vorteil von `tkinter` ist, dass es in der Standardbibliothek enthalten ist und somit nicht separat installiert werden muss. Demgegenüber steht, dass Tk nicht mehr zeitgemäß ist und sich eher zum Schreiben von Prototypen oder kleineren GUI-Anwendungen eignet. Als Gegenpol zu Tk werden wir in diesem Kapitel eine projektorientierte Einführung in das umfangreiche Qt-Framework bringen.

Sie haben in Abschnitt 23.1 bereits einen Überblick darüber erhalten, was das Framework *Qt* (sprich *cute*, dt. »pfiffig«) und damit auch die Python-Bindings *PyQt* ausmacht. Dieser grobe Überblick wird hier verfeinert, wobei wir die Konzepte und Stärken von Qt vor allem von der technischen Seite her beleuchten werden.

Dazu beschäftigen wir uns im nächsten Abschnitt zunächst mit der Frage, wo und wie Qt und PyQt bezogen und installiert werden können. Danach geben wir eine Übersicht über die grundlegenden Konzepte des Qt-Frameworks.

23.3.1 Installation

Um Qt-Anwendungen mit Python schreiben zu können, müssen zwei Komponenten installiert werden: das Qt-Framework und die zugehörigen Python-Bindings. Sollten Sie beides noch nicht auf Ihrem PC installiert haben und ein Windows-Betriebssystem einsetzen, können Sie das Komplettpaket von der Website des PyQt-Entwicklers unter <http://www.riverbankcomputing.com> herunterladen. Eine separate Qt-Installation ist dann nicht mehr vonnöten. Alternativ können Sie PyQt von der CD installieren, die diesem Buch beiliegt.

Wenn aus gewissen Gründen das Komplettpaket für Sie nicht in Frage kommt, beispielsweise weil Sie Linux einsetzen, können Qt und PyQt getrennt voneinander installiert werden. Qt kann auf der Website <http://qt.nokia.com> von Nokia kostenlos heruntergeladen werden. Alternativ können Sie Qt von der CD installieren, die diesem Buch beiliegt.

Qt und PyQt stehen unter einem dualen Lizenzsystem. Projekte, die Sie mit den angebotenen freien Versionen von Qt und PyQt entwickeln, dürfen Sie nur unter einer ebenfalls freien Lizenz, beispielsweise der GPL, veröffentlichen. Um ein kommerzielles Programm veröffentlichen zu dürfen, muss eine Lizenzgebühr entrichtet werden.

**Hinweis**

Sollten Sie während der Installation von Qt eine Warnung bekommen, dass keine MinGW-Installation auf Ihrem Rechner gefunden wurde, können Sie diese ignorieren, sofern Sie ausschließlich mit PyQt arbeiten möchten. MinGW ist eine freie Compilerdistribution für C und C++.

23.3.2 Grundlegende Konzepte von Qt

Als Einführung in die Programmierung mit Qt bzw. PyQt wird in diesem Abschnitt eine Übersicht über die wichtigsten Konzepte und Stärken des Qt-Frameworks gegeben.

Umfang

Ihnen wird bereits aufgefallen sein, dass im Zusammenhang mit Qt nicht von einem Toolkit, sondern von einem Framework gesprochen wurde. Das hängt damit zusammen, dass der überwiegende Teil der Klassen, die das Qt-Framework enthält, nichts mit der Programmierung grafischer Benutzeroberflächen zu tun hat, sondern anderweitige nützliche Funktionalität bereitstellt. So enthält das Qt-Framework beispielsweise Klassen zum Arbeiten mit XML-Daten oder zur Netzkommunikation. Viele dieser Klassen sind zwar in Kombination mit Python aufgrund der Standardbibliothek faktisch überflüssig, bieten aber in Programmiersprachen wie C++, das über keine so umfangreiche Standardbibliothek verfügt, einen erheblichen Mehrwert.

Signale und Slots

Einer der größten Unterschiede zu anderen GUI-Toolkits ist das Signal-und-Slot-Prinzip, das Qt zur Kommunikation einzelner Objekte untereinander einsetzt. Bei jedem Ereignis, das in einem Qt-Objekt auftritt, beispielsweise beim Anklicken einer Schaltfläche, wird ein *Signal* gesendet, das dann von verbundenen *Slots* empfangen werden kann. Signale und Slots sind flexibler als das Überladen von Methoden, das in anderen Toolkits zur Kommunikation verwendet wird.

Näheres zum Signal-und-Slot-Prinzip erfahren Sie in Abschnitt 23.4.

Layouts

Das Qt-Framework unterstützt *Layouts* in der grafischen Oberfläche eines Programms. Mithilfe eines Layouts lassen sich Steuerelemente relativ zueinander automatisch positionieren. Diese Gruppe von Steuerelementen kann dann gemeinsam

verschoben oder in der Größe verändert werden, ohne ihre relative Ausrichtung zueinander zu verlieren.

Näheres zu Layouts erfahren Sie in Abschnitt 23.3.3.

Zeichenfunktionen

Qt stellt umfangreiche Funktionalität zum Zeichnen in der grafischen Benutzeroberfläche bereit. So erlaubt Qt das Zeichnen verschiedenster Formen mit verschiedensten Arten der Füllung oder des Linienstils. Darüber hinaus bietet Qt Möglichkeiten zur Transformation von Zeichnungen mithilfe von Transformationsmatrizen, was erstaunliche Effekte ermöglicht und wodurch sich Qt von vielen anderen GUI-Toolkits abhebt. Ebenso ermöglicht Qt das Lesen und Schreiben vieler Grafikformate, darunter vor allem auch des Vektorformats SVG.

Näheres zum Zeichnen mittels Qt erfahren Sie in Abschnitt 23.6.

Das Model-View-Konzept

Qt implementiert das *Model-View-Konzept*, das eine Trennung von Form und Inhalt ermöglicht. So ist es in Qt möglich, die Daten, die ein Programm verarbeitet, in einer eigenen Klassenstruktur zu speichern – getrennt von der Funktionalität, die diese Daten anzeigt. Auf diese Weise wird der Programmaufbau insgesamt übersichtlicher, und die unabhängige Weiterentwicklung und Wiederverwertung einzelner Komponenten wird erleichtert.

Näheres zum Model-View-Konzept erfahren Sie in Abschnitt 23.7.

QML

Seit Version 4.7 enthält Qt mit QML eine an CSS angelehnte Skriptsprache, die das Gestalten von grafischen Oberflächen und das Erstellen von einfachen Programmlogiken auf abstrakter Ebene ermöglicht.

Werkzeuge

Der nächste herausragende Bereich von Qt sind die mitgelieferten Werkzeuge. Bei einer Qt-Installation werden für gewöhnlich die Programme *Qt Designer*, *Qt Assistant* und *Qt Linguist* mit installiert. Ersteres ist ein Programm zur komfortablen Gestaltung von grafischen Benutzeroberflächen. Wir werden im Laufe dieses Kapitels noch auf den Qt Designer zurückkommen.

Die ebenfalls mitgelieferten Programme Qt Assistant und Qt Linguist werden in diesem Buch nicht besprochen. Es sei nur erwähnt, dass es sich bei Qt Assistant um ein

Tool zum Lesen der Qt-Hilfe und bei Qt Linguist um ein Hilfsmittel zur Lokalisierung von Qt-Anwendungen handelt.

Dokumentation

Als letzter Punkt ist die Dokumentation des Qt-Frameworks zu nennen, die entweder Ihrer Qt-Installation beiliegt oder im Internet unter <http://doc.qt.nokia.com> zu finden ist. Die Qt-Dokumentation ist sehr umfangreich und mit vielen Beispielen versehen.

Mit der PyQt-Installation erhalten Sie zudem eine auf Python abgestimmte Version der Qt-Dokumentation, in der alle Schnittstellen und ein Teil der Beispiele für Python beschrieben werden. Auch diese Dokumentation ist mittlerweile umfangreich genug, um mit ihr arbeiten zu können. Beachten Sie, dass in der PyQt-Dokumentation viele Beispiele noch nicht von C++ nach Python übersetzt worden sind.

23.3.3 Entwicklungsprozess

In diesem Abschnitt wird der vollständige Entwicklungsprozess einer einfachen PyQt-Anwendung dargestellt. Auf dem hier erarbeiteten Wissen werden wir später aufbauen, wenn es an eine komplexere Anwendung geht.

Bei dem Beispielprogramm, das in diesem Kapitel entwickelt wird, handelt es sich um ein Formular, das einige persönliche Daten vom Benutzer einliest.

Erstellen des Dialogs

In diesem Beispiel sollen Vorname, Nachname, Geburtsdatum und Adresse des Benutzers eingelesen und gespeichert werden. Zusätzlich soll der Benutzer die allgemeinen Geschäftsbedingungen unseres Pseudo-Portals akzeptieren müssen und optional einen Newsletter bestellen können.

Auf Basis dieser Voraussetzungen können wir im nächsten Schritt einen *Dialog* erstellen. Ein Dialog ist ein einzelnes Fenster einer grafischen Oberfläche und enthält mehrere *Widgets*. Unter einem Widget (dt. »Dingsbums«) versteht man ein einzelnes Steuer- oder Bedienelement der grafischen Oberfläche. Ein Widget kann beliebig viele untergeordnete Widgets enthalten, sogenannte *Children* (dt. »Kinder«). Im Dialog aus Abbildung 23.37 sind beispielsweise die Eingabefelder, die ihrerseits ebenfalls Widgets sind, dem Gruppierungswidget »Persönliche Daten« untergeordnet.

Abbildung 23.37 zeigt den Hauptdialog unserer ersten Anwendung und erklärt die Namen und Bedeutungen der einzelnen Widgets.

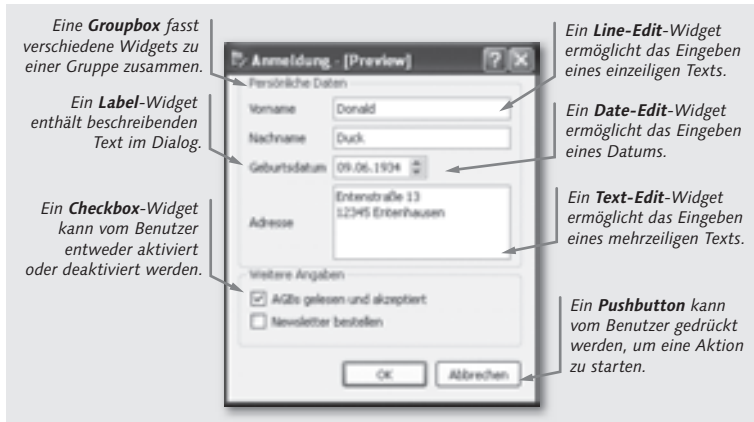


Abbildung 23.37 Die verschiedenen Widgets des Hauptdialogs

Jetzt können wir damit beginnen, den Dialog zu erstellen. Qt stellt dafür ein komfortables Entwicklungswerkzeug, den *Qt Designer*, bereit. Mithilfe des Qt Designers lassen sich auch komplexe Dialoge problemlos editieren.

Unter Windows finden Sie das Programm Qt Designer im Startmenü unter dem Eintrag »Qt« bzw. »PyQt«. Sollten Sie Linux oder MacOS X einsetzen, müssen Sie den Qt Designer gegebenenfalls über den Befehl `designer` aus einer Shell heraus starten.

Nach dem Starten des Designers wird die Möglichkeit angeboten, ein sogenanntes *Template* (dt. »Schablone«) zu laden. Wir entscheiden uns in diesem Fall für das Template »Dialog without buttons«, das einen vollständig leeren Dialog bereitstellt. Danach präsentiert sich der Qt Designer so wie in Abbildung 23.38.

Die Arbeitsfläche des Qt Designers lässt sich grob in drei Bereiche unterteilen:

- ▶ In der Mitte finden Sie Ihre Dialogvorlage, die zu diesem Zeitpunkt noch leer ist.
- ▶ Auf der linken Seite befindet sich eine Liste aller verfügbaren Steuerelemente. Um eines dieser Steuerelemente zum Dialog hinzuzufügen, wählen Sie es aus und ziehen es auf den Dialog. Nachdem ein Steuerelement zum Dialog hinzugefügt worden ist, kann es beliebig positioniert oder in seiner Größe verändert werden.
- ▶ Auf der rechten Seite können die Eigenschaften des aktuell ausgewählten Steuerelements bearbeitet werden.

Wir beginnen damit, die beiden Schaltflächen »OK« und »Abbrechen« im Dialog zu platzieren. Dazu ziehen Sie zwei Buttons aus der Liste am linken Rand und platzieren sie im Dialog. Die Aufschrift der Schaltflächen lässt sich durch einen Doppelklick auf den bisherigen Text verändern. Um die Schaltfläche »OK« zur Standardschaltfläche des Dialogs zu machen, ändern Sie die Eigenschaft »default« des Buttons in der Liste auf der rechten Seite auf »true«. Ziehen Sie danach zwei Groupboxen auf den Dialog, und pas-

sen Sie Größe und Position Ihren Wünschen an. Der Titel einer Groupbox lässt sich durch einen Doppelklick auf den bisherigen Text anpassen (siehe Abbildung 23.39).

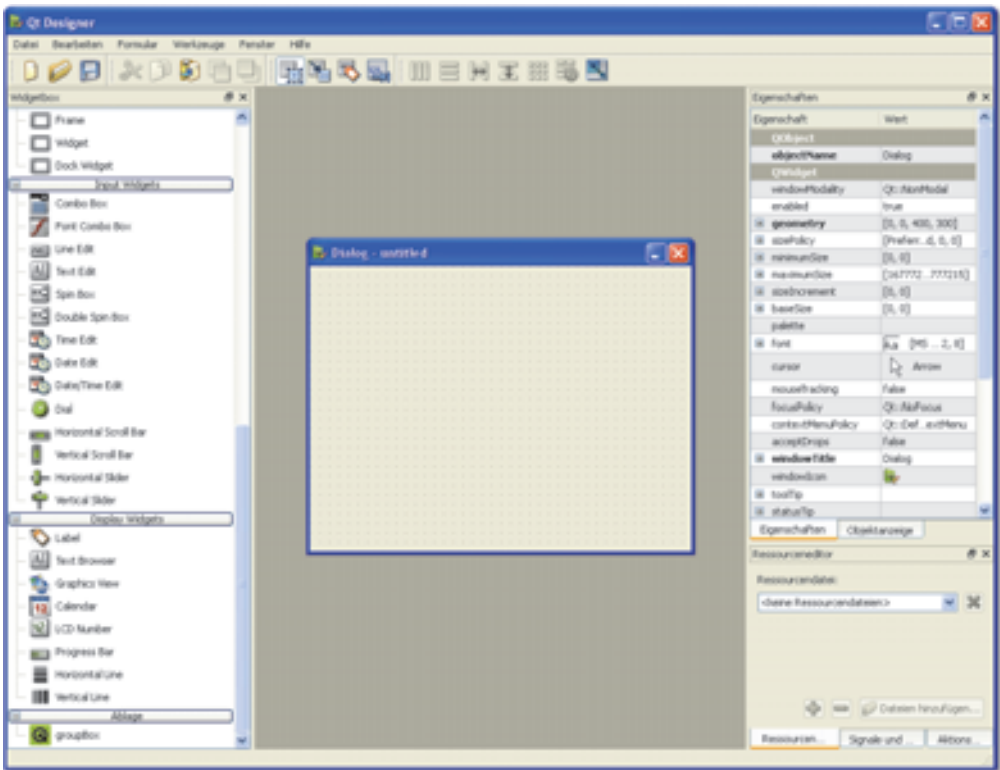


Abbildung 23.38 Der Qt Designer

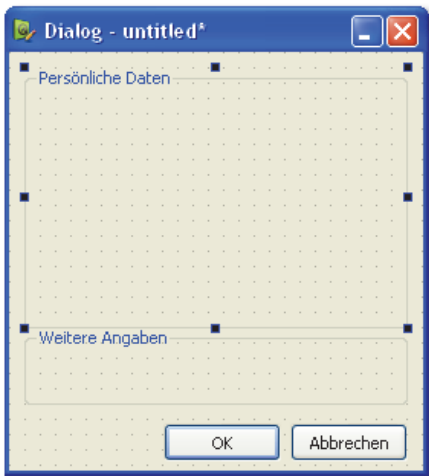


Abbildung 23.39 Dialog mit Groupboxen

Jetzt könnten im Prinzip weitere Widgets zum Dialog hinzugefügt werden, indem sie in eine der beiden Gruppen platziert werden. Das macht sie automatisch zu untergeordneten Widgets, die in ihrer jeweiligen Gruppe absolut positioniert werden. Damit sich die untergeordneten Widgets bei einer Größenänderung des Hauptdialogs automatisch anpassen, erstellen wir für sie ein Layout. Um ein Layout zu erstellen, markieren Sie alle Widgets, die dazugehören sollen, und wählen eine der Schaltflächen aus der Layout-Toolbar (siehe Abbildung 23.40).

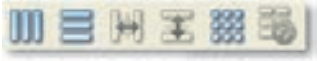


Abbildung 23.40 Die Layout-Toolbar

Ein Klick auf das erste Icon ordnet die Steuerelemente horizontal an, ein Klick auf das zweite Icon vertikal. Die beiden folgenden Icons erstellen einen sogenannten *Splitter* (dt. »Trenner«) zwischen den Steuerelementen, der später vom Benutzer verschoben werden kann. Dies ist für unseren Dialog nicht weiter wichtig und wird deshalb von uns nicht benötigt. Interessant ist noch die dritte Schaltfläche, die die markierten Steuerelemente tabellarisch anordnet. Ein Klick auf das letzte Icon löst ein bestehendes Layout auf.

Wenn Sie sich den Inhalt der Groupbox »Persönliche Daten« im geplanten Dialog noch einmal ansehen, werden Sie feststellen, dass es sich um einen zweispaltigen tabellarischen Aufbau handelt, bei dem stets eine Beschreibung der anzugebenden Daten links und das entsprechende Eingabefeld rechts steht.

Abbildung 23.41 Tabellarischer Aufbau

Um diese Anordnung der Steuerelemente durch ein Layout zu erzielen, fügen wir die Steuerelemente zunächst an ihrem ungefähren Platz in den Dialog ein. Dabei finden Sie die Beschriftungen unter dem Namen *Label* und die Eingabefelder unter den Namen *Editbox*, *Datebox* und *Textbox* im Steuerelemente-Menü des Qt Designers.

Nachdem die Steuerelemente ungefähr an ihrem Platz sind, markieren wir sie und klicken auf das oben beschriebene Icon in der Toolbar zum Aufbau eines tabellarischen Layouts. Es ist egal, in welcher Reihenfolge die Steuerelemente zuvor markiert wurden,

da Qt die Position im Layout anhand ihrer vorherigen Position im Dialog abschätzt. Kleinere Ungenauigkeiten in der Positionierung werden dabei automatisch korrigiert. Das fertige Layout wird durch einen roten Rahmen angezeigt und kann jetzt als Ganzes neu positioniert oder in der Größe verändert werden. Sie werden feststellen, dass alle enthaltenen Steuerelemente bei einer Größenänderung automatisch mit angepasst werden.

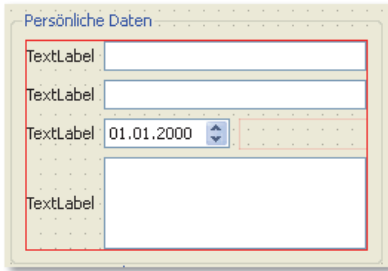


Abbildung 23.42 Nach dem Festlegen eines Layouts

Beachten Sie, dass dem Date Edit-Widget automatisch nur so viel Platz eingeräumt wurde, wie zum Anzeigen des Datums benötigt wird. Der dahinter eingefügte Leerraum wird durch ein dünnes rotes Rechteck gekennzeichnet. Wenn Sie selbst solche Aussparungen in ein Layout einbauen möchten, können Sie dies mithilfe eines *Spacers* tun, den Sie im Steuerelemente-Menü finden. Ein Spacer verhält sich wie ein Widget, wird aber im späteren Dialog nicht angezeigt.

Um den Text der Label-Widgets zu bearbeiten, klicken Sie, wie bei einer Groupbox, doppelt auf den bisherigen Text.

Analog zum eben Besprochenen können Sie das Hinzufügen der beiden Checkboxes angehen. In diesem Fall sollten Sie ein vertikales Layout wählen (siehe Abbildung 23.43).

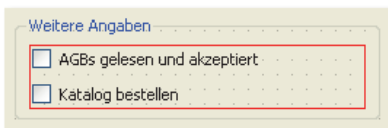


Abbildung 23.43 Checkboxes im vertikalen Layout

Damit ist der Dialog äußerlich gestaltet, doch eine wichtige Sache fehlt noch. Jedes Steuerelement benötigt einen Namen, unter dem es nachher im Programm angesprochen werden kann. Dieser Name kann im Qt Designer festgelegt werden. Um einem Widget einen neuen Namen zu verpassen, markieren Sie es, öffnen mit einem Rechtsklick das entsprechende Kontextmenü und wählen den Menüpunkt **OBJEKT-NAMEN ÄNDERN**.

Die folgende Tabelle listet die im Beispiel vergebenen Namen für alle wichtigen Steuerelemente auf. Steuerelemente, die nur aus Layoutgründen existieren, beispielsweise die Groupboxen und Labels, müssen nicht benannt werden, da später keine Operationen mit ihnen durchgeführt werden.

Steuerelement	Name
der Dialog selbst	Hauptdialog
das Line-Edit-Widget »Vorname«	vorname
das Line-Edit-Widget »Nachname«	nachname
das Date-Edit-Widget »Geburtsdatum«	geburtsdatum
das Text-Edit-Widget »Adresse«	adresse
das Checkbox-Widget »AGB«	agb
das Checkbox-Widget »Newsletter«	newsletter
das Button-Widget »OK«	buttonOK
das Button-Widget »Abbrechen«	buttonAbbrechen

Tabelle 23.28 Die Namen der wichtigen Steuerelemente

Nachdem alle Namen vergeben wurden, ist das Layout des Hauptdialogs fertig und kann im Projektverzeichnis gespeichert werden. Im Falle dieses Beispielsprogramms soll der Dialog unter dem Dateinamen *hauptdialog.ui* gespeichert werden.

Schreiben des Programms

Es gibt zwei Wege, den erstellten Dialog aus einem Programm heraus zu nutzen:

- Der in einer UI-Datei gespeicherte Dialog kann mithilfe des bei PyQt mitgelieferten Programms *pyuic4* in eine Python-Klasse überführt werden. Diese Klasse kann dann von einem Programm eingebunden und verwendet werden. Das Programm *pyuic4* bekommt den Pfad zur UI-Datei als Kommandozeilenparameter übergeben und gibt die daraus erstellte Python-Klasse standardmäßig auf dem Bildschirm aus. Um die Ausgabe in eine Programmdatei umzulenken, kann die Kommandozeilenoption *-o* verwendet werden:¹³

```
C:\Python32\Lib\site-packages\PyQt4\pyuic4 -o hauptdialog.py hauptdialog.ui
```

¹³ Bei vielen Linux-Distributionen kann das Konvertierungsprogramm nach der Installation über das Kommando *pyuic4* direkt gestartet werden, ohne einen Pfad angeben zu müssen.

- Die zweite, an dieser Stelle aufgrund ihrer Einfachheit bevorzugte Möglichkeit ist das Laden einer UI-Datei zur Laufzeit eines PyQt-Programms. Diese Variante wird im Folgenden anhand eines Beispielprogramms erläutert.

Um den vorbereiteten Dialog zu verwenden, erstellen wir die Programmdatei *programm.py*. Das einfachste Programm, das den Hauptdialog anzeigt und sonst keine weiteren Operationen durchführt, sieht folgendermaßen aus:

```
import sys
from PyQt4 import QtGui, uic

class MeinDialog(QtGui.QDialog):
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.ui = uic.loadUi("hauptdialog.ui", self)

app = QtGui.QApplication(sys.argv)
dialog = MeinDialog()
dialog.show()
sys.exit(app.exec_())
```

Zunächst werden alle benötigten Module eingebunden. Das sind insbesondere das Modul `QtGui`, in dem alle Klassen des Qt-Frameworks gekapselt sind, die thematisch mit grafischen Benutzeroberflächen zusammenhängen, und das Modul `uic`, mit dessen Hilfe die erstellte UI-Datei gelesen werden kann. Danach wird die Klasse `MeinDialog` erstellt, die von der Basisklasse aller Qt-Dialoge, `QDialog`, abgeleitet wird.

Der Konstruktor von `MeinDialog` ruft neben dem `QDialog`-Konstruktor die Funktion `loadUi` auf, um den Hauptdialog aus der UI-Datei *hauptdialog.ui* zu laden. Die als zweiter Parameter übergebene Instanz, in diesem Fall die `MeinDialog`-Instanz selbst, fungiert als Elternwidget des neuen Dialogs. Die Funktion `loadUi` gibt das erzeugte Widget zurück.

Im darauffolgenden Codeblock wird eine Instanz der Klasse `QApplication` erstellt, die den Rahmen einer Anwendung mit grafischer Benutzeroberfläche bereitstellt. Dazu gehört beispielsweise die *Main-Event-Loop*, die Hauptschleife der Anwendung. Dem Konstruktor der Klasse `QApplication` werden die Kommandozeilenparameter `sys.argv` übergeben. Jede Qt-Anwendung darf immer nur eine einzige `QApplication`-Instanz haben, unabhängig davon, wie viele Dialoge später angezeigt werden sollen.

Nachdem der Rahmen für die Qt-Anwendung erstellt wurde, kann der Hauptdialog erzeugt werden, indem die Klasse `MeinDialog` instanziiert wird. Durch die von der Basisklasse `QDialog` vererbte Methode `show` wird der Dialog sichtbar. Zu guter Letzt muss die eben angesprochene Main-Event-Loop durch die Methode `exec_` der

QApplication-Instanz gestartet werden. Da wir im Moment keine weiteren Operationen durchführen möchten, nachdem der Dialog vom Benutzer geschlossen wurde, geben wir den Rückgabewert von `app.exec_` direkt an `sys.exit` weiter und beenden damit das Beispielprogramm. Das Programm blockiert nach Aufruf von `app.exec_` solange, bis der Benutzer den Hauptdialog schließt.

Dieses einfache Beispielprogramm wird im Folgenden sinnvoll erweitert. So sollen die vom Benutzer eingegebenen Werte ausgelesen und in das neben der Benutzeroberfläche existierende Konsolenfenster geschrieben werden. Anhand der dazu notwendigen Erweiterungen wird im Folgenden das *Signal-und-Slot*-Konzept von Qt erklärt.

23.4 Signale und Slots

Beim Schreiben eines Programms mit grafischer Benutzeroberfläche wird das Prinzip der ereignisgesteuerten Programmierung angewandt. Dieses Prinzip sieht nicht vor, dass ein Programm sequentiell von oben nach unten abgearbeitet wird, sondern führt beim Auftreten bestimmter Ereignisse einen Codeabschnitt aus, der vom Programmierer für dieses Ereignis vorgesehen wurde. Die Anwendung der ereignisgesteuerten Programmierung ist im Falle einer grafischen Benutzeroberfläche notwendig, da hier der Benutzer das Programm steuert und nicht das Programm den Benutzer, wie es bei einer Konsolenanwendung der Fall war. Der Benutzer steuert das Programm durch seine Eingaben, die im Programm in Form von Ereignissen ankommen. Wann und in welcher Reihenfolge der Benutzer seine Eingaben macht, ist durch das Programm nicht vorgegeben.

In Qt finden sich zwei Techniken der ereignisgesteuerten Programmierung: zum einen sogenannte *Events* und zum anderen *Signale und Slots*. Beide Techniken werden im Folgenden besprochen.

Jedes Widget in der grafischen Benutzeroberfläche wird programmintern durch eine Instanz einer entsprechenden Qt-Klasse repräsentiert. Jede dieser Klassen bietet sogenannte *Eventhandler* an. Das sind Methoden, die der Programmierer in einer abgeleiteten Klasse überschreiben kann, um beim Eintreten eines speziellen Ereignisses (engl. *event*) eigenen Code ausführen zu können. Events werden nur für wenige Ereignisse verwendet, die aber häufig eintreten. Ein Beispiel für ein solches Ereignis ist das `paintEvent`, das immer dann eintritt, wenn der Inhalt eines Widgets neu gezeichnet werden muss. Das Widget reagiert auf das Event durch Ausführung seines Eventhandlers. Dies kann unter Umständen sehr häufig passieren. Ein Beispiel für die Implementation eines Eventhandlers finden Sie in Abschnitt 23.6 im Zusammenhang mit der Zeichenfunktionalität von Qt.

Neben den Events bietet das Qt-Framework *Signale und Slots* für die Behandlung von Ereignissen an. Dieses zentrale Konzept zur Kommunikation von Qt-Objekten ist womöglich das größte Unterscheidungsmerkmal zwischen Qt und anderen GUI-Toolkits.

Ein *Signal* wird von einem Widget gesendet, wenn ein bestimmtes Ereignis, beispielsweise eine Benutzereingabe, eingetreten ist. Es gibt für jedes Widget in Qt vordefinierte Signale für die meisten Anwendungsfälle. Zusätzlich ist es möglich, eigene Signale zu selbstbestimmten Ereignissen zu senden.

Um ein Signal zu empfangen, muss ein *Slot* (dt. »Steckplatz«) eingerichtet werden. Ein Slot ist eine Funktion oder Methode, die immer dann aufgerufen wird, wenn ein bestimmtes Signal gesendet wird. Dazu muss ein Slot mit einem Signal verbunden werden. Es ist möglich, einen Slot mit mehreren Signalen zu verbinden.

Im Folgenden wird das Beispiel des letzten Kapitels zu einer sinnvollen Anwendung erweitert. Diese Anwendung soll die Daten, die der Benutzer in den Dialog eingibt, in das parallel geöffnete Konsolenfenster ausgeben, sofern der Benutzer die Eingaben durch Klicken der OK-Schaltfläche bestätigt. Beim Klicken auf Abbrechen sollen keine Daten ausgegeben werden.

```
import sys
from PyQt4 import QtGui, QtCore, uic

class MeinDialog(QtGui.QDialog):
    def __init__(self):
        QtGui.QDialog.__init__(self)
        self.ui = uic.loadUi("hauptdialog.ui", self)

        # Slots einrichten
        self.ui.buttonOK.clicked.connect(self.onOK)
        self.ui.buttonAbbrechen.clicked.connect(self.onAbbrechen)

    def onOK(self):
        # Daten auslesen
        print("Vorname: {}".format(self.ui.vorname.text()))
        print("Nachname: {}".format(self.ui.nachname.text()))
        print("Adresse: {}".format(
            self.ui.adresse.toPlainText()))
        datum = self.ui.geburtsdatum.date().toString(
            "dd.MM.yyyy")

        print("Geburtsdatum: {}".format(datum))
```

```

    if self.ui.agb.checkState():
        print("AGBs akzeptiert")
    if self.ui.newsletter.checkState():
        print("Katalog bestellt")

    self.close()

    def onAbbrechen(self):
        print("Schade")
        self.close()

```

Im Konstruktor der Dialogklasse `MeinDialog` werden die `clicked`-Signale für den Ok- und den Abbrechen-Button mit den dafür vorgesehenen Slots verbunden. Die `clicked`-Signale werden immer dann ausgelöst, wenn der Benutzer die Schaltfläche betätigt. Die Signale, die ein Widget bereitstellt, sind als Attribute in der entsprechenden Widget-Instanz enthalten. Um ein Signal mit einem Slot zu verbinden, wird die Methode `connect` des Signals aufgerufen und der Slot als Parameter übergeben. Im Beispielprogramm werden die `clicked`-Signale der Schaltflächen `ui.buttonOK` und `ui.buttonAbbrechen` mit den Slots `onOK` und `onAbbrechen` verbunden. Diese werden ab jetzt immer dann aufgerufen, wenn der Benutzer die assoziierte Schaltfläche betätigt.¹⁴



Hinweis

In diesem Fall sind die verbundenen Signale parameterlos. Es gibt komplexere Signale, die einen oder mehrere Parameter übergeben. In einem solchen Fall muss auch der verbundene Slot eine entsprechende Anzahl von Parametern erwarten.

In der Methode `onOK` sollen die Eingaben des Benutzers aus den verschiedenen Widgets des Hauptdialogs ausgelesen werden. Jedes dieser Widgets wird durch eine Instanz einer entsprechenden Qt-Klasse repräsentiert. Die Namen dieser Instanzen haben wir zuvor im Qt-Designer festgelegt. Welches Widget dabei welchen Namen bekam, können Sie in der Tabelle in Abschnitt 23.3.3 nachlesen.

Über die angesprochenen Attribute können wir den Inhalt der Steuerelemente auslesen. Wie dies geschieht, ist von Widget zu Widget verschieden. So kann beispielsweise auf den Inhalt eines Line-Edit-Widgets über die Methode `text` zugegriffen werden. Erwähnenswert ist noch, dass die Methode `date` der Date-Edit-Instanz `geburtsdatum` das gespeicherte Datum nicht direkt in Form eines Strings, sondern in Form einer `QDate`-Instanz zurückgibt. Diese muss durch Aufruf der Methode `toString` in einen

¹⁴ Dieser komfortable Weg, Signale und Slots zu verbinden, ist eine Neuerung von PyQt und im C++-Framework Qt so nicht möglich. Sollten Sie mit Qt bereits vertraut sein, sei Ihnen gesagt, dass die klassische Variante des Verbindens von Signalen und Slots auch mit PyQt funktioniert, allerdings den Nachteil mit sich bringt, dass die C++-Schnittstelle der Signale angegeben werden muss.

String konvertiert werden. Zum Schluss, nachdem alle Daten ausgelesen und ausgegeben wurden, wird der Dialog durch Aufruf der Methode `close` geschlossen.

Im zweiten Slot, `onAbbrechen`, sind, abgesehen vom Schließen des Dialogs, keine weiteren Operationen vonnöten.

```
app = QtGui.QApplication(sys.argv)
dialog = MeinDialog()
dialog.show()
sys.exit(app.exec_())
```

Bei dem Code, der die Applikations- und Dialogklasse instanziiert und die Main-Event-Loop startet, handelt es sich um denselben, der schon im letzten Beispielprogramm seinen Dienst getan hat.

Hinweis
Wie das Beispiel demonstriert, öffnet auch ein Python-Programm mit grafischer Benutzeroberfläche unter Windows immer noch ein Konsolenfenster, in das mittels `print` geschrieben werden kann.
Das mag in einigen Fällen wünschenswert sein, ist jedoch häufig störend, wenn die Kommunikation mit dem Benutzer vollständig über die grafische Oberfläche ablaufen soll. Wenn Sie nicht wünschen, dass ein Konsolenfenster geöffnet wird, können Sie die Dateiendung der Python-Programmdatei von `.py` nach `.pyw` ändern. Dann werden alle Ausgaben in die Konsole unterdrückt, und es wird kein Konsolenfenster geöffnet.

[+]

23.5 Wichtige Widgets

Nachdem wir anhand eines kleinen Beispielprogramms in die Arbeit mit Qt eingestiegen sind, werden in diesem Abschnitt die wichtigsten Widgetklassen vorgestellt, die Qt zur Verfügung stellt. Die Verwendung der Klassen kann hier nicht vollständig erläutert werden. Wenn Sie eine umfassende Beschreibung dieser Klassen benötigen, ist die Qt- bzw. PyQt-Dokumentation Ihr Freund und Helfer.

Widget	Beschreibung	Seite
<i>QCheckBox</i>	ein Widget zur Aktivierung bzw. Deaktivierung einer Option	840
<i>QComboBox</i>	eine ausklappbare Liste	841

Tabelle 23.29 In diesem Abschnitt behandelte Widgets

Widget	Beschreibung	Seite
<i>QDateEdit</i>	ein Feld zur Datumsangabe	842
<i>QDateTimeEdit</i>	ein Feld zur Datums- und Zeitangabe	842
<i>QDial</i>	ein Drehregler	847
<i>QDialog</i>	die Basisklasse eines Dialogs	843
<i>QLineEdit</i>	ein einzeiliges Eingabefeld	844
<i>QListView</i>	eine Liste im Kontext der Model-View-Programmierung	844
<i>QListWidget</i>	eine klassische Liste	844
<i>QProgressBar</i>	ein Fortschrittsbalken	845
<i>QPushButton</i>	eine Schaltfläche	846
<i>QRadioButton</i>	ein Widget zur Auswahl einer Option aus mehreren Möglichkeiten	846
<i>QSlider</i>	ein Schieberegler	847
<i>QTextEdit</i>	ein mehrzeiliges Eingabefeld	848
<i>QTimeEdit</i>	ein Feld zur Zeitangabe	842
<i>QWidget</i>	eine Basisklasse aller Widgets	848

Tabelle 23.29 In diesem Abschnitt behandelte Widgets (Forts.)

Im Folgenden wird für jedes Steuerelement eine kurze Erklärung sowie eine Tabelle gegeben, die die wichtigsten Methoden und Signale auflistet.

23.5.1 QCheckBox

Die Klasse *QCheckBox* repräsentiert eine *Checkbox* in der grafischen Benutzeroberfläche. Eine *Checkbox* ist ein Steuerelement, das vom Benutzer entweder aktiviert oder deaktiviert werden kann und dabei in seiner Bedeutung unabhängig von anderen *Checkboxen* ist.

Methode	Beschreibung
<i>checkState()</i>	Gibt den Status der <i>Checkbox</i> zurück. Gültige Status sind <code>QtCore.Qt.Checked</code> , <code>QtCore.Qt.Unchecked</code> oder <code>QtCore.Qt.PartiallyChecked</code> .

Tabelle 23.30 Wichtige Methoden von *QCheckBox*

Methode	Beschreibung
<i>setCheckState(state)</i>	Setzt den Status der Checkbox. Hier können die Status gesetzt werden, die von <i>checkState</i> zurückgegeben werden.
<i>isTristate()</i>	Gibt <code>True</code> zurück, wenn es sich um eine dreiwertige Checkbox handelt, andernfalls <code>False</code> .
<i>setTristate([on])</i>	Wenn für <i>on</i> der Wert <code>True</code> übergeben wird, erhält die Checkbox einen zusätzlichen dritten Zustand.

Tabelle 23.30 Wichtige Methoden von QCheckBox (Forts.)

Signal	Beschreibung
<i>stateChanged(int)</i>	Wird gesendet, wenn der Benutzer den Status der Checkbox ändert. Der neue Status wird als Parameter übergeben.

Tabelle 23.31 Wichtige Signale von QCheckBox

23.5.2 QComboBox

Die Klasse `QComboBox` repräsentiert eine Combobox, auch bekannt als Dropdown-Menü, in der grafischen Benutzeroberfläche.

Methode	Beschreibung
<i>addItem([icon,]text)</i>	Fügt ein Element mit dem Namen <i>text</i> zur Combobox hinzu. Optional kann eine <code>QIcon</code> -Instanz übergeben werden, um ein Icon neben dem Elementtext anzeigen zu lassen.
<i>addItems(texts)</i>	Fügt mehrere Elemente zur Combobox hinzu. Für <i>texts</i> muss eine Liste von Strings übergeben werden.
<i>clear()</i>	Löscht alle Elemente der Combobox.
<i>currentIndex()</i>	Gibt den Index des ausgewählten Elements zurück.
<i>currentText()</i>	Gibt den Text des ausgewählten Elements zurück.
<i>setModel(model)</i>	Setzt ein <code>QAbstractItemModel</code> . Näheres zur Model-View-Architektur von Qt erfahren Sie in Abschnitt 23.7 (S. 863)
<i>setView(itemView)</i>	Setzt einen <code>QAbstractItemView</code> .

Tabelle 23.32 Wichtige Methoden von QComboBox

Signal	Beschreibung
<i>currentIndexChanged(int)</i>	Wird gesendet, wenn sich das ausgewählte Element ändert. Der Index dieses Elements wird als Parameter übergeben.

Tabelle 23.33 Wichtige Signale von QComboBox

23.5.3 QDateEdit, QTimeEdit, QDateTimeEdit

Die Klassen QDateEdit, QTimeEdit und QDateTimeEdit repräsentieren Widgets, die eine Datumsangabe bzw. Zeitangabe vom Benutzer einlesen. Dem Benutzer wird zur Eingabe ein Kalender angezeigt.

Die folgenden Methoden beziehen sich auf den Umgang mit Datumsangaben bei QDateEdit- und QDateTimeEdit-Widgets. Die Methoden *setMinimumTime*, *setMaximumTime*, *minimumTime*, *maximumTime*, *setTime* und *time* funktionieren analog für Zeitangaben bei QDateTimeEdit- und TimeEdit-Widgets.

Methode	Beschreibung
<i>setCalendarPopup(enable)</i>	Aktiviert bzw. deaktiviert die Benutzereingabe über einen Kalender.
<i>setMinimumDate(date)</i>	Setzt das kleinstmögliche Datum. Für <i>date</i> kann eine QDate- oder eine QDateTime.date-Instanz übergeben werden.
<i>setMaximumDate(date)</i>	Setzt das größtmögliche Datum.
<i>minimumDate()</i>	Gibt das kleinstmögliche Datum als QDate-Instanz zurück.
<i>maximumDate()</i>	Gibt das größtmögliche Datum als QDate-Instanz zurück.
<i>setDate(date)</i>	Setzt das im Widget angezeigte Datum.
<i>date()</i>	Gibt das gewählte Datum als QDate-Instanz zurück.

Tabelle 23.34 Wichtige Methoden von QDateEdit

Signal	Beschreibung
<i>dateChanged(QDate)</i>	Wird gesendet, wenn ein neues Datum ausgewählt wurde.
<i>dateTimeChanged(QDateTime)</i>	Wird gesendet, wenn ein neues Datum oder ein neuer Zeitpunkt ausgewählt wurde.
<i>timeChanged(QTime)</i>	Wird gesendet, wenn ein neuer Zeitpunkt ausgewählt wurde.

Tabelle 23.35 Wichtige Signale von QDateEdit, QTimeEdit und QDateTimeEdit

23.5.4 QDialog

Die Klasse `QDialog` repräsentiert einen Dialog in der grafischen Benutzeroberfläche. Ein Dialog kann wie jede andere Widgetklasse verwendet werden, mit dem Unterschied, dass sich Dialoge nicht in andere Widgets einbetten lassen. Grundsätzlich unterscheidet man zwei Arten von Dialogen – modale und nicht modale Dialoge:

- Unter einem *modalen Dialog* versteht man einen Dialog, der sich im Vordergrund der Anwendung platziert und sich den Eingabefokus nimmt. Der Benutzer kann keine anderen Dialoge der Anwendung bedienen, während ein modaler Dialog geöffnet ist. Ein modaler Dialog bietet sich also für eine wichtige Teilkommunikation mit dem Benutzer an, die für den weiteren Programmablauf essenziell ist.
- Dem gegenüber steht der *nicht modale Dialog*. Wird ein Dialog nicht modal geöffnet, kann er parallel zum restlichen Teil der Anwendung bedient werden. Ein bekanntes Beispiel für einen nicht modalen Dialog ist der »Suchen und Ersetzen«-Dialog in einem Textverarbeitungsprogramm, bei dem es dem Benutzer möglich sein muss, während des geöffneten Dialogs Änderungen im Hauptfenster vorzunehmen.

Um einen Dialog modal anzuzeigen, wird die Methode `exec` aufgerufen. Diese blockiert den Programmfluss so lange, bis der Benutzer den Dialog beendet hat. Der Rückgabewert von `exec` gibt an, in welcher Form der Dialog beendet wurde. Es wird eine der beiden Konstanten `QtCore.Qt.Accepted` und `QtCore.Qt.Rejected` zurückgegeben, wobei die erste ein Beenden des Dialogs über den OK- und die zweite ein Beenden über den ABBRECHEN-Button repräsentiert. Innerhalb der Dialogklasse können die Methoden `accept` und `reject` aufgerufen werden, um den Dialog mit dem entsprechenden Rückgabewert zu beenden.

Ein nicht modaler Dialog wird mithilfe der Methode `show` angezeigt. Diese Methode kehrt sofort zurück, ohne auf das Beenden des Dialogs zu warten, und ermöglicht somit das parallele Verarbeiten von Dialog und Hauptanwendung.

23.5.5 QLineEdit

Die Klasse `QLineEdit` repräsentiert ein einzeliliges Eingabefeld in der grafischen Benutzeroberfläche.

Methode	Beschreibung
<i>setReadOnly(ro)</i>	Macht das Eingabefeld schreibgeschützt bzw. hebt den Schreibschutz auf.
<i>setEchoMode(echoMode)</i>	Steuert, in welcher Form die Benutzereingabe angezeigt wird. Mögliche Modi sind <code>QLineEdit.NoEcho</code> , in dem keine Anzeige erfolgt, <code>QLineEdit.Password</code> , in der jedes Zeichen durch einen dicken Punkt ersetzt wird, und <code>QLineEdit.Normal</code> für eine unmodifizierte Anzeige.
<i>setText(text)</i>	Setzt den Text des Eingabefelds.
<i>text()</i>	Gibt den Text des Eingabefelds zurück.

Tabelle 23.36 Wichtige Methoden von QLineEdit

Signal	Beschreibung
<i>textChanged(QString)</i>	Wird gesendet, wenn der Text des Eingabefelds verändert wurde.

Tabelle 23.37 Wichtige Signale von QLineEdit

23.5.6 QListWidget, QListView

Die Klassen `QListWidget` und `QListView` repräsentieren Widgets zur Anzeige von Listen. Während es sich bei `QListWidget` um ein klassisches Widget handelt, verwendet `QListView` die Model-View-Architektur. Eine Einführung in das Konzept der Model-View-Programmierung anhand des `QListView`-Widgets finden Sie in Abschnitt 23.7. Ein `QListWidget` wird immer dann eingesetzt, wenn das Erstellen einer Model-View-Architektur für die anzuzeigenden Daten unverhältnismäßig aufwendig wäre und die Flexibilität, die eine Model-View-Architektur bietet, nicht benötigt wird.

Methode	Beschreibung
<i>addItem(label)</i>	Fügt einen Eintrag mit dem Namen <i>label</i> ans Ende der Liste an.
<i>addItems(labels)</i>	Fügt mehrere Einträge mit den Namen <i>labels</i> ans Ende der Liste an. Für <i>labels</i> kann eine Liste von Strings übergeben werden.
<i>currentItem()</i>	Gibt den momentan ausgewählten Eintrag als <i>QListWidgetItem</i> -Instanz zurück. Der Name des Eintrags kann über die Methode <i>text</i> des Items herausgefunden werden.

Tabelle 23.38 Wichtige Methoden von QListWidget

Signal	Beschreibung
<i>currentItemChanged</i> (<i>QListWidgetItem</i> , <i>QListWidgetItem</i>)	Wird gesendet, wenn ein neuer Eintrag ausgewählt wurde. Als Parameter werden der neu ausgewählte Eintrag und der zuvor ausgewählte Eintrag übergeben.

Tabelle 23.39 Wichtige Signale von QListWidget

23.5.7 QProgressBar

Die Klasse *QProgressBar* repräsentiert einen Fortschrittsbalken in einer grafischen Benutzeroberfläche. Ein Fortschrittsbalken zeigt den Fortschritt einer langwierigen Operation an. Das *QProgressBar*-Widget erlaubt dabei keine Interaktion mit dem Benutzer.

Methode	Beschreibung
<i>setMinimum(minimum)</i>	Setzt die ganzzahlige untere Grenze des Wertebereichs.
<i>setMaximum(maximum)</i>	Setzt die ganzzahlige obere Grenze des Wertebereichs.
<i>setValue(value)</i>	Setzt den aktuell darzustellenden Wert. Der Wert <i>value</i> muss ganzzahlig sein und in den angegebenen Grenzen liegen.
<i>setTextVisible(visible)</i>	Legt fest, ob zusätzlich zum Balken eine Prozentangabe angezeigt werden soll. Für <i>visible</i> muss ein boolescher Wert übergeben werden.

Tabelle 23.40 Wichtige Methoden von QProgressBar

Methode	Beschreibung
<i>setOrientation(orientation)</i>	Setzt die Orientierung des Fortschrittsbalkens. Möglich sind die Werte <code>QtCore.Qt.Horizontal</code> und <code>QtCore.Qt.Vertical</code> für eine horizontale bzw. vertikale Ausrichtung.

Tabelle 23.40 Wichtige Methoden von `QProgressBar` (Forts.)

23.5.8 `QPushButton`

Die Klasse `QPushButton` repräsentiert eine Schaltfläche in der grafischen Benutzeroberfläche. Eine Schaltfläche ist ein beschriftetes Widget, auf das der Benutzer klicken kann, um eine Aktion auszulösen.

Methode	Beschreibung
<i>setText(text)</i>	Setzt die Beschriftung der Schaltfläche.
<i>setDefault(default)</i>	Macht den Button zu einem <i>Default Button</i> . Eine solche Schaltfläche wird immer dann aktiviert, wenn der Benutzer auf die Return- oder Leertaste drückt.

Tabelle 23.41 Wichtige Methoden von `QPushButton`

Signal	Beschreibung
<i>clicked()</i>	Wird gesendet, wenn der Benutzer auf die Schaltfläche geklickt hat.

Tabelle 23.42 Wichtige Signale von `QPushButton`

23.5.9 `QRadioButton`

Die Klasse `QRadioButton` repräsentiert einen Radiobutton in der grafischen Benutzeroberfläche. Ein Radiobutton wird verwendet, um den Benutzer eine Auswahl aus mehreren vorgegebenen Möglichkeiten treffen zu lassen. Alle Radiobuttons eines Dialogs, die über dasselbe Elternwidget verfügen, gehören zu einer Gruppe. Innerhalb einer Gruppe kann immer nur ein Radiobutton aktiviert sein. Wenn der Benutzer einen anderen aktiviert, wird der vorher aktivierte deaktiviert. Um mehrere Gruppen von Radiobuttons in demselben Dialog unterzubringen, müssen diese über verschiedene Elternwidgets verfügen. Dazu bietet sich das *Frame-*

Widget, repräsentiert durch die Klasse `QFrame`, an, das einem unsichtbaren Rahmen im *Widget* entspricht.

Methode	Beschreibung
<i>isChecked()</i>	Gibt den Status des Radiobuttons zurück.

Tabelle 23.43 Wichtige Methoden von `QRadioButton`

Signal	Beschreibung
<i>toggled()</i>	Wird gesendet, wenn der Radiobutton aktiviert oder deaktiviert wurde.

Tabelle 23.44 Wichtige Signale von `QRadioButton`

23.5.10 `QSlider`, `QDial`

Die Klasse `QSlider` repräsentiert einen Slider in der grafischen Benutzeroberfläche. Bei einem Slider handelt es sich um einen Schieberegler, wie Sie ihn beispielsweise von einem Mischpult her kennen. Grundsätzlich wird ein Slider dazu verwendet, den Benutzer einen ganzzahligen Wert auswählen zu lassen, der innerhalb eines bestimmten Bereichs liegen muss. Die Klasse `QDial` funktioniert ähnlich wie `QSlider`, entspricht aber einem Drehregler statt einem Schieberegler.

Methode	Beschreibung
<i>setMinimum(minimum)</i>	Setzt die ganzzahlige untere Grenze des Wertebereichs.
<i>setMaximum(maximum)</i>	Setzt die ganzzahlige obere Grenze des Wertebereichs.
<i>setValue(value)</i>	Setzt den aktuell darzustellenden Wert. Der Wert <i>value</i> muss ganzzahlig sein und in den angegebenen Grenzen liegen.
<i>value()</i>	Gibt den momentan eingestellten Wert zurück.
<i>setOrientation(orientation)</i>	Setzt die Orientierung des Sliders. Möglich sind die Werte <code>QtCore.Qt.Horizontal</code> und <code>QtCore.Qt.Vertical</code> für eine horizontale bzw. vertikale Ausrichtung.

Tabelle 23.45 Wichtige Methoden von `QSlider` und `QDial`

Signal	Beschreibung
<i>valueChanged()</i>	Wird gesendet, wenn der eingestellte Wert des Sliders bzw. des Drehreglers verändert wurde.

Tabelle 23.46 Wichtige Signale von QSlider und QDial

23.5.11 QTextEdit

Die Klasse `QTextEdit` repräsentiert ein mehrzeiliges Eingabefeld in der grafischen Benutzeroberfläche. In ein solches Eingabefeld kann der Benutzer beliebigen Text schreiben.

Methode	Beschreibung
<i>setPlainText(text)</i>	Setzt den Text des Eingabefeldes.
<i>toPlainText()</i>	Gibt den Text des Eingabefeldes zurück.
<i>setReadOnly(ro)</i>	Versetzt das Eingabefeld in den schreibgeschützten Modus bzw. zurück.

Tabelle 23.47 Wichtige Methoden von QTextEdit

Signal	Beschreibung
<i>textChanged()</i>	Wird gesendet, wenn der Text des Eingabefeldes verändert wurde.

Tabelle 23.48 Wichtige Signale von QTextEdit

23.5.12 QWidget

Die Klasse `QWidget` ist die Basisklasse aller Steuerelemente und implementiert einen Großteil der steuerelementübergreifenden Funktionalität. Allgemein können Widgets in zwei Sorten unterteilt werden:

- Der häufigste Fall ist ein *untergeordnetes Widget*. Ein solches untergeordnetes Widget wird relativ zu seinem Elternwidget (engl. *parent*) positioniert und kann nicht über die Ränder des Parent-Widgets hinausragen.
- Ein Widget ohne Parent-Widget ist ein *Fenster* (engl. *window*). Theoretisch kann jede Instanz einer von `QWidget` abgeleiteten Klasse als Fenster dienen, beispielsweise also auch ein Button oder eine Checkbox. Praktisch ergibt dies jedoch wenig

Sinn, und es werden nur geeigneteren Klassen wie beispielsweise `QDialog` als Fenster verwendet.

Die folgende Tabelle listet die wichtigsten Eventhandler auf, die beim Auftreten verschiedener Events aufgerufen werden. Ein Eventhandler kann eingerichtet werden, indem eine selbsterstellte Klasse, die von `QWidget` direkt oder von einer Steuerelementklasse erbt, die genannte Methode implementiert.

Event	Beschreibung
<i>paintEvent(event)</i>	Wird aufgerufen, wenn das Widget neu gezeichnet werden muss. Die Methode bekommt eine Instanz der Klasse <code>QPaintEvent</code> übergeben, die beispielsweise den neu zu zeichnenden Bereich enthält.
<i>resizeEvent(event)</i>	Wird aufgerufen, wenn das Widget in seiner Größe verändert wurde. Der Methode wird eine Instanz der Klasse <code>QResizeEvent</code> übergeben, die unter anderem die alte und neue Größe des Widgets enthält.
<i>mousePressEvent(event)</i>	Wird aufgerufen, wenn der Benutzer mit der Maus auf das Widget klickt. Der Methode wird eine Instanz der Klasse <code>QMouseEvent</code> übergeben, die beispielsweise die Koordinaten des Mausklicks enthält.
<i>mouseDoubleClickEvent(event)</i>	Wird aufgerufen, wenn der Benutzer doppelt auf das Widget klickt. Beachten Sie, dass vor diesem Event immer ein <i>mousePressEvent</i> , ein <i>mouseReleaseEvent</i> und möglicherweise ein <i>mouseMoveEvent</i> auftreten. Es ist aus naheliegenden Gründen nicht möglich, bereits beim ersten Klick zwischen einem einfachen und einem doppelten Klick zu unterscheiden. Es wird eine <code>QMouseEvent</code> -Instanz übergeben.
<i>mouseMoveEvent(event)</i>	Wird aufgerufen, wenn der Benutzer die Maus bewegt, während er eine Maustaste gedrückt hält. Wenn zuvor die Methode <i>setMouseTracking</i> aufgerufen und <code>True</code> übergeben wurde, wird auch dann ein <i>mouseMoveEvent</i> erzeugt, wenn der Benutzer die Maus bewegt, ohne eine Taste gedrückt zu halten.

Tabelle 23.49 Events eines Widgets

Event	Beschreibung
<i>keyPressEvent(event)</i>	Wird aufgerufen, wenn das Widget den Eingabefokus hat und der Benutzer eine Taste drückt. Wenn die Taste ausreichend lange gedrückt wird, wird die Methode <i>keyPressEvent</i> mehrfach aufgerufen. Es wird eine Instanz der Klasse <i>QKeyEvent</i> übergeben, über die beispielsweise die gedrückte Taste ausgelesen werden kann.

Tabelle 23.49 Events eines Widgets (Forts.)

23.6 Zeichenfunktionalität

In den vorangegangenen Abschnitten wurden das Qt-Framework anhand eines kleinen Beispielprogramms eingeführt und daraufhin die wichtigsten Widgets besprochen. In diesem Abschnitt wird das Zeichnen in Qt behandelt. Thema des letzten Abschnitts zu Qt wird die Model-View-Architektur sein.

Wenn Sie ein eigenes Widget erstellen, also eine Klasse definieren, die von einem Steuerelement oder direkt von *QWidget* erbt, haben Sie die Möglichkeit, selbst beliebige Inhalte in das Widget zu zeichnen. Das ist besonders dann interessant, wenn eine Anwendung Inhalte in einem Widget anzeigen möchte, für die es im Qt-Framework keine vorgefertigte Klasse gibt. Das kann zum Beispiel ein Diagramm oder eine spezifische Grafik sein.

Im Folgenden beschäftigen wir uns zunächst mit den Grundlagen des Zeichnens in Qt und bringen danach einige einfache Formen, beispielsweise einen Kreis oder ein Rechteck, auf den Bildschirm.

Die in den folgenden Unterkapiteln präsentierten Beispielklassen verstehen sich im folgenden Kontext:

```
from PyQt4 import QtGui, QtCore
import sys

class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

app = QtGui.QApplication(sys.argv)
widget = MeinWidget()
```

```

widget.resize(150, 150)
widget.show()
sys.exit(app.exec_())

```

Wir werden in den Abschnitten jeweils nur die Klasse `MeinWidget` neu implementieren. Um aus diesen Beispielen ein lauffähiges PyQt-Programm zu erstellen, muss die vorgestellte Klasse in den obigen Kontext eingefügt werden.

23.6.1 Werkzeuge

Um innerhalb eines Widgets zeichnen zu können, muss der Eventhandler `paintEvent` implementiert werden. Dabei handelt es sich um eine Methode, die vom Qt-Frame-work immer dann aufgerufen wird, wenn das Widget teilweise oder vollständig neu gezeichnet werden muss. Das passiert beispielsweise dann, wenn das Anwendungsfenster teilweise verdeckt oder minimiert war und vom Benutzer in den Vordergrund geholt wurde. Die Methode `paintEvent` bekommt eine Instanz der Klasse `QPaintEvent` übergeben, die unter anderem den Bereich des Widgets enthält, der neu gezeichnet werden soll.¹⁵

Innerhalb der `paintEvent`-Methode muss ein *Painter* (dt. »Maler«) erzeugt werden, mit dessen Hilfe die Zeichenoperationen durchgeführt werden können. Bei einem Painter handelt es sich um eine Instanz der Klasse `QtGui.QPainter`. Ein grundlegendes Widget, das das Paint-Event implementiert und einen Painter erzeugt, sieht folgendermaßen aus:

```

class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)

    def paintEvent(self, event):
        painter = QtGui.QPainter(self)

```

23

Zum Zeichnen gibt es in Qt neben dem Painter zwei grundsätzliche Werkzeuge: den Pen und den Brush.

Als *Pen* (dt. »Stift«) wird eine Instanz der Klasse `QtGui.QPen` bezeichnet. Um einen Pen zu verwenden, muss dieser dem Painter mithilfe der Methode `setPen` bekannt gegeben werden. Grundsätzlich wird ein Pen zum Zeichnen von Linien, beispielsweise für Umrundungen bestimmter Figuren, verwendet. Dazu benötigt ein Pen im Wesentli-

¹⁵ In unseren einfachen Beispielen werden wir beim Aufrufen der `paintEvent`-Methode stets die komplette Grafik neu zeichnen. Je komplexer und aufwendiger eine Grafik jedoch ist, desto eher sollte man nur den durch die `QPaintEvent`-Instanz spezifizierten Bereich tatsächlich neu zeichnen.

chen drei Informationen: die Linienfarbe, die Liniendicke und den Linienstil. Ein Pen wird folgendermaßen erzeugt:

```
pen = QtGui.QPen(QtGui.QColor(255,0,0))
```

Dem Konstruktor des Pens wird eine Instanz der Klasse QColor übergeben, um die Farbe des Pens, in diesem Fall Rot,¹⁶ zu spezifizieren. Nachdem ein Pen erzeugt worden ist, kann seine Liniendicke bzw. der Linienstil durch die Methoden `setWidth` und `setStyle` festgelegt werden:

```
pen.setWidth(7)
pen.setStyle(QtCore.Qt.DashLine)
```

Die der Methode `setWidth` übergebene ganze Zahl entspricht der Liniendicke in Pixeln, die eine mit diesem Pen gezeichnete Linie später auf dem Bildschirm haben wird. Der Methode `setStyle` können verschiedene Konstanten übergeben werden, die jeweils einen bestimmten Linienstil vorschreiben. Eine Auswahl dieser Konstanten finden Sie in der folgenden Tabelle:

Konstante	Beschreibung
QtCore.Qt.SolidLine	Eine durchgezogene Linie. Dies ist die Standardeinstellung und braucht nicht explizit gesetzt zu werden.
QtCore.Qt.DashLine	eine gestrichelte Linie
QtCore.Qt.DotLine	eine gepunktete Linie
QtCore.Qt.DashDotLine	eine Linie, die abwechselnd gestrichelt und gepunktet ist

Tabelle 23.50 Linienstile eines Pens

Das zweite wichtige Werkzeug zum Zeichnen ist der *Brush* (dt. »Pinsel«), mit dessen Hilfe Flächen gefüllt werden. Ein Brush spezifiziert, ähnlich wie ein Pen, zunächst einmal die Füllfarbe. Analog zum Pen wird ein Brush folgendermaßen erzeugt:

```
brush = QtGui.QBrush(QtGui.QColor(0,0,255))
```

Auch dem Konstruktor des Brushes wird der Farbwert, in diesem Fall Blau, in Form einer QColor-Instanz übergeben. Nachdem der Brush erzeugt worden ist, kann auch hier mit der Methode `setStyle` ein Stil festgelegt werden. Hier können beispielsweise

¹⁶ Eine RGB-Farbangabe besteht aus drei einzelnen Werten zwischen 0 und 255. Der erste übergebene Wert spezifiziert den Rot-, der zweite den Grün- und der dritte den Blauanteil der Farbe.

Schraffierungen, Farbverläufe oder Texturen gesetzt werden, worauf wir an dieser Stelle aber nicht näher eingehen werden.

Allgemein gilt, dass Pens und Brushes ausgewählt werden müssen, bevor sie benutzt werden können. Dazu werden die Methoden `setPen` bzw. `setBrush` des `Painter`s mit der jeweiligen `QPen`- bzw. `QBrush`-Instanz als Parameter aufgerufen. Eine darauffolgende Zeichenoperation wird dann mit den ausgewählten Werkzeugen durchgeführt. Es können immer nur ein Brush und ein Pen gleichzeitig selektiert sein.

23.6.2 Koordinatensystem

Bevor es ans Zeichnen geht, müssen wir uns Gedanken über das in Qt verwendete Koordinatensystem machen. Dieses lehnt sich an andere GUI-Toolkits an und ist in Abbildung 23.44 dargestellt.

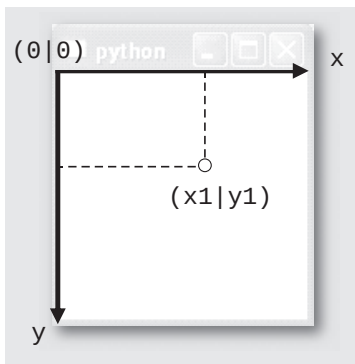


Abbildung 23.44 Das Qt-Koordinatensystem

Der Ursprung des Koordinatensystems liegt in der oberen linken Ecke des Widgets, und die Y-Achse zeigt, im Gegensatz zu dem in der Mathematik verwendeten kartesischen Koordinatensystem, nach unten. Die Einheit des Koordinatensystems ist Pixel.

Jedes Widget verfügt über ein eigenes lokales Koordinatensystem, dessen Ursprung stets relativ zur Position des Widgets in dessen oberer linken Ecke liegt. Das hat den Vorteil, dass eine Zeichnung nicht angepasst werden muss, wenn das Widget in seiner Position auf dem Bildschirm oder innerhalb eines anderen Widgets verändert wird.

23.6.3 Einfache Formen

Das Qt-Framework bietet eine Reihe von abstrakten Zeichenoperationen, die das Zeichnen einfacher geometrischer Formen, wie beispielsweise eines Rechtecks oder einer Ellipse, ermöglichen. Grundsätzlich sind Zeichenoperationen als Methoden der Klasse `QPainter`, also eines `Painter`s, implementiert.

Wir beginnen damit, ein Rechteck zu zeichnen. Dazu wird die Methode `drawRect` eines `Painters` verwendet. Bevor ein Rechteck gezeichnet werden kann, sollten ein `Pen` für den Rand des Rechtecks und ein `Brush` für die Füllung erzeugt und ausgewählt werden:

```
class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.pen = QtGui.QPen(QtGui.QColor(0,0,0))
        self.pen.setWidth(3)
        self.brush = QtGui.QBrush(QtGui.QColor(255,255,255))

    def paintEvent(self, event):
        painter = QtGui.QPainter(self)
        painter.setPen(self.pen)
        painter.setBrush(self.brush)
        painter.drawRect(10, 10, 130, 130)
```

Im Konstruktor der Widget-Klasse `MeinWidget` werden `Pen` und `Brush` angelegt, die zum Zeichnen des Rechtecks verwendet werden sollen. In diesem Fall handelt es sich um einen schwarzen Stift mit einer Dicke von drei Pixeln sowie um einen weißen Pinsel. In der Methode `paintEvent` werden nach dem Erzeugen des `Painters` die beiden Werkzeuge ausgewählt. Danach wird mittels `drawRect` ein Rechteck auf dem Bildschirm gemalt. Die übergebenen Parameter kennzeichnen der Reihe nach die X-Koordinate der oberen linken Ecke, die Y-Koordinate der oberen linken Ecke, die Breite des Rechtecks sowie die Höhe des Rechtecks. Alle Werte werden in Pixel angegeben.

Auf dem Bildschirm erscheint das Rechteck, genauer betrachtet, ein Quadrat (siehe Abbildung 23.45).



Abbildung 23.45 Ein mit `drawRect` gezeichnetes Quadrat

Auf ähnliche Weise können noch weitere Figuren gezeichnet werden, deren Form durch Angabe eines umschließenden Rechtecks beschrieben ist. So braucht beispielsweise nur der Methodenname `drawRect` ausgetauscht zu werden, um ein Rechteck mit runden Ecken (`drawRoundRect`) oder eine Ellipse (`drawEllipse`) zu zeichnen.

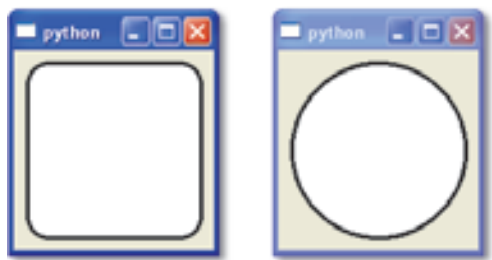


Abbildung 23.46 Ein Rechteck mit runden Ecken und eine Ellipse

Um eine dieser Figuren in ihrer Größe an das Widget anzupassen, kann die parameterlose Methode `rect` einer Widgetklasse verwendet werden, die die Dimensionen des Widgets als `QRect`-Instanz¹⁷ zurückgibt. Auf diese Weise ist es beispielsweise möglich, das gesamte Widget mit einer Form zu füllen.

Neben diesen drei grundlegenden Formen existiert eine Reihe weiterer Methoden zum Zeichnen spezieller Formen. Die folgende Tabelle gibt eine Übersicht über die wichtigsten dieser Methoden, die in diesem Kapitel nicht weiter besprochen werden.

Methode	Beschreibung
<code>drawArc(x, y, w, h, a, alen)</code>	Zeichnet einen geöffneten Bogen mit dem Startwinkel <i>a</i> und dem Spannwinkel <i>alen</i> . Die anderen Parameter spezifizieren das umgebende Rechteck. »Geöffnet« bedeutet in diesem Fall, dass die beiden Enden des Bogens nicht durch eine Linie miteinander verbunden sind.
<code>drawChord(x, y, w, h, a, alen)</code>	Zeichnet einen geschlossenen Bogen. »Geschlossen« bedeutet, dass die beiden Enden des Bogens durch eine Linie miteinander verbunden sind.
<code>drawConvexPolygon(point, ...)</code>	Zeichnet ein konvexes Polygon. ¹⁸ Es können entweder eine <code>QPolygon</code> -Instanz oder beliebig viele <code>QPoint</code> -Instanzen übergeben werden.

Tabelle 23.51 Methoden eines Painters

¹⁷ Die Klasse `QRect` beschreibt ein Rechteck und verfügt unter anderem über die parameterlosen Methoden `x`, `y`, `width` und `height`, mit denen auf die Koordinaten der oberen linken Ecke und die Dimensionen des Rechtecks zugegriffen werden kann.

¹⁸ Ein Polygon ist eine Fläche, die durch einen Linienzug begrenzt ist. Eine Fläche heißt konvex, wenn die Verbindungslinie zwischen je zwei Punkten in der Fläche vollständig innerhalb der Fläche verläuft. Ein nicht-konvexes Polygon ist wesentlich aufwendiger darzustellen als ein konvexes.

Methode	Beschreibung
<i>drawLine(x1, y1, x2, y2)</i>	Zeichnet eine Linie von (x1, y1) nach (x2, y2).
<i>drawLines(line)</i>	Zeichnet einen Linienzug. Dieser kann in Form einer Liste von <i>QPoint</i> -Instanzen übergeben werden.
<i>drawPie(x, y, w, h, a, alen)</i>	Zeichnet einen Ausschnitt einer Ellipse, der umgangssprachlich als »Tortenstück« bezeichnet wird.
<i>drawPolygon(point, ...)</i>	Zeichnet ein beliebiges Polygon. Diese Methode ist allgemeiner als <i>drawConvexPolygon</i> .

Tabelle 23.51 Methoden eines Painters (Forts.)

23.6.4 Grafiken

Neben dem Zeichnen der grundlegenden geometrischen Formen ermöglicht es das Qt-Framework über die Klasse *QImage*, Grafiken der verschiedensten Formate von der Festplatte zu laden und mithilfe eines Painters anzuzeigen.¹⁹ Das folgende Beispielprogramm lädt die Grafik *buch.png* und zeigt sie mithilfe der Methode *drawImage* des Painters an:

```
class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.grafik = QtGui.QImage("buch.png")
        self.ziel = QtCore.QRect(10, 10, 130, 130)
        self.quelle = QtCore.QRect(0, 0,
                                     self.grafik.width(),
                                     self.grafik.height())

    def paintEvent(self, event):
        painter = QtGui.QPainter(self)
        painter.drawImage(self.ziel, self.grafik, self.quelle)
```

Im Konstruktor der Widgetklasse *MeinWidget* wird zunächst unter Angabe des Pfades der zu ladenden Grafik eine Instanz der Klasse *QImage* erzeugt. Danach werden zwei Rechtecke namens *self.quelle* und *self.ziel* erzeugt. Das Rechteck *self.ziel* spezifiziert das Rechteck im Widget, in das die Grafik gezeichnet werden soll. Das Recht-

¹⁹ Qt »versteht« unter anderem die Dateiformate BMP, GIF, JPG und PNG.

eck `self.quelle` entspricht dem Ausschnitt der Grafik, der dabei gezeichnet werden soll. In diesem Fall umschließt das Quellrechteck das gesamte Bild.

Das mit diesem Code erstellte Widget sieht so aus wie in Abbildung 23.47.



Abbildung 23.47 Eine Grafik in einem Widget

Hinweis

In diesem Fall wurde ein nicht quadratisches Bild auf eine quadratische Fläche gezeichnet und somit beim Zeichnen leicht gestreckt. Um dies zu vermeiden, muss das Seitenverhältnis des Zielrechtecks an das des Quellrechtecks angepasst werden.



23.6.5 Text

Nachdem wir sowohl geometrische Formen als auch Grafiken in ein Widget zeichnen können, fehlt noch das Zeichnen von Text, beispielsweise für die Beschriftung von Diagrammen.

Zum Zeichnen von Text in einem Widget wird die Methode `drawText` eines `Painter`s verwendet. Im folgenden Beispiel wird der Text »Hallo Welt« im Widget zentriert ausgegeben:

```
class MeinWidget(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.font = QtGui.QFont("Helvetica", 16)
        self.pen = QtGui.QPen(QtGui.QColor(0,0,255))

    def paintEvent(self, event):
        painter = QtGui.QPainter(self)
        painter.setPen(self.pen)
```

```
painter.setFont(self.font)
painter.drawText(self.rect(), QtCore.Qt.AlignCenter,
                 "Hallo Welt")
```

Im Konstruktor der Klasse `MeinWidget` wird zunächst eine Instanz der Klasse `QFont` erzeugt. Diese Klasse repräsentiert einen Schrifttyp in einer bestimmten Größe. Zudem wird ein `Pen` erzeugt, der die Schriftfarbe vorgibt, in der der Text geschrieben werden soll.

In der Methode `paintEvent` wird zunächst, wie gehabt, ein `Painter` erzeugt. Danach werden mittels `setFont` und `setPen` Schriftart und Stift ausgewählt. Durch einen Aufruf der Methode `drawText` wird der Text gezeichnet. Die Methode bekommt ein Rechteck als ersten und eine Positionsangabe innerhalb dieses Rechtecks als zweiten Parameter übergeben. Als dritter Parameter wird der zu schreibende Text übergeben. Zur Positionierung des Textes innerhalb des angegebenen Rechtecks können mehrere Konstanten mithilfe des binären ODER verknüpft werden. Die wichtigsten dieser Konstanten sind in folgender Tabelle aufgelistet und kurz erläutert:

Konstante	Beschreibung
<code>QtCore.Qt.AlignLeft</code>	Richtet den Text am linken Rand des Rechtecks aus.
<code>QtCore.Qt.AlignRight</code>	Richtet den Text am rechten Rand des Rechtecks aus.
<code>QtCore.Qt.AlignHCenter</code>	Richtet den Text horizontal zentriert im Rechteck aus.
<code>QtCore.Qt.AlignTop</code>	Richtet den Text am oberen Rand des Rechtecks aus.
<code>QtCore.Qt.AlignBottom</code>	Richtet den Text am unteren Rand des Rechtecks aus.
<code>QtCore.Qt.AlignVCenter</code>	Richtet den Text vertikal zentriert im Rechteck aus.
<code>QtCore.Qt.AlignCenter</code>	Richtet den Text horizontal und vertikal zentriert im Rechteck aus.

Tabelle 23.52 Konstanten zur Positionierung des Textes

Das mit diesem Code erstellte Widget ist in Abbildung 23.48 dargestellt.



Abbildung 23.48 Mit `drawText` kann Text gezeichnet werden

Es gibt noch eine zweite, vereinfachte Variante, `drawText` zu verwenden. Dabei werden ebenfalls drei Parameter übergeben: die X- und Y-Koordinate, an die der Text geschrieben werden soll, und ein String, der den Text enthält. Wenn im obigen Beispielprogramm der Aufruf von `drawText` durch folgende Codezeile ersetzt wird

```
painter.drawText(0, 16, "Hallo Welt")
```

dann erscheint der Text in der oberen linken Ecke des Fensters. Beachten Sie, dass sich die Koordinaten, die bei der zweiten Variante von `drawText` übergeben werden, auf die untere linke Ecke des Textes beziehen, sodass der Text nicht an die Position 0/0 geschrieben werden kann. Der als Y-Koordinate übergebene Wert von 16 Pixeln entspricht der gewählten Schriftgröße, weswegen der Text direkt unter dem oberen Rand des Widgets erscheint.

23.6.6 Eye-Candy

Eingangs wurde erwähnt, dass das Qt-Framework unter anderem in Bezug auf seine Zeichenfunktionalität aus der Masse der GUI-Toolkits heraussticht. Zugegebenermaßen sind die bislang besprochenen Grundlagen zum Zeichnen in einem Qt-Widget zwar wichtig, aber auch nicht besonders beeindruckend. Funktionalität zum Zeichnen von grundlegenden geometrischen Formen, Grafiken und Text finden Sie so oder so ähnlich auch in vielen anderen Toolkits. Aus diesem Grund möchten wir in diesem Kapitel einige Aspekte der Zeichenfunktionalität von Qt in den Vordergrund holen und als »Eye-Candy« (dt. »Blickfang, Augenweide«) präsentieren. Die hier besprochenen Aspekte des Zeichnens in Qt dienen als Demonstration der Zeichenfunktionalität und sollen Stichwörter liefern, unter denen in der Qt-Dokumentation näher nachgeforscht werden kann.

Farbverläufe

Abgesehen von einem flächigen Farbanstrich, kann ein Brush einen Bereich auch mit komplexeren Strukturen füllen. So kann beispielsweise das Innere eines Rechtecks mit einem Farbverlauf gefüllt werden, wie in Abbildung 23.49 zu sehen ist.



Abbildung 23.49 Ein Farbverlauf mit QLinearGradient

Um einen Brush zu erstellen, der Flächen mit einem Farbverlauf füllt, muss zunächst eine Instanz einer *Gradient*-Klasse (dt. »Gefälle«) erzeugt werden. Eine solche Klasse enthält alle Informationen, die benötigt werden, um einen Farbverlauf zu zeichnen. Es existieren drei verschiedene Gradient-Klassen, die jeweils einen eigenen Typus von Farbverlauf beschreiben:

Klasse	Beschreibung
<i>QtGui.QConicalGradient</i>	Beschreibt einen konischen Farbverlauf. Das Ergebnis ähnelt der Draufsicht eines Kegels.
<i>QtGui.QLinearGradient</i>	Beschreibt einen linearen Farbverlauf. Ein solcher wurde im Beispielwidget aus Abbildung 23.49 verwendet.
<i>QtGui.QRadialGradient</i>	Beschreibt einen kreisförmigen Farbverlauf.

Tabelle 23.53 Gradient-Klassen

Nachdem eine Instanz einer Gradient-Klasse mit den erforderlichen Informationen über den Farbverlauf erzeugt wurde, kann diese dem Konstruktor eines Brushes als Parameter übergeben werden. Ein auf diese Weise erzeugter Brush kann dann verwendet werden, um eine Fläche mit einem Farbverlauf zu füllen.

Transparenz

Das Qt-Framework unterstützt sowohl bei einem Brush als auch bei einem Pen das sogenannte *Alpha-Blending*. Darunter versteht man das Zeichnen von teilweise

transparenten Strukturen. Abbildung 23.50 zeigt zwei sich überschneidende, semi-transparente Kreise.

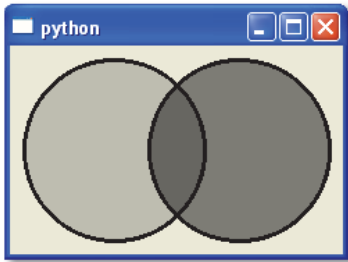


Abbildung 23.50 Alpha-Blending

Zum Verwenden von Alpha-Blending reicht es aus, bei der Erzeugung eines Brushes bzw. eines Pens eine `QColor`-Instanz mit einem Transparenzwert zu übergeben. Dieser Transparenzwert wird dem `QColor`-Konstruktor als vierter Parameter übergeben. Hier sind ganzzahlige Werte zwischen 0 (vollständig transparent) und 255 (opak) möglich.

Diese Möglichkeiten zur Darstellung von Transparenzen lassen sich auch im Zusammenhang mit den bereits besprochenen Farbverläufen für interessante Effekte nutzen (siehe Abbildung 23.51).



Abbildung 23.51 Transparenzeffekt

In diesem Beispielwidget wird eine von einem Rechteck überlagerte Grafik angezeigt. Das Innere dieses Rechtecks ist mit einem Farbverlauf-Brush gezeichnet. Die Ziel-farbe dieses Farbverlaufs ist vollständig transparent.

Anti-Aliasing

Wenn Sie sich das Beispielwidget mit den beiden überlappenden, teilweise transparenten Kreisen noch einmal ansehen, werden Sie feststellen, dass man die einzelnen Pixel, aus denen der Umriss der Kreise besteht, erkennen kann. Die Kreise sehen des-

wegen nicht besonders ansprechend aus. In vielen Fällen soll eine solche oder ähnliche Zeichnung »sauber« aussehen. Zu diesem Zweck existiert eine Technik namens *Anti-Aliasing*, von der Sie vielleicht schon im Zusammenhang mit Computerspielen gehört haben. Beim Anti-Aliasing werden die Randbereiche einer Zeichnung geglättet, sodass einzelne Pixel nicht mehr auszumachen sind. Das Qt-Framework bietet grundlegende Unterstützung zum Zeichnen mit Anti-Aliasing.

Das Transparenz-Beispiel mit aktiviertem Anti-Aliasing ist in Abbildung 23.52 zu sehen.

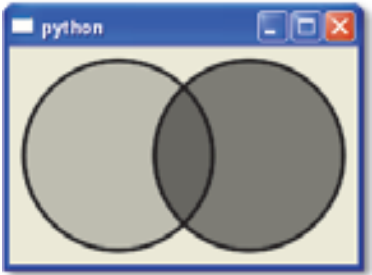


Abbildung 23.52 Anti-Aliasing

Um Anti-Aliasing bei einem Painter zu aktivieren, wird die Codezeile

```
painter.setRenderHints(QtGui.QPainter.Antialiasing)
```

verwendet, wobei `painter` eine `QPainter`-Instanz ist.

Transformationen

Eine weitere interessante Möglichkeit, die Qt bietet, sind *Transformationen*, die mithilfe von *Transformationsmatrizen* auf beliebige zu zeichnende Formen angewendet werden können. Eine Transformationsmatrix wird durch die Klasse `QMatrix` repräsentiert.

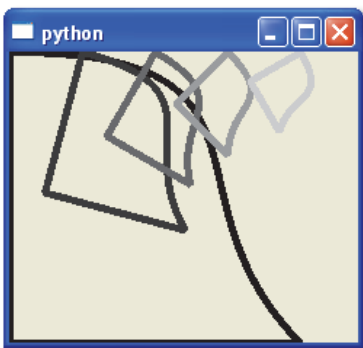


Abbildung 23.53 Matrixtransformationen

Im Beispiel aus Abbildung 23.53 wurde zunächst eine Figur erstellt. Da es sich dabei nicht um eine vorgefertigte Form handelt, muss die Figur mithilfe eines sogenannten *Painter Path* zu einer Einheit zusammengefügt werden. Ein *Painter Path* ist eine Instanz der Klasse `QPainterPath`. Die Form dieses Beispiels besteht aus zwei Linien und einer *Beziérkurve*²⁰.

Nachdem sowohl die Transformationsmatrix als auch der *Painter Path* erstellt worden sind, kann die Matrix auf den *Painter Path* angewandt und der resultierende, transformierte *Painter Path* schließlich gezeichnet werden. Im Beispiel wurde die Matrix in fünf Iterationsschritten immer wieder verändert und die entstandene Figur jeweils mit einem unterschiedlichen Pen gezeichnet.

23.7 Model-View-Architektur

Mit der Version 4 wurde die *Model-View-Architektur* in das Qt-Framework eingeführt. Die grundlegende Idee dieser Art der Programmierung ist es, Form und Inhalt voneinander zu trennen. Bezogen auf Qt, bedeutet das, dass Klassen, die Daten enthalten, von Klassen getrennt werden sollen, die diese Daten anzeigen. So gibt es eine *Modellklasse*, die ein bekanntes Interface für die gespeicherten Daten bereitstellt, und eine *Viewklasse*, die über die Modellklasse auf die Daten zugreift und diese auf der grafischen Oberfläche anzeigt. Es wird nicht vorausgesetzt, dass die Daten tatsächlich in der Modellklasse gespeichert sind, sondern nur, dass die Modellklasse Methoden bereitstellt, um auf die Daten zuzugreifen. Die Daten selbst können in einer Datenbank oder Datei stehen.

Das Aufteilen der Programmlogik in Modell- und Viewklassen hat den Vorteil, dass das Programm insgesamt einfacher und besser strukturiert wird. Außerdem führen Änderungen in der Datenrepräsentation nicht dazu, dass die Anzeigeklasse angepasst werden muss. Umgekehrt ist es der Modellklasse egal, in welcher Form die von ihr bereitgestellten Daten am Bildschirm angezeigt werden.

Das Verhältnis zwischen Modell- und Viewklasse wird in Abbildung 23.54 veranschaulicht.

Das Qt-Framework bietet einige Klassen, die dem Programmierer beim Erstellen einer Model-View-Architektur helfen. Darunter finden sich Basisklassen sowohl für die Modell- als auch für die Viewklassen. Im Folgenden wird eine rudimentäre Adressbuch-Applikation als Beispiel für eine Model-View-Architektur besprochen.

²⁰ Eine Beziérkurve ist eine Kurve, die durch eine mathematische Funktion mit, im Falle einer kubischen Beziérkurve, vier Parametern beschrieben wird. Beziérkurven können auch in vielen Grafikprogrammen erstellt werden.

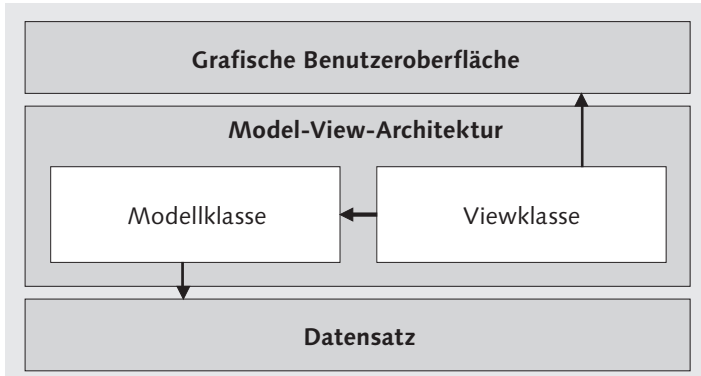


Abbildung 23.54 Die Model-View-Architektur

23.7.1 Beispielprojekt: Ein Adressbuch

In diesem Abschnitt bieten wir einen praxisorientierten Einstieg in die Programmierung einer Model-View-Architektur anhand eines einfachen Beispielprogramms. Dazu dient ein grafisches Adressbuch, das beim Starten mehrere Adresssätze aus einer Textdatei einliest und dann grafisch auf dem Bildschirm darstellt. Intern werden die Datensätze durch eine Modellklasse eingelesen und aufbereitet. Eine Viewklasse kümmert sich dann um die Anzeige der Daten.

Im ersten Schritt konzentrieren wir uns dabei auf das Einlesen und Anzeigen der Daten. Danach werden durch Erweiterungen des Beispielprogramms weitere Aspekte der Model-View-Programmierung in Qt vorgestellt. Die vorläufige Anwendung, die in diesem Kapitel entwickelt wird, ist in Abbildung 23.55 dargestellt.



Abbildung 23.55 Ein Adressbuch

Die Adressdaten sollen aus einer Datei des folgenden Formats ausgelesen werden:

Donald Duck
don@ld.de
Pechvogelstraße 13
12345 Entenhausen
01234/313

Dagobert Duck
d@gobert.de
Geldspeicherweg 42
12345 Entenhausen
0190/123456
[...]

Die Adressdaten sind also zeilenweise in einer Datei gespeichert. Zwei Einträge im Adressbuch werden durch eine Leerzeile in der Quelldatei voneinander getrennt. Abgesehen davon, dass der Name der Person, zu der der Eintrag gehört, in der ersten Zeile des Eintrags steht, gibt es keine weiteren Anforderungen an die Formatierung der Daten.²¹

Das Adressbuch stellt eine Beispielimplementation für eine Model-View-Architektur dar:

- Die Modellklasse hat die Aufgabe, die Quelldatei mit den Adressdaten einzulesen und eine Schnittstelle bereitzustellen, über die auf diese Daten zugegriffen werden kann.
- Die Viewklasse greift auf die in der Modellklasse gespeicherten Daten zu und präsentiert diese dann in geeigneter Form auf dem Bildschirm. Da es sich bei dem Adressbuch im Prinzip um eine Liste von Adresseinträgen handelt, können wir hier auf die Basisklasse `QListView` des Qt-Frameworks zurückgreifen, die die grundlegende Funktionalität zum Anzeigen von Modelldaten mit Listenstruktur bereitstellt. Lägen die Daten in einer anderen Struktur vor, könnten die Basisklassen `QTreeView` oder `QTableView` verwendet werden, die eine baumartige bzw. tabellarische Struktur der Daten visualisieren.

Abbildung 23.56 stellt die Programmstruktur grafisch dar.

²¹ Tatsächlich ist das Dateiformat für den vorgestellten Verwendungszweck eher ungeeignet, da es beispielsweise für das Programm, das die Datei einliest, keine effiziente Möglichkeit gibt, die einzelnen Teilinformationen des Eintrags zuzuordnen, beispielsweise also die E-Mail-Adresse herauszufiltern. Das Dateiformat wird hier jedoch aufgrund seiner Einfachheit verwendet.

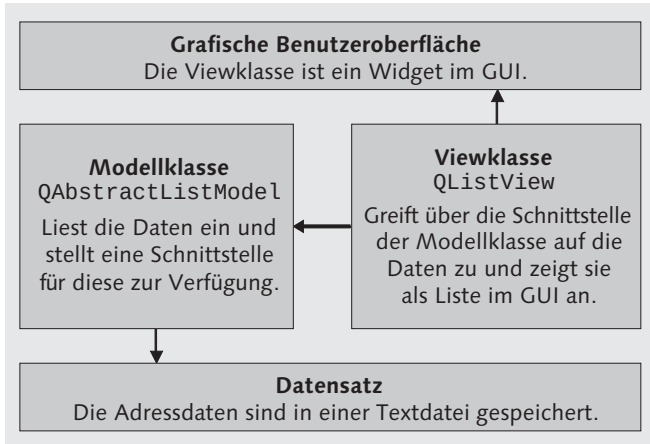


Abbildung 23.56 Die Model-View-Architektur des Beispielprogramms

Der Quellcode der Modellklasse befindet sich in der Programmdatei *modell.py* und sieht folgendermaßen aus:

```

from PyQt4 import QtCore

class Modell(QtCore.QAbstractListModel):
    def __init__(self, dateiname):
        QtCore.QAbstractListModel.__init__(self)
        self.datensatz = []

        # Lade Datensatz
        with open(dateiname) as f:
            lst = []
            for zeile in f:
                if not zeile.strip():
                    self.datensatz.append(lst)
                    lst = []
                else:
                    lst.append(zeile.strip())
            if lst:
                self.datensatz.append(lst)

    def rowCount(self, parent=QtCore.QModelIndex()):
        return len(self.datensatz)

    def data(self, index, role=QtCore.Qt.DisplayRole):
        return self.datensatz[index.row()]
  
```

Es wird die Modellklasse `Modell` definiert, die von der Basisklasse `QtCore.QabstractListModel` erbt. Diese Basisklasse implementiert die grundlegende Funktionalität einer Modellklasse für einen Datensatz mit Listenstruktur.

Im Konstruktor der Klasse `Modell` werden die Adressdaten aus einer Textdatei des oben beschriebenen Formats geladen. Dazu bekommt der Konstruktor den Dateinamen dieser Datei übergeben. Da das Dateiformat, in dem die Daten vorliegen, sehr einfach ist, ist auch der Einlesevorgang vergleichsweise simpel und braucht nicht näher erläutert zu werden. Wichtig ist aber, dass die einzelnen Einträge des Adressbuchs klassenintern in einer Liste gespeichert werden, die durch das Attribut `self.datensatz` referenziert wird. Jeder Eintrag dieser Liste ist wiederum eine Liste von Strings, die den Zeilen des Eintrags entsprechen.

Am Ende der Klassendefinition werden noch zwei Methoden definiert, die jede Modellklasse implementieren muss. Diese Methoden bilden die Schnittstelle, über die die Viewklasse später auf die in der Modellklasse gespeicherten Daten zugreifen kann.

Die Methode `rowCount` muss die Anzahl der Elemente als ganze Zahl zurückgeben, die der Datensatz enthält. Der dabei übergebene Parameter `parent` spielt an dieser Stelle keine Rolle.

Die Methode `data` wird von der Viewklasse aufgerufen, um auf ein bestimmtes Element des Datensatzes zuzugreifen. Welches das ist, wird über den Parameter `index` mitgeteilt. Bei `index` handelt es sich aber nicht um eine ganze Zahl, sondern um eine `QModelIndex`-Instanz. Auf den numerischen Index kann über die Methode `row` dieser Instanz zugegriffen werden.

Die zur Modellklasse passende Viewklasse ist in der Programmdatei `view.py` enthalten und sieht folgendermaßen aus:

```
class View(QtGui.QListView):
    def __init__(self, modell, parent=None):
        QtGui.QListView.__init__(self, parent)
        self.delegate = ViewDelegate()
        self.setItemDelegate(self.delegate)
        self.setModel(modell)
        self.setVerticalScrollMode(QtGui.QListView.ScrollPerPixel)
```

Die Viewklasse `View` wird von der Basisklasse `QtGui.QListView` abgeleitet. Diese Basisklasse stellt die Funktionalität bereit, die benötigt wird, um einen listenartigen Datensatz grafisch darzustellen. Alternativ hätten auch die Klassen `QTreeView` und `QTableView` als Basisklassen dienen können, wenn zur Darstellung der Daten eine baumartige oder tabellarische Struktur besser gewesen wäre.

Dem Konstruktor der Klasse `View` wird eine Instanz der soeben definierten Modellklasse `Modell` übergeben, deren Inhalt grafisch dargestellt werden soll. Um die Daten

jedoch tatsächlich anzuzeigen, wird eine weitere Klasse benötigt, der sogenannte *Delegate* (dt. »Abgesandter«). Ein Delegate, den wir im Anschluss an die Viewklasse besprechen, wird der Viewklasse über die Methode `setItemDelegate` zugewiesen. Die Delegate-Klasse enthält die Zeichenroutinen für ein Element des Datensatzes.

Zum Schluss wird noch das Modell mittels `setModel` eingebunden und, was eher eine kosmetische Angelegenheit ist, der Scrollmodus auf »pixelweise« gesetzt. Im Normalzustand verschiebt das `QListView`-Widget den Inhalt beim Scrollen immer um ganze Einträge, was bei großen Einträgen nicht schön aussieht.

Noch zu besprechen ist die Delegate-Klasse, die das Zeichnen der Einträge übernimmt. Dazu kann sie über die Schnittstelle der Modellklasse auf den eingelesenen Datensatz zugreifen. In der Grafik, die eingangs die Model-View-Architektur des Beispielprogramms veranschaulichte, wurde die Delegate-Klasse aus Gründen der Einfachheit weggelassen. Das möchten wir an dieser Stelle nachholen und zeigen in Abbildung 23.57, wie sich die Delegate-Klasse in die Model-View-Architektur integriert.

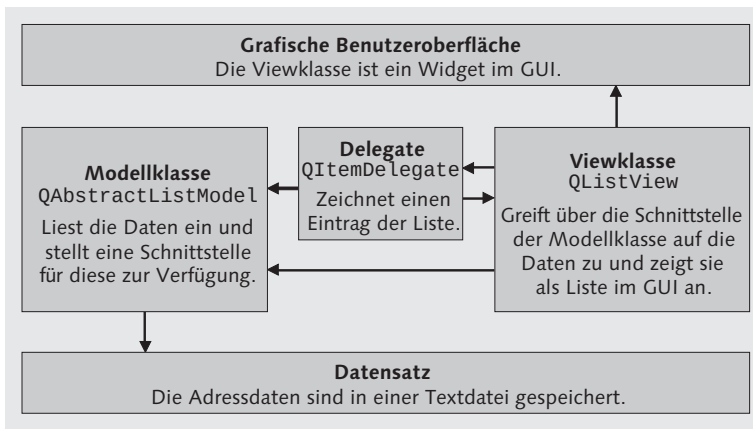


Abbildung 23.57 Die Model-View-Architektur des Beispielprogramms

Die Delegate-Klasse positioniert sich als Hilfsklasse zwischen der View- und der Modellklasse. Die Viewklasse ruft die Methode `paint` der Delegate-Klasse für jeden Eintrag im Datensatz auf und stellt aus den Einzelzeichnungen das `ListView` Widget zusammen, das an der grafischen Benutzeroberfläche angezeigt wird. Wie bei der View- und Modellklasse, existiert im Qt-Framework eine Basisklasse, von der eine selbstdefinierte Delegate-Klasse abgeleitet werden muss.

Um einen Eintrag adäquat zeichnen zu können, kann die Delegate-Klasse über die von der Modellklasse bereitgestellte Schnittstelle auf den Datensatz zugreifen. Selbstverständlich kann auch die Viewklasse selbst auf diesem Wege Daten des Datensatzes lesen.

Im Folgenden wird das Beispielprogramm um die noch fehlende Delegate-Klasse erweitert. Diese ist in der gleichen Programmdatei definiert wie die Viewklasse. Da die Delegate-Klasse vergleichsweise umfangreich ist, werden wir sie Methode für Methode besprechen:

```
from PyQt4 import QtGui, QtCore

class ViewDelegate(QtGui.QItemDelegate):
    def __init__(self):
        QtGui.QItemDelegate.__init__(self)

        self.rahmenStift = QtGui.QPen(QtGui.QColor(0,0,0))
        self.titelTextStift = QtGui.QPen(
            QtGui.QColor(255,255,255))
        self.titelFarbe = QtGui.QBrush(QtGui.QColor(120,120,120))
        self.textStift = QtGui.QPen(QtGui.QColor(0,0,0))
        self.titelSchriftart = QtGui.QFont("Helvetica", 10,
            QtGui.QFont.Bold)
        self.textSchriftart = QtGui.QFont("Helvetica", 10)

        self.zeilenHoehe = 15
        self.titelHoehe = 20
        self.abstand = 4
        self.abstandInnen = 2
        self.abstandText = 4
```

Im Konstruktor der Klasse `ViewDelegate` werden die Attribute initialisiert, die zum Zeichnen eines Adresseintrags von Bedeutung sind. Dazu zählen zum einen die Zeichenwerkzeuge, mit denen der Adresseintrag gezeichnet werden soll, und zum anderen Konstanten, die Richtgrößen zum Zeichnen eines Eintrags festlegen. Um zu besprechen, welches Attribut wofür gedacht ist, vergegenwärtigen wir uns anhand von Abbildung 23.58 noch einmal, wie ein Eintrag im späteren Programm aussieht.



Abbildung 23.58 Ein Eintrag im Adressbuch

Die folgende Tabelle listet alle Attribute der Klasse `ViewDelegate` mit einer kurzen Beschreibung der jeweiligen Bedeutung auf.

Attribut	Beschreibung
rahmenStift	der Stift, mit dem der dünne schwarze Rahmen um den Eintrag gezeichnet wird
titelTextStift	der Stift, mit dem die Überschrift geschrieben wird
titelFarbe	der Pinsel, mit dem das graue Rechteck unter der Überschrift gezeichnet wird
titelSchriftart	die Schriftart, in der die Überschrift geschrieben wird
textStift	der Stift, mit dem die Adressdaten geschrieben werden
textSchriftart	die Schriftart, in der die Adressdaten geschrieben werden
zeilenHoehe	die Höhe einer Adressdaten-Zeile in Pixel
titelHoehe	die Höhe der Überschrift in Pixel
abstand	der Abstand eines Eintrags vom Dialogrand und von anderen Einträgen in Pixel
abstandInnen	der Abstand zwischen dem grauen Rechteck unter der Überschrift und der Umrandung des Eintrags in Pixel
abstandText	der Abstand des Textes von der Umrandung des Eintrags auf der linken Seite in Pixel

Tabelle 23.54 Attribute der Klasse ViewDelegate

Damit ist der Konstruktor vollständig beschrieben. Es folgt die Methode `sizeHint`, die jede Delegate-Klasse implementieren muss.

```
def sizeHint(self, option, index):
    anz = len(index.data())
    return QtCore.QSize(170,
                        self.zeilenHoehe*anz + self.titelHoehe)
```

Die Methode wird von der Viewklasse aufgerufen, um die Höhe und die Breite eines einzelnen Eintrags in Erfahrung zu bringen. Dabei bekommt sie zwei Parameter übergeben: `option` und `index`.

Für den Parameter `option` wird eine Instanz der Klasse `QStyleOptionViewItem` übergeben, die verschiedene Anweisungen enthalten kann, in welcher Form der Eintrag gezeichnet werden soll. Da diese Formatanweisungen möglicherweise auch Einfluss auf die Maße eines Eintrags haben, werden sie auch der Funktion `sizeHint` überge-

ben. In unserem Beispielprogramm ist der Parameter `option` nicht von Belang und wird nicht weiter erläutert.

Mit dem zweiten Parameter, `index`, wird das Element spezifiziert, dessen Dimensionen zurückgegeben werden sollen. Für `index` wird eine Instanz der Klasse `QModelIndex` übergeben. Wichtig ist vor allem die Methode `data` der `QModelIndex`-Instanz, über die auf die Daten des Eintrags zugegriffen werden kann.

Die Implementierung der Methode `sizeHint` berechnet Breite und Höhe eines Eintrags und gibt die Werte als `QSize`-Instanz zurück. Die Breite liegt dabei bei konstanten 170 Pixeln.²²

Die folgende Methode `paint` muss von einer Delegate-Klasse implementiert werden und wird immer dann aufgerufen, wenn ein einzelner Eintrag neu gezeichnet werden muss. Es wird pro `paint`-Aufruf immer nur ein Eintrag gezeichnet.

```
def paint(self, painter, option, index):
    rahmen = option.rect.adjusted(self.abstand, self.abstand,
                                  -self.abstand, -self.abstand)
    rahmenTitel = rahmen.adjusted(self.abstandInnen,
                                   self.abstandInnen, -self.abstandInnen+1, 0)
    rahmenTitel.setHeight(self.titelHoehe)
    rahmenTitelText = rahmenTitel.adjusted(self.abstandText,
                                             0, self.abstandText, 0)

    datensatz = index.data()
    painter.save()
    painter.setPen(self.rahmenStift)
    painter.drawRect(rahmen)
    painter.fillRect(rahmenTitel, self.titelFarbe)

    # Titel schreiben
    painter.setPen(self.titelTextStift)
    painter.setFont(self.titelSchriftart)
    painter.drawText(rahmenTitelText,
                     QtCore.Qt.AlignLeft | QtCore.Qt.AlignVCenter,
                     datensatz[0])

    # Adresse schreiben
    painter.setPen(self.textStift)
    painter.setFont(self.textSchriftart)
```

²² Dabei handelt es sich um eine Vereinfachung des Beispielprogramms. Alternativ kann die Breite des Eintrags anhand der längsten Zeile berechnet werden. Dazu verwendet man die Methode `width` einer `QFontMetrics`-Instanz.

```
for i, eintrag in enumerate(datensatz[1:]):
    painter.drawText(rahmenTitel.x() + self.abstandText,
                    rahmenTitel.bottom() + (i+1)*self.zeilenHoehe,
                    eintrag)
painter.restore()
```

Die Methode `paint` bekommt die drei Parameter `painter`, `option` und `index` übergeben. Für den Parameter `painter` wird eine `QPainter`-Instanz übergeben, die zum Zeichnen verwendet werden soll. Die beiden Parameter `option` und `index` haben die gleiche Bedeutung wie bei der Methode `sizeHint`.

In der Methode `paint` werden zunächst einige Rechtecke berechnet, die nachher zum Zeichnen des Eintrags verwendet werden. Beachten Sie, dass `option.rect` die Zeichenfläche für den Eintrag in Form einer `QRect`-Instanz beschreibt. Alle Zeichenoperationen sollten sich also an diesem Rechteck ausrichten. Die angelegten lokalen Referenzen haben die folgenden Bedeutungen:

Attribut	Beschreibung
<code>rahmen</code>	das Rechteck, um das der dünne schwarze Rahmen gezogen wird
<code>rahmenTitel</code>	das Rechteck der grau hinterlegten Titelzeile
<code>rahmenTitelText</code>	das Rechteck, in das der Text in der Titelzeile geschrieben wird

Tabelle 23.55 Lokale Referenzen in der Methode `paint`

Nachdem die lokalen Referenzen angelegt wurden, wird der Status des Painters mittels `save` gespeichert, um ihn am Ende der Methode mittels `restore` wiederherstellen zu können. Ein als Parameter übergebener Painter sollte immer in den Ausgangszustand zurückversetzt werden, nachdem die Zeichenoperationen durchgeführt wurden, da sonst ein ungewollter Seiteneffekt in einer übergeordneten Funktion auftreten könnte.

Danach werden mithilfe der Methoden `drawRect` und `fillRect` des Painters der Rahmen um den Eintrag und die grau hinterlegte Titelzeile gezeichnet. Jetzt fehlen nur noch die Beschriftungen. Dazu werden zunächst die passende Schriftart und das gewünschte Stiftwerkzeug mittels `setFont` und `setPen` ausgewählt. Die Titelzeile des Eintrags wird im Rechteck `rahmenTitelText` linksbündig und vertikal zentriert mit der fetten Schriftart `titelSchriftart` und einem weißen Pen geschrieben.

Die Methode `drawText` des Painters kann in mehreren Varianten aufgerufen werden. So ist es beispielsweise möglich, (wie bei der Titelzeile) ein Rechteck und eine Positi-

onsanweisung innerhalb dieses Rechtecks zu übergeben oder (wie bei den Adresszeilen des Eintrags) direkt die Koordinaten anzugeben, an die der Text geschrieben werden soll.

Zu guter Letzt werfen wir noch einen Blick auf das Hauptprogramm, das in der Programmdatei *programm.py* steht:

```
from PyQt4 import QtGui
import sys
import modell
import view

m = modell.Modell("adressbuch.txt")

app = QtGui.QApplication(sys.argv)
liste = view.View(m)
liste.resize(200, 500)
liste.show()
sys.exit(app.exec_())
```

Nachdem die lokalen Module *modell* und *view* eingebunden wurden, wird eine Instanz der Klasse *Modell* erzeugt, die den Datensatz aus der Datei *adressbuch.txt* repräsentiert.

Die Viewklasse *View* dient als einziges Widget der Applikation gleichzeitig als Fensterklasse. Bevor das Widget mittels *show* angezeigt wird, setzen wir seine Größe durch Aufruf der Methode *resize* auf einen sinnvollen Wert (200 Pixel breit und 500 Pixel hoch).

Wenn das Hauptprogramm ausgeführt wird, können Sie sehen, dass sich die Basisklasse *QListView* der Viewklasse tatsächlich um Feinheiten wie das Scrollen von Einträgen oder das Anpassen der Einträge bei einer Größenänderung kümmert (siehe Abbildung 23.59).



Abbildung 23.59 Scrollen im Adressbuch

23.7.2 Auswählen von Einträgen

Nachdem die Adressdaten als Liste angezeigt wurden, stellt sich die Frage, in welcher Form Benutzerinteraktionen umgesetzt werden können. In diesem Abschnitt wird das Beispielprogramm dahingehend erweitert, dass der Benutzer einen Eintrag des Adressbuchs auswählen kann. Im darauffolgenden Abschnitt behandeln wir dann das Editieren von Einträgen.

An der Grundstruktur des Beispielprogramms muss dafür nicht viel verändert werden, denn genau genommen ist das Auswählen im vorherigen Beispielprogramm schon möglich gewesen, wir haben bis jetzt nur alle Einträge der Liste gleich gezeichnet. Was noch fehlt, ist also die grafische Hervorhebung des ausgewählten Eintrags.

Das Adressbuch mit einem ausgewählten Eintrag ist in Abbildung 23.60 dargestellt.



Abbildung 23.60 Ein ausgewählter Eintrag im Adressbuch

Der ausgewählte Eintrag unterscheidet sich in der Farbe der Titelleiste und des Hintergrundes von den anderen Einträgen. Statt in Grau werden diese Flächen bei ausgewählten Einträgen blau gezeichnet.

Dazu legen wir im Konstruktor der Delegate-Klasse zunächst einen neuen Brush mit den gewünschten Blautönen als Farbe an:

```
def __init__(self):
    QtGui.QItemDelegate.__init__(self)
    [...]
    self.titelFarbeAktiv = QtGui.QBrush(QtGui.QColor(0,0,120))
    self.hintergrundFarbeAktiv = QtGui.QBrush(
        QtGui.QColor(230,230,255))
    [...]
```

Jetzt muss nur noch beim Zeichnen eines Eintrags, also in der Methode `paint`, unterschieden werden, ob es sich bei dem zu zeichnenden Eintrag um den momentan ausgewählten handelt oder nicht. Dies lässt sich anhand des Attributs `state` der `QStyleOptionViewItem`-Instanz feststellen, die beim Aufruf der Methode `paint` für den Parameter `option` übergeben wird.

Wir ändern also das Zeichnen des grauen Titelrechtecks in folgenden Code:

```
if option.state & QtGui.QStyle.State_Selected:
    painter.fillRect(rahmen, self.hintergrundFarbeAktiv)
    painter.fillRect(rahmenTitel, self.titelFarbeAktiv)
else:
    painter.fillRect(rahmenTitel, self.titelFarbe)
```

Dieser Code muss vor dem Zeichnen des dünnen schwarzen Rahmens stehen:

```
painter.setPen(self.rahmenStift)
painter.drawRect(rahmen)
```

Das waren schon alle notwendigen Schritte, um es dem Benutzer zu erlauben, einen Eintrag des Adressbuchs auszuwählen.

23.7.3 Bearbeiten von Einträgen

Nachdem wir uns damit beschäftigt haben, wie die Adressdaten in einem `QListView`-Widget angezeigt werden können, und das Beispielprogramm dahingehend erweitert haben, dass ein Eintrag vom Benutzer ausgewählt werden kann, liegt die Frage nahe, ob wir dem Benutzer auch das Bearbeiten eines Datensatzes erlauben können. Es ist zwar nicht so einfach wie das Selektieren im vorherigen Abschnitt, doch auch für das Editieren eines Eintrags bietet die Model-View-Architektur von Qt eine komfortable Schnittstelle an.

Im späteren Programm wird das Bearbeiten eines Eintrags so aussehen, wie es in Abbildung 23.61 gezeigt ist.



Abbildung 23.61 Bearbeiten eines Adresseintrags

Um das Bearbeiten von Einträgen zu ermöglichen, müssen die einzelnen Einträge des Datensatzes von der Modellklasse zunächst explizit als editierbar gekennzeichnet werden. Dazu muss die Modellklasse die Methode `flags` implementieren:

```
def flags(self, index):
    return QtCore.Qt.ItemIsSelectable | QtCore.Qt.ItemIsEditable\
           | QtCore.Qt.ItemIsEnabled
```

Diese Methode wird immer dann aufgerufen, wenn das `QListView-Widget` nähere Informationen über den Eintrag erhalten will, der durch die `QModelIndex-Instanz` `index` spezifiziert wird. In unserem Fall werden unabhängig vom Index des Eintrags pauschal die Flags `ItemIsSelectable`, `ItemIsEditable` und `ItemIsEnabled` zurückgegeben, die für einen selektierbaren, editierbaren und aktivierten Eintrag stehen. Standardmäßig – also wenn die Methode `flags` nicht implementiert wird – erhält jeder Eintrag die Flags `ItemIsSelectable` und `ItemIsEnabled`.

Zusätzlich zur Methode `flags` sollte die Modellklasse die Methode `setData` implementieren, die die Aufgabe hat, die vom Benutzer veränderten Einträge in den Datensatz zu übernehmen.

```
def setData(self, index, value, role=QtCore.Qt.EditRole):
    self.datensatz[index.row()] = value
    self.layoutChanged.emit()
    return True
```

Die Methode bekommt den Index des veränderten Eintrags und den veränderten Inhalt dieses Eintrags übergeben. Der zusätzliche Parameter `role` soll an dieser Stelle nicht weiter interessieren. Im Körper der Methode wird der alte Eintrag in dem in der Modellklasse gespeicherten Datensatz `self.datensatz` durch den veränderten ersetzt. Danach wird das Signal `layoutChanged` gesendet, das die Viewklasse dazu veranlasst, die Anzeige vollständig neu aufzubauen. Das ist sinnvoll, da sich durch die Änderungen des Benutzers die Zeilenzahl und damit die Höhe des jeweiligen Eintrags verändert haben könnte.

Das sind alle Änderungen, die an der Modellklasse vorgenommen werden müssen, um das Editieren eines Eintrags zu erlauben. Doch auch die Delegate-Klasse muss einige zusätzliche Methoden implementieren. Dabei handelt es sich um die Methoden `createEditor`, `setEditorData`, `updateEditorGeometry`, `setModelData` und `eventFilter`, die im Folgenden besprochen werden.

Die Methode `createEditor` wird aufgerufen, wenn der Benutzer doppelt auf einen Eintrag klickt, um diesen zu editieren. Die Methode `createEditor` muss ein Widget zurückgeben, das dann statt des entsprechenden Eintrags zum Editieren angezeigt wird.

```
def createEditor(self, parent, option, index):
    return QtGui.QTextEdit(parent)
```

Die Methode bekommt die bereits bekannten Parameter `option` und `index` übergeben, die den zu editierenden Eintrag spezifizieren. Zusätzlich wird für `parent` das Widget übergeben, das als Elternwidget des Editorwidgets eingetragen werden soll. In diesem Fall erstellen wir ein `QTextEdit`-Widget, in dem der Benutzer den Eintrag bearbeiten soll.

Die Methode `setEditorData` wird vom `QListView`-Widget aufgerufen, um das von `createEditor` erzeugte Widget mit Inhalt zu füllen.

```
def setEditorData(self, editor, index):
    editor.setPlainText("\n".join(index.data()))
```

Dazu bekommt die Methode das Editorwidget in Form des Parameters `editor` und den bekannten Parameter `index` übergeben, der den zu editierenden Eintrag spezifiziert. Im Methodenkörper werden die Daten des zu editierenden Eintrags ausgelesen und mittels `join` zu einem einzigen String zusammengefügt. Dieser String wird dann durch Aufruf der Methode `setPlainText` in das `QTextEdit`-Widget geschrieben.

Die Methode `updateEditorGeometry` wird vom `QListView`-Widget aufgerufen, um die Größe des Editorwidgets festlegen zu lassen.

```
def updateEditorGeometry(self, editor, option, index):
    rahmen = option.rect.adjusted(self.abstand, self.abstand,
                                   -self.abstand, -self.abstand)
    editor.setGeometry(rahmen)
```

Die Methode bekommt die bekannten Parameter `option` und `index` und zusätzlich das Editorwidget `editor` übergeben. In diesem Fall verpassen wir dem Editorwidget mittels `setGeometry` die gleiche Größe, die der entsprechende Eintrag gehabt hätte, wenn er normal gezeichnet werden würde.

Die Methode `setModelData` wird aufgerufen, wenn das Editieren durch den Benutzer erfolgt ist, um die veränderten Daten aus dem Editorwidget auszulesen und an die Modellklasse weiterzureichen.

```
def setModelData(self, editor, model, index):
    model.setData(index, editor.toPlainText().split("\n"))
```

Die Methode bekommt sowohl das Editorwidget als auch die Modellklasse in Form der Parameter `editor` und `model` übergeben. Zusätzlich wird eine `QModelIndex`-Instanz übergeben, die den editierten Eintrag spezifiziert. In der Methode wird der Text des `QTextEdit`-Widgets ausgelesen und in einzelne Zeilen unterteilt. Danach wird die vorhin angelegte Methode `setData` der Modellklasse aufgerufen.

Damit ist die grundlegende Funktionalität zum Editieren eines Eintrags implementiert. Allerdings werden Sie beim Ausführen des Programms feststellen, dass die `[Enter]`-Taste beim Editieren eines Eintrags sowohl eine neue Zeile beginnt als auch das Editieren des Eintrags beendet. Das ist nicht besonders glücklich und sollte behoben werden. Dazu implementieren wir die Methode `eventFilter`, die immer dann aufgerufen wird, wenn ein Event eintritt. Ein Event ist beispielsweise das Drücken einer Taste während des Editierens eines Eintrags.

```
def eventFilter(self, editor, event):
    if event.type() == QtCore.QEvent.KeyPress \
        and event.key() in (QtCore.Qt.Key_Return,
                           QtCore.Qt.Key_Enter):
        return False
    return QtGui.QItemDelegate.eventFilter(self, editor, event)
```

Die Methode bekommt das Editorwidget `editor` und eine `QEvent`-Instanz übergeben, die das aufgetretene Event spezifiziert. Im Körper der Methode wird überprüft, ob es sich bei dem Event um einen Tastendruck handelt und wenn ja, ob es sich bei der gedrückten Taste um die `Enter`- oder die `↵`-Taste handelt.²³ Nur wenn es sich bei dem Event nicht um eine gedrückte `Enter`- oder `↵`-Taste handelt, wird die Standardimplementierung der Methode aufgerufen, beispielsweise soll also bei gedrückter `Esc`-Taste weiterhin das Editieren des Eintrags abgebrochen werden. Im Falle der `Enter`- oder der `↵`-Taste wird nichts dergleichen unternommen.

²³ Es gibt einen Unterschied zwischen diesen beiden Tasten. `Enter` finden Sie auf der Tastatur in der unteren rechten Ecke des Nummernblocks, während die `↵`-Taste diejenige ist, die Sie verwenden, um in einem Text eine neue Zeile zu beginnen.

Kapitel 24

Wissenschaftliches Rechnen

*»High Technology is essentially mathematical technology.«
– Enquete-Kommission der Amerikanischen Akademie der
Wissenschaften*

In diesem Abschnitt werden wir Ihnen Module vorstellen, mit deren Hilfe Python zur numerischen Lösung von mathematischen Problemen verwendet werden kann. Die Möglichkeiten sind mit Umgebungen wie *MATLAB* oder *Scilab* vergleichbar.

Mithilfe der hier vorgestellten Module `numpy` und `scipy` lassen sich beispielsweise Gleichungen und Optimierungsprobleme lösen, Integrale berechnen, statistische Berechnungen durchführen oder Differentialgleichungen simulieren. Die Ergebnisse können mit dem Modul `matplotlib` visualisiert werden.

Die Berechnungen werden intern durch hardwarenahe Routinen durchgeführt, sodass sich bei geschickter Programmierung effiziente Programme schreiben lassen.

In diesem Kapitel wird Ihnen keine erschöpfende Erklärung der mannigfaltigen Fähigkeiten von `numpy`, `scipy` und `matplotlib` präsentiert, sondern Sie werden anhand eines einfachen Beispiels an den prinzipiellen Umgang mit den Modulen herangeführt. Außerdem finden Sie am Schluss des Kapitels eine Übersicht über ihren strukturellen Aufbau von `scipy` sowie weitere Beispiele für die Verwendung des Datentyps `numpy.ndarray`.

Ziel dieses Kapitels ist es, Ihnen einen Überblick zu geben, ob `numpy`, `scipy` und `matplotlib` für Ihre konkreten Programme nützlich sind. Nach der Lektüre der hier gegebenen Einleitung sind Sie dann in der Lage, gezielt auf die Onlinedokumentationen zurückzugreifen.

Die folgende Tabelle enthält eine kurze Beschreibung der drei Module.

Modul	Beschreibung
numpy	Stellt einen flexiblen Datentyp für mehrdimensionale Arrays zur Verfügung, mit dessen Hilfe umfangreiche Rechnungen effizient durchgeführt werden können. Die Onlinedokumentation finden Sie auf der Seite http://docs.scipy.org/doc/ unter »Numpy Reference Guide«.
scipy	Implementiert aufbauend auf numpy vielfältige mathematische Operationen. Die Onlinedokumentation finden Sie auf der Seite http://docs.scipy.org/doc/ unter »Scipy Reference Guide«.
matplotlib	Visualisiert Daten grafisch. Dabei ist es erklärtes Ziel der Entwickler, hochwertige Grafiken zu erzeugen, die sich beispielsweise in wissenschaftlichen Veröffentlichungen verwenden lassen. Die Onlinedokumentation finden Sie auf der Seite http://matplotlib.sourceforge.net/ unter »docs«.

Tabelle 24.1 Die Module für numerische Berechnungen und Visualisierung

Die Module sind nicht in der Standardbibliothek von Python enthalten und müssen daher separat installiert werden. Wie dies funktioniert, erfahren Sie im folgenden Abschnitt.

24.1 Installation

Die neusten Versionen der Module `numpy` und `scipy` finden Sie auf der offiziellen Homepage unter <http://www.scipy.org/Download/> bzw. auf den zugehörigen Download-Seiten von Sourceforge:

- <http://sourceforge.net/projects/numpy/files/NumPy/>
- <http://sourceforge.net/projects/scipy/files/scipy/>

Dort werden sowohl der Quellcode als auch vorkompilierte Pakete für Windows und Mac OSX angeboten.

Falls Sie unter Linux arbeiten, gibt es mit hoher Wahrscheinlichkeit vorgefertigte Pakete für Ihre Distribution, die `numpy` und `scipy` enthalten. Wir empfehlen Ihnen, auf diese Pakete zurückzugreifen.

Alternativ finden Sie den Quellcode von `numpy` und `scipy` sowie vorkompilierte Pakete für Windows und Mac OSX auf der CD, die diesem Buch beiliegt.

Auf der Seite <http://matplotlib.sourceforge.net/> finden Sie die neueste Version von matplotlib unter »download«.¹

24.2 Das Modellprogramm

Wir betrachten im Folgenden ein einfaches Modellprogramm, das zu einer Funktion $f(x) = x^3 - 10\sin(x) - 4$ die Ableitung und eine Stammfunktion numerisch berechnet und grafisch im Intervall $[-3, 3]$ darstellt. Anhand dieses Modellprogramms werden typische Vorgehensweisen beim Umgang mit `numpy`, `scipy` und `matplotlib` erläutert.

```
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt

import scipy.misc
import scipy.integrate

def f(x):
    return x**3 - 10*np.sin(x) - 4

def df(x):
    return sp.misc.derivative(f, x)

@np.vectorize
def F(x):
    return sp.integrate.quad(f, 0, x)[0]

X = np.linspace(-3, 3, 100)

Y = f(X)
Y1 = df(X)
Y2 = F(X)
```

¹ Zum Zeitpunkt der Drucklegung dieses Buchs gab es noch keine offizielle Version von matplotlib, die mit Python 3 funktioniert. Trotzdem ist es möglich, die aktuelle Entwicklerversion mit Python 3 zu verwenden. Eine Anleitung, wie Sie diese auf Ihrem Rechner installieren können, finden Sie auf der beiliegenden Buch-CD.

Falls Sie warten möchten, bis die offizielle Version von matplotlib zu Python 3 kompatibel ist, können Sie in der Zwischenzeit auf Python 2 umsteigen, da alle Beispiele dieses Kapitels auch unter Python 2 funktionieren.


```
plt.plot(X, Y, linewidth=2, label="f")
plt.plot(X, Y1, linewidth=2, linestyle="dashed", label="f'")
plt.plot(X, Y2, linewidth=2, linestyle="dotted", label="F")
plt.legend()
plt.show()
```

Abbildung 24.1 zeigt die entstehende Grafik.

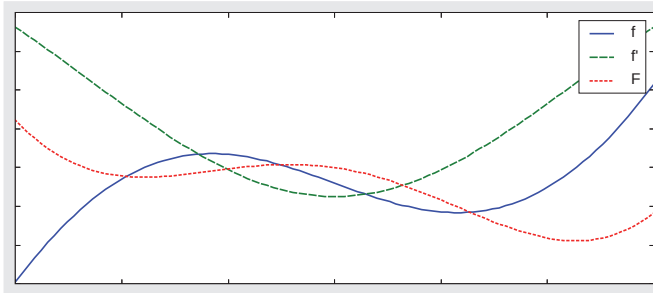


Abbildung 24.1 Die Funktion f mit Ableitung f' und Stammfunktion F

Dieses einfache Beispielprogramm zeigt bereits die wichtigsten Besonderheiten, die man beim Arbeiten mit `numpy`, `scipy` und `matplotlib` beachten sollte. Wir werden das Programm nun stückweise besprechen.

24.2.1 Der Import von `numpy`, `scipy` und `matplotlib`

In den ersten vier Zeilen werden die Module `numpy`, `scipy` und `matplotlib` importiert und mit Kurznamen versehen.

```
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Diese Art, die Module zu importieren, hat sich in der Praxis bewährt und wurde daher als Konvention eingeführt. Wir werden in den Beispielen in diesem Kapitel immer annehmen, dass die Module in dieser Weise importiert wurden, und empfehlen Ihnen, die Konvention auch in Ihren eigenen Programmen zu befolgen.

Im Anschluss an die grundlegenden `import`-Anweisungen werden zusätzlich noch die Module `scipy.misc` und `scipy.integrate` geladen.

Da `scipy` sehr umfangreich ist, ist es nicht sinnvoll, dass jedes Programm alle Funktionen auf einmal importiert. Aus diesem Grund ist `scipy` in mehrere Untermodule unterteilt, die bei Bedarf eingebunden werden müssen. In unserem Beispiel verwenden

den wir die Funktion `derivative` aus dem Untermodul `scipy.misc` zum Berechnen der Ableitung. Die Stammfunktion ermitteln wir mithilfe der Funktion `quad` aus dem Modul `scipy.integrate`. Eine Übersichtstabelle über die Module von `scipy` finden Sie auf Seite 899.

24.2.2 Vektorisierung und der Datentyp `numpy.ndarray`

Nach den `import`-Anweisungen werden drei Funktionen `f`, `df` und `F` definiert, die die Funktion `f`, ihre Ableitung `f'` und eine Stammfunktion `F` auswerten. Für die Arbeit mit `numpy` und `scipy` ist es wichtig, dass die Funktionen `f`, `df` und `F` nicht nur für einzelne Zahlen, sondern für ganze Arrays von Zahlen funktionieren. Ein *Array* ist dabei eine geordnete Folge von gleichartigen Objekten, in der Regel von Zahlen. Bevor wir auf das Zusammenspiel von Arrays und Funktionen eingehen, besprechen wir kurz den elementaren Umgang mit Arrays.²

Um einen Array zu erzeugen, kann die Funktion `numpy.array` verwendet werden, der man eine Liste mit den gewünschten Elementen des Arrays übergibt.

```
>>> a = np.array([0.5, 1, 2, 3])
>>> a
array([ 0.5,  1. ,  2. ,  3. ])
```

Das Besondere an Instanzen des Datentyps `numpy.ndarray` ist, dass mit ihnen wie mit Zahlen gerechnet werden kann. Die Operationen werden dabei elementweise durchgeführt.³

```
>>> a+a
array([ 1.,  2.,  4.,  6.])
>>> a*a
array([ 0.25,  1. ,  4. ,  9. ])
>>> a**3
array([ 0.125,  1. ,  8. ,  27. ])
>>> 4*a
array([ 2.,  4.,  8., 12.] )
```

Neben den Rechenoperationen können auch Funktionen auf Arrays angewendet werden, indem jedes Element des Arrays in die Funktion eingesetzt wird und die Ergebnisse als neuer Array zusammengefasst werden.

² Genauer zum Umgang mit Arrays finden Sie später ab Seite 891.

³ Sind `A` ein zweidimensionaler und `x` ein eindimensionaler Array, so bewirkt `A*x` *keine* Matrix-Vektor-Multiplikation, wie sie in der linearen Algebra üblich ist. Für die Matrix-Vektor-Multiplikation können Sie auf `numpy.dot` zurückgreifen: `np.dot(A,x)`

Beispielsweise können wir die Funktion f aus unserem Beispielprogramm auf den Array a anwenden.

```
>>> f(a)
array([-8.66925539, -11.41470985, -5.09297427, 21.58879992])
```

Hier werden die Werte $f(0.5)$, $f(1)$, $f(2)$ und $f(3)$ berechnet, in einem neuen Array abgelegt und zurückgegeben.

Dieses Prinzip, die gleiche Operation auf einen ganzen Array von Zahlen anzuwenden, nennt man *Vektorisierung*. Wenn eine Funktion außer Zahlen auch Arrays verarbeiten kann, wird sie als *vektoriert* bezeichnet.

Die Funktionen des Moduls `math` der Python-Standardlibrary wie beispielsweise `sin`, `cos`, `tan` oder `exp` sind nicht vektoriert, weshalb `numpy` und `scipy` ihre eigenen vektorisierten Versionen dieser Funktionen mitbringen. In unserem Beispiel verwenden wir daher statt `math.sin` die Funktion `numpy.sin`, die in der Lage ist, für einen Array von Zahlen die Sinuswerte zu berechnen.

Mit dem Wissen über Arrays und Vektorisierung können wir die nächsten beiden Zeilen des Modellprogramms verstehen.

```
X = np.linspace(-3, 3, 100)
Y = f(X)
```

Zunächst generiert die Funktion `numpy.linspace` einen neuen Array X , auf den anschließend die Funktion f elementweise angewendet wird. Die berechneten Funktionswerte werden im Array Y gespeichert.

Mithilfe von `numpy.linspace` kann ein Array mit einer bestimmten Anzahl von Elementen erzeugt werden, die alle den gleichen Abstand haben. Über die ersten beiden Parameter von `numpy.linspace` werden das größte und kleinste Element des Arrays festgelegt. Der folgende Aufruf erzeugt beispielsweise einen Array mit 5 Elementen im Intervall von -1 bis 1.

```
>>> np.linspace(-1,1,5)
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

In unserem Modellprogramm wird also ein Array erzeugt, der 100 Zahlen aus dem Intervall von -3 bis 3 enthält und zu jeder dieser Zahlen den entsprechenden Wert der Funktion f berechnet. Die folgende Zeile `Y1 = df(X)` bestimmt dann die Ableitung von f an den Stellen, die im Array X gespeichert sind. Bei der Definition von `df` nutzen wir aus, dass die Funktion `scipy.misc.derivative` vektoriert ist und daher auch mit einem Array als Parameter funktioniert.

Vektorisieren nicht-vektorisierter Funktionen mittels `numpy.vectorize`

Wenn eine Funktion wie oben `f` oder `df` nur aus vektorisierten Operationen zusammengesetzt ist, ist auch das Ergebnis automatisch wieder vektorisiert. Es gibt allerdings auch Fälle, in denen die Vektorisierung nicht automatisch garantiert wird. Anstatt sich selbst Gedanken darüber zu machen, wie eine Funktion am besten vektorisiert werden kann, bietet `numpy` dazu ein praktisches Hilfsmittel an.

Als Beispiel betrachten wir eine Funktion mit dem Namen `clip_positive`, die den Wert einer Zahl zurückgeben soll, wenn diese positiv ist, und 0, falls die Zahl einen negativen Wert hat. Eine mögliche Implementation dieser Funktion zeigt das folgende Listing.

```
>>> def clip_positive(x):
...     if x > 0: return x
...     else: return 0

>>> clip_positive(10)
10
>>> clip_positive(-5)
0
>>> clip_positive(2)
2
```

Wir möchten dieser Funktion auch Arrays als Wert für `x` übergeben, sodass ein neuer Array erzeugt wird, der für positive Einträge von `x` den Eintrag selbst und für negative Einträge den Wert 0 enthält. So wie die Funktion jetzt implementiert ist, funktioniert sie jedoch nicht für Arrays, da sich die Fallunterscheidung in dieser Form nicht auf Arrays übertragen lässt.

Abhilfe schafft der Function Decorator⁴ `numpy.vectorize`, der eine Funktion so anpasst, dass sie neben einzelnen Zahlen auch Arrays als Parameter akzeptiert.

```
>>> @np.vectorize
... def clip_positive(x):
...     if x > 0: return x
...     else: return 0
>>> clip_positive(10)
array(10)
>>> clip_positive(-5)
array(0.0)
>>> b = np.array([4, -3, 0.7, -10, 8])
>>> clip_positive(b)
array([ 4. ,  0. ,  0.7,  0. ,  8. ])
```

⁴ Näheres über die Verwendung von Function Decorators erfahren Sie auf Seite 373.

Wie Sie sehen, funktioniert die Funktion `clip_positive` nun sowohl mit einzelnen Zahlen als auch mit ganzen Arrays als Parameter. Allerdings wird jetzt auch dann ein Array erzeugt, wenn für den Parameter `x` eine Zahl übergeben wurde. Dies ist deshalb kein Problem, da sich Arrays der Länge 1 wie Zahlen verhalten und daher auch in weiteren Berechnungen wie solche verwendet werden können.

Mit diesem Wissen können wir die Implementation der Funktion `F` in unserem Modellprogramm verstehen. Das folgende Listing wiederholt die Definition aus dem Programm.

```
@np.vectorize
def F(x):
    return sp.integrate.quad(f, 0, x)[0]
```

Die Funktion `scipy.integrate.quad` erwartet als ersten Parameter die zu integrierende Funktion. Mit den beiden folgenden Parametern werden die Integrationsgrenzen angegeben. Der Rückgabewert von `scipy.integrate.quad` ist ein Tupel, bestehend aus zwei Elementen, wobei der erste Eintrag eine Näherung des Integrals und der zweite Eintrag eine obere Schranke für den Approximationsfehler darstellt.

```
>>> sp.integrate.quad(f, 0, 4)
(31.463563791363885, 7.668063971742951e-13)
```

Die Funktion `F` liefert somit für ein `x` das Integral über `f` von 0 bis `x`, also $F(x) = \int_0^x f(t) dt$.

Sie ist damit nach dem Hauptsatz der Differential- und Integralrechnung eine Stammfunktion von `f`. Durch den Function Decorator `numpy.vectorize` wird die Funktion für die Verwendung von Arrays als Parameterwerte erweitert.

Die Zeile `Y2 = F(X)` des Modellprogramms bestimmt die Werte einer Stammfunktion von `f` an den Stellen, die im Array `X` gespeichert sind.



Hinweis

Die vektorisierten Funktionen von `numpy` und `scipy` sind hardwarenah implementiert und dadurch sehr effizient. Wenn Sie eine Funktion mittels `numpy.vectorize` vektorisieren, geht dieser Geschwindigkeitsvorteil verloren.

Sie sollten daher, soweit möglich, versuchen, bei laufzeitkritischen Funktionen auf die schnellen Routinen von `numpy` und `scipy` zurückzugreifen.

24.2.3 Visualisieren von Daten mit `matplotlib.pyplot`

Nachdem nun die gewünschten Daten berechnet wurden, möchten wir sie grafisch darstellen. Wir verwenden dazu das Modul `matplotlib.pyplot`, das an die Zeichenschnittstelle von *MATLAB* angelehnt ist.

Man kann sich diese Schnittstelle wie ein leeres Blatt Papier vorstellen, zu dem durch Aufrufe von Funktionen aus dem Modul `matplotlib.pyplot` neue grafische Elemente hinzugefügt werden. Ist das Zeichnen beendet, wird das Ergebnis mit der Funktion `matplotlib.pyplot.show` auf dem Bildschirm angezeigt.

Die wichtigste Funktion ist `matplotlib.pyplot.plot`, mit der sich beliebige Daten visualisieren lassen. Im einfachsten Fall übergibt man der Funktion `plot` eine Liste von Zahlen.

```
>>> plt.plot([1, 3, 5, 2, 1, 4, 6, 7, 3])
[<matplotlib.lines.Line2D object at 0xb813acc>]
>>> plt.show()
```

Diese Zahlen werden nun als Funktionswerte aufgefasst, wobei die Indizes der Liste, also 0, 1, 2, ..., 8, als zugehörige Werte auf der x-Achse interpretiert werden. Zwischen den angegebenen Funktionswerten wird linear interpoliert, sodass die einzelnen Datenpunkte durch gerade Linien miteinander verbunden werden.

Abbildung 24.2 veranschaulicht das Ergebnis.

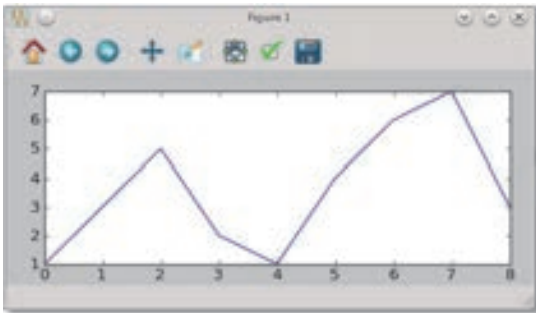


Abbildung 24.2 Ein einfacher Plot einer Liste von Zahlen

Die Werkzeugleiste am oberen Rand des Fensters bietet eine Reihe von Operationen, um die Grafik zu verändern. Außerdem kann die Grafik in verschiedenen Grafikformaten (beispielsweise PNG, PS, EPS, PDF, SVG) gespeichert werden.

In der Regel möchte man die Werte auf der x-Achse, die zu den Funktionswerten gehören, explizit angeben und nicht automatisch über die Indizes ermitteln lassen. Dazu übergibt man der Funktion `plot` zwei Listen oder Arrays, wobei der erste Parameter die x-Werte und der zweite Parameter die zugehörigen y-Werte enthält.

Um beispielsweise eine Periode der Sinusfunktion zu zeichnen, erzeugen wir mit `numpy.linspace` zunächst einen Array, der Datenpunkte im Intervall $[0, 2\pi]$ enthält, und wenden anschließend die Funktion `numpy.sin` darauf an. Die Ergebnisse geben wir dann an `plot` weiter.

```
>>> X = np.linspace(0, 2*np.pi, 200)
>>> Y = np.sin(X)
>>> plt.plot(X, Y)
[<matplotlib.lines.Line2D object at 0xb819eac>]
>>> plt.show()
```

Das Resultat ist in Abbildung 24.3 dargestellt.

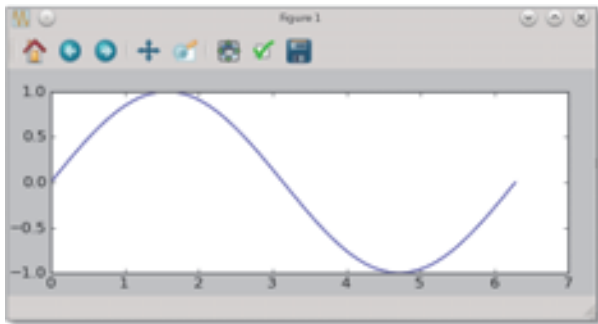


Abbildung 24.3 Plot der Sinusfunktion

Möchte man mehrere Funktionen in einer Grafik darstellen, kann man `plot` mehrmals aufrufen, bevor das Ergebnis mittels `show` angezeigt wird, wie dies auch in unserem Modellprogramm gemacht wird.

Über zusätzliche Parameter der Funktion `plot` kann das Aussehen der Grafik angepasst werden. Das folgende Listing zeigt die Befehle, mit denen in unserem Modellprogramm die grafische Ausgabe erzeugt wird. Dabei verwenden wir den Parameter `linewidth`, um die Dicke der Linien anzupassen, und verändern über `linestyle` das Aussehen der Linien. Mit dem Parameter `label` kann jedem Plot eine Zeichenfolge zugeordnet werden, die in der Legende des Plots angezeigt wird. Der Aufruf der Funktion `legend` sorgt dafür, dass die Legende sichtbar wird.

```
plt.plot(X, Y, linewidth=2, label="f")
plt.plot(X, Y1, linewidth=2, linestyle="dashed", label="f'")
plt.plot(X, Y2, linewidth=2, linestyle="dotted", label="F")
plt.legend()
plt.show()
```

In Abbildung 24.4 wird das Ausgabefenster des Modellprogramms dargestellt.

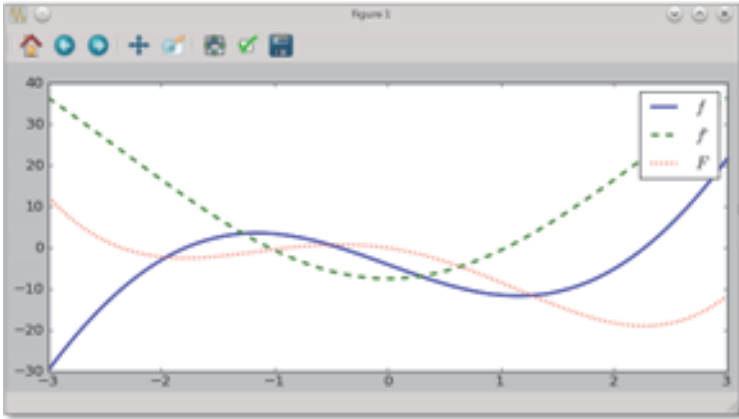


Abbildung 24.4 Endgültige Ausgabe des Modellprogramms

Das Modul `matplotlib` ist in der Lage, komplexe 2D- und 3D-Visualisierungen zu erzeugen. Die verfügbaren Möglichkeiten gehen weit über den hier präsentierten Einstieg hinaus. Wir verweisen Sie für weitere Informationen auf die Onlinedokumentation auf der Seite <http://matplotlib.sourceforge.net/>.

24.3 Überblicke über die Module numpy und scipy

Dieser Abschnitt gibt Ihnen Hinweise zur praktischen Arbeit mit dem Datentyp `numpy.ndarray`. Außerdem wird eine Übersichtstabelle der Untermodule von `scipy` präsentiert.

24.3.1 Überblick über den Datentyp `numpy.ndarray`

Das Herzstück von `numpy` und `scipy` ist der Datentyp `numpy.ndarray`, der mehrdimensionale Arrays verwalten kann. Wir werden Ihnen daher in diesem Kapitel die Grundlagen für den Umgang mit `numpy.ndarray` vermitteln. Diese Einführung ist als erster Einstieg gedacht und deckt nicht alle Möglichkeiten ab, die Ihnen `numpy.ndarray` bietet.

Für weiterführende Informationen verweisen wir Sie auf die Onlinedokumentation von `numpy` auf der Seite <http://docs.scipy.org/doc/> unter »Numpy Reference Guide«.

Die Form und der Datentyp eines Arrays

Ein *Array* ist eine Zusammenfassung mehrerer gleichartiger Elemente zu einer Einheit, wobei auf die Elemente über ganzzahlige Indizes zugegriffen werden kann. Wir werden uns in den Beispielen dieses Kapitels aus Gründen der Übersicht auf ein- und zweidimensionale Arrays beschränken.

Jeder Array hat eine *Form* und einen *Datentyp*. Dabei legt die Form die Dimensionen des Arrays fest, und der Datentyp gibt an, welche Art von Daten die Elemente des Arrays speichern können. Sowohl die Form als auch der Datentyp werden beim Erzeugen des Arrays festgelegt und können anschließend nicht mehr verändert werden. Die folgende Tabelle listet die wichtigsten verfügbaren Datentypen für Elemente von `numpy.ndarray`-Instanzen auf.⁵

Name	Beschreibung der möglichen Werte
<code>numpy.bool</code>	boolesche Werte (True oder False), die intern jeweils als ein Byte gespeichert werden
<code>numpy.int</code>	ganze Zahlen, wobei je nach Plattform 32 oder 64 Bits zur Speicherung verwendet werden
<code>numpy.float</code>	Gleitkommazahlen doppelter Genauigkeit, für die 64 Bits Speicherplatz verwendet werden
<code>numpy.complex</code>	komplexe Zahlen, wobei sowohl Real- als auch Imaginärteil als 64-Bit-Gleitkommazahl gespeichert werden

Tabelle 24.2 Übersicht über die wichtigsten Datentypen von `numpy`.

Neben diesen Basistypen gibt es noch eine Reihe weiterer Typen, die sich genauso verhalten wie die Basisdatentypen, allerdings eine andere Menge an Speicher belegen. So bezeichnet `int8` beispielsweise einen Typ, der ganze Zahlen mit 8 Bits speichern kann. Für Details zu diesen Typen verweisen wir Sie auf die Onlinedokumentation von `numpy`.

Bei der Erzeugung von Arrays sollten Sie immer darauf achten, den richtigen Typ anzugeben, da ansonsten Ungenauigkeiten auftreten können, wenn der gewählte Datentyp nicht den Wertebereich abbilden kann, mit dem Sie arbeiten möchten.

Erzeugen von Arrays und grundlegende Eigenschaften

Im Modellprogramm des letzten Abschnitts wurden bereits eindimensionale Arrays mithilfe der Funktionen `numpy.array` und `numpy.linspace` erzeugt, wie es das folgende Beispiel zeigt.

```
>>> np.array([-1, 4, -5, 7])
array([-1,  4, -5,  7])
>>> np.linspace(0, 1, 11)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

⁵ Die Datentypen heißen genauso wie die eingebauten numerischen Datentypen von Python, da sie ähnliche Werte speichern können. Es handelt sich dennoch um besondere Datentypen für den Umgang mit `numpy`.

Jeder Array besitzt ein Attribut `dtype`, das den Datentyp des Arrays speichert.

```
>>> x = np.array([-1, 4, -5, 7])
>>> x.dtype
dtype('int32')
>>> y = np.array([-1, 4, -5.2, 7])
>>> y.dtype
dtype('float64')
```

Wie Sie sehen, hängt der gewählte Datentyp davon ab, welche Werte der Funktion `numpy.array` übergeben wurden. Dabei wählt `numpy` immer einen möglichst einfachen Typ, der alle übergebenen Werte speichern kann. Dies kann eine ungewollte Einschränkung sein, wenn ein Array nach der Erzeugung andere Werte speichern soll. Beispielsweise kann der Array `x` des obigen Beispiels keine Gleitkommazahlen speichern.

Um den Datentyp eines Arrays explizit festzulegen, kann man den meisten Funktionen zur Erzeugung von Arrays einen Schlüsselwortparameter `dtype` übergeben.

```
>>> x = np.array([-1,4,-5,7], dtype=np.float)
>>> x
array([-1.,  4., -5.,  7.])
>>> x.dtype
dtype('float64')
```

Die Form als zweite grundlegende Eigenschaft eines Arrays wird in dem Attribut `shape` gespeichert.

```
>>> x.shape
(4,)
```

Das Attribut `shape` speichert ein Tupel, das für jede Dimension des Arrays die Anzahl der Komponenten speichert. Im Beispiel hat der Array `x` demnach eine Dimension mit einer Komponente.

Um einen Array mit mehreren Dimensionen zu erzeugen, kann man der Funktion `numpy.array` eine verschachtelte Struktur von iterierbaren Objekten übergeben. Im einfachsten Fall ist dies eine Liste, die weitere Listen enthält.

```
>>> b = np.array([[1,2], [3,4], [5,6]])
>>> b
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
    [5, 6]])  
>>> b.shape  
(3, 2)
```

In diesem Beispiel wird ein Array mit zwei Dimensionen, also eine Matrix, erzeugt, die zeilenweise die Elemente der übergebenen Listen beinhaltet.

Es gibt eine Reihe spezieller Funktionen, mit denen Arrays einer bestimmten Gestalt erzeugt werden können. Die folgende Tabelle gibt eine kurze Übersicht über einige dieser Funktionen.

Name	Beschreibung
<i>array(object)</i>	Erzeugt einen Array aus den von <i>object</i> referenzierten Daten. Häufig ist <i>object</i> eine Referenz auf eine verschachtelte Liste.
<i>linspace(start, stop, num)</i>	Erzeugt einen Array, der <i>num</i> äquidistante Datenpunkte von <i>start</i> bis <i>stop</i> (jeweils inklusive) enthält.
<i>empty(shape)</i>	Erzeugt einen Array mit den Ausmaßen <i>shape</i> . Der Parameter <i>shape</i> ist dabei ein Tupel, das für jede Dimension die Anzahl der gewünschten Komponenten enthält. Die Werte des Arrays werden nicht initialisiert, weshalb sie beliebige Zahlen enthalten können.
<i>eye(N[, M[, k]])</i>	Erzeugt einen zweidimensionalen Array mit <i>N</i> Zeilen und <i>M</i> Spalten, der überall mit Ausnahme der Hauptdiagonale Nullen enthält. Auf der Hauptdiagonale wird der Wert 1 gesetzt. Über den Parameter <i>k</i> kann die Diagonale mit dem Wert 1 verschoben werden. Wird der Parameter <i>M</i> nicht angegeben, wird ein Array mit <i>N</i> Zeilen und <i>N</i> Spalten erzeugt.
<i>ones(shape)</i>	Erzeugt einen Array der Form <i>shape</i> , dessen Elemente alle den Wert 1 haben.
<i>zeros(shape)</i>	Erzeugt einen Array der Form <i>shape</i> , dessen Elemente alle den Wert 0 haben.
<i>diag(v[, k])</i>	Erzeugt einen Array, der auf der Hauptdiagonale die Werte der Sequenz <i>v</i> besitzt und dessen sonstige Elemente mit 0 aufgefüllt werden. Über den Parameter <i>k</i> kann die Diagonale verschoben werden, die die Werte von <i>v</i> annehmen soll.

Tabelle 24.3 Einige der Funktionen zum Erzeugen von Arrays

Als Beispiel möchten wir eine 9×9-Matrix erzeugen, die auf Hauptdiagonalen die Werte von 1 bis 9 und auf den beiden Nebendiagonalen den Wert 5 hat. Alle anderen Werte sollen mit 3 belegt sein.

```
>>> M = 3*(np.ones((9,9)) - np.eye(9,9))
>>> M += 2*(np.eye(9,9,1) + np.eye(9,9,-1))
>>> M += np.diag(range(1,10))
```

Dazu erzeugen wir zunächst mit `np.ones` und `np.eye` eine Matrix `M`, die überall bis auf die Hauptdiagonale den Wert 3 hat. Anschließend addieren wir zu den beiden Nebendiagonalen mittels `np.eye` den Wert 2 hinzu, um auf die gewünschte 5 zu kommen. Zum Schluss erzeugen wir mithilfe von `np.diag` eine Diagonalmatrix mit den Werten von 0 bis 9 auf der Hauptdiagonalen und addieren diese zu `M`, um das Endergebnis zu erhalten.

```
>>> M
array([[ 1.,  5.,  3.,  3.,  3.,  3.,  3.,  3.,  3.],
       [ 5.,  2.,  5.,  3.,  3.,  3.,  3.,  3.,  3.],
       [ 3.,  5.,  3.,  5.,  3.,  3.,  3.,  3.,  3.],
       [ 3.,  3.,  5.,  4.,  5.,  3.,  3.,  3.,  3.],
       [ 3.,  3.,  3.,  5.,  5.,  5.,  3.,  3.,  3.],
       [ 3.,  3.,  3.,  3.,  5.,  6.,  5.,  3.,  3.],
       [ 3.,  3.,  3.,  3.,  3.,  5.,  7.,  5.,  3.],
       [ 3.,  3.,  3.,  3.,  3.,  3.,  5.,  8.,  5.],
       [ 3.,  3.,  3.,  3.,  3.,  3.,  3.,  5.,  9.]])
```

Dieses Vorgehen, mithilfe der vorgefertigten Bausteine komplexere Arrays zusammenzubauen, ist typisch für den Umgang mit `numpy.ndarray`.

Zugriff auf die Elemente eines Arrays

Auf die einzelnen Elemente eines Arrays kann wie bei Listen über einen Index zugegriffen werden. Auch das Slicing funktioniert bei eindimensionalen Arrays analog zum Slicing bei Listen.⁶

```
>>> a = np.array([-1,4,-5,7])
>>> a[2]
-5
>>> a[2:]
array([-5,  7])
```

⁶ Es gibt noch weiterführende Slicing-Möglichkeiten für Arrays mit mehr als zwei Dimensionen. Für die Details verweisen wir Sie auf die Onlinedokumentation von `numpy`.

Da `numpy.ndarray` ein *mutable* Datentyp ist, können die Werte eines Arrays auch nach der Erzeugung verändert werden.

```
>>> a[2] = 1337
>>> a
array([ -1,    4, 1337,    7])
>>> a[0:2] = [-20, -20]
>>> a
array([ -20, -20, 1337,    7])
```

Hat ein Array mehrere Dimensionen, so bestehen die Indizes für einzelne Elemente aus mehreren Zahlen.

```
>>> a = np.zeros((5,5))
>>> a[0,1] = 1
>>> a[3,2] = 2
>>> a[1,3] = 3
>>> a
array([[ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  3.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  2.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Auch das Slicing funktioniert mit mehrdimensionalen Arrays, indem für jede Dimension ein Bereich angegeben wird. Auf diese Weise kann man beispielsweise gezielt einzelne Blöcke einer Matrix verändern.

```
>>> a[0:3,0:3] = 5*np.ones((3,3))
>>> a[3:,3:] = 11
>>> a
array([[ 5.,  5.,  5.,  0.,  0.],
       [ 5.,  5.,  5.,  3.,  0.],
       [ 5.,  5.,  5.,  0.,  0.],
       [ 0.,  0.,  2., 11., 11.],
       [ 0.,  0.,  0., 11., 11.]])
```

Dabei kann man einem ganzen Block eine Zahl zuweisen, wie es in der zweiten Zeile des obigen Beispiels gezeigt wird. In diesem Fall wird jedem Eintrag des betreffenden Blocks der Wert 11 zugewiesen.

Genauso ist es möglich, aus einer bestehenden Matrix einen Teil zu extrahieren.

```
>>> b = a[1:4,1:4]
>>> b
array([[ 5.,  5.,  3.],
       [ 5.,  5.,  0.],
       [ 0.,  2., 11.]])
```

Allerdings ist hier Vorsicht geboten, denn die Matrizen *a* und *b* teilen sich dieselben Daten, sodass Änderungen an der Matrix *b* auch die Matrix *a* betreffen und umgekehrt. Um einen Teil einer Matrix zu kopieren, dient die Methode `copy` des Datentyps `numpy.ndarray`. Das folgende Beispiel extrahiert den gleichen Bereich aus *a* wie das Beispiel zuvor, wobei die Daten diesmal kopiert werden.

```
>>> b = a[1:4,1:4].copy()
```

Zum Abschluss dieser Einführung geben wir eine Übersicht über eine Auswahl von Attributen und Methoden des Datentyps `numpy.ndarray`. Dabei ist *a* eine Instanz von `numpy.ndarray`.

Zur Übersicht wurden einige Parameterlisten vereinfacht. Für eine vollständige Liste verweisen wir Sie auf die Onlinedokumentation.

Name	Beschreibung
Attribute	
<i>a.shape</i>	Beschreibt die Form des Arrays <i>a</i> als Tupel.
<i>a.ndim</i>	Gibt die Anzahl der Dimensionen des Arrays <i>a</i> an.
<i>a.size</i>	Gibt die Gesamtanzahl der Elemente des Arrays <i>a</i> an.
<i>a.itemsize</i>	Gibt an, wie viele Bytes jedes einzelne Element des Arrays <i>a</i> im Speicher belegt.
<i>a.dtype</i>	Beschreibt den Datentyp der Elemente des Arrays <i>a</i> .
<i>a.nbytes</i>	Gibt den Speicher in Bytes an, den die Elemente von <i>a</i> insgesamt belegen.
<i>a.real</i>	Gibt einen Array zurück, der die Realteile der Elemente von <i>a</i> als Elemente enthält.
<i>a.imag</i>	Gibt einen Array zurück, der die Imaginärteile der Elemente von <i>a</i> als Elemente enthält.

Tabelle 24.4 Auswahl von Attributen und Methoden des Datentyps `numpy.ndarray`

Name	Beschreibung
Generelle Methoden	
<i>a.tolist()</i>	Erzeugt eine verschachtelte Liste, die die Elemente von <i>a</i> enthält.
<i>a.dump(file)</i>	Schreibt den Array <i>a</i> so in das Dateiobjekt <i>file</i> , dass er anschließend mit dem Modul <code>pickle</code> wieder ausgelesen werden kann. Näheres zum Modul <code>pickle</code> finden Sie auf Seite 595.
<i>a.astype(dtype)</i>	Erzeugt eine Kopie des Arrays <i>a</i> . Die Elemente des resultierenden Arrays haben dabei den Datentyp <i>dtype</i> .
<i>a.copy()</i>	Erzeugt eine Kopie des Arrays <i>a</i> .
<i>a.fill(value)</i>	Setzt alle Elemente von <i>a</i> auf den Wert <i>value</i> .
Methoden für die Größenanpassung	
<i>a.reshape(shape)</i>	Gibt einen Array der Form <i>shape</i> zurück, der die gleichen Daten wie <i>a</i> enthält. Teilt sich das zurückgegebene Array und seine Daten mit <i>a</i> .
<i>a.resize(new_shape)</i>	Verändert die Form von <i>a</i> in-place zu der Form <i>new_shape</i> .
<i>a.transpose()</i>	Gibt einen Array zurück, der die an der Hauptdiagonalen gespiegelten Werte von <i>a</i> enthält. Dabei teilen sich der zurückgegebene Array und <i>a</i> dieselben Daten.
<i>a.flatten()</i>	Kopiert die Elemente des Arrays <i>a</i> in einen Array mit einer Dimension, indem er Daten aus mehreren Dimensionen aneinander hängt.
<i>a.ravel()</i>	wie <i>a.flatten</i> , außer dass keine Kopie erzeugt wird, sodass sich der zurückgegebene Array die Daten mit <i>a</i> teilt
Berechnungsmethoden	
<i>a.argmax()</i>	Gibt den Index eines Elements von <i>a</i> zurück, das den größten Wert in <i>a</i> hat.
<i>a.max</i>	Gibt den größten Wert in <i>a</i> zurück.
<i>a.argmin()</i>	Gibt den Index eines Elements von <i>a</i> zurück, das den kleinsten Wert in <i>a</i> hat.
<i>a.min()</i>	Gibt den kleinsten Wert in <i>a</i> zurück.

Tabelle 24.4 Auswahl von Attributen und Methoden des Datentyps `numpy.ndarray` (Forts.)

Name	Beschreibung
<i>a.clip(min, max)</i>	Gibt einen Array zurück, bei dem die Werte von <i>a</i> durch <i>min</i> und <i>max</i> limitiert wurden.
<i>a.conj()</i>	Erzeugt einen neuen Array aus dem Array <i>a</i> , indem alle Elemente komplex konjugiert werden.
<i>a.sum()</i>	Gibt die Summe aller Elemente von <i>a</i> zurück.
<i>a.mean()</i>	Gibt den Mittelwert aller Elemente in <i>a</i> zurück.
<i>a.var()</i>	Berechnet die Varianz aller Elemente von <i>a</i> .
<i>a.std()</i>	Berechnet die Standardabweichung aller Elemente von <i>a</i> .
<i>a.prod()</i>	Gibt das Produkt aller Elemente von <i>a</i> zurück.
<i>a.all()</i>	Gibt <code>True</code> zurück, wenn der Wahrheitswert aller Elemente von <i>a</i> <code>True</code> ist. Ist dies nicht der Fall, wird <code>False</code> zurückgegeben.
<i>a.any()</i>	Gibt <code>True</code> zurück, wenn es in <i>a</i> ein Element gibt, dessen Wahrheitswert <code>True</code> ist. Ansonsten wird <code>False</code> zurückgegeben.

Tabelle 24.4 Auswahl von Attributen und Methoden des Datentyps `numpy.ndarray` (Forts.)

Der geschickte Umgang mit dem Datentyp `numpy.ndarray` erfordert einige Übung. Hat man sich aber einmal an die Besonderheiten der Vektorisierung gewöhnt, kann man sehr kompakte und effiziente Programme damit schreiben.

24.3.2 Überblick über scipy

Das Modul `scipy` bietet eine Reihe von Untermodulen für bestimmte Arten von Berechnungen. Die folgende Tabelle verschafft Ihnen einen Überblick, welche Module vorhanden sind und welche Funktionalitäten sie bereitstellen.

Nähere Informationen zu den angegebenen Modulen können Sie der Onlinedokumentation von `scipy` unter <http://docs.scipy.org/doc/scipy/reference/> entnehmen.

Untermodul	Beschreibung
<code>cluster</code>	Bietet Funktionen zur Clusteranalyse, um Datensätze zu kategorisieren.
<code>constants</code>	Stellt mathematische und physikalische Konstanten bereit

Tabelle 24.5 Untermodule von `scipy`

fftpack	Bietet Funktionen für die diskrete Fourier-Transformation (implementiert mit der Fast Fourier Transform). Außerdem werden Differentialoperatoren und Faltungen bereitgestellt.
integrate	Stellt Funktionen bereit, um Integrale numerisch zu approximieren. Dabei können sowohl Funktionsobjekte als auch Listen mit diskreten Funktionswerten integriert werden. Zusätzlich gibt es Funktionen, mit deren Hilfe gewöhnliche Differentialgleichungen in einer oder mehreren Dimensionen numerisch gelöst werden können.
interpolate	Stellt Funktionen für die Interpolation bereit. Dabei werden unter anderem Polynominterpolation (Lagrange und Taylor) sowie Splines in einer und mehreren Dimensionen unterstützt.
io	Bietet Komfortfunktionen, um Daten aus verschiedenen Formaten zu lesen und zu schreiben. Unterstützt werden Daten der Programme <i>MATLAB</i> und <i>IDL</i> sowie das Format <i>Matrix Market file</i> und <i>Arff</i> -Dateien. Außerdem können Daten im <i>NetCDF</i> -Format sowie <i>WAV</i> -Sounddateien gelesen und geschrieben werden.
linalg	Bietet Funktionen aus der linearen Algebra an. Dazu gehören Funktionen zum Lösen von linearen Gleichungen, zur Berechnung von Eigenwerten und Eigenvektoren sowie Standardzerlegungen von Matrizen (beispielsweise LU, Cholesky, SVD). Außerdem werden Matrizen-Funktionen wie beispielsweise die Matrixexponentialfunktion implementiert und Funktionen zur Erzeugung spezieller Matrizen (beispielsweise die Hilbert-Matrix) angeboten.
misc	Bietet generelle Funktionen an, die in keine der anderen Kategorien passen. Dazu gehören beispielsweise Funktionen zur Berechnung der Fakultät oder der numerischen Differentiation.
ndimage	Stellt Funktionen bereit, mit denen Bilddaten verarbeitet werden können. Es gibt Filter- und Interpolationsfunktionen sowie Funktionen für Morphologie und Analyse.

Tabelle 24.5 Untermodule von scipy (Forts.)

odr	Implementiert Regressionsroutinen für die orthogonale Regression, mit der auch Messfehler in den unabhängigen Variablen ausgeglichen werden können.
optimize	Bietet Funktionen, um Extremwertaufgaben zu lösen und Nullstellen zu finden. Es werden Probleme in einer und mehreren Dimensionen mit und ohne Nebenbedingungen unterstützt.
signal	Stellt Funktionen für die Signalverarbeitung bereit. Dazu gehören beispielsweise Filter- und Faltungsfunktionen sowie B-Splines und Wavelets.
sparse	Bietet Funktionen für den effizienten Umgang mit dünnbesetzten Matrizen.
spatial	Bietet Funktionen, um Bereichsanfragen zu beantworten (k-d-Baum). Außerdem wird eine Klasse für die Delaunay-Triangulierung angeboten.
special	Stellt eine Menge spezieller Funktionen wie beispielsweise die Gamma-Funktion oder Orthogonalpolynome bereit.
stats	Bietet vielfältige statistische Funktionen und Wahrscheinlichkeitsverteilungen.
weave	Ermöglicht es, C/C++-Code in Python-Programme einzubetten.

Tabelle 24.5 Untermodule von scipy (Forts.)

Zusätzlich zu den Modulen, die in `scipy` enthalten sind, gibt es eine Reihe weiterer Module, die aus verschiedenen Gründen nicht zum Standardumfang von `scipy` gehören. Diese Module sind unter der Bezeichnung *SciKits* zusammengefasst und können unter der Internetadresse <http://scikits.appspot.com/> heruntergeladen werden.

Kapitel 25

Anbindung an andere Programmiersprachen

»Wenn die Sprache nicht stimmt, ist das, was gesagt wird, nicht das, was gemeint ist.«
– Konfuzius

Dieser Abschnitt beschäftigt sich mit der Interoperabilität zwischen Python und anderen Programmiersprachen, hier ausschließlich C. Dabei gibt es zwei wesentliche Szenarien:

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 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 wird in Abschnitt 25.3, »Python als eingebettete Scriptsprache«, verfolgt.

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 laufzeitkritische 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 Dateierweiterung *.dll* unter Windows weggelassen werden

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.

muss, wenn eine System-Bibliothek geladen werden soll. Unter Linux sieht 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

Die normale `printf`-Funktion erwartet einen `bytes`-String. 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.

Hinweis

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 der Python-Datentypen `str` oder `bytes` übergeben dürfen. Das liegt daran, dass diese zu den unveränderlichen Datentypen gehören und auch von einer C-Funktion nicht verändert 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. Wie Sie diesen zu einer dynamischen Bibliothek kompilieren, hängt von Ihrem Betriebssystem und Ihrer Build-Umgebung ab und soll nicht

² Die Funktion `printf` schreibt nicht nach `sys.stdout`, sondern in den tatsächlichen `stdout`-Stream des Betriebssystems. 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.

Gegenstand dieses Buchs sein. Sie finden dazu nach einer kurzen Recherche im Internet genügend Anleitungen.

```
// Berechnet die Fakultät einer ganzen Zahl
```

```
int fakultaet(int n)
{
    int i;
    int ret = 1;
    for(i = 2; i <= n; i++)
        ret *= i;
    return ret;
}
```

```
// Berechnet die Länge eines Vektors im  $\mathbb{R}^3$ 
```

```
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*-Algorithmus³, 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 Beispielprogramme 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`⁴ 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))
120
```

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)
<code>c_char</code>	<code>char</code>	<code>bytes</code> (ein Zeichen)
<code>c_wchar</code>	<code>wchar_t</code>	<code>str</code> (ein Zeichen)
<code>c_byte</code>	<code>char</code>	<code>int</code>

Tabelle 25.1 Datentypen des Moduls `ctypes`

³ Der Bubblesort-Algorithmus ist einer der einfachsten Sortieralgorithmen. Der Name rührt daher, dass die zu sortierenden Elemente wie Luftblasen im Wasser nach oben steigen, bis sie ihre Endposition erreichen. Der Bubblesort-Algorithmus hat quadratische Laufzeit und ist damit nicht besonders effizient.

⁴ 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_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
c_ulong	unsigned long	int
c_longlong	__int64, long long	int
c_ulonglong	unsigned __int64, unsigned long long	int
c_float	float	float
c_double	double	float
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 instanziiert 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(b"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 einen 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 über-

geben 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 erleichtern.

Funktion	Bedeutung	Seite
<i>addressof(obj)</i>	Gibt die Speicheradresse der Instanz <i>obj</i> zurück. Für den Parameter <i>obj</i> muss eine Instanz eines ctypes-Datentyps übergeben werden.	–
<i>byref(obj[, offset])</i>	Erzeugt einen Pointer auf die Instanz <i>obj</i> eines ctypes-Datentyps. Der zurückgegebene Pointer kann einer C-Funktion übergeben werden.	912
<i>cast(obj, type)</i>	Bildet den Cast-Operator von C in Python ab.	912
<i>create_string_buffer(init_or_size[, size])</i>	Erzeugt einen veränderlichen String-Buffer, in den von einer C-Funktion heraus geschrieben werden kann.	912
<i>create_unicode_buffer(init_or_size[, size])</i>	wie <i>create_string_buffer</i> , mit dem Unterschied, dass ein veränderliches Unicode-Array erzeugt wird (Datentyp <code>c_wchar</code>)	–
<i>sizeof(obj_or_type)</i>	Bildet den <code>sizeof</code> -Operator von C auf Python ab.	913
<i>string_at(address[, size])</i>	Gibt den String zurück, der an der Speicheradresse <i>address</i> steht.	913
<i>wstring_at(address[, size])</i>	wie <i>string_at</i> , mit dem Unterschied, dass ein Unicode-String zurückgegeben wird	–

Tabelle 25.2 Funktionen des Moduls ctypes

Die Beispiele verstehen sich in folgendem Kontext:

```
>>> import ctypes
>>> bib = ctypes.CDLL("MSVCRT")
```

ctypes.byref(obj[, offset])

Diese Funktion erzeugt einen Pointer auf die Instanz *obj* eines `ctypes`-Datentyps. Der zurückgegebene Pointer kann einer C-Funktion übergeben werden. Ein für *offset* übergebenes ganzzahliges Offset wird auf den Pointer aufaddiert.

```
>>> x = ctypes.c_int()
>>> bib.scanf(b"%d", ctypes.byref(x))
27
1
>>> x
c_int(27)
```

Im Beispiel wurde die Funktion `scanf` der Standard C Library verwendet, um eine ganze Zahl vom Benutzer einzulesen. Dazu muss ein Pointer auf eine `int`-Variable übergeben werden. Dieser wird über die Funktion `byref` erzeugt. Wie Sie sehen, ist die eingegebene 27 tatsächlich in die Variable `x` geschrieben worden. Bei der 1 handelt es sich um den Rückgabewert von `scanf`.

ctypes.cast(obj, type)

Die Funktion `cast` bildet den Cast-Operator von C in Python ab. Die Funktion gibt eine neue Instanz des `ctypes`-Pointer-Datentyps *type* zurück, die auf die gleiche Speicherstelle verweist wie *obj*.

```
>>> s = ctypes.c_char_p(b"Hallo Welt")
>>> vp = ctypes.cast(s, ctypes.c_void_p)
>>> vp.value
140011580525696
>>> ctypes.cast(vp, ctypes.c_char_p).value
b'Hallo Welt'
```

Hier wurde ein `char`-Pointer in einen `void`-Pointer und dieser wieder zurück in einen `char`-Pointer gecastet.

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. Im Falle eines Strings wird der Buffer ein Zeichen größer gemacht 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.

```
>>> s = ctypes.create_string_buffer(20)
>>> bib.printf(s, b"Die Zahl ist: %d", 12)
16
>>> s.value
b'Die Zahl ist: 12'
```

Im Beispiel wurde ein String-Buffer der Länge 20 erzeugt und mittels *printf* ein formatierter Text hineingeschrieben.

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.sizeof(ctypes.c_int)
4
>>> ctypes.sizeof(ctypes.c_char)
1
>>> ctypes.sizeof(ctypes.c_double(1.7))
8
```

ctypes.string_at(address[, size])

Diese Funktion 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.

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. Jetzt behandeln wir das Schreiben von Python-Modulen in C. 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 Verschiebechiffre, auch »Cäsar-Verschlüsselung« genannt, durchführen soll. Die Funktion soll später folgendermaßen verwendet werden können:

```
>>> chiffre.caesar("HALLOWELT", 13)
'UNYYBJRYG'
```

Dabei entspricht der zweite Parameter der Anzahl Stellen, um die jeder Buchstabe des ersten Parameters verschoben wird.

Im Folgenden werden wir eine Extension schreiben, die das Modul *chiffre* inklusive der Funktion *caesar* für ein Python-Programm bereitstellt. Die Quelldatei der Erweiterung lautet *chiffre.c*. Der Quelltext der Erweiterung 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.⁵

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` enthält:

```
static PyModuleDef ChiffreModule =
{
    PyModuleDef_HEAD_INIT,
    "chiffre",
    "Performs insane encryption operations",
```

⁵ Das Schlüsselwort `static` bedeutet in diesem Fall, dass der Name `chiffre_caesar` nur innerhalb der Quelldatei *chiffre.c* mit der angelegten Funktion verknüpft ist. Dieses Verhalten wird »internal linkage« genannt.


```

-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.

Die Funktion muss `PyInit_chiffre` genannt werden, wobei `chiffre` bei einem anderen Modulnamen entsprechend anzupassen ist.

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`, wird nur dann benötigt, wenn die Funktion `chiffre_caesar` eine Python-Methode implementiert, 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 Standard-encodings 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 ausschließlich 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 `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 große Bekanntheit genießt.⁶ 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

Um eine Extension zu erzeugen, verwenden wir das Paket `distutils`, das die dazu notwendigen Schritte automatisiert. Zum Kompilieren der Extension muss ein C-Compiler auf Ihrem System installiert sein. Unter Linux wird von `distutils` der GCC- und unter Windows der MSVC-Compiler von Visual Studio⁷ verwendet. Sollten Sie ein Windows-System einsetzen und Visual Studio 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]
)
```

⁶ Was aber nicht heißt, dass er besonders sicher wäre.

⁷ Das Visual Studio kann in einer eingeschränkten Version kostenlos von Microsoft bezogen werden.

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 Installationsscript kompiliert, gelinkt und in die Distribution aufgenommen werden sollen.

Jetzt kann das Installationsscript wie gewohnt ausgeführt werden und kompiliert bzw. installiert die Erweiterung automatisch.

Neben dem Schlüsselwortparameter `sources` können bei der Instanziierung der Klasse `Extension` noch weitere Parameter übergeben werden, die in der folgenden Tabelle kurz erläutert werden.

Parameter	Bedeutung
<i>include_dirs</i>	eine Liste von Verzeichnissen, die für das Kompilieren der Erweiterung benötigte Header-Dateien enthalten
<i>define_macros</i>	Eine Liste von Tupeln, über die beim Kompilieren der Erweiterung bestimmte Makros mit bestimmten Werten definiert werden können. Das Tupel muss folgende Struktur haben: ("MAKRONAME", "Wert").
<i>undef_macros</i>	eine Liste von Makronamen, die beim Kompilieren nicht definiert sein sollen
<i>libraries</i>	eine Liste von Bibliotheksnamen, gegen die die Erweiterung gelinkt werden soll
<i>library_dirs</i>	eine Liste von Verzeichnissen, in denen nach den bei <i>libraries</i> angegebenen Bibliotheken gesucht wird

Tabelle 25.3 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 verträgt es sich damit also, wenn man Python-Instanzen in einem C-Programm verwendet?

Grundsätzlich sollten Sie sich von dem C-Idiom verabschieden, im Besitz einer bestimmten Instanz bzw. in diesem Fall einer `PyObject`-Struktur zu sein. Vielmehr können Sie allenfalls im Besitz einer Referenz auf eine Instanz bzw. eines Pointers auf eine `PyObject`-Struktur sein. Damit implementiert die Python API im Grunde das Speichermodell von Python in C. Im Gegensatz zum Speichermodell von Python erhöht bzw. verringert sich der Referenzzähler einer Instanz jedoch nicht automatisch, sondern muss in einer C-Extension explizit mitgeführt werden. Dazu können die Makros `Py_INCREF` und `Py_DECREF` der Python API folgendermaßen verwendet werden:

```
PyObject *string = 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.

Jetzt bleibt 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 soge-

nanntes *Memory Leak*⁸, und Memory Leaks sind kein erstrebenswerter Umstand in einem Programm.

Die zweite Möglichkeit sind sogenannte *geliehene Referenzen* (engl. *borrowed references*). Solche Referenzen gehören Ihnen nicht. Das bedeutet, dass sie nicht von Ihnen freigegeben werden müssen. Ein Beispiel für geliehene Referenzen sind Funktionsparameter. 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 daraufhin möglicherweise auf bereits freigegebene Speicherbereiche zugegriffen wird. 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 kennen gelernt, in C geschriebene Programmteile von einem Python-Programm aus aufzurufen und somit beispielsweise laufzeitkritische Teile in ein C-Programm auszulagern.

In diesem Abschnitt soll der entgegengesetzte Weg beschritten werden: Wir möchten Python-Programme aus einem C/C++-Programm heraus ausführen, Python also als eingebettete Scriptsprache (engl. *embedded script language*) verwenden. Auf diese Weise können wir bestimmte Teile des Programms in Python schreiben, für die Python aufgrund seiner Flexibilität 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:

⁸ Ein Memory Leak ist ein Fehler in einer Anwendung, durch den ein allozierter Speicherbereich von dieser Anwendung nicht mehr verwendet oder freigegeben werden kann, beispielsweise weil alle Pointer auf diesen Bereich überschrieben wurden. Solche Speicherbereiche können erst nach Beendigung der Anwendung vom Betriebssystem freigegeben werden.

```
#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 später auszuführenden Python-Code enthält.

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 *python3.lib* im Unterordner *libs* Ihrer Python-Installation zu finden.⁹ Wenn sowohl das Kompilieren als auch das Linken ohne Probleme vonstatten gegangen sind, werden Sie feststellen, dass das Programm tatsächlich funktioniert:

```
Guten Tag, die Zufallszahl ist: 64
Das war ... Python
```

Das Python-Script läuft bislang völlig autonom, und es können keine Werte zwischen ihm und dem C-Programm ausgetauscht werden. Aber gerade die Interaktion mit dem Hauptprogramm macht die Qualität einer eingebetteten Scriptsprache aus.

⁹ 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 3.2-Bibliothek mit der Option `-lpython3` einbinden.

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 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 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 Sequenzen *a* und *b* übergeben und gibt eine der beiden zurück. Die Entscheidung, welche der beiden Sequenzen zurückgegeben wird, hängt davon ab, welche das in der jeweiligen Ordnungsrelation kleinste Element enthält. 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 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 müssen wir dabei ein geöffnetes Dateiobjekt übergeben.

Das Python-Script, das von diesem C-Programm aufgerufen wird, kann 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.

Funktion	Beschreibung	Seite
<i>PyObject *Py_BuildValue(const char *format, ...)</i>	Erzeugt und initialisiert eine Instanz eines Python-Datentyps.	929
<i>void Py_Finalize()</i>	Deinitialisiert den Interpreter. Diese Funktion sollte beim Embedding des Python-Interpreters aufgerufen werden, wenn der Interpreter nicht mehr benötigt wird.	–
<i>void Py_INCREF(PyObject *o)</i>	Erhöht den Reference Count der Instanz <i>o</i> um 1. Für <i>o</i> darf nicht der NULL-Pointer übergeben werden.	–
<i>void Py_DECREF(PyObject *o)</i>	Verringert den Reference Count der Instanz <i>o</i> um 1. Für <i>o</i> darf nicht der NULL-Pointer übergeben werden.	–
<i>void Py_Initialize()</i>	Initialisiert den Python-Interpreter und muss aufgerufen werden, bevor der Interpreter beim Embedding eingesetzt wird.	–
<i>void Py_XINCREF(PyObject *o)</i>	Wie <i>Py_INCREF</i> , nur für <i>o</i> darf der NULL-Pointer übergeben werden.	–
<i>void Py_XDECREF(PyObject *o)</i>	Wie <i>Py_DECREF</i> , nur für <i>o</i> darf der NULL-Pointer übergeben werden.	–
<i>int PyArg_ParseTuple(PyObject *args, const char *format, ...)</i>	Liest die einer Funktion übergebenen Argumente aus und speichert sie in lokalen Variablen.	930
<i>void PyErr_SetString(PyObject *type, const char *message)</i>	Wirft eine Python-Exception.	931
<i>PyObject *PyImport_ImportModuleEx(char *name, PyObject *globals, PyObject *locals, PyObject *fromlist)</i>	Importiert ein Python-Modul.	931
<i>int PyMem_Free(void *p)</i>	Gibt den allokierten Speicherblock frei, auf den <i>p</i> zeigt.	–
<i>void *PyMem_Malloc(size_t n)</i>	Allokiert einen Speicherbereich der Größe <i>n</i> und gibt einen Pointer auf diesen Bereich zurück.	–

Tabelle 25.4 Funktionen der Python-API

Funktion	Beschreibung	Seite
<i>PyObject *PyObject_CallObject(PyObject *callable_object, PyObject *args)</i>	Ruft das Objekt <i>callable_object</i> mit den Parametern <i>args</i> auf.	931
<i>PyObject *PyObject_GetAttrString(PyObject *o, const char *attr_name)</i>	Gibt eine Referenz auf das Attribut mit dem Namen <i>attr_name</i> der Instanz <i>o</i> zurück. Im Fehlerfall wird NULL zurückgegeben.	–
<i>int PyRun_SimpleFile(FILE *fp, const char *filename)</i>	Führt eine Python-Programmdatei aus.	931
<i>int PyRun_SimpleString(const char *command)</i>	wie <i>PyRun_SimpleFile</i> mit dem Unterschied, dass der auszuführende Code aus dem String <i>command</i> statt aus einer Datei gelesen wird	–
<i>char *PyBytes_AsString(PyObject *o)</i>	Gibt den internen C-Buffer des bytes-Strings <i>o</i> zurück. Sie dürfen nur dann in den zurückgegebenen Buffer schreiben, wenn Sie ihn zuvor selbst erzeugt haben.	–
<i>PyObject *PyBytes_FromStringAndSize(const char *v, Py_ssize_t len)</i>	Erzeugt eine bytes-Instanz mit der Länge <i>len</i> und dem Inhalt des C-Strings <i>v</i> . Die Funktion gibt eine Referenz auf die erzeugte Instanz zurück.	–
<i>void PySys_SetPath(wchar_t *path)</i>	Setzt die Verzeichnisliste <i>sys.path</i> .	932

Tabelle 25.4 Funktionen der Python-API (Forts.)

PyObject *Py_BuildValue(const char *format, ...)

Diese Funktion 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 <i>str</i> aus einem C-String (<i>char *</i>).
"y"	Erzeugt eine Instanz des Python-Datentyps <i>bytes</i> aus einem C-String (<i>char *</i>).

Tabelle 25.5 Mögliche Angaben im Formatstring

Formatstring	Beschreibung
"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>).
"f"	Erzeugt eine Instanz des Python-Datentyps <code>float</code> aus einer C-Gleitkommazahl (<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> . Der Formatstring "[iii]" erzeugt zum Beispiel eine Liste mit drei ganzzahligen Elementen.
"{...}"	Erzeugt eine Instanz des Python-Datentyps <code>dict</code> . Der Formatstring "{i:s,i:s}" erzeugt ein Dictionary mit zwei Schlüssel-Wert-Paaren, die jeweils aus einer ganzen Zahl als Schlüssel und einem String als Wert bestehen.

Tabelle 25.5 Mögliche Angaben im Formatstring (Forts.)

Ein C-String wird stets kopiert, wenn er an die Funktion `Py_BuildValue` übergeben wird, um einen Python-String zu erzeugen. Das bedeutet insbesondere, dass Sie jeden dynamisch allokierten String wieder freigeben müssen, selbst wenn er an die Funktion `Py_BuildValue` übergeben wurde.

int `PyArg_ParseTuple(PyObject *args, const char *format, ...)`

Die Funktion `PyArg_ParseTuple` liest die einer Funktion übergebenen Positionsargumente 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.

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.

Der Reference Count einer Standard-Exception muss nicht erhöht werden, wenn sie an *PyErr_SetString* übergeben wird.

PyObject *PyImport_ImportModuleEx(char *name, PyObject *globals, PyObject *locals, PyObject *fromlist)

Diese C-Funktion lädt ein Python-Modul und ist äquivalent zu einem Aufruf der Built-in Funktion *__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.

Das lokale Programmverzeichnis muss zuerst mittels *PySys_SetPath* einrichtet werden, bevor lokale Module eingebunden werden können.

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 an die Python-Funktion übergeben werden sollen, kann für *args* *NULL* übergeben werden.

Das Ergebnis des Objektaufrufs wird als Rückgabewert zurückgegeben. Bei einem Fehler 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.

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.

Der für *path* übergebene String muss eine Liste von Verzeichnissen sein. Unter Windows wird dabei ein Semikolon als Trennzeichen verwendet und unter Unix-ähnlichen Systemen ein Doppelpunkt.

Kapitel 26

Insiderwissen

»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

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. Das Modul enthält im Wesentlichen die Funktion `open`, die im Folgenden erklärt wird.

`open(url[, new[, autoraise]])`

Diese Funktion 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` der Wert `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.

```
>>> import webbrowser
>>> webbrowser.open("http://www.galileo-press.de", 2)
True
```

26.2 Funktionsschnittstellen vereinfachen – `functools`

Das Modul `functools` enthält Funktionen und Dekoratoren, mit deren Hilfe sich Funktionen und Klassen auf einer abstrakten Ebene modifizieren lassen. In diesem Kapitel werden die Vereinfachung von Funktionsschnittstellen, das Hinzufügen eines Caches und das Vervollständigen der Ordnungsrelation besprochen.

26.2.1 Funktionsschnittstellen vereinfachen

Im Modul `functools` ist die Funktion *partial* enthalten, mit der sich Funktionsschnittstellen vereinfachen lassen. Betrachten Sie dazu die folgende Funktion:

```
def f(a, b, c, d):
    print("{} {} {} {}".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.

`partial(func[, *args][, **kwargs])`

Die Funktion *partial* bekommt ein Funktionsobjekt übergeben, dessen Schnittstelle vereinfacht werden soll. Zusätzlich werden die zu fixierenden Positions- und Schlüsselwortparameter übergeben.

Die Funktion *partial* gibt 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 fixierten Parameter automatisch ergänzt.

Dies soll das folgende Beispiel anhand der oben definierten Funktion *f* demonstrieren:

```
>>> def f(a, b, c, d):
...     print("{} {} {} {}".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. Der Funktion `f` wird dieser Parameter gemeinsam mit den drei fixierten Parametern übergeben.

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("{} {} {} {}".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.

26.2.2 Caches

Mithilfe des Dekorators *lru_cache*, der im Modul `functools` enthalten ist, lässt sich eine Funktion mit einem Cache versehen. Ein *Cache* ist ein Speicher, der vergangene Funktionsaufrufe sichert. Wenn eine Parameterbelegung beim Funktionsaufruf bereits vorgekommen ist, kann das Ergebnis aus dem Cache gelesen werden, und die Funktion muss nicht noch einmal ausgeführt werden. Dieses Prinzip kann besonders bei rechenintensiven und häufig aufgerufenen Funktionen einen großen Laufzeitvorteil bringen.



Hinweis

Wenn ein Funktionsergebnis aus dem Cache gelesen wird, wird die Funktion nicht ausgeführt. Das Cachen ergibt also nur Sinn, wenn die Funktion frei von Seiteneffekten und deterministisch ist, also das Ergebnis bei der gleichen Parameterbelegung stets dasselbe ist.

@lru_cache([maxsize])

Der Dekorator `lru_cache` versteht eine Funktion mit einem LRU Cache mit *maxsize* Einträgen. Bei einem *LRU Cache* (für *least recently used*) verdrängt ein neuer Eintrag stets den am längsten nicht mehr aufgetretenen Eintrag, sofern der Cache vollständig gefüllt ist. Wenn für *maxsize* der Wert `None` übergeben wird, hat der Cache keine Maximalgröße und kann unbegrenzt wachsen.

Im folgenden Beispiel wird die Funktion `fak` zur Berechnung der Fakultät einer ganzen Zahl definiert und mit einem Cache versehen:

```
>>> import functools
>>> @functools.lru_cache(20)
... def fak(n):
...     res = 1
...     for i in range(2, n+1):
...         res *= i
...     return res
...
>>> [fak(x) for x in [7, 5, 12, 3, 5, 7, 3]]
[5040, 120, 479001600, 6, 120, 5040, 6]
```

Mithilfe der Methode `cache_info`, die der Dekorator `lru_cache` dem dekorierten Funktionsobjekt hinzufügt, lassen sich Informationen über den aktuellen Status des Caches erhalten:

```
>>> fak.cache_info()
CacheInfo(hits=3, misses=4, maxsize=20, currsize=4)
```

Das Ergebnis ist ein benanntes Tupel mit den folgenden Einträgen:

Eintrag	Beschreibung
hits	die Anzahl der Funktionsaufrufe, deren Ergebnisse aus dem Cache gelesen wurden

Tabelle 26.1 Einträge im `CacheInfo`-Tupel

Eintrag	Beschreibung
misses	die Anzahl der Funktionsaufrufe, deren Ergebnisse nicht aus dem Cache gelesen wurden
maxsize	die maximale Größe des Caches
currsize	die aktuelle Größe des Caches

Tabelle 26.1 Einträge im CacheInfo-Tupel (Forts.)

Zusätzlich zu `cache_info` verfügt ein mit `lru_cache` dekoriertes Funktionsobjekt über die Methode `cache_clear`, die den Cache leert.

Hinweis

Intern wird der Cache in Form eines Dictionarys realisiert, bei dem die Parameterbelegung eines Funktionsaufrufs als Schlüssel verwendet wird. Aus diesem Grund dürfen nur Instanzen von hashbaren Datentypen an ein Funktionsobjekt übergeben werden, das den hier vorgestellten LRU Cache verwendet.



26.2.3 Ordnungsrelationen vervollständigen

Eine Klasse, auf der eine Ordnungsrelation definiert sein soll, für die also die Vergleichsoperatoren `<`, `<=`, `>`, `>=` funktionieren sollen, muss jede der entsprechenden magischen Methoden `__lt__`, `__le__`, `__gt__` und `__ge__` implementieren, obwohl eine dieser Methoden bereits ausreichen würde, um die Ordnungsrelation zu beschreiben.

Der Dekorator `total_ordering`, der im Modul `functools` enthalten ist, erweitert eine Klasse, die nur eine der oben genannten magischen Methoden und zusätzlich die Methode `__eq__` bereitstellt, um die jeweils anderen Vergleichsmethoden. Das folgende Beispiel demonstriert die Verwendung des Dekorators:

```
>>> @functools.total_ordering
... class MeinString(str):
...     def __eq__(self, other):
...         return max(self) == max(other)
...
...     def __lt__(self, other):
...         return max(self) < max(other)
...
>>> MeinString("Hallo") > MeinString("Welt")
```

```
False
>>> MeinString("Hallo") <= MeinString("Welt")
True
```

Die Klasse `MeinString` erbt von dem eingebauten Datentyp `str`. Instanzen von `MeinString` sollen anhand des größten enthaltenen Buchstabens miteinander verglichen werden. Dazu sind die Methoden `__eq__` für den Gleichheitsoperator und `__lt__` für den Kleiner-Operator implementiert. Da die Klasse mit `total_ordering` dekoriert wurde, können auch die nicht explizit implementierten Vergleichsoperatoren verwendet werden.

26.3 Weitere Datentypen – collections

Das Modul `collections` der Standardbibliothek enthält Datentypen und Funktionen, die auf den komplexeren Basisdatentypen aufbauen und diese für bestimmte Einsatzzwecke besser nutzbar machen. Die folgende Tabelle listet den Inhalt des `collections`-Moduls auf.

Funktion bzw. Datentyp	Beschreibung	Seite
<i>Counter</i>	ein Dictionary zum Erfassen von Häufigkeiten	938
<i>defaultdict</i>	ein Dictionary, das einen Standardwert für nicht enthaltene Schlüssel unterstützt	941
<i>deque</i>	eine doppelt verkettete Liste	942
<i>namedtuple</i>	eine Funktion zur Erzeugung von benannten Tupeln	944
<i>OrderedDict</i>	ein Dictionary, das die Reihenfolge, in der die Schlüssel-Wert-Paare eingefügt wurden, beim Iterieren beibehält	945

Tabelle 26.2 Inhalt des `collections`-Moduls

26.3.1 Zählen von Häufigkeiten

Oft ist man an der Häufigkeitsverteilung der Elemente eines iterierbaren Objekts interessiert, beispielsweise daran, wie oft einzelne Buchstaben in einem String auftreten.¹ Dieses Problem lässt sich elegant mit einem Dictionary lösen:

1 Über so eine Häufigkeitsanalyse lässt sich beispielsweise das Cäsar-Verschlüsselungsverfahren brechen. Dabei macht man es sich zunutze, dass die Buchstaben in natürlichsprachigem Text nicht gleichmäßig verteilt sind. So ist das `e` der in der deutschen Sprache am häufigsten vorkommende Buchstabe.

```
>>> t = "Dies ist der Text"
>>> d = {}
>>> for c in t:
...     if c in d:
...         d[c] += 1
...     else:
...         d[c] = 1
...
>>> d
{' ': 3, 'e': 3, 'D': 1, 'i': 2, 'T': 1, 's': 2, 'r': 1, 't': 2, 'x': 1, 'd': 1}
```

Der Datentyp `Counter` des Moduls `collections` ist ein Dictionary, das bei einem Zugriff mit einem unbekannten Schlüssel `k` automatisch ein Schlüssel-Wert-Paar `k : 0` hinzufügt. Mit diesem Datentyp lässt sich der obige Code vereinfachen:

```
>>> t = "Dies ist der Text"
>>> d = collections.Counter()
>>> for c in t:
...     d[c] += 1
...
>>> d
Counter({' ': 3, 'e': 3, 'i': 2, 's': 2, 't': 2, 'D': 1, 'T': 1, 'r': 1, 'x': 1, 'd': 1})
```

Im Beispiel liegen die Daten bereits in Form eines iterierbaren Objekts vor. In einem solchen Fall kann dieses Objekt bei der Instanziierung von `Counter` übergeben werden. Damit wird das Beispiel zu einem Einzeiler:

```
>>> collections.Counter("Dies ist der Text")
Counter({' ': 3, 'e': 3, 'i': 2, 's': 2, 't': 2, 'D': 1, 'T': 1, 'r': 1, 'x': 1, 'd': 1})
```

Der Datentyp `Counter` stellt zusätzlich zu der Funktionalität eines Dictionarys einige Methoden bereit, die im Folgenden besprochen werden. Die Beispiele verstehen sich im Kontext der im vorherigen Beispiel angelegten `Counter`-Instanz `d`.

d.elements()

Diese Methode gibt einen Iterator über die Elemente einer `Counter`-Instanz zurück. Dabei wird jedes Element so oft durchlaufen, wie sein aktueller Zählerstand ist.


```
>>> list(d.elements())
[' ', ' ', ' ', 'e', 'e', 'e', 'D', 'i', 'i', 'T', 's', 's', 'r', 't', 't',
'x', 'd']
```

d.most_common([n])

Diese Methode gibt eine Liste der n häufigsten Elemente zurück. Die Liste besteht dabei aus Tupeln, die das jeweilige Element und dessen Häufigkeit enthalten.

```
>>> d.most_common(3)
[(' ', 3), ('e', 3), ('i', 2)]
```

Wenn der Parameter n nicht angegeben wird, enthält die zurückgegebene Liste alle Elemente.

d.subtract([iterable-or-mapping])

Diese Methode subtrahiert die Häufigkeiten der Elemente von *iterable-or-mapping* von den Häufigkeiten in d . So lassen sich beispielsweise die Zeichen finden, in deren Häufigkeiten sich deutsch- und englischsprachige Texte am besten von einander unterscheiden lassen:

```
>>> import collections
>>> ger = collections.Counter(
...     open("deutsch.txt", "r").read().lower())
>>> eng = collections.Counter(
...     open("englisch.txt", "r").read().lower())
>>> eng.most_common(5)
[('e', 10890), ('a', 8567), ('i', 7879), ('n', 7676),
('t', 7530)]
>>> ger.most_common(5)
[('e', 15234), ('n', 9495), ('i', 7629), ('r', 7275),
('a', 5959)]
>>> eng.subtract(ger)
>>> eng.most_common(5)
[('o', 3633), ('a', 2608), ('t', 1734), ('y', 1504), ('p', 1238)]
```

Zunächst wird der Inhalt zweier gleich großer Dateien, die jeweils einen deutschen und einen englischen Text enthalten, eingelesen und eine Häufigkeitsanalyse mithilfe des `Counter`-Datentyps durchgeführt. Die Texte sind, abgesehen von Umlauten, frei von Sonderzeichen. Mithilfe der *subtract*-Methode werden die deutschen Buchstabenhäufigkeiten von den englischen abgezogen. Anhand des Ergebnisses lässt sich erkennen, dass sich ein englischer Text offenbar gut anhand der absolu-

ten Häufigkeiten der Buchstaben o und a von einem deutschen unterscheiden lässt.²

d.update([iterable-or-mapping])

Die Funktion *update* verhält sich wie *subtract*, mit dem Unterschied, dass die in *iterable-or-mapping* enthaltenen Häufigkeiten nicht subtrahiert, sondern auf die Häufigkeiten von *d* addiert werden.

26.3.2 Dictionarys mit Standardwerten

Der Datentyp `defaultdict` ist eine Verallgemeinerung des im vorherigen Abschnitt besprochenen Datentyps `Counter`. Bei einer `Counter`-Instanz wird beim Zugriff auf einen nicht vorhandenen Schlüssel *k* automatisch das Schlüssel-Wert-Paar `{k : 0}` zum Dictionary hinzugefügt. Eine `defaultdict`-Instanz fügt in diesem Fall das Schlüssel-Wert-Paar `{k : x()}` hinzu, wobei *x* für einen beliebigen Datentyp steht.

Der Wert-Datentyp wird beim Instanzieren übergeben. Das `defaultdict` kann danach wie ein normales Dictionary verwendet werden.

Wenn Sie beispielsweise die Wörter eines Textes nach ihrer Länge gruppieren möchten, können Sie dies unter Verwendung des Basisdatentyps `dict` folgendermaßen erreichen:

```
>>> t = "if for else while elif with not and or try except"
>>> d = {}
>>> for wort in t.split(" "):
...     if len(wort) in d:
...         d[len(wort)].append(wort)
...     else:
...         d[len(wort)] = [wort]
...
>>> d
{2: ['if', 'or'],
 3: ['for', 'not', 'and', 'try'],
 4: ['else', 'elif', 'with'],
 5: ['while'],
 6: ['except']}
```

Mit einem `defaultdict` lässt sich dies einfacher lösen.

² Interessanter als die absolute Häufigkeitsverteilung ist eigentlich die relative Häufigkeit, bei der das *y* sehr signifikant ist. Dieser Buchstabe stellt im Deutschen nur 0,04 % der Zeichen, während es im Englischen 1,974 % sind. Anhand dieses Unterschiedes lassen sich die beiden Sprachen mit einer großen Sicherheit voneinander unterscheiden.

```
>>> import collections
>>> t = "if for else while elif with not and or try except"
>>> d = collections.defaultdict(list)
>>> for wort in t.split(" "):
...     d[len(wort)].append(wort)
...
>>> d
defaultdict(<class 'list'>,
{2: ['if', 'or'],
3: ['for', 'not', 'and', 'try'],
4: ['else', 'elif', 'with'],
5: ['while'],
6: ['except']})
```

26.3.3 Doppelt verkettete Listen

Betrachten wir einmal die beiden folgenden Varianten, eine Liste mit den Zahlen von 0 bis 99 zu füllen. In der ersten Version werden die Zahlen einzeln jeweils ans Ende der Liste angefügt:

```
l = []
for x in range(10000):
    l.append(x)
```

Bei der zweiten Variante werden die Zahlen jeweils an den Anfang der Liste eingefügt:

```
l = []
for x in range(9999, -1, -1):
    l.insert(0, x)
```

Beide Beispiele erzeugen die gleiche Liste, doch es zeigt sich, dass die erste Variante etwa 34-mal schneller läuft als die zweite. Das liegt daran, dass der Basisdatentyp `list` im Hinblick auf den Zugriff über Indizes und das Anfügen neuer Elemente am Ende der Liste optimiert wurde. Beim Einfügen eines Wertes am Anfang einer Liste müssen alle in der Liste enthaltenen Elemente umkopiert, praktisch um eine Stelle nach rechts verschoben werden.

Der Datentyp `deque` (für *double-ended queue*) des Moduls `collections` implementiert eine *doppelt verkettete Liste*, die das effiziente Einfügen von Elementen am Anfang oder am Ende unterstützt. Wenn in den obigen Beispielen eine `deque`-Instanz statt einer Liste verwendet würde, würde sich kein Unterschied in der Laufzeit erkennenbar machen.

Neben den Methoden *append*, *count*, *extend*, *pop*, *remove* und *reverse*, die vom Basisdatentyp *list* her bekannt sind, verfügt der Datentyp *deque* über die folgenden zusätzlichen Methoden.

Methode	Beschreibung
<i>appendleft(x)</i>	Fügt das Element <i>x</i> am Anfang der Liste ein.
<i>clear()</i>	Leert die Liste.
<i>extendleft(iterable)</i>	Fügt die Elemente aus <i>iterable</i> am Anfang der Liste ein.
<i>popleft()</i>	Gibt das erste Element zurück und entfernt es aus der Liste.
<i>rotate(n)</i>	Rotiert die Liste um <i>n</i> Elemente. Das bedeutet, dass die letzten <i>n</i> Elemente der Liste gelöscht und in umgekehrter Reihenfolge am Anfang eingefügt werden.

Tabelle 26.3 Methoden des Datentyps *deque*

Das folgende Beispiel zeigt die Verwendung von *deque*-Instanzen:

```
>>> d = collections.deque([1,2,3,4])
>>> d.appendleft(0)
>>> d.append(5)
>>> d
deque([0, 1, 2, 3, 4, 5])
>>> d.extendleft([-2, -1])
>>> d
deque([-1, -2, 0, 1, 2, 3, 4, 5])
>>> d.rotate(4)
>>> d
deque([2, 3, 4, 5, -1, -2, 0, 1])
```

Darüber hinaus ist es möglich, bei der Instanziierung einer doppelt verketteten Liste eine Maximallänge festzulegen. Wenn in einer maximal langen Liste ein Element auf einer Seite angefügt wird, wird das letzte Element auf der anderen Seite entfernt.

```
>>> d = collections.deque([1,2,3,4], 4)
>>> d
deque([1, 2, 3, 4], maxlen=4)
>>> d.append(5)
>>> d
deque([2, 3, 4, 5], maxlen=4)
```

```
>>> d.appendleft(1)
>>> d
deque([1, 2, 3, 4], maxlen=4)
```

Auf die Maximallänge einer `deque`-Instanz kann über das Attribut `maxlen` zugegriffen werden.



Hinweis

Die Flexibilität, die `deque` beim Anfügen und Entfernen von Elementen bietet, hat seinen Preis. Zwar unterstützt der Datentyp `deque` den Elementzugriff über Indizes, dieser ist im Vergleich zum Basisdatentyp `list` jedoch im Allgemeinen langsam.

26.3.4 Benannte Tupel

Viele Funktionen der Standardbibliothek geben ihr Ergebnis in Form eines benannten Tupels zurück. Das ist ein Tupel, dessen Felder jeweils einen Namen tragen. Auf die Werte des Tupels kann über die Feldnamen zugegriffen werden, weswegen benannte Tupel zu einem lesbareren Code führen. Ein Beispiel für ein benanntes Tupel ist `version_info` im Modul `sys` der Standardbibliothek, das die Version des Python-Interpreters angibt.

```
>>> sys.version_info
sys.version_info(major=3, minor=2, micro=2, releaselevel='final', serial=0)
```

Auf die Felder eines benannten Tupels kann wie auf Attribute zugegriffen werden:

```
>>> sys.version_info.major
3
```

Ein benanntes Tupel kann mithilfe der Funktion `namedtuple` des Moduls `collections` erzeugt werden.

`namedtuple(typename, field_names[, verbose[, rename]])`

Die Funktion `namedtuple` erzeugt einen neuen, vom Basisdatentyp `tuple` abgeleiteten Datentyp mit dem Namen `typename`. Instanzen dieses Datentyps erlauben es, über feste Namen auf die einzelnen Elemente des zugrundeliegenden Tupels zuzugreifen.

Über den Parameter `field_names` werden die Namen der Elemente des Tupels festgelegt. Hier kann entweder eine Liste von Strings übergeben werden oder ein einzelner String, in dem die Namen durch Leerzeichen oder Kommata voneinander getrennt

stehen. Die Namen müssen den Regeln eines Python-Bezeichners folgen und dürfen zusätzlich nicht mit einem Unterstrich beginnen. Die Feldnamen sind nach der Erzeugung des Datentyps in dem statischen Attribut `_fields` gespeichert.

```
>>> Buch = collections.namedtuple("Buch",
                                   ["titel", "autor", "seitenzahl", "ISBN"])
>>> py = Buch("Python 3", "Ernesti und Kaiser",
              788, "978-3-8362-1412-4")
>>> py.autor
'Ernesti und Kaiser'
>>> py[1]
'Ernesti und Kaiser'
>>> Buch._fields
('titel', 'autor', 'seitenzahl', 'ISBN')
```

Bei der Instanziierung eines benannten Tupels, in diesem Fall des Datentyps `Buch`, müssen Werte für alle Felder des Tupels angegeben werden. Danach kann sowohl über einen Index als auch über den zugeordneten Namen auf ein Feld zugegriffen werden.

Wenn für den Parameter *rename* der Wert `True` übergeben wird, werden ungültige Feldnamen akzeptiert. Auf solche Felder kann dann nur noch über ihren Index zugegriffen werden. Der Parameter *rename* ist mit `False` vorbelegt, was im Falle ungültiger Feldnamen eine `ValueError`-Exception auslöst.

Der mit `False` vorbelegte Parameter *verbose* steuert, ob nach dem Erzeugen des Datentyps dessen Klassendefinition ausgegeben wird.

26.3.5 Sortierte Dictionarys

Der Datentyp `OrderedDict` aus dem Modul `collections` implementiert ein Dictionary, das die Reihenfolge der Schlüssel beibehält. Bei dem Basisdatentyp `dict` ist die Reihenfolge der Schlüssel, beispielsweise beim Iterieren über alle Schlüssel-Wert-Paare, willkürlich:

```
>>> d1 = dict([("a", 1), ("b", 2), ("c", 3)])
>>> d2 = collections.OrderedDict([("a", 1), ("b", 2), ("c", 3)]) >>> d1
{'a': 1, 'c': 3, 'b': 2}
>>> d2
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

Der Datentyp `OrderedDict` stellt zusätzlich zur Funktionalität eines normalen Dictionarys die folgenden Methoden bereit.

popitem([last])

Diese Methode entfernt das letzte Schlüssel-Wert-Paar aus dem Dictionary. Dieses Paar wird zurückgegeben. Wenn für den mit `True` vorbelegten Parameter *last* der Wert `False` übergeben wird, wird statt des letzten das erste Schlüssel-Wert-Paar des Dictionarys entfernt und zurückgegeben.

```
>>> d2.popitem()
('c', 3)
OrderedDict([('a', 2), ('b', 2)])
>>> d2.popitem(False)
('a', 2)
>>> d2
OrderedDict([('b', 2)])
```

move_to_end(key[, last])

Die Methode *move_to_end* verschiebt das Schlüssel-Wert-Paar mit dem Schlüssel *key* ans Ende bzw. an den Anfang des Dictionarys. An welche der beiden Stellen das Paar verschoben wird, kann wie bei *popitem* über den Parameter *last* gesteuert werden.

26.4 Interpretieren von Binärdaten – struct

Wenn eine Datei im Binärmodus gelesen wird, kommt der Dateiinhalt im Programm als Bytefolge an. Ein Byte besteht aus acht Bit und repräsentiert somit eine Zahl zwischen 0 und 255. Werte, die nicht in diesen Zahlenraum passen, werden durch eine Folge von Bytes dargestellt. In Abschnitt 8.4 haben wir eine Grafik im Bitmap-Dateiformat binär eingelesen, um die Höhe, die Breite und die Farbtiefe des Bildes zu erhalten:

```
def bytes2int(b):
    res = 0
    for x in b[::-1]:
        res = (res << 8) + x
    return res

with open("bild.bmp", "rb") as f:
    f.seek(18)
    print("Breite:", bytes2int(f.read(4)), "px")
    print("Höhe:", bytes2int(f.read(4)), "px")

    f.seek(2, 1)
    print("Farbtiefe:", bytes2int(f.read(2)), "bpp")
```

Da Bilder eine Kantenlänge von mehr als 255 Pixeln haben können, werden Höhe und Breite des Bildes im Bitmap-Format durch eine Folge von vier Bytes gespeichert. Zur Umrechnung dieser Bytefolge in eine Zahl dient die Funktion `bytes2int`, die die Bytefolge rückwärts durchläuft, das jeweils aktuelle Zwischenergebnis in seiner Binärdarstellung um acht Bits nach links verschiebt und dann den nächsten Bytewert aufaddiert. Auf diese Weise lassen sich vier 8-Bit-Werte zu einem 32-Bit-Wert zusammenfügen.

Pythons Standardbibliothek enthält das Modul `struct`, das uns das Interpretieren von Bytefolgen abnimmt. Das obige Beispiel lässt sich unter Verwendung des Moduls `struct` folgendermaßen schreiben:

```
import struct
with open("bild.bmp", "rb") as f:
    f.seek(18)
    werte = struct.unpack("iihh", f.read(12))
    print("Breite:", werte[0], "px")
    print("Höhe:", werte[1], "px")
    print("Farbtiefe:", werte[3], "bpp")
```

Zunächst wird die Leseposition wie gehabt auf das 18. Byte gesetzt. Danach lesen wir die folgenden 12 Byte in einem Rutsch und übergeben sie zusammen mit einer Formatbeschreibung an die Funktion `unpack` des Moduls `struct`. Die Funktion `unpack` zerlegt die Bytefolge anhand der Formatbeschreibung und fügt die Einzelwerte zusammen. Das von `unpack` zurückgegebene Tupel enthält die interpretierten Werte.

Die folgende Tabelle listet die wichtigsten Formatangaben zusammen mit der Größe und dem Python-Datentyp auf, in den sie konvertiert werden.

Angabe	Datentyp	Größe	Vorzeichenbehftet
c	bytes	1 Byte ³	–
b	int	1 Byte	Ja
B	int	1 Byte	Nein
?	bool	1 Byte	–
h	int	2 Byte	Ja
H	int	2 Byte	Nein

Tabelle 26.4 Formatangaben bei `unpack`

³ Die Formatangabe `c` steht für ein einzelnes Zeichen, während `s` für einen String von beliebiger Länge steht.

Angabe	Datentyp	Größe	Vorzeichenbehaftet
i	int	4 Byte	Ja
I	int	4 Byte	Nein
l	int	4 Byte	Ja
L	int	4 Byte	Nein
q	int	8 Byte	Ja
Q	int	8 Byte	Nein
f	float	4 Byte	Ja
d	float	8 Byte	Ja
s	bytes	–	–

Tabelle 26.4 Formatangaben bei unpack (Forts.)

Abgesehen von der Funktion *unpack* enthält das Modul *struct* noch einige weitere Funktionen, die im Folgenden kurz beschrieben werden.

pack(fmt, *values)

Diese Funktion ist das Gegenstück zu *unpack*: Sie kodiert die übergebenen Werte anhand der Formatangaben *fmt* in eine Bytefolge.

```
>>> struct.pack("iif", 12, 34, 5.67)
b'\x0c\x00\x00\x00"\x00\x00\x00\xa4p\xb5@'
```

pack_into(fmt, buffer, offset, *values)

Diese Funktion arbeitet wie *pack*, schreibt die kodierten Daten aber an der Stelle *offset* in die bytearray-Instanz *buffer*.

unpack_from(fmt, buffer[, offset])

Diese Funktion arbeitet wie *unpack*, liest die Daten aber erst ab der Stelle *offset* aus der Bytefolge. Mithilfe von *unpack_from* lässt sich im obigen Bitmap-Beispiel der *seek*-Aufruf einsparen.

calcsize(fmt)

Diese Funktion gibt die Größe der Bytefolge zurück, die durch einen Aufruf von *pack* mit der Formatangabe *fmt* entstehen würde.

26.5 Versteckte Passworteingaben – `getpass`

Das Modul `getpass` ist sehr überschaubar und ermöglicht 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([prompt[, stream]])`

Die Funktion `getpass` liest, ähnlich wie etwa `input`, eine Eingabe vom Benutzer ein und gibt diese als String zurück. Der Unterschied zu `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 `prompt` 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'
```

26.6 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 Komman-

dointerpreter. 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 {}.{}.{}".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 {}: {}: {} Uhr".format(z[3], z[4], z[5]))
        return False

    def do_timer(self, prm):
        if prm == "start":
            self.startTime = time.time()
        elif prm == "get":
            print("Es sind {} Sekunden vergangen.".format(
                int(time.time() - self.startTime)))

    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 vier Kommandos unterstützen: `date` zum Ausgeben des aktuellen Datums, `time` zum Ausgeben der aktuellen Uhrzeit, `timer` zum Initialisieren und Auslesen einer Stoppuhr 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`, `do_timer` 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 an dieser Stelle nicht näher erläutert zu werden.

Wichtig ist, dass eine Kommandomethode anhand des Rückgabewertes 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. Der Rückgabewert `False` einiger Methoden ist im obigen Beispiel überflüssig, 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. Wenn der Benutzer ein Fragezeichen oder den Befehl `help`, gefolgt von einem Kommandonamen, eingibt, 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 instanziiert 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 10.10.2011
==> time
Es ist 19:26:50 Uhr
==> time
Es ist 19:26:54 Uhr
==> timer start
==> timer get
Es sind 5 Sekunden vergangen.
==> 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 weitere Möglichkeiten, um das Verhalten der Konsole genau an Ihre Bedürfnisse anzupassen.

26.7 Dateiinterface für Strings – `io.StringIO`

Das Modul `io` der Standardbibliothek enthält die Klasse `StringIO`, die die Schnittstelle eines Dateiobjekts bereitstellt, intern aber auf einem String arbeitet.

Das ist beispielsweise dann nützlich, wenn eine Funktion ein geöffnetes Dateiobjekt als Parameter erwartet, um dort hineinzuschreiben, Sie die geschriebenen Daten aber lieber in Form eines Strings vorliegen haben würden. Hier kann in der Regel eine

StringIO-Instanz übergeben werden, sodass die geschriebenen Daten danach als String weiterverwendet werden können.

```
>>> from io import StringIO
>>> pseudodatei = StringIO()
```

Dem Konstruktor kann optional ein String übergeben werden, der den anfänglichen Inhalt der Datei enthält. Von nun an kann die zurückgegebene Instanz, referenziert durch `pseudodatei`, wie ein Dateiojekt verwendet werden.

```
>>> pseudodatei.write("Hallo Welt")
>>> print(" Hallo Welt", file=pseudodatei)
```

Neben der Funktionalität eines Dateiojekts bietet eine Instanz der Klasse `StringIO` eine zusätzliche Methode namens *getvalue*, durch die auf den internen String zugegriffen werden kann:

```
>>> pseudodatei.getvalue()
'Hallo Welt Hallo Welt\n'
```

Wie ein Dateiojekt auch, sollte eine `StringIO`-Instanz durch Aufruf der Methode `close` geschlossen werden, wenn sie nicht mehr gebraucht wird.

```
>>> pseudodatei.close()
```


Kapitel 27

Von Python 2 nach Python 3

*»Mehr als die Vergangenheit interessiert mich die Zukunft,
denn in ihr gedenke ich zu leben.«*
– Albert Einstein

Die Programmiersprache Python befindet sich in fortlaufender Entwicklung und hat seit der Veröffentlichung der Version 1.0 im Januar 1994 viele Veränderungen erfahren. In diesem Kapitel werden wir die Entwicklung der Sprache Python beschreiben. Dabei beginnen wir mit der Version 2.0, die im Jahr 2000 veröffentlicht wurde. Alle vorherigen Versionen von Python sind heute nur noch von historischem Interesse.

Im Anschluss daran wird der große Sprung auf Version 3.0 im Detail besprochen. Python 2 ist auch heute noch im Einsatz, weswegen Sie sich über die bestehenden Unterschiede im Klaren sein sollten.

Die folgende Tabelle nennt die wichtigen Neuerungen der einzelnen Python-Versionen. Diese Liste ist nicht vollständig, die Änderungen sind zu zahlreich, um sie hier vollständig wiederzugeben. Eine umfassende Auflistung aller Änderungen einer Python-Version finden Sie in der entsprechenden Dokumentation unter dem Titel »What's New in Python x.y«.

Version	Jahr	Wesentliche Neuerungen
2.0	2000	▶ der <code>unicode</code>-Datentyp für Unicode-Strings ▶ List Comprehensions ▶ erweiterte Zuweisungen ▶ Garbage Collection für zyklische Referenzen
2.1	2001	▶ das <code>__future__</code>-Modul ▶ Vergleichsoperatoren können einzeln überladen werden. ▶ Die <code>import</code>-Anweisung arbeitet auch unter Windows und Mac OS case-sensitive.

Tabelle 27.1 Entwicklung von Python

Version	Jahr	Wesentliche Neuerungen
2.2	2001	▶ verschachtelte Namensräume ▶ New-Style Classes erlauben das Erben von eingebauten Datentypen. ▶ Mehrfachvererbung ▶ Properties ▶ das Iterator-Konzept ▶ statische Methoden und Klassenmethoden ▶ automatische Konvertierung der Datentypen <code>long</code> und <code>int</code>
2.3	2003	▶ der <code>set</code>-Datentyp (zunächst noch im <code>sets</code>-Modul, ab 2.4 als eingebauter Datentyp) ▶ Generatorfunktionen ▶ der Datentyp <code>bool</code>
2.4	2004	▶ Generator Expressions ▶ Funktions- und Klassendekoratoren ▶ das <code>decimal</code>-Modul
2.5	2006	▶ Conditional Expressions ▶ die relative <code>import</code>-Anweisung ▶ Vereinheitlichung der <code>try-except-finally</code>-Anweisung. Zuvor konnten <code>except</code>- und <code>finally</code>-Zweige nicht gemeinsam verwendet werden.
2.6	2008	Die Version 2.6 wurde zeitgleich zu 3.0 entwickelt und enthält die Neuerungen aus Version 3.0, die die Abwärtskompatibilität der Sprache nicht beeinträchtigen: ▶ die <code>with</code>-Anweisung ▶ das <code>multiprocessing</code>-Paket ▶ eine neue Syntax zur String-Formatierung ▶ <code>print</code> wird zu einer Funktion. ▶ Literale für Zahlen im Oktal- und Binärsystem
2.7	2010	Die Version 2.7 ist die letzte Version von Python 2 und beinhaltet einige der Neuerungen von Python 3.1 bzw. 3.2: ▶ Dictionary- und Set Comprehensions ▶ das <code>argparse</code>-Modul

Tabelle 27.1 Entwicklung von Python (Forts.)

Version	Jahr	Wesentliche Neuerungen
3.0	2008	▶ Änderungen, die nicht auch in Version 2.6 enthalten sind: ▶ Viele eingebaute Funktionen und Methoden geben jetzt Iteratoren statt Listen zurück. ▶ Der Datentyp <code>long</code> verschwindet. ▶ der <code>bytes</code>-Datentyp für Binärdaten ▶ Der Datentyp <code>str</code> speichert jetzt ausschließlich Unicode-Strings. ▶ Dictionary Comprehensions ▶ reine Schlüsselwortparameter ▶ die <code>nonlocal</code>-Anweisung ▶ ein Literal für den Datentyp <code>set</code> ▶ Viele Module der Standardbibliothek wurden entfernt, umbenannt oder verändert.
3.1	2009	▶ der Datentyp <code>OrderedDict</code> im <code>collections</code>-Modul ▶ die Methode <code>bit_length</code> des Datentyps <code>int</code> ▶ die automatische Nummerierung von Platzhaltern bei der String-Formatierung ▶ ein neuer Algorithmus zur internen Repräsentation von <code>float</code>-Werten
3.2	2011	▶ Die <code>*.pyc</code>-Kompilate werden jetzt in einem Unterverzeichnis namens <code>__pycache__</code> gesammelt. ▶ Viele Module der Standardbibliothek wurden erweitert.

Tabelle 27.1 Entwicklung von Python (Forts.)

Im Folgenden wird der einschneidende Versionssprung von Python 2 auf Python 3 besprochen. Das Besondere an der Version 3.0 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 umzu-

steigen. Aus diesem Grund besprechen wir in diesem Kapitel übersichtlich die wichtigsten Unterschiede zwischen Python 2 und 3.

Python 3 beinhaltet ein Tool namens 2to3, das das Umstellen eines größeren Projekts 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 und 3 aufgeführt.



Hinweis

Mit Python 3 wurden auch viele subtile Änderungen, beispielsweise an den Schnittstellen vieler Module der Standardbibliothek, vorgenommen, die hier naheliegenderweise nicht alle erläutert werden können. Antworten auf solch detaillierte Fragen finden Sie in den Online-Dokumentationen auf der Python-Website <http://www.python.org>.

27.1.1 Ein-/Ausgabe

In puncto Ein-/Ausgabe gab es zwei auffällige, aber schnell erklärte Änderungen:

- Das Schlüsselwort `print` aus Python 2 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	Python 3
<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 27.2 `print` ist jetzt eine Funktion.

Nähere Informationen, insbesondere zu den Schlüsselwortparametern der Built-in Function `print`, finden Sie in Abschnitt 9.7 (S. 248).

- Die zweite auffällige Änderung bezüglich Ein-/Ausgabe betrifft die Built-in Function `input`. Die `input`-Funktion aus Python 3 entspricht der `raw_input`-Funktion aus Python 2. Eine Entsprechung für die `input`-Funktion aus Python 2 gibt es in Python 3 als Built-in Function nicht, doch ihr Verhalten kann mithilfe von `eval` nachgebildet werden:

Python 2	Python 3
<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 27.3 `input` vs. `raw_input`

Dies waren die beiden gewöhnungsbedürftigsten Änderungen, was die Bildschirm-ausgabe bzw. Tastatureingabe betrifft.

27.1.2 Iteratoren

Obwohl Python 2 das Iterator-Konzept bereits unterstützt, geben 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 wird diese Liste aber nur in einer `for`-Schleife durchlaufen. Dies kann durch Verwendung von Iteratoren eleganter und speicherschonender durchgeführt werden. Aus diesem Grund geben viele der Funktionen und Methoden, die in Python 2 eine Liste zurückgeben, in Python 3 einen auf die Gegebenheiten zugeschnittenen Iterator zurück. Um diese Objekte in eine Liste zu überführen, können sie einfach der Built-in Function `list` übergeben werden.

Damit verhält sich zum Beispiel die `range`-Funktion aus Python 3 gerade so wie die `xrange`-Funktion aus Python 2. Ein Aufruf von `list(range(...))` in Python 3 ist äquivalent zur `range`-Funktion aus Python 2.

Python 2	Python 3
<pre>>>> xrange(5) xrange(5)</pre>	<pre>>>> range(5) range(0, 5)</pre>
<pre>>>> range(5) [0, 1, 2, 3, 4]</pre>	<pre>>>> list(range(5)) [0, 1, 2, 3, 4]</pre>

Tabelle 27.4 range gibt jetzt einen Iterator zurück.

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 existieren in Python 3 nicht mehr.

27.1.3 Strings

Die wohl grundlegendste Änderung in Python 3 ist die Umdeutung des Datentyps *str*. In Python 2 existieren zwei Datentypen für Strings: *str* und *unicode*. Während ersterer zum Speichern beliebiger Byte-Folgen verwendet werden kann, war letzterer für Unicode-Text zuständig.

In Python 3 ist der Datentyp *str* ausschließlich für Text zuständig und mit dem *unicode*-Datentyp aus Python 2 vergleichbar. Zum Speichern von *byte*-Folgen gibt es in Python 3 die Datentypen *bytes* und *bytearray*, wobei es sich bei *bytes* um einen unveränderlichen und bei *bytearray* um einen veränderlichen Datentyp handelt. In Python 3 existiert weder das *u*-Literal für Unicode-Strings noch der Datentyp *unicode*.

Die Datentypen *bytes* und *str* sind in Python 3 klarer voneinander abgegrenzt, als es bei den Datentypen *str* und *unicode* in Python 2 der Fall ist. 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 und *str* in Python 3 exemplarisch auf:

Python 2	Python 3
<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>

Tabelle 27.5 Strings

Python 2	Python 3
<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 27.5 Strings (Forts.)

Die stärkere Abgrenzung von str und bytes in Python 3 hat Auswirkungen auf die Standardbibliothek. So dürfen Sie beispielsweise zur Netzwerkkommunikation nur 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 also auch unter Betriebssystemen von Interesse, die diese beiden Modi von sich aus gar nicht unterscheiden.

27.1.4 Ganze Zahlen

In Python 2 existieren 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 gibt es nur noch einen solchen Datentyp namens int, der sich aber wie long aus Python 2 verhält. Die Unterscheidung zwischen int und long ist auch in Python 2 für den Programmierer im Wesentlichen schon uninteressant, da die beiden Datentypen automatisch ineinander konvertiert werden.

Eine zweite Änderung erfolgt in Bezug auf die Division ganzer Zahlen. In Python 2 wird in diesem Fall eine ganzzahlige Division (Integer-Division) durchgeführt, das Ergebnis ist also wieder eine ganze Zahl. In Python 3 ist das Ergebnis der Division zweier Ganzzahlen eine Gleitkommazahl. Für die Integer-Division existiert hier der Operator //:

Python 2	Python 3
<pre>>>> 10 / 4 2</pre>	<pre>>>> 10 / 4 2.5</pre>
<pre>>>> 10 // 4 2</pre>	<pre>>>> 10 // 4 2</pre>

Tabelle 27.6 Integer-Division

Python 2	Python 3
<pre>>>> 10.0 / 4 2.5</pre>	<pre>>>> 10.0 / 4 2.5</pre>

Tabelle 27.6 Integer-Division (Forts.)

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	Python 3
<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 27.7 Exception Handling

Dabei ist anzumerken, dass die im obigen Beispiel unter Python 3 für das Werfen der Exception verwendete Syntax unter Python 2 auch funktioniert. Die unter Python 2 angegebene Syntax ist aber mit Python 3 aus der Sprache entfernt worden.

27.1.6 Standardbibliothek

Mit Python 3 wurde auch in der Standardbibliothek gründlich aufgeräumt. Viele Module, die kaum verwendet wurden, sind entfernt worden, andere umbenannt oder mit anderen zu Paketen zusammengefasst. Die folgende Tabelle listet die Module auf, die in Python 3 umbenannt wurden:

Python 2	Python 3
<code>_winreg</code>	<code>winreg</code>
<code>ConfigParser</code>	<code>configparser</code>
<code>Copy_reg</code>	<code>copyreg</code>
<code>cPickle</code>	<code>_pickle</code>
<code>Queue</code>	<code>queue</code>

Tabelle 27.8 Unbenannte Module der Standardbibliothek

Python 2	Python 3
<i>SocketServer</i>	<i>socketserver</i>
<i>markupbase</i>	<i>_markupbase</i>
<i>repr</i>	<i>reprlib</i>
<i>test.test_support</i>	<i>test.support</i>
<i>thread</i>	<i>_thread</i>

Tabelle 27.8 Unbenannte Module der Standardbibliothek (Forts.)

Die meisten der oben aufgeführten Module werden in diesem Buch nicht thematisiert, da sie sehr speziell sind. 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	Module aus Python 2
<i>dbm</i>	<i>anydbm, dbhash, dbm, dumbdbm, gdbm, whichdb</i>
<i>html</i>	<i>HTMLParser, htmlentitydefs</i>
<i>http</i>	<i>httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib</i>
<i>tkinter</i>	abgesehen von <i>turtle</i> alle Module, die etwas mit Tkinter zu tun haben
<i>urllib</i>	<i>urllib, urllib2, urlparse, robotparse</i>
<i>xmlrpc</i>	<i>xmlrpclib, DocXMLRPCServer, SimpleXMLRPCServer</i>

Tabelle 27.9 Verschobene Module der Standardbibliothek

27.1.7 Neue Sprachelemente in Python 3

Mit Python 3 wurden eine Reihe neuer Sprachelemente in die Sprache aufgenommen, die wir in der folgenden Tabelle kurz auflisten möchten. In der zweiten Spalte finden Sie einen Verweis, in welchem Abschnitt dieses Buchs die Neuerung besprochen wird:

Sprachelement	Abschnitt	Seite
Literal für Oktal- und Binärzahlen	7.3.4	96
Literal für Mengen	7.7	185
nonlocal-Anweisung	9.3.3	227
Dict- und Set Comprehensions	12.2	346
with-Anweisung	12.7	368
Function Annotations	12.8	371

Tabelle 27.10 Neue Sprachelemente

27.2 Automatische Konvertierung

Um die Migration von Python 2 nach Python 3 auch bei größeren Projekten zu vereinfachen, gibt es in der Python 3-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-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 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 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 Quellcodedateien 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 Kommandozeilen-Schalter *-w*, die angegebene Quelldatei direkt zu modifizieren. Der ursprüngliche Python 2-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 unter Python 3 lauffähig ist.

Statt einer einzelnen Programmdatei können Sie dem 2to3-Script auch eine 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
-d	--doctests_only	Ist dieser Schalter gesetzt, so werden ausschließlich die in der angegebenen Quelldatei enthaltenen Doctests nach Python 3 konvertiert. Standardmäßig werden Doctests nicht angerührt. Näheres über Doctests erfahren Sie in Abschnitt 20.5.1 (S. 713).
-f FIX	--fix=FIX	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 xrange durch range.
-x NOFIX	--nofix=NOFIX	Das Gegenstück zu -f. Hier bestimmen Sie, welche Fixes nicht angewandt werden dürfen.

Tabelle 27.11 Kommandozeilenoptionen von 2to3

Schalter	Alternativ	Beschreibung
-l	--list-fixes	Durch Setzen dieses Schalters erhalten Sie eine Liste aller verfügbaren Fixes.
-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 <code>Future Import 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>datei-name.py.bak</i> angelegt.
-n	--nobackups	Wenn dieser Schalter gesetzt ist, wird auf das Anlegen der Backup-Datei verzichtet.

Tabelle 27.11 Kommandozeilenoptionen von 2to3 (Forts.)

Anhang

»Alles, was einen Anfang hat, hat auch ein Ende –
und meistens hat das, was ein Ende hat, auch eine Fortsetzung.«
(Sprichwort)

A.1 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.2 Built-in Functions

Python enthält eine Menge Built-in Functions, 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	Seite
<code>__import__</code>	Bindet ein Modul oder Paket ein.	267
<code>abs</code>	Berechnet den Betrag einer Zahl.	235
<code>all</code>	Prüft, ob alle Elemente einer Sequenz <code>True</code> ergeben.	235

Tabelle A.2 Built-in Functions in Python

Built-in Function	Beschreibung	Seite
<i>any</i>	Prüft, ob mindestens ein Element einer Sequenz <code>True</code> ergibt.	235
<i>ascii</i>	Erzeugt einen druckbaren String, der das übergebene Objekt beschreibt. Dabei werden Sonderzeichen maskiert, sodass die Ausgabe nur ASCII-Zeichen enthält.	236
<i>bin</i>	Gibt einen String zurück, der die übergebene Ganzzahl als Binärzahl darstellt.	236
<i>bool</i>	Erzeugt einen booleschen Wert.	236
<i>bytearray</i>	Erzeugt eine neue <code>bytearray</code> -Instanz.	236
<i>bytes</i>	Erzeugt eine neue <code>bytes</code> -Instanz.	237
<i>callable</i>	Gibt an, ob eine Instanz aufrufbar ist.	–
<i>chr</i>	Gibt das Zeichen mit einem bestimmten Unicode-Codepoint zurück.	238
<i>classmethod</i>	Erzeugt eine Klassenmethode.	300
<i>complex</i>	Erzeugt eine komplexe Zahl.	238
<i>delattr</i>	Löscht ein bestimmtes Attribut einer Instanz.	324
<i>dict</i>	Erzeugt ein Dictionary.	239
<i>dir</i>	Gibt eine Liste aller Attribute eines Objekts zurück.	–
<i>divmod</i>	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>	239
<i>enumerate</i>	Gibt einen Aufzählungsiterator für die übergebene Sequenz zurück.	239
<i>eval</i>	Wertet einen Python-Ausdruck aus.	367
<i>exec</i>	Wertet einen Python-Ausdruck aus.	366
<i>filter</i>	Ermöglicht es, bestimmte Elemente einer Liste herauszufiltern.	240
<i>float</i>	Erzeugt eine Gleitkommazahl.	241

Tabelle A.2 Built-in Functions in Python (Forts.)

Built-in Function	Beschreibung	Seite
<i>format</i>	Formatiert einen Wert mit der angegebenen Formatangabe.	241
<i>frozenset</i>	Erzeugt eine unveränderliche Menge.	241
<i>getattr</i>	Gibt ein bestimmtes Attribut einer Instanz zurück.	323
<i>globals</i>	Gibt ein Dictionary mit allen Referenzen des globalen Namensraums zurück.	242
<i>hasattr</i>	Überprüft, ob eine Instanz über ein bestimmtes Attribut verfügt.	322
<i>hash</i>	Gibt den Hash-Wert einer Instanz zurück.	242
<i>help</i>	Startet die eingebaute interaktive Hilfe von Python.	243
<i>hex</i>	Gibt den Hexadezimalwert einer ganzen Zahl in Form eines Strings zurück.	243
<i>id</i>	Gibt die Identität einer Instanz zurück.	243
<i>input</i>	Liest einen String von der Tastatur ein.	243
<i>int</i>	Erzeugt eine ganze Zahl.	244
<i>isinstance</i>	Prüft, ob ein Objekt Instanz einer bestimmten Klasse ist.	325
<i>issubclass</i>	Prüft, ob eine Klasse von einer bestimmten Basis-klasse erbt.	326
<i>iter</i>	Erzeugt ein Iterator-Objekt.	357, 362
<i>len</i>	Gibt die Länge einer bestimmten Instanz zurück.	244
<i>list</i>	Erzeugt eine Liste.	245
<i>locals</i>	Gibt ein Dictionary zurück, das alle Referenzen des lokalen Namensraums enthält.	245
<i>map</i>	Wendet eine Funktion auf jedes Element einer Liste an.	245
<i>max</i>	Gibt das größte Element einer Sequenz zurück.	247
<i>min</i>	Gibt das kleinste Element einer Sequenz zurück.	248
<i>next</i>	Gibt das nächste Element des übergebenen Iterators zurück.	353

Tabelle A.2 Built-in Functions in Python (Forts.)

Built-in Function	Beschreibung	Seite
<i>oct</i>	Gibt den Oktalwert einer ganzen Zahl in Form eines Strings zurück.	248
<i>open</i>	Erzeugt ein Dateiojekt.	204
<i>ord</i>	Gibt den Unicode-Code eines bestimmten Zeichens zurück.	248
<i>pow</i>	Führt eine Potenzoperation durch.	248
<i>print</i>	Gibt die übergebenen Objekte auf dem Bildschirm in anderen Ausgabegeräten aus.	49, 248
<i>property</i>	Erzeugt ein Managed Attribute.	297
<i>range</i>	Erzeugt einen Iterator zurück, mit dem gezählt werden kann.	249, 357
<i>repr</i>	Gibt eine String-Repräsentation einer Instanz zurück.	250
<i>reversed</i>	Erzeugt einen Iterator, der ein iterierbares Objekt rückwärts durchläuft.	251
<i>round</i>	Rundet eine Zahl.	251
<i>set</i>	Erzeugt ein Set.	185, 251
<i>setattr</i>	Setzt ein bestimmtes Attribut einer Instanz auf einen bestimmten Wert.	324
<i>sorted</i>	Sortiert ein iterierbares Objekt.	251
<i>staticmethod</i>	Erzeugt eine statische Methode.	299
<i>str</i>	Erzeugt einen String.	252
<i>sum</i>	Gibt die Summe aller Elemente einer Sequenz zurück.	253
<i>tuple</i>	Erzeugt ein Tupel.	253
<i>type</i>	Gibt den Datentyp einer Instanz zurück.	77, 254
<i>vars</i>	Gibt das Dictionary <code>x.__dict__</code> zurück, wenn eine Instanz <code>x</code> übergeben wird. Ohne Argument ist <i>vars</i> äquivalent zu <i>locals</i> .	–
<i>zip</i>	Fasst mehrere Sequenzen zu Tupeln zusammen, um sie beispielsweise mit einer <code>for</code> -Schleife zu durchlaufen.	254

Tabelle A.2 Built-in Functions in Python (Forts.)

Index

@-Zeichen 374

A

ABC 27
 __abs__ 317
 abs 235
 __add__ 314, 320
 all 235
 Alpha-Blending 860
 __and__ 314
 and 46
 __annotations__ 371
 Anonyme Funktion 230
 Anti-Aliasing 861
 Anweisung 51
 Anweisungskopf 51
 Anweisungskörper 51
 any 235
 argparse 512
 Argument 210, 512
 Arithmetischer Operator 93
 Array 909
 as 258
 ASCII 166
 ascii 236
 assert 377
 Assoziativer Array 174
 atexit 523
 Attribut 273, 278, 552
 Klassenattribut 300
 Property-Attribut 296
 Augmented Assignment 94
 Ausdruck 45
 arithmetischer 45
 boolescher 105
 logischer 45
 Automatisiertes Testen 713

B

Backslash 378
 Basisdatentyp 87
 bool 104
 bytearray 142

Basisdatentypbytes (Forts.)

complex 110
 dict 174
 float 41, 102
 frozenset 185, 195
 int 40, 96
 list 43, 128
 NoneType 92
 set 185, 193
 str 42, 142
 tuple 139
 Baum 554
 Bezeichner 44
 Beziérkurve 863
 Big Endian 504
 Bildschirmausgabe 49
 bin 236
 Binärdistribution 740, 748
 Binärer Operator 313
 Binärsystem 99
 Bindigkeit 89
 Bindings 760
 Bit-Operation 99
 Bitverschiebung 101
 Bitweises ausschließendes ODER 100
 Bitweises Komplement 101
 Bitweises ODER 100
 bitweises UND 100
 Blockkommentar 55
 __bool__ 303
 bool 104, 107, 236
 Boolescher Ausdruck 105
 boolescher Operator 47
 Boolescher Wert 104
 Borrowed Reference 922
 break 66
 Breakpoint 693
 Brush 852
 Bubblesort 907
 Bug 691
 Built-in Function 48, 231, 969
 __import__ 267
 abs 235
 all 235
 any 235
 ascii 236

Built-in Function (Forts.)

<i>bin</i>	236
<i>bool</i>	236
<i>bytearray</i>	236
<i>bytes</i>	237
<i>chr</i>	238
<i>classmethod</i>	300
<i>complex</i>	238
<i>delattr</i>	323
<i>dict</i>	239
<i>divmod</i>	239
<i>enumerate</i>	239
<i>eval</i>	367
<i>exec</i>	365
<i>filter</i>	240
<i>float</i>	241
<i>frozenset</i>	241
<i>getattr</i>	322
<i>globals</i>	242
<i>hasattr</i>	323
<i>hash</i>	242
<i>help</i>	243, 363
<i>hex</i>	243
<i>id</i>	243
<i>input</i>	243
<i>int</i>	244
<i>isinstance</i>	323
<i>issubclass</i>	323
<i>iter</i>	354
<i>len</i>	126, 244
<i>list</i>	245
<i>locals</i>	245
<i>map</i>	245
<i>max</i>	126, 247
<i>min</i>	126, 248
<i>oct</i>	248
<i>open</i>	199, 204
<i>ord</i>	248
<i>pow</i>	248
<i>print</i>	248
<i>property</i>	297
<i>range</i>	71, 249
<i>repr</i>	250
<i>reversed</i>	251
<i>round</i>	251
<i>set</i>	251
<i>setattr</i>	322
<i>sorted</i>	251
<i>staticmethod</i>	299
<i>str</i>	252
<i>sum</i>	253

Built-in Function (Forts.)

<i>tuple</i>	253
<i>type</i>	254
<i>zip</i>	254
Busy Waiting	627
Button	783
byte	115
bytearray	115, 142, 236
Byte-Code	28, 36
<i>__bytes__</i>	303
bytes	142, 237
bz2	550

C

C	903
Cache	935
<i>__call__</i>	303, 375
Call by Reference	222
Call by Value	222
Callstack	343, 722
Canvas	806
case sensitive	45
Checkbox	840
Checkbutton	786
Children	554
chr	238
class	275
classmethod	300
Client	610, 611
Client-Server-System	610, 613
cmath	385
cmd	949
Codepage	166
Codepoint	168
collections	938
Combobox	841
Compiler	28, 36
<i>__complex__</i>	303, 318
complex	238
Comprehension	346, 755
<i>Dict</i>	348
<i>List</i>	346
<i>Set</i>	349
Conditional Expression	63
Container	319
<i>__contains__</i>	320
continue	68
Control	759
copy	519
cProfile	730

Critical Section	533, 631
CSV	599
ctypes	904
cx_Freeze	750

D

Dämon-Thread	545
Date Edit	842
Datei	197
<i>Deskriptor</i>	205
<i>temporäre</i>	604
Dateiähnliches Objekt	483
Dateideskriptor	205
Dateidialog	819
Dateiobjekt	199, 204, 635
Dateizugriffsrechte	486
Datenbank	576
<i>Cursor</i>	580
<i>Join</i>	586
<i>Query</i>	576
<i>relationale</i>	576
<i>Transaktion</i>	582
Datenstrom	197
Datentyp	77, 907
<i>immutable</i>	83, 141
<i>mutable</i>	83
datetime	460
Daylight Saving Time	452
Deadlock	536
<i>_debug_</i>	377
Debugging	691
<i>Breakpoint</i>	693
<i>Post Mortem</i>	694
decimal	399
def	211
<i>_del_</i>	302
del	82, 130, 178
<i>_delattr_</i>	306
delattr	323
Delegate	868
<i>_delitem_</i>	319
Deserialisieren	595
Dezimalsystem	97
Dialog	829, 835, 843
<i>_dict_</i>	306
dict	174, 239
Dict Comprehension	348
Dictionary	174, 321, 941
<i>sortiertes</i>	945
Differenzmenge	190
Distribution	739
distutils	739
<i>_divmod_</i>	314
divmod	239
DLL	904
Docstring	297, 363, 713
doctest	713
Document Object Model	553
Doktorarbeit	124
DOM	553
<i>Child</i>	554
<i>Elternelement</i>	554
<i>Geschwisterelement</i>	554
<i>Kindelement</i>	554
<i>Knoten</i>	553
<i>Node</i>	553
<i>Parent</i>	554
<i>Sibling</i>	554
<i>Wurzel</i>	555
DST	452
Dualsystem	97, 99
Duck-Typing	317
Dynamik Link Library	904
Dynamische Bibliothek	904

E

Echte Teilmenge	188
Eingabeaufforderung	512
Eingebaute Funktion	231
Eingebettete Scriptsprache	922
Einrückung	52
Einrückungstiefe	52
Einwegkodierung	447
ElementTree	570
elif	61
else	62, 66, 338
Elternelement	554
E-Mail	655
<i>Header</i>	672
email	671
Embedded Script Language	922
Encoding-Deklaration	173
<i>_enter_</i>	319
Entry-Widget	789
Entwicklungsumgebung	31
enumerate	239
EOF	198
<i>_eq_</i>	313

Erweiterte Zuweisung	94, 315
Escape-Sequenz	144, 421
ESMTP	655
eval	367
Event	546, 771, 836, 878
EventHandler	836
except	336
Exception	330, 506, 757, 918, 962
Exception Chaining	344
Exception Handling	329
exec	365
__exit__	319
Exit Code	509
Exponent	103
Exponentialschreibweise	103
Extension	740, 744, 914, 926

F

Fakultät	68
Fallunterscheidung	59
False	46, 105
Farbverlauf	860
Fenster	759
Fibonacci-Zahlen	354
File Transfer Protocol	646
filter	240
finally	338
Flag	431
__float__	303, 318
float	102, 241
__floordiv__	314
Font	822
for	70, 346
format	241
Fortschrittsbalken	845
Frame-Objekt	703
from	259
frozenset	185, 195, 241
FTP	646, 655
<i>Modus</i>	647
ftplib	646
Function Annotation	371
Function Decorator	374
functools	934
Funktion	47, 113, 209
<i>anonyme</i>	230
<i>eingebaute</i>	231
<i>lokale</i>	229
<i>rekursive</i>	231

Funktionsaufruf	47, 210, 756
Funktionsiterador	362
Funktionskörper	211
Funktionsname	211
Funktionsparameter	48, 215
Funktionsschnittstelle	210
__future__	380

G

Ganze Zahl	40, 96, 961
Garbage Collection	81
__ge__	313
Geliehene Referenz	922
Generator	350
Generator Expression	352
Geplantes Sprachelement	380
Geschwisterelement	554
__getattr__	306
getattr	322
__getattribute__	306
__getitem__	319
getpass	949
Getter-Methode	295
gettext	442
<i>Sprachkompilat</i>	444
Gleitkommazahl	41, 102, 386, 403
global	227
Globale Referenz	225
Globale Variable	527
Globaler Namensraum	225
globals	242
GNU gettext API	442
Goldener Schnitt	354
Grafische Benutzeroberfläche	759
__gt__	313
Gtk	760
GUI	759
Guido van Rossum	27
gzip	549

H

hasattr	323
__hash__	303
hash	242
hashable	305
Hash-Funktion	446
Hash-Kollision	447

hashlib.....446
 Hash-Wert.....175, 446
 help.....243, 363
 hex.....243
 Hexadezimalsystem.....97
 History.....39
 Hook.....507

I

`__iadd__`.....316, 320
`__iand__`.....316
 id.....243
 Identität.....80
 Identitätenvergleich.....81
 IDLE.....31, 692
 if.....60, 347
`__ifloordiv__`.....316
`__ilshift__`.....316
 Imaginärteil.....110
 IMAP4.....665
 Mailbox.....665
 imaplib.....665
 immutable.....83, 141, 754
`__imod__`.....316
`__import__`.....267
 import.....257, 266, 380
 Absolute import-Anweisung.....265
 Relative import-Anweisung.....265
`__imul__`.....316, 320
 in.....118, 178, 188
 in place.....133
 Index.....121
`__index__`.....304
 inf.....104, 403
`__init__`.....302, 303
 Inlining.....756
 in-place.....121
 input.....243
 inspect.....694
 Installationsscript.....742
 Instanz.....48, 75, 275
 Datentyp.....77
 Identität.....80
 Wert.....78
 Instanziierung.....48, 275
`__int__`.....318
 int.....96, 244
 Integer-Division.....41

Interaktiver Modus.....31, 39
 Internationalisierung.....442
 Interpreter.....28, 37
`__invert__`.....317
 io.StringIO.....952
`__ior__`.....316
 IP-Adresse.....609
`__ipow__`.....316
 IPv6.....620
`__irshift__`.....316
 is.....81
 isinstance.....323
 ISO-Kalender.....465
 issubclass.....323
`__isub__`.....316
`__iter__`.....320, 354
 iter.....354
 Iterator.....353, 363, 959
 Iterator-Protokoll.....70, 354
 Iterierbares Objekt.....70, 353
 itertools.....405
`__itruediv__`.....316
`__ixor__`.....316

J

Join.....586

K

Keyword Argument.....114, 216, 758
 Kindelement.....554
 Klasse.....274
 Klassenattribut.....300
 Klassenmethode.....300
 Knoten.....553
 Kommandozeilen-Interpreter.....949
 Kommandozeilenparameter.....512
 Kommentar.....55
 Kommunikationssocket.....610
 Kompilat.....36
 Komplexe Zahl.....110
 Imaginärteil.....110
 konjugierte.....112
 Realteil.....110
 Konsole.....512
 Konsolenanwendung.....34
 Konstruktor.....277

Kontextmanager	318, 369
Kontrollstruktur	59
<i>Conditional Expression</i>	63
<i>Fallunterscheidung</i>	59
<i>Schleife</i>	64
Konvexes Polygon	813
Koordinatensystem	807, 853
Koordinierte Weltzeit	451
Körperloses Tag	553

L

Label	790
LabelFrame	791
lambda	230
Laufzeitmessung	727
Laufzeitmodell	75
Laufzeitoptimierung	753
Laufzeitverhalten	726
Layout	827
Lazy Evaluation	64, 109
__le__	313
Leichtgewichtprozess	527
__len__	319
len	126, 244
Line Edit	844
list	115, 128, 245
List Comprehension	346, 755
Listbox	792
Liste	43, 128
<i>doppelt verkettete</i>	942
List-Widget	844
Little Endian	504
locals	245
Locking	531, 539
Lock-Objekt	533, 539
Logdatei	706
logging	706
Logging Handler	711
Logischer Ausdruck	45
Logischer Operator	105
<i>Logische Negierung</i>	105
<i>Logisches ODER</i>	106
<i>Logisches UND</i>	105
Lokale Funktion	229
Lokale Referenz	225
Lokaler Namensraum	225
Lokales Modul	260
Lokalisierung	442
Lokalzeit	451
long	96
Lookup	757
__lshift__	314
__lt__	313

M

Magic Attribute	302
__annotations__	371
__dict__	306
__doc__	364
__slots__	306
Magic Line	35
Magic Method	302
__abs__	317
__add__	314, 320
__and__	314
__bytes__	303
__call__	303, 375
__complex__	303, 318
__contains__	320
__del__	302
__delattr__	306
__delitem__	319
__div__	314
__divmod__	314
__enter__	319, 370
__eq__	313
__exit__	319, 370
__float__	303, 318
__floordiv__	314
__ge__	313
__getattr__	306
__getattribute__	306
__getitem__	319, 361
__gt__	313
__hash__	303
__iadd__	316, 320
__iand__	316
__idiv__	316
__ifloordiv__	316
__ilshift__	316
__imod__	316
__imul__	316, 320
__index__	304, 318
__init__	302, 303
__int__	318
__invert__	317
__ior__	316
__ipow__	316

Magic Method (Forts.)	
__irshift__	316
__isub__	316
__iter__	320, 354
__ixor__	316
__le__	313
__len__	319
__lshift__	314
__lt__	313
__mod__	314
__mul__	314, 320
__ne__	313
__neg__	317
__next__	354
__nonzero__	303
__or__	314
__pos__	317
__pow__	314
__radd__	315, 320
__rand__	315
__rdiv__	315
__rdivmod__	315
__repr__	302
__rfloordiv__	315
__rlshift__	315
__rmod__	315
__rmul__	315, 320
__ror__	315
__round__	303, 318
__rpow__	315
__rrshift__	315
__rshift__	314
__rsub__	315
__rxor__	315
__setattr__	306
__setitem__	319
__str__	303, 341
__sub__	314
__xor__	314
Mailbox	665
Main-Event-Loop	835
MANIFEST-Datei	747
Mantis	103
map	245, 755
Mapping	173
Matching	415, 435, 439
Match-Objekt	429, 435
math	385
MATLAB	881
matplotlib	881, 889
max	126, 247
MD5	448
Mehrfachvererbung	294
Member	273
Memory Leak	922
Menge	185
Differenz	190
Schnitt	190
Symmetrische Differenz	191
Vereinigung	189
Menu	796
Menubutton	799
Menüleiste	796
MessageBox	822
Method Table	915
Methode	49, 113, 273
Klassenmethode	300
statische	299
Überschreiben	282
Methodendefinition	276
MFC	759
Migration	
2to3	964
MIME	671
min	126, 248
__mod__	314
Modaler Dialog	843
Modellklasse	863
Model-View-Konzept	828, 863
Modifier	772
Modul	257, 740
lokales	260
Modularisierung	257
__mul__	314, 320
Multicall	687
Multiplexender Server	610, 627
Multitasking	525
mutable	83, 754
N	
Namenskonflikt	261, 264
Namensraum	225, 258
globaler	225
lokaler	225
nan	104, 403
__ne__	313
__neg__	317
Netzwerk-Byte-Order	625
__next__	354
Nichtkonvexes Polygon	813

Nicht-modaler Dialog	843
Node	553
None	92
nonlocal	228
not	46, 105
not in	119, 178, 188
numpy	881
<i>ndarray</i>	885, 891

O

Objekt	269, 273
Objektorientierung	269
oct	248
Oktalsystem	97
open	199, 204
Operand	88
Operator	40, 45, 88, 105
<i>arithmetischer</i>	45, 93
<i>binärer</i>	313
<i>Bit-Operator</i>	99
<i>boolescher</i>	47
<i>logischer</i>	105
<i>unärer</i>	316
<i>Vergleichsoperator</i>	46, 95, 311
Operatorrangfolge	89
Optimierung	753
Option	512
Option Menu	800
__or__	314
or	47
ord	248
Ordnungsrelation	126, 937
os	481
os.path	490
OSI-Schichtenmodell	607
Out-of-Band Data	627

P

Packer	763, 767
Padding	770
Painter	851
Painter Path	863
Paket	262, 740
Paralleler Server	610
Parameter	114, 215
<i>entpacken</i>	221
<i>optional</i>	115, 215

Parent	554
Parser	552
pass	73
Passworтеingabe	949
PDB	691
Pen	851
Pfad	484
pickle	595
PIL	810
Pipe	424
platform	511
Plattformunabhängigkeit	28
Polygon	813, 855
POP3	660
poplib	660
Port	609
__pos__	317
Positional Argument	216
Post-Mortem Debugger	694
__pow__	314
pow	248
pprint	703
Primzahl	537
print	49, 248, 958
Profiler	730
Programmdatei	34
Programmierparadigma	28
Progress Bar	845
property	297
Property-Attribut	296
Protokollebene	607
Prozess	525
PSF-Lizenz	29
Push Button	846
PyGtk	760
PyQt	760, 826
<i>Alpha-Blending</i>	860
<i>Anti-Aliasing</i>	861
<i>Beziérkurve</i>	863
<i>Brush</i>	852
<i>Checkbox</i>	840
<i>Combobox</i>	841
<i>Date Edit</i>	842
<i>Delegate</i>	868
<i>Dialog</i>	829, 835, 843
<i>Event</i>	836, 878
<i>EventHandler</i>	836
<i>Farbverlauf</i>	860
<i>Grafiken</i>	856
<i>Koordinatensystem</i>	853
<i>Layout</i>	827

PyQt (Forts.)	
<i>Line-Edit</i>	844
<i>List-Widget</i>	844
<i>Main-Event-Loop</i>	835
<i>Modellklasse</i>	863
<i>Model-View-Konzept</i>	863
<i>Painter</i>	851
<i>Painter Path</i>	863
<i>Pen</i>	851
<i>Progress Bar</i>	845
<i>Push Button</i>	846
<i>QML</i>	828
<i>Radiobutton</i>	846
<i>Signal</i>	827, 836
<i>Slider</i>	847
<i>Slot</i>	827, 836
<i>Spacer</i>	833
<i>Splitter</i>	832
<i>Text</i>	857
<i>Text Edit</i>	848
<i>Transformation</i>	862
<i>Transparenz</i>	860
<i>UI-Datei</i>	835
<i>Viewklasse</i>	863
<i>Widget</i>	829, 839, 848
<i>Zeichnen</i>	850
PySide	761
Python 2	955
<i>Konvertierung</i>	964
Python API	29, 914
Python Software Foundation	28
Python-Shell	31
Python-Website	31
Q	
QML	828
Qt	760, 826
Qt Designer	830
Quantor	418
<i>genügsamer</i>	423
Quellcodedistribution	739, 746
Quelltext	34
Query	576
Queue	542, 610
R	
__radd__	315, 320
Radiobutton	788, 846
raise	334
__rand__	315
random	394
range	71, 249
Rapid Prototyping	30
Raw String	146
raw_input	959
__rdivmod__	315
re	415
Realteil	110
Reference Count	82, 509, 921
Referenz	75
<i>geliehene</i>	922
<i>globale</i>	225
<i>lokale</i>	225
Referenzzähler	82
Regular Expression Objekt	431
Regulärer Ausdruck	415
<i>Alternative</i>	424
<i>Extension</i>	424
<i>Flag</i>	431
<i>Gruppe</i>	423
<i>Matching</i>	435, 439
<i>Match-Objekt</i>	435
<i>Quantor</i>	418, 423
<i>Searching</i>	438
<i>Sonderzeichen</i>	421
<i>Syntax</i>	416
<i>Zeichenklasse</i>	417, 420
<i>Zeichenliteral</i>	416
Regular-Expression-Objekt	432
Rekursion	231
Rekursionstiefe	231
Rekursive Funktion	231
Relationale Datenbank	576
RE-Objekt	432
__repr__	302
repr	250
Reserviertes Wort	45, 969
return	213
reversed	251
__rfloordiv__	315
__rlshift__	315
__rmod__	315
__rmul__	315, 320
__ror__	315
ROT13	919
__round__	303, 318
round	251
RPM	748
__rpow__	315

<code>__rrshift__</code>	315
<code>__rshift__</code>	314
<code>__rsub__</code>	315
<code>__rtruediv__</code>	315
Rückgabewert	48, 113, 211
<code>__rxor__</code>	315

S

SAX	565
Schlafender Thread	526
Schleife	64, 755
<i>break</i>	66
<i>continue</i>	68
<i>else</i>	66
<i>for</i>	70
<i>while</i>	65
Zählschleife	71
Schleifenkörper	64
Schlüssel-Wert-Paar	174
Schlüsselwort	45, 969
Schlüsselwortparameter	114, 216
<i>reiner</i>	219
Schnittmenge	190
Schnittstelle	909, 934
Schriftart	822
Scilab	881
scipy	881, 899
Scrollbar	801
Searching	415, 438
Seiteneffekt	84, 138, 222, 519
<code>select</code>	627
<code>self</code>	276
Semikolon	379
Serialisieren	595
Serieller Server	610
Server	610
<i>multiplexer</i>	610, 627
<i>paralleler</i>	610
<i>serieller</i>	610
<code>set</code>	185, 193, 251
Set Comprehension	349
<code>__setattr__</code>	306
<code>setattr</code>	322
<code>__setitem__</code>	319
Setter-Methode	295
SHA	448
Shared Object	904
Shebang	35
Shell	512
shutil	495
Sibling	554
Signal	827, 836
Simple API for XML	565
Slicing	123
Slider	847
Slot	827, 836
<code>__slots__</code>	306
SMTP	655
<code>smtplib</code>	655
Socket	609
<i>blockierender</i>	616
<i>Byte-Order</i>	625
<i>IPv6</i>	620
<i>Kommunikationssocket</i>	610
<i>nicht-blockierender</i>	617
<i>Verbindungssocket</i>	610
socketserver	630
Sommerzeit	452
<code>sorted</code>	251
Sortiervverfahren	
<i>stabiles</i>	135
Spacer	833
Speicherzugriffsfehler	922
Spinbox	802
Splitter	832
Sprachkompilat	444
SQL	576
SQL Injection	583
sqlite3	579
Stabiles Sortiervverfahren	135
Standardbibliothek	28, 385, 962
<i>__thread</i>	528
<i>argparse</i>	512
<i>atexit</i>	523
<i>cmath</i>	385
<i>cmd</i>	949
<i>collections</i>	938
<i>copy</i>	519
<i>cProfile</i>	730
<i>csv</i>	599
<i>ctypes</i>	904
<i>datetime</i>	460
<i>decimal</i>	399
<i>distutils</i>	739
<i>doctest</i>	713
<i>ElementTree</i>	570
<i>email</i>	671
<i>ftplib</i>	646
<i>functools</i>	934
<i>getpass</i>	949

Standardbibliothek (Forts.)		
<code>gettext</code>	442	
<code>gzip</code>	549	
<code>hashlib</code>	446	
<code>imaplib</code>	665	
<code>inspect</code>	694	
<code>io.StringIO</code>	952	
<code>itertools</code>	405	
<code>logging</code>	706	
<code>math</code>	385	
<code>os</code>	481, 490	
<code>os.path</code>	490	
<code>pickle</code>	595	
<code>platform</code>	511	
<code>poplib</code>	660	
<code>pprint</code>	703	
<code>random</code>	394	
<code>re</code>	415, 428	
<code>select</code>	627	
<code>shutil</code>	495, 501	
<code>smtplib</code>	655	
<code>socket</code>	609	
<code>socketserver</code>	630	
<code>sqlite3</code>	579	
<code>struct</code>	946	
<code>sys</code>	501	
<code>telnetlib</code>	676	
<code>tempfile</code>	604	
<code>threading</code>	536	
<code>time</code>	451	
<code>timeit</code>	727	
<code>Tkinter</code>	760	
<code>tkinter</code>	762	
<code>trace</code>	734	
<code>traceback</code>	722	
<code>unittest</code>	718	
<code>urllib.parse</code>	640	
<code>urllib.request</code>	634	
<code>webbrowser</code>	933	
<code>xml</code>	551	
<code>xml.dom</code>	556	
<code>xml.etree.ElementTree</code>	570	
<code>xmlrpc</code>	680	
Standarddialog	819	
staticmethod	299	
Statische Methode	299	
stdin	198	
stdout	198	
Steuerelement	759, 779	
Steuerelementvariable	764	
Steuerzeichen	144	
str	115, 142, 252	
Stream	197	
String	42, 142, 960	
Stringformatierung	156	
StringIO	952	
struct	946	
<code>__sub__</code>	314	
sum	253	
Symmetrische Differenzmenge	191	
Syntax	51	
Syntaxanalyse	552	
sys	501	
T		
Tabulatorweite	52	
Tag	551	
<code>körperloses</code>	553	
tar	551	
TCP	614	
Teilmenge	188	
Telnet	676	
telnetlib	676	
tempfile	604	
Temporäre Datei	604	
Term	40	
Terminator	362	
Test		
<code>automatisierter</code>	713	
Text Edit	848	
Text-Widget	804	
Thread	525, 527	
<code>schlafend</code>	526	
<code>_thread</code>	528	
threading	536	
time	451	
timeit	727	
Tk	760	
Tkinter	760, 762	
<code>Button</code>	783	
<code>Canvas</code>	806	
<code>Checkbutton</code>	786	
<code>Entry-Widget</code>	789	
<code>Event</code>	771	
<code>Font</code>	822	
<code>Label</code>	790	
<code>LabelFrame</code>	791	
<code>Listbox</code>	792	
<code>Menu</code>	796	
<code>Menubutton</code>	799	

Tkinter (Forts.)	
Menüleiste	796
MessageBox	822
Modifier	772
Option Menu	800
Packer	763, 767
Padding	770
Radiobutton	788
Schriftart	822
Scrollbar	801
Spinbox	802
Standarddialog	819
Steuerelement	779
Steuerelementvariable	764
Text-Widget	804
Widget	780
Toolkit	759
PyGtk	760
PyQt	760, 826
PySide	761
Tkinter	760
wxPython	761
Toplevel-Tag	562
trace	734
Traceback	330, 508
traceback	722
Traceback-Objekt	371, 506, 722
Tracer	734
Transaktion	582
Transformation	862
Transmission Control Protocol	614
Transparenz	860
Trolltech	760
True	46, 104
__truediv__	314
try	336
Tupel	140
benanntes	944
tuple	115, 139, 253
Tuple Packing	140
Tuple Unpacking	140
type	77, 254

U

Überdeckungsanalyse	734
UDP	612
UI-Datei	835

Unärer Operator	316
Unicode	168
Uniform Resource Locator	634
Unit Test	718
unittest	718
Unix-Epoche	451
Unix-Timestamp	451
Unveränderlicher Datentyp	83
URL	634, 635, 933
urllib.parse	640
urllib.request	634
User Datagram Protocol	612
UTC	452
UTF	168

V

Variable	43
globale	527
Vektorisierung	886
Veränderlicher Datentyp	83
Verbindungssocket	610
Vereinigungsmenge	189
Vererbung	280
Mehrfachvererbung	294
Vergleich	46
Vergleichsoperator	46, 95, 311
Viewklasse	863
Virtuelle Maschine	37

W

Wahlfreier Zugriff	553
Wahrheitswert	46
Wallis'sches Produkt	528
Warteschlange	542
webbrowser	933
Wert	
boolescher	104
Wertevergleich	78
while	65
Whitespace	146, 420
Widget	759, 780, 829, 839, 848
Wissenschaftliche Schreibweise	103
with	202, 368
Worker-Thread	542
Wurzel	555
wxPython	761

X

XML	551
<i>Attribut</i>	552
<i>Deklaration</i>	551
<i>Parser</i>	552
<i>SAX</i>	565
<i>Tag</i>	551
xml.dom	556
XML-RPC	680
<i>Client</i>	685
<i>Multicall</i>	687
<i>Server</i>	681
__xor__	314

Y

yield	351

Z

Zahl	
<i>ganze</i>	40, 96, 961
<i>Gleitkommazahl</i>	41, 102
<i>komplexe</i>	110
Zahlensystem	97
Zählschleife	71
Zeichenkette	42
Zeichenklasse	417, 420
Zeichenliteral	416
Zeilenkommentar	55
Zeitscheibe	526
ZIP	550
zip	254
zlib	549
Zuweisung	43, 94
<i>erweiterte</i>	94, 315