

Johannes Ernesti, Peter Kaiser

Python

Das umfassende Handbuch

Auf einen Blick

1	Einleitung	21
2	Überblick über Python	27
3	Die Arbeit mit Python	31
4	Der interaktive Modus	39
5	Grundlegendes zu Python-Programmen	47
6	Kontrollstrukturen	55
7	Das Laufzeitmodell	69
8	Basisdatentypen	79
9	Benutzerinteraktion und Dateizugriff	165
10	Funktionen	177
11	Modularisierung	217
12	Objektorientierung	229
13	Weitere Spracheigenschaften	273
14	Mathematik	319
15	Strings	337
16	Datum und Zeit	375
17	Schnittstelle zum Betriebssystem	395
18	Parallele Programmierung	427
19	Datenspeicherung	449
20	Netzwerkkommunikation	505
21	Debugging	579
22	Distribution von Python-Projekten	621
23	Optimierung	641
24	Grafische Benutzeroberflächen	649
25	Python als serverseitige Programmiersprache im WWW mit Django	719
26	Anbindung an andere Programmiersprachen	761
27	Insiderwissen	787
28	Zukunft von Python	795
A	Anhang	799

Inhalt

Teil I

1	Einleitung	21
1.1	Warum haben wir dieses Buch geschrieben?	21
1.2	Was leistet dieses Buch und was nicht?	22
1.3	Wie ist dieses Buch aufgebaut?	23
1.4	Wer sollte dieses Buch wie lesen?	24
1.5	Danksagung	25
2	Überblick über Python	27
2.1	Geschichte und Entstehung	27
2.2	Grundlegende Konzepte	28
2.3	Einsatzmöglichkeiten und Stärken	29
2.4	Aktuelle Einsatzgebiete	30
3	Die Arbeit mit Python	31
3.1	Die Verwendung von Python	31
3.1.1	Windows	33
3.1.2	Linux	33
3.1.3	Mac OS X	33
3.2	Tippen, kompilieren, testen	34
3.2.1	Shebang	35
3.2.2	Interne Abläufe	36
4	Der interaktive Modus	39
4.1	Ganze Zahlen	40
4.2	Gleitkommazahlen	41
4.3	Zeichenketten	41
4.4	Variablen	42
4.5	Logische Ausdrücke	44
4.6	Bildschirm Ausgaben	45

5	Grundlegendes zu Python-Programmen	47
5.1	Grundstruktur eines Python-Programms	47
5.2	Das erste Programm	49
5.3	Kommentare	51
5.4	Der Fehlerfall	52
6	Kontrollstrukturen	55
6.1	Fallunterscheidungen	55
6.1.1	If, elif, else	55
6.1.2	Conditional expressions	59
6.2	Schleifen	60
6.2.1	While-Schleife	60
6.2.2	Vorzeitiger Abbruch einer Schleife	62
6.2.3	Vorzeitiger Abbruch eines Schleifendurchlaufs	63
6.2.4	For-Schleife	64
6.3	Die pass-Anweisung	68
7	Das Laufzeitmodell	69
7.1	Die Struktur von Instanzen	71
7.2	Referenzen und Instanzen freigeben	75
7.3	Mutable vs. immutable Datentypen	77
8	Basisdatentypen	79
8.1	Operatoren	79
8.2	Das Nichts – NoneType	81
8.3	Numerische Datentypen	82
8.3.1	Ganzzahlen – int, long	85
8.3.2	Gleitkommazahlen – float	90
8.3.3	Boolesche Werte – bool	92
8.3.4	Komplexe Zahlen – complex	97
8.4	Methoden und Parameter	100
8.5	Sequenzielle Datentypen	102
8.5.1	Listen – list	110
8.5.2	Unveränderliche Listen – tuple	120
8.5.3	Strings – str, unicode	122
8.6	Mappings	145
8.6.1	Dictionary – dict	146

8.7	Mengen	155
8.7.1	Mengen – set	161
8.7.2	Unveränderliche Mengen – frozenset	163

9 Benutzerinteraktion und Dateizugriff 165

9.1	Bildschirmausgaben	165
9.2	Tastatureingaben	166
9.3	Dateien	168
9.3.1	Datenströme	168
9.3.2	Daten aus einer Datei auslesen	169
9.3.3	Daten in eine Datei schreiben	172
9.3.4	Verwendung des Dateiobjekts	174

10 Funktionen 177

10.1	Schreiben einer Funktion	179
10.2	Funktionsparameter	183
10.2.1	Optionale Parameter	183
10.2.2	Schlüsselwortparameter	184
10.2.3	Beliebige Anzahl von Parametern	185
10.2.4	Seiteneffekte	188
10.3	Zugriff auf globale Variablen	191
10.4	Lokale Funktionen	193
10.5	Anonyme Funktionen	193
10.6	Rekursion	194
10.7	Vordefinierte Funktionen	195

Teil II

11 Modularisierung 217

11.1	Einbinden externer Programmbibliotheken	217
11.2	Eigene Module	219
11.2.1	Modulinterne Referenzen	221
11.3	Pakete	221
11.3.1	Importieren aller Module eines Pakets	224
11.3.2	Relative Importanweisungen	224
11.4	Built-in Functions	226

12 Objektorientierung	229
12.1 Klassen	234
12.1.1 Definieren von Methoden	236
12.1.2 Konstruktor, Destruktor und die Erzeugung von Attributen	237
12.1.3 Private Member	240
12.1.4 Versteckte Setter und Getter	244
12.1.5 Statische Member	246
12.2 Vererbung	249
12.2.1 Mehrfachvererbung	253
12.3 Magic Members	257
12.3.1 Allgemeine Magic Members	257
12.3.2 Datentypen emulieren	264
12.4 Objektphilosophie	271
13 Weitere Spracheigenschaften	273
13.1 Exception Handling	273
13.1.1 Eingebaute Exceptions	274
13.1.2 Werfen einer Exception	277
13.1.3 Abfangen einer Exception	278
13.1.4 Eigene Exceptions	283
13.1.5 Erneutes Werfen einer Exception	284
13.2 List Comprehensions	287
13.3 Docstrings	289
13.4 Generatoren	290
13.5 Iteratoren	296
13.6 Interpreter im Interpreter	304
13.7 Geplante Sprachelemente	306
13.8 Die with-Anweisung	307
13.9 Function Decorator	310
13.10 assert	313
13.11 Weitere Aspekte der Syntax	314
13.11.1 Umbrechen langer Zeilen	314
13.11.2 Zusammenfügen mehrerer Zeilen	315
13.11.3 String conversions	316

Teil III

14 Mathematik	319
14.1 Mathematische Funktionen – math, cmath	319
14.2 Zufallszahlengenerator – random	325
14.3 Präzise Dezimalzahlen – decimal	331
14.3.1 Verwendung des Datentyps	332
14.3.2 Nichtnumerische Werte	334
14.3.3 Das Context-Objekt	335
15 Strings	337
15.1 Arbeiten mit Zeichenketten – string	337
15.1.1 Ein einfaches Template-System	339
15.2 Reguläre Ausdrücke – re	340
15.2.1 Syntax regulärer Ausdrücke	341
15.2.2 Verwendung des Moduls	352
15.2.3 Ein einfaches Beispielprogramm – Searching	361
15.2.4 Ein komplexeres Beispielprogramm – Matching	362
15.3 Lokalisierung von Programmen – gettext	365
15.3.1 Beispiel für die Verwendung von gettext	366
15.4 Hash-Funktionen – hashlib	369
15.4.1 Verwendung des Moduls	371
15.4.2 Beispiel	372
15.5 Dateinterface für Strings – StringIO	373
16 Datum und Zeit	375
16.1 Elementare Zeitfunktionen – time	375
16.2 Komfortable Datumsfunktionen – datetime	381
16.2.1 datetime.date	383
16.2.2 datetime.time	386
16.2.3 datetime.datetime	389
17 Schnittstelle zum Betriebssystem	395
17.1 Funktionen des Betriebssystems – os	395
17.1.1 Zugriff auf den eigenen Prozess und andere Prozesse	396
17.1.2 Zugriff auf das Dateisystem	397
17.2 Umgang mit Pfaden – os.path	403

17.3	Zugriff auf die Laufzeitumgebung – sys	407
17.3.1	Konstanten	408
17.3.2	Exceptions	410
17.3.3	Hooks	411
17.3.4	Sonstige Funktionen	413
17.4	Informationen über das System – platform	414
17.4.1	Funktionen	415
17.5	Kommandozeilenparameter – optparse	415
17.5.1	Taschenrechner – ein einfaches Beispiel	416
17.5.2	Weitere Verwendungsmöglichkeiten	418
17.6	Kopieren von Instanzen – copy	420
17.7	Zugriff auf das Dateisystem – shutil	423
17.8	Das Programmende – atexit	425

18 Parallele Programmierung 427

18.1	Prozesse, Multitasking und Threads	427
18.2	Die Thread-Unterstützung in Python	430
18.3	Das Modul thread	430
18.3.1	Datenaustausch zwischen Threads – locking	432
18.4	Das Modul threading	437
18.4.1	Locking im threading-Modul	439
18.4.2	Worker-Threads und Queues	443
18.4.3	Ereignisse definieren – threading.Event	446
18.4.4	Eine Funktion zeitlich versetzt ausführen – threading.Timer	446

19 Datenspeicherung 449

19.1	Komprimierte Dateien lesen und schreiben – gzip	449
19.2	XML	451
19.2.1	DOM – Document Object Model	453
19.2.2	SAX – Simple API for XML	464
19.2.3	ElementTree	469
19.3	Datenbanken	473
19.3.1	Pythons eingebaute Datenbank – sqlite3	477
19.3.2	MySQLdb	492
19.4	Serialisierung von Instanzen – pickle	494
19.5	Das Tabellenformat CSV – csv	498
19.6	Temporäre Dateien – tempfile	503

20 Netzwerkkommunikation 505

20.1	Socket API	507
20.1.1	Client/Server-Systeme	508
20.1.2	UDP	510
20.1.3	TCP	512
20.1.4	Blockierende und nicht-blockierende Sockets	515
20.1.5	Verwendung des Moduls	516
20.1.6	Netzwerk-Byte-Order	521
20.1.7	Multiplexende Server – select	522
20.1.8	SocketServer	525
20.2	Zugriff auf Ressourcen im Internet – urllib	529
20.2.1	Verwendung des Moduls	529
20.3	Einlesen einer URL – urlparse	534
20.4	FTP – ftplib	537
20.5	E-Mail	544
20.5.1	SMTP – smtplib	545
20.5.2	POP3 – poplib	548
20.5.3	IMAP4 – imaplib	553
20.5.4	Erstellen komplexer E-Mails – email	559
20.6	Telnet – telnetlib	564
20.7	XML-RPC	567
20.7.1	Der Server	568
20.7.2	Der Client	571
20.7.3	Multicall	574
20.7.4	Einschränkungen	575

21 Debugging 579

21.1	Der Debugger	579
21.2	Inspizieren von Instanzen – inspect	582
21.2.1	Datentypen, Attribute und Methoden	583
21.2.2	Quellcode	584
21.2.3	Klassen und Funktionen	586
21.3	Formatierte Ausgabe von Instanzen – pprint	590
21.4	Logdateien – logging	592
21.4.1	Das Meldungsformat anpassen	594
21.4.2	Logging Handler	596
21.5	Automatisiertes Testen	598
21.5.1	Testfälle in Docstrings – doctest	598
21.5.2	Unit Tests – unittest	602

21.6	Traceback-Objekte – traceback	606
21.7	Analyse des Laufzeitverhaltens	609
21.7.1	Laufzeitmessung – timeit	609
21.7.2	Profiling – cProfile	612
21.7.3	Tracing – trace	616

Teil IV

22 Distribution von Python-Projekten 621

22.1	Erstellen von Distributionen – distutils	621
22.1.1	Schreiben des Moduls	623
22.1.2	Das Installationsscript	624
22.1.3	Erstellen einer Quellcodedistribution	628
22.1.4	Erstellen einer Binärdistribution	629
22.1.5	Beispiel für die Verwendung einer Distribution	630
22.2	Erstellen von EXE-Dateien – py2exe	631
22.3	Automatisches Erstellen einer Dokumentation – epydoc	633
22.3.1	Docstrings und ihre Formatierung für epydoc	635

23 Optimierung 641

23.1	Die Optimize-Option	642
23.2	Strings	642
23.3	Funktionsaufrufe	644
23.4	Schleifen	644
23.5	C	645
23.6	Lookup	645
23.7	Lokale Referenzen	646
23.8	Exceptions	647
23.9	Keyword arguments	647

24 Grafische Benutzeroberflächen 649

24.1	Toolkits	649
24.2	Einführung in PyQt	651
24.2.1	Installation	651
24.2.2	Grundlegende Konzepte von Qt	652
24.3	Entwicklungsprozess	655
24.3.1	Erstellen des Dialogs	655
24.3.2	Schreiben des Programms	663

24.4	Signale und Slots	665
24.5	Überblick über das Qt-Framework	668
24.6	Zeichenfunktionalität	670
24.6.1	Werkzeuge	670
24.6.2	Koordinatensystem	672
24.6.3	Einfache Formen	673
24.6.4	Grafiken	676
24.6.5	Text	677
24.6.6	Eye-Candy	679
24.7	Model-View-Architektur	683
24.7.1	Beispielprojekt: Ein Adressbuch	684
24.7.2	Auswählen von Einträgen	695
24.7.3	Editieren von Einträgen	696
24.8	Wichtige Widgets	700
24.8.1	QCheckBox	700
24.8.2	QComboBox	701
24.8.3	QDateEdit	702
24.8.4	QDateTimeEdit	703
24.8.5	QDial	704
24.8.6	QDialog	704
24.8.7	QGLWidget	705
24.8.8	QLineEdit	706
24.8.9	QListView	707
24.8.10	QListWidget	707
24.8.11	QProgressBar	708
24.8.12	QPushButton	709
24.8.13	QRadioButton	709
24.8.14	QScrollArea	710
24.8.15	QSlider	711
24.8.16	QTableView	711
24.8.17	QTableWidget	712
24.8.18	QTabWidget	713
24.8.19	QTextEdit	713
24.8.20	QTimeEdit	714
24.8.21	QTreeView	715
24.8.22	QTreeWidget	715
24.8.23	QWidget	716

25 Python als serverseitige Programmiersprache im WWW mit Django	719
25.1 Installation	720
25.2 Konzepte und Besonderheiten im Überblick	723
25.3 Erstellen eines neuen Django-Projekts	724
25.4 Erstellung der Applikation	727
25.5 Djangos Administrationsoberfläche	736
25.6 Unser Projekt wird öffentlich	740
25.7 Djangos Template-System	747
25.8 Verarbeitung von Formulardaten	757
26 Anbindung an andere Programmiersprachen	761
26.1 Dynamisch ladbare Bibliotheken – ctypes	762
26.1.1 Ein einfaches Beispiel	762
26.1.2 Die eigene Bibliothek	763
26.1.3 Schnittstellenbeschreibung	767
26.1.4 Verwendung des Moduls	768
26.2 Schreiben von Extensions	769
26.2.1 Ein einfaches Beispiel	770
26.2.2 Exceptions	773
26.2.3 Erzeugen der Extension	774
26.2.4 Reference Counting	776
26.3 Python als eingebettete Skriptsprache	777
26.3.1 Ein einfaches Beispiel	778
26.3.2 Ein komplexeres Beispiel	779
26.3.3 Python-API-Referenz	782
27 Insiderwissen	787
27.1 Dateien direkt mit einem bestimmten Encoding lesen	787
27.2 URLs im Standardbrowser öffnen – webbrowser	788
27.3 Funktionsschnittstellen vereinfachen – functools	789
27.4 Versteckte Passworтеingaben – getpass	790
27.5 Kommandozeilen-Interpreter – cmd	791
28 Zukunft von Python	795
28.1 Python 3000	795
28.2 Python 2.6	796

A Anhang	799
A.1 Entwicklungsumgebungen	799
A.1.1 Eclipse	799
A.1.2 Eric4	800
A.1.3 Komodo IDE	801
A.1.4 Wing IDE	801
A.2 Reservierte Wörter	803
A.3 Operatorrangfolge	803
A.4 Built-in Exceptions	804
A.5 Built-in Functions	808
Index	811

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

3 Die Arbeit mit Python

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

3.1 Die Verwendung von Python

Die jeweils aktuelle Version von Python können Sie von der offiziellen Python-Website unter <http://www.python.org> als Installationsdatei für Ihr Betriebssystem herunterladen und installieren. Alternativ können Sie Python 2.5.1 von der CD installieren, die diesem Buch beiliegt.

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

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

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

Bei *IDLE* (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 bie-

tet IDLE dem Programmierer einige Komfortfunktionen wie beispielsweise das farbige Hervorheben bestimmter Code-Elemente (»Syntax Highlighter«) oder eine automatische Vervollständigung von Code.

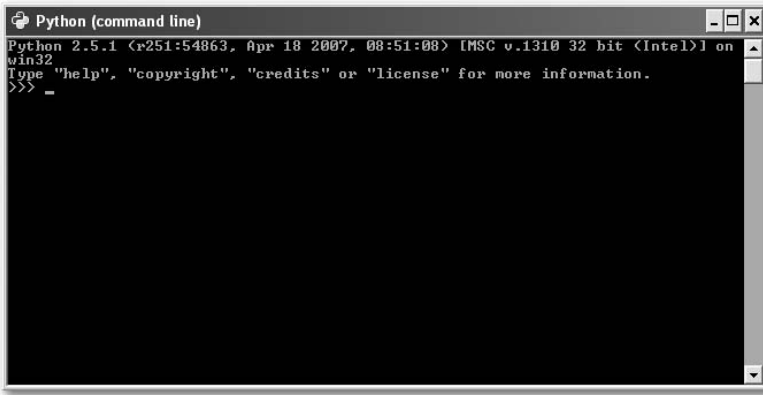


Abbildung 3.1 Python im interaktiven Modus (Python Shell)

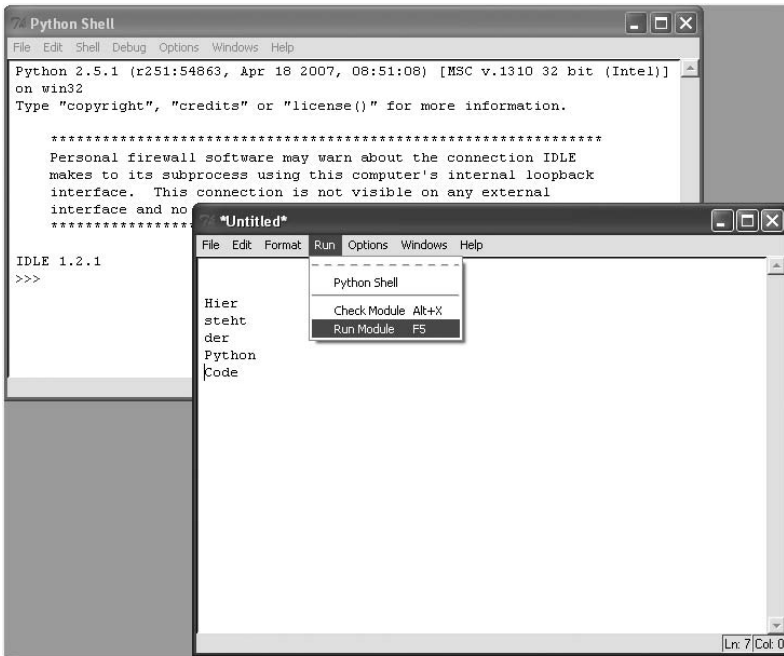


Abbildung 3.2 Die Entwicklungsumgebung IDLE

Wenn Sie mit IDLE nicht zufrieden sind, finden Sie eine Übersicht über die wichtigsten Python-Entwicklungsumgebungen im Anhang dieses Buchs. Zudem befin-

det sich auf der offiziellen Python-Website unter <http://wiki.python.org/moin/PythonEditors> eine umfassende Auflistung aller Entwicklungsumgebungen und Editoren für Python.

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

3.1.1 Windows

Sie finden die Windows-Installationsdatei von Python 2.5.1 auf der dem Buch beigelegten CD-ROM.

Nach der Installation von Python unter Windows finden Sie im Wesentlichen zwei neue Einträge im Startmenü: »Python (command line)« und »IDLE (Python GUI)«. Ersteres startet den interaktiven Modus von Python in der Kommandozeile (»schwarzes Fenster«) und Letzteres die grafische Entwicklungsumgebung IDLE.

3.1.2 Linux

Sie finden den Quellcode von Python 2.5.1 auf der dem Buch beigelegten CD-ROM.

Beachten Sie, dass Python bei vielen Linux-Distributionen bereits im Lieferumfang enthalten ist oder sich mit dem jeweiligen Paketmanager der Distribution bequem nachinstallieren lässt. Sollten Sie eine Distribution ohne Paketmanager einsetzen oder sollte Python 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.

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

3.1.3 Mac OS X

Sie finden die Mac OS X-Installationsdatei von Python 2.5.1 auf der dem Buch beigelegten CD-ROM.


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

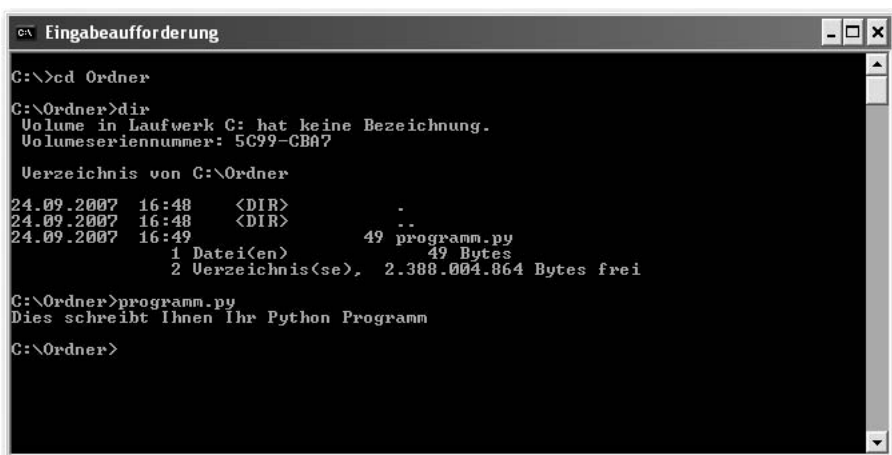
3.2 Tippen, kompilieren, testen

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

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

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

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



```

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



```

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

```

Abbildung 3.4 Ausführen eines Python-Programms unter Linux

3.2.1 Shebang

Unter einem Unix-ähnlichen Betriebssystem wie beispielsweise Linux können Python-Programmdateien mithilfe eines sogenannten *Shebangs*, auch *Magic Line*

genannt, direkt ausführbar gemacht werden. Dazu muss die erste Zeile der Programmdatei in der Regel folgendermaßen lauten:

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

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

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

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

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

```
chmod +x dateiname
```

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

3.2.2 Interne Abläufe

Bislang haben Sie einen ungefähren Begriff davon, was Python ausmacht und wo die Stärken der Programmiersprache liegen. Außerdem wurde das theoretische Grundwissen zum Erstellen und Ausführen einer Python-Programmdatei vermittelt. Doch in den vorherigen Abschnitten sind Begriffe wie Compiler oder Interpreter gefallen, ohne tatsächlich erklärt worden zu sein. In diesem Abschnitt möchten wir uns daher den internen Vorgängen widmen, die beim Ausführen einer Python-Programmdatei ablaufen. Die folgende Grafik soll veranschaulichen, 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. In Falle von Python übersetzt der Compiler von der Sprache Python in den sogenannten *Byte-Code*. Dabei steht es dem Compiler frei, den generierten Byte-Code im Arbeitsspeicher zu behalten oder als *programm.pyc* auf der Festplatte zu speichern.

Beachten Sie, dass das vom Compiler generierte Kompilat, im Gegensatz zu beispielsweise C- oder C++-Kompilaten, nicht direkt auf dem Prozessor ausgeführt

werden kann. Zur Ausführung des Byte-Codes wird eine weitere Abstraktionsschicht, der sogenannte *Interpreter*, benötigt. Der Interpreter, häufig auch *virtuelle Maschine* (engl. *virtual machine*) genannt, liest den vom Compiler erzeugten Byte-Code ein und führt ihn aus.

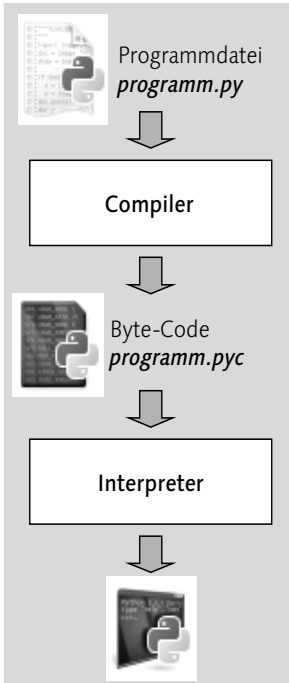


Abbildung 3.5 Kompilieren und Interpretieren einer Programmdatei

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 auch immer langsamer als ein vergleichbares C-Programm, das direkt auf dem Prozessor ausgeführt wird.

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

5 Grundlegendes zu Python-Programmen

In diesem Kapitel werden wir einige Grundlagen zur Programmierung in Python erarbeiten. Insbesondere werden Sie Ihr erstes »richtiges« Python-Programm erstellen, das nicht mehr nur im interaktiven Modus läuft.

5.1 Grundstruktur eines Python-Programms

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

Um Ihnen ein Gefühl für die Sprache Python zu vermitteln, möchten wir zunächst einen Überblick über ihre Syntax geben. Dazu ist zu sagen, dass Python dem Programmierer sehr genaue Vorgaben macht, wie er seinen Quellcode zu strukturieren hat. Obwohl erfahrene Programmierer darin eine Einschränkung sehen mögen, kommt diese Eigenschaft gerade Programmierneulingen zugute, denn unstrukturierter und unübersichtlicher Code ist eine der größten Fehlerquellen in der Programmierung.

Grundsätzlich besteht ein Python-Programm aus einzelnen *Anweisungen*, die im einfachsten Fall genau eine Zeile im Quelltext einnehmen. Folgende Anweisung gibt beispielsweise einen Text auf dem Bildschirm aus:

```
print "Hallo Welt"
```

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

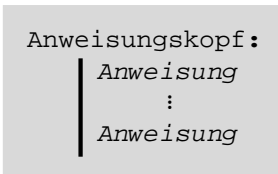


Abbildung 5.1 Struktur einer mehrzeiligen Anweisung

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

```

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

```

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

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


Achtung!

Ein Programm, in dem sowohl Leerzeichen als auch Tabulatoren verwendet wurden, kann vom Python-Interpreter anstandslos übersetzt werden, da jeder Tabulator intern durch acht Leerzeichen ersetzt wird. Dies kann aber zu schwer auffindbaren Fehlern führen, denn viele Editoren verwenden standardmäßig eine Tabulatorweite von vier Leerzeichen. Dadurch scheinen bestimmte Quellcodeabschnitte gleich weit eingerrückt, 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!"
...
>>>
```

Beachten Sie, dass Sie, auch wenn eine Zeile mit ... beginnt, die aktuelle Einrückungstiefe berücksichtigen müssen. Das Ende des Anweisungskörpers kann der Interpreter nicht automatisch erkennen, da er beliebig viele Anweisungen enthalten kann. Deswegen muss ein Anweisungskörper im interaktiven Modus durch Drücken der -Taste beendet werden.

5.2 Das erste Programm

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

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

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

Kommen wir vom Ablaufprotokoll zur konkreten Implementierung in Python (siehe Abbildung 5.2).

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

Initialisierung

Hier werden die für das Spiel benötigten Variablen angelegt. Python unterscheidet zwischen verschiedenen *Variablentypen*, wie Zeichenketten, Ganz- oder

Fließkommazahlen. Der Typ einer Variablen wird zur Laufzeit des Programms anhand des ihr zugewiesenen Wertes bestimmt. Es ist also nicht nötig, einen Variablentyp explizit anzugeben. Eine Variable kann im Laufe des Programms ihren Typ ändern.

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

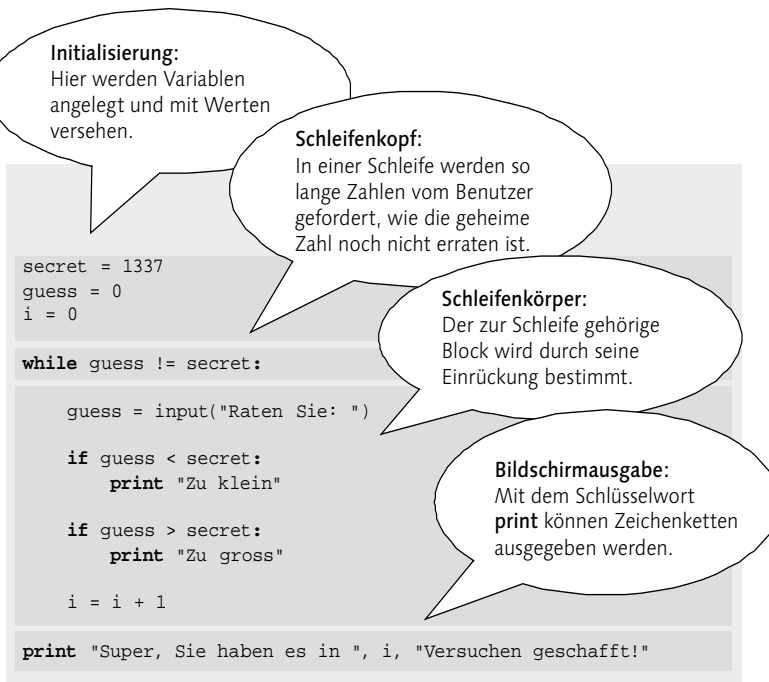


Abbildung 5.2 Zahlenraten, ein einfaches Beispiel

Schleifenkopf

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

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

Schleifenkörper

In der ersten Zeile des Schleifenkörpers wird eine Zahl vom Spieler eingelesen und in der Variablen `guess` gespeichert. Dazu wird die Zeile `input("Raten Sie: ")` benötigt. Der String "Raten Sie: " wird dabei vor der Eingabe ausgegeben und dient dazu, den Benutzer zur Eingabe der Zahl aufzufordern.

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

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

Bildschirmausgabe

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

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

5.3 Kommentare

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

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

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

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

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

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

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

5.4 Der Fehlerfall

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

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

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

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

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

Neben dem Syntaxfehler gibt es eine ganze Reihe weiterer Fehlertypen, die hier nicht alle im Detail besprochen werden sollen. Wir möchten jedoch noch auf den

`IndentationError` (dt. *Einrückungsfehler*) hinweisen, da er gerade bei Python-Anfängern häufig auftritt. Versuchen Sie dazu einmal folgendes Programm auszuführen:

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

Sie sehen, dass die letzte Zeile eigentlich einen Schritt weiter eingerückt sein müsste. So, wie das Programm jetzt geschrieben wurde, hat die `if`-Anweisung keinen Anweisungskörper. Das ist nicht zulässig, und es tritt ein `Indentation Error` auf:

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

`IndentationError: expected an indented block`

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

»Divide et impera!«
– Julius Caesar

11 Modularisierung

Unter Modularisierung versteht man die Aufteilung des Quelltextes in einzelne Teile, sogenannte *Module*. Grundsätzlich gibt es zwei Arten von Modulen:

Zum einen kann jedes Python-Programm sogenannte *Bibliotheken* (engl. *Libraries*) einbinden. Eine Bibliothek dient häufig einem ganz bestimmten Zweck, wie etwa der Arbeit mit Dateien eines bestimmten Dateiformats, und stellt üblicherweise Datentypen oder Funktionen bereit, die nach dem Einbinden verwendet werden können. Es ist möglich, eigene Bibliotheken zu schreiben oder eine Bibliothek eines Drittanbieters zu installieren. Ein gutes Argument für Python ist die umfangreiche *Standardbibliothek*, die im Lieferumfang enthalten ist. Sie bietet eine hohe Grundfunktionalität, die in jeder Python-Umgebung verfügbar ist.

Die zweite Möglichkeit zur Modularisierung sind lokale Module. Darunter versteht man die Kapselung einzelner Programmteile – auch hier üblicherweise Datentypen oder Funktionen – in eigene Programmdateien. Diese Dateien können wie Bibliotheken eingebunden werden, sind aber in keinem anderen Python-Programm verfügbar. Diese Form der Modularisierung hilft bei der Programmierung ungemein, da sie dem Programmierer die Möglichkeit gibt, sehr langen Programmcode überschaubar auf verschiedene Programmdateien aufzuteilen.

11.1 Einbinden externer Programmbibliotheken

Eine Bibliothek, sei es ein Teil der Standardbibliothek oder eine selbst geschriebene, kann mithilfe der `import`-Anweisung eingebunden werden. Wir werden in den Beispielen hauptsächlich das Modul `math` der Standardbibliothek verwenden. Das ist ein Modul, das mathematische Funktionen wie `sin` oder `cos` sowie mathematische Konstanten wie `pi` bereitstellt. Um sich diese Funktionalität in einem Programm zunutze machen zu können, ist folgende `import`-Anweisung nötig:

```
import math
```

Eine `import`-Anweisung besteht aus dem Schlüsselwort `import`, gefolgt von einem Modulnamen. Es können mehrere Module gleichzeitig eingebunden werden, indem diese, durch Kommata getrennt, hinter das Schlüsselwort geschrieben werden:

```
import math, random
```

Dies ist äquivalent zu:

```
import math
import random
```

Obwohl eine `import`-Anweisung prinzipiell überall im Quellcode stehen kann, ist es der Übersichtlichkeit halber sinnvoll, alle Module zu Beginn des Quelltextes einzubinden.

Nachdem eine Bibliothek eingebunden wurde, wird für diese ein neuer Namensraum mit dem Namen der Bibliothek erstellt. Über diesen Namensraum sind alle Funktionen, Datentypen und Konstanten der Bibliothek im Programm nutzbar. Mit einem Namensraum kann wie mit einer Instanz umgegangen werden, und die Funktionen der Bibliothek können wie Methoden des Namensraums verwendet werden. So bindet folgendes Beispielprogramm die Bibliothek `math` ein und berechnet den Sinus von der Kreiszahl π :

```
import math
print math.sin(math.pi)
```

Es ist möglich, den Namen des Namensraums durch eine `import/as`-Anweisung festzulegen:

```
import math as mathematik
print mathematik.sin(mathematik.pi)
```

Beachten Sie, dass dieser Name keine zusätzliche Option ist, sondern das Modul `math` nun ausschließlich über den Namensraum `mathematik` erreichbar ist.

Des Weiteren kann die `import`-Anweisung so verwendet werden, dass kein eigener Namensraum für die eingebundene Bibliothek erzeugt wird, sondern alle Elemente dieser Bibliothek im globalen Namensraum des Programms zur Verfügung stehen:

```
from math import *
print sin(pi)
```

Wenn die `import`-Anweisung in dieser Weise verwendet wird, sollten Sie beachten, dass keine Referenzen, Funktionen oder Instanzen des einzubindenden Moduls in den aktuellen Namensraum importiert werden, wenn sie mit einem Unterstrich beginnen. Diese Elemente eines Moduls werden als privat und damit als modulintern angesehen.

Hinweis

Der Sinn von Namensräumen ist es, thematisch abgegrenzte Bereiche, also zum Beispiel eine Bibliothek, zu kapseln und über einen gemeinsamen Namen anzusprechen. Wenn Sie den kompletten Inhalt einer Bibliothek in den globalen Namensraum eines Programms einbinden, kann es vorkommen, dass die Bibliothek mit eventuell vorhandenen Referenzen interferiert. In einem solchen Fall werden die bereits bestehenden Referenzen kommentarlos überschrieben, wie das folgende Beispiel zeigt:

```
>>> pi = 1234
>>> pi
1234
>>> from math import *
>>> pi
3.1415926535897931
```

Aus diesem Grund ist es immer sinnvoll, eine Bibliothek, wenn sie vollständig eingebunden wird, in einem eigenen Namensraum zu kapseln und damit die Anzahl der im globalen Namensraum eingebundenen Elemente möglichst gering zu halten.

Im Hinweiskasten wurde gesagt, dass man die Anzahl der in den globalen Namensraum importierten Objekte möglichst gering halten sollte. Aus diesem Grund ist die oben geschriebene Form der `from/import`-Anweisung nicht gerade praktikabel. Es ist aber möglich, anstatt des Sterns eine Liste von zu importierenden Elementen der Bibliothek anzugeben:

```
from math import sin, pi
print sin(pi)
```

In diesem Fall werden ausschließlich die Funktion `sin` und die Konstante `pi` in den globalen Namensraum importiert. Auch hier ist es möglich, durch ein dem Namen nachgestelltes `as` einen eigenen Namen festzulegen:

```
from math import sin as hallo, pi as welt
print hallo(welt)
```

So viel zum Einbinden externer Bibliotheken. Sie werden die Standardbibliothek von Python im dritten Teil dieses Buches noch ausführlich kennenlernen.

11.2 Eigene Module

Nachdem Sie in die unendlichen Weiten der `import`-Anweisung eingeführt wurden, möchten wir uns damit beschäftigen, wie Module selbst erstellt und eingebunden werden können. Beachten Sie, dass es sich hier nicht um eine Bibliothek handelt, die in jedem Python-Programm zur Verfügung steht, sondern um ein Modul, das nur lokal in Ihrem Python-Programm genutzt werden kann. Von der

Verwendung her unterscheiden sich Module und Bibliotheken kaum. In diesem Abschnitt soll ein Programm erstellt werden, das eine ganze Zahl einliest, deren Fakultät und Kehrwert berechnet und die Ergebnisse ausgibt. Die mathematischen Berechnungen sollen dabei nicht nur in Funktionen, sondern auch in einem eigenen Modul gekapselt werden. Dazu schreiben wir diese zunächst in eine Datei namens *mathematik.py*:

```
def fak(n):
    ergebnis = 1
    for i in xrange(2, n+1):
        ergebnis *= i
    return ergebnis

def kehr(n):
    return 1.0 / n
```

Die Funktionen sollten selbsterklärend sein. Beachten Sie, dass die Datei *mathematik.py* selbst keinerlei Code ausführt, sondern nur Funktionen bereitstellt, die aus anderen Modulen heraus aufgerufen werden können.

Jetzt erstellen wir eine Programmdatei namens *programm.py*, in der das Hauptprogramm stehen soll. Beide Dateien müssen sich im selben Verzeichnis befinden. Im Hauptprogramm importieren wir zunächst das lokale Modul *mathematik*. Der Modulname eines lokalen Moduls entspricht dem Dateinamen der zugehörigen Programmdatei ohne Dateiendung. Beachten Sie, dass der Modulname den Regeln der Namensgebung eines Bezeichners folgen muss. Das bedeutet insbesondere, dass, abgesehen von dem Punkt vor der Dateiendung, kein Punkt im Dateinamen erlaubt ist.

```
import mathematik

while True:
    zahl = int(raw_input("Geben Sie eine ganze Zahl ein: "))
    print "Fakultaet: ", mathematik.fak(zahl)
    print "Kehrwert: ", mathematik.kehr(zahl)
```

Sie sehen, dass das lokale Modul im Hauptprogramm wie eine Bibliothek importiert und verwendet werden kann.

Durch das Erstellen eigener Module kann es leicht zu Namenskonflikten mit der Standardbibliothek kommen. Beispielsweise hätten wir unsere obige Programmdatei auch *math.py* und das Modul demzufolge *math* nennen können. Dieses Modul stünde im Konflikt mit der Bibliothek *math*. Für solche Fälle ist dem Interpreter eine Reihenfolge vorgegeben, nach der er zu verfahren hat, wenn ein Modul oder eine Bibliothek importiert werden soll:

- ▶ Zunächst wird der lokale Programmordner nach einer Datei mit dem entsprechenden Namen durchsucht. In dem oben geschilderten Konfliktfall würde bereits im ersten Schritt feststehen, dass ein lokales Modul namens `math` existiert. Wenn ein solches lokales Modul existiert, wird dieses eingebunden und keine weitere Suche durchgeführt.
- ▶ Wenn kein lokales Modul des angegebenen Namens gefunden wurde, wird die Suche auf Bibliotheken ausgeweitet.
- ▶ Wenn auch keine Bibliothek mit dem angegebenen Namen gefunden werden konnte, wird ein `ImportError` erzeugt:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named abc
```

11.2.1 Modulinterne Referenzen

In jedem Modul existieren globale Variablen, die Informationen über das Modul selbst enthalten. An dieser Stelle soll ein Überblick über diese recht überschaubare Anzahl von Referenzen gegeben werden. Beachten Sie, dass es sich jeweils um zwei Unterstriche vor und hinter dem Namen der Referenz handelt.

Referenz	Beschreibung
<code>__builtins__</code>	Referenziert ein Dictionary, das die Namen aller eingebauten Typen und Funktionen als Schlüssel und die mit den Namen verknüpften Instanzen als Werte enthält.
<code>__file__</code>	Referenziert einen String, der den Namen der Programmdatei des Moduls inklusive Pfad enthält. Nicht bei Modulen der Standardbibliothek verfügbar.
<code>__name__</code>	Referenziert einen String, der den Namen des Moduls enthält.

Tabelle 11.1 Globale Variablen in einem Modul

11.3 Pakete

Python ermöglicht es Ihnen, mehrere Module in einem sogenannten *Paket* zu kapseln. Das ist vorteilhaft, wenn diese Module thematisch zusammengehören. Ein Paket kann, im Gegensatz zu einem einzelnen Modul, beliebig viele weitere Pakete enthalten, die ihrerseits wieder Module bzw. Pakete enthalten können.

Um ein Paket zu erstellen, muss im Wesentlichen ein Unterordner im Programmverzeichnis erzeugt werden. Der Name des Ordners entspricht dem Namen des Pakets. Zusätzlich muss in diesem Ordner eine Programmdatei namens `__init__.py`

existieren. (Beachten Sie, dass es sich um jeweils zwei Unterstriche vor und hinter »init« handelt.) Diese Datei darf leer, muss aber vorhanden sein und enthält Initialisierungscode, der beim Einbinden des Paketes einmalig ausgeführt wird. Ein Programm mit mehreren Paketen und Unterpaketen hat also eine solche oder ähnliche Verzeichnisstruktur, wie Sie in Abbildung 11.1 sehen.

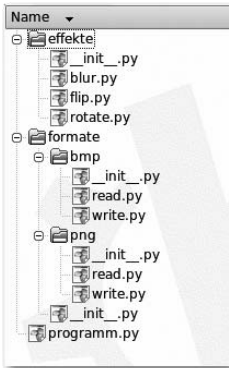


Abbildung 11.1 Paketstruktur eines Beispielprogramms

Im weiteren Verlauf des Kapitels werden wir immer wieder auf dieses Beispiel zurückkommen, deswegen soll es hier kurz besprochen werden. Es handelt sich um die Verzeichnisstruktur eines fiktiven Bildbearbeitungsprogramms. Das Hauptprogramm befindet sich in der Datei *programm.py*. Neben dem Hauptprogramm existieren im Programmverzeichnis zwei Pakete:

- ▶ Das Paket *effekte* soll bestimmte Effekte auf ein bereits geladenes Bild anwenden können. Dazu enthält das Paket neben der Datei *__init__.py* drei Module, die jeweils einen grundlegenden Effekt durchführen sollen. Es handelt sich um die Module *blur* (zum Verwischen des Bildes), *flip* (zum Spiegeln des Bildes) und *rotate* (zum Drehen des Bildes).
- ▶ Das Paket *formate* soll dazu in der Lage sein, bestimmte Grafikformate zu lesen und schreiben. Dazu definiert es in seiner *__init__.py* zwei Funktionen namens *leseBild* und *schreibeBild*. Wir möchten nicht näher auf Funktionsschnittstellen oder Ähnliches eingehen, sondern relativ abstrakt bleiben. Damit das Lesen und Schreiben von Grafiken diverser Formate möglich ist, enthält das Paket *formate* zwei Unterpakete namens *bmp* und *png*, die je zwei Module zum Lesen bzw. Schreiben des entsprechenden Formats enthalten.

Im Hauptprogramm sollen zunächst die Pakete *effekte* und *formate* eingebunden und verwendet werden. Dies ermöglicht die *import*-Anweisung:

```
import effekte, formate
```

bzw:

```
import effekte
import formate
```

Beachten Sie, dass es zu einem Namenskonflikt kommt, wenn beispielsweise neben dem Paket `effekte` ein Modul gleichen Namens, also eine Programmdatei namens `effekte.py`, existiert. Es ist grundsätzlich so, dass bei Namensgleichheit ein Paket Vorrang vor einem Modul hat, es also keine Möglichkeit mehr gibt, das Modul zu importieren.

Die `import`-Anweisung veranlasst, dass die Programmdatei `__init__.py` des einzubindenden Paketes ausgeführt und der Inhalt dieser Datei als Modul in einem eigenen Namensraum verfügbar gemacht wird. So könnte nach den obigen `import`-Anweisungen folgendermaßen auf die Funktionen `leseBild` und `schreibeBild` zugegriffen werden:

```
formate.leseBild()
formate.schreibeBild()
```

Um das nun geladene Bild zu modifizieren, soll diesmal ein Modul des Paketes `effekte` geladen werden. Auch dies ist mit der `import`-Anweisung möglich. Der Paketname wird durch einen Punkt vom Modulnamen getrennt. Auf diese Weise kann ein Modul aus einer beliebigen Paketstruktur importiert werden:

```
import effekte.blur
```

In diesem Fall wurde das Paket `effekte` vorher eingebunden. Wenn dies nicht der Fall gewesen wäre, so würde das Importieren von `effekte.blur` dafür sorgen, dass zunächst das Paket `effekte` eingebunden und die dazugehörige `__init__.py` ausgeführt werden würde. Danach wird das Untermodul `blur` eingebunden. Das Modul kann fortan wie jedes andere verwendet werden:

```
effekte.blur.verschwemmeBild()
```

Auf diese Weise kann auch direkt auf ein Modul zugegriffen werden, das tiefer in der Paketstruktur steht. Hier greifen wir beispielsweise auf das Modul `formate.bmp.write` zu und rufen die Funktion `schreibeBMP` auf, die in diesem Modul definiert werden soll.

```
import formate.bmp.write
formate.bmp.write.schreibeBMP()
```

Auf ein eingebundenes Modul oder Paket kann nur über seinen vollen Namen, also inklusive aller übergeordneten Pakete, zugegriffen werden. Allerdings ist es auch hier möglich, durch `as` ein Alias zu vergeben:

```
import formate.bmp.write as w
w.schreibeBMP()
```

11.3.1 Importieren aller Module eines Pakets

Bisher konnte mit

```
from abc import *
```

der gesamte Inhalt eines Moduls in den aktuellen Namensraum importiert werden. Dies funktioniert für Pakete nicht. Der Grund dafür ist, dass einige Betriebssysteme, darunter vor allem Windows, bei Datei- und Ordnernamen nicht zwischen Groß- und Kleinschreibung unterscheiden – Python aber sehr wohl. Angenommen, die obige Anweisung würde wie gehabt funktionieren und `abc` wäre ein Paket, so wäre es beispielsweise unter Windows völlig unklar, ob ein Untermodul namens `modul` als `Modul`, `MODUL` oder `modul` eingebunden werden soll.

Aus diesem Grund importiert die obige Anweisung nicht alle im Paket enthaltenen Module in den aktuellen Namensraum, sondern importiert nur das Paket an sich und führt den Initialisierungscode in `__init__.py` aus. Sowohl alle in dieser Datei angelegten Elemente als auch alle Elemente von eventuell vorher importierten Modulen dieses Pakets werden in den aktuellen Namensraum eingeführt.

Es gibt zwei Möglichkeiten, das gewünschte Verhalten der obigen Anweisung zu erreichen. Beide müssen vom Autor des Pakets implementiert werden.

Zum einen können alle Module des Pakets innerhalb der `__init__.py` per `import`-Anweisung importiert werden. Dies hätte zur Folge, dass sie beim Einbinden des Paketes und damit nach dem Ausführen des Codes der `__init__.py`-Datei eingebunden wären.

Zum anderen kann dies durch Anlegen einer Referenz namens `__all__` geschehen. Diese muss eine Liste von Strings mit den zu importierenden Modulnamen referenzieren:

```
__all__ = ["blur", "flip", "rotate"]
```

Es liegt im Ermessen des Programmierers, welches Verhalten `from abc import *` bei seinen Paketen zeigen soll. Beachten Sie aber, dass das Importieren des kompletten Modul- bzw. Paketinhalts in den aktuellen Namensraum zu unerwünschten Namenskonflikten führen kann. Aus diesem Grund sollten Sie importierte Module stets in einem eigenen Namensraum führen.

11.3.2 Relative Importanweisungen

Bei der Verwendung der klassischen `import`-Anweisung, die bislang besprochen wurde, kann es, wie bereits angemerkt wurde, zu Namenskonflikten zwischen der Standardbibliothek und einem lokalen Modul oder Paket kommen. Wenn

also beispielsweise ein lokales Modul namens `math` existiert, gibt es keine saubere Möglichkeit mehr, das gleichnamige Modul der Standardbibliothek einzubinden.

Um solchen Konfliktfällen entgegenzuwirken, soll sich das Verhalten der `import`-Anweisung in künftigen Python-Versionen dahingehend ändern, dass im Konfliktfall das globale Modul bzw. Paket eingebunden wird. Das explizite Einbinden eines lokalen Moduls oder Pakets geschieht über eine sogenannte *relative Importanweisung*. Auf diese Weise können Namenskonflikte vermieden werden.

Seit Version 2.5 von Python sind relative Importanweisungen in die Sprache aufgenommen worden. Das Verhalten der normalen `import`-Anweisung wurde jedoch nicht geändert. Allerdings ist es möglich, Python 2.5 in einen Modus zu schalten, in dem sich die `import`-Anweisung nach den zukünftigen Regeln verhält. Dazu muss folgende Codezeile zu Beginn der jeweiligen Programmdatei geschrieben werden:

```
from __future__ import absolute_imports
```

Näheres zum Modul `__future__` und seiner Bedeutung erfahren Sie in Abschnitt 13.7.

Eine relative Importanweisung wird folgendermaßen geschrieben:

```
from . import math
```

Als Basis für die relative Importanweisung dient die `from/import`-Anweisung, bei der ein Punkt für das aktuelle Paket steht. Im Beispiel würde also das lokale Modul `math` eingebunden, das sich im selben Paket befindet wie das Modul, in dem die relative Importanweisung steht.

Um ein Modul oder Paket einzubinden, das sich in einem Unterpaket befindet, wird folgende relative Importanweisung verwendet:

```
from .math import trig
```

Dieses Mal wurde vorausgesetzt, dass ein Unterpaket `math` existiert, in dem es ein Modul oder Paket namens `trig`, beispielsweise für trigonometrische Funktionen, gibt.

Mithilfe einer relativen Importanweisung ist es auch möglich, ein Modul oder Paket einzubinden, das sich in einem Paket befindet, das in der Pakethierarchie höher angeordnet ist. Im folgenden Beispiel wird das Modul `trig` des direkt übergeordneten Pakets `math` eingebunden, indem dem Paketnamen ein weiterer Punkt vorangestellt wurde:

```
from ..math import trig
```

Beachten Sie bei der Verwendung von relativen Importanweisungen, dass diese nur innerhalb einer Paketstruktur verwendet werden dürfen. Eine Programmdatei, die direkt ausgeführt wird, befindet sich nicht in einer Paketstruktur und darf deshalb keine relativen Importanweisungen enthalten. Dies bezieht sich insbesondere auch auf den interaktiven Modus.

11.4 Built-in Functions

Es existieren zwei Built-in Functions, die sich auf Modularisierung, also auf das Einbinden von Modulen und Paketen beziehen. Diese Funktionen wurden in Abschnitt 10.7, »Vordefinierte Funktionen«, nicht erläutert, da das Konzept der Modularisierung Ihnen zu diesem Zeitpunkt noch nicht bekannt war. Aus diesem Grund soll die Beschreibung der Built-in Functions `__import__` und `reload` an dieser Stelle nachgeholt werden. Beachten Sie, dass diese beiden Funktionen nur in wenigen Fällen benötigt werden und daher an dieser Stelle nur oberflächlich erläutert werden. Ausführliche Informationen über die Funktionen finden Sie in der Python-Dokumentation.

`__import__(name[, globals[, locals[, fromlist[, level]]])`

Die Built-in Function `__import__` wird von der `import`-Anweisung verwendet, um ein Modul oder Paket einzubinden. Die Funktion existiert hauptsächlich, damit sie vom Programmierer überschrieben werden kann, um das Verhalten der `import`-Anweisung zu verändern. Zum Überschreiben der Funktion muss eine neue Funktion mit gleicher Schnittstelle erstellt und dem Namen `__import__` zugewiesen werden.

Die Funktion `__import__` bindet das Modul oder Paket *name* ein und gibt den erzeugten Namensraum zurück. Dabei kann für *globals* und *locals* jeweils ein Dictionary übergeben werden, das alle Referenzen des globalen bzw. lokalen Namensraums enthält. Ein solches Dictionary wird von den Built-in Functions `globals` und `locals` erstellt. Für den vierten Parameter, *fromlist*, kann eine Liste mit Namen übergeben werden, die aus dem Modul *name* eingebunden werden sollen. Der fünfte Parameter, *level*, gibt an, ob absolutes oder relatives Importverhalten verwendet werden soll (vgl. Abschnitt 11.3.2). Der voreingestellte Wert von `-1` weist die Funktion `__import__` dazu an, sowohl absolutes als auch relatives Importverhalten zu zeigen. Ein Wert von `0` würde absolutes Importverhalten vorschreiben, während ein positiver Wert größer null die Anzahl der übergeordneten Verzeichnisse festlegt, die beim relativen Importverhalten mit 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)
```

reload(module)

Mithilfe der Funktion `reload` kann ein bereits eingebundenes Modul oder Paket erneut geladen und initialisiert werden. Das ist besonders dann sinnvoll, wenn ein selbst geschriebenes Modul in einer Sitzung im interaktiven Modus eingebunden ist und während der Sitzung Änderungen am Quelltext des Moduls durchgeführt wurden. In einem solchen Fall kann die Funktion `reload` dazu verwendet werden, das Modul neu einzubinden, ohne die Sitzung im interaktiven Modus beenden zu müssen. Beachten Sie, dass ein Modul nicht mit einer erneuten `import`-Anweisung neu eingebunden werden kann, da diese zuerst überprüft, ob das Modul bereits eingebunden und initialisiert wurde.

Allgemein ist von der Verwendung von `reload` vor allem im Quelltext eines Programms, also außerhalb des interaktiven Modus, abzuraten, da das Neuinitialisieren eines Moduls Probleme hervorrufen kann. Diese Probleme sind zum Teil sehr speziell und sollen hier nicht weiter erläutert werden. Eine genaue Beschreibung der möglichen Probleme finden Sie in der Python-Dokumentation.

Index

`__debug__` 314
`__future__` 307

A

ABC 27
Accessibility 654
ACP 736
Adaption 488
Administrationsoberfläche 736
Alpha-Blending 680
and 45
Anonyme Funktionen 193
Anti-Aliasing 681
Anweisung 47
Anweisungskopf 47
Anweisungskörper 47
Argument 178, 415
Arithmetischer Ausdruck 43
Arithmetischer Operator 43
as 218
ASCII 137
assert 313
Attribut 233, 239, 452
Automatisierter Test 598

B

Backslash 314
Basisdatentypen 79
 bool 92
 complex 97
 dict 146
 float 90
 frozenset 155, 163
 int 85
 list 110
 long 85
 set 155, 161
 str 122
 tuple 120
 unicode 122, 140
Basisklasse 250
Baum 454
BDFL 28

Behindertengerechte Oberflächen 654
Betaverteilung 329
Bezeichner 43
Beziérkurve 682
Bibliothek 217
Big-Endian 408
Bildschirmausgabe 45
Binárdistribution 629
Binársystem 87
Bindigkeit 80
Bindings 649
Bit-Operationen 87
 Bitverschiebung 90
 Bitweises ausschließendes ODER 89
 Bitweises Komplement 89
 Bitweises ODER 88
 Bitweises UND 88
Blockierender Socket 515
Blockkommentar 52, 289
Boolesche Werte 92
Boolescher Ausdruck 93
Boolescher Operator 45
break 62
Breakpoint 581
Brush 672
Bubblesort 764
Buchstabe 104
Bug 579
Built-in Functions 102
Built-in-Funktionen 195
Busy Waiting 522
Byte-Code 28, 36

C

C 761
Call Stack 285
Callback-Funktion 465
Call-by-Reference 188
Call-by-Value 188
Callstack 606
Case sensitive 43
Children 454, 655
class 235
Client 508, 510

Client/Server-System 492, 508
 Codepage 137
 Codepoint 140
 Compiler 28, 36
 Conditional Expression 59
 continue 63
 Control 649
 Critical Section 434, 527
 CSV 498
 CSV-Dialekt 498
 Cursor 477

D

Dämon-Thread 445
 Dateiähnliches Objekt 529
 Dateien 168, 169
 Dateiobjekt 169, 174, 529
 Datenbank 474
 Datenstrom 168
 Datentyp 71
 Deadlock 437
 Debugger 579
 Debugging 579
 def 179
 del 76, 111, 149
 Delegate 688
 Deserialisieren 494
 Destruktor 238
 Dezimalsystem 86
 Dialog 655
 Dictionary 146
 Differenzmenge 159
 Distribution 621
 Django 720
 Django-Applikationen 723, 727
 Django-Projekt 723
 DLL 762
 Docstring 289, 635
 Document Object Model → DOM 453
 DOM 453
 Drittanbieterbibliotheken
 MySQLdb 492
 py2exe 631
 DST 376
 Dualsystem 87
 Dynamic Link Library 762
 Dynamische Website 719

E

Echte Teilmengen 158
 Eclipse 799
 Eingabeaufforderung 415
 Eingebettete Skriptsprache 777
 Einrückung 48
 Eins-zu-eins-Relation 730
 Eins-zu-viele-Relation 729
 Einwegfunktion 370
 Eleganz 641
 ElementTree 469
 elif 56
 else 57, 61, 67, 281
 Elternelement 454
 E-Mail-Header 560
 Embedded Script Language 778
 Encoding 787
 Encoding-Deklaration 145
 Entwicklungsumgebung 31, 799
 Entwicklungswebserver 725
 Epytext 636
 Eric 652, 800
 Erweiterte Zuweisung 83
 Escape-Sequenzen 124, 346
 ESMTTP 545
 Event 446, 665, 699
 Eventhandler 665
 except 279
 Exception Handling 273
 Exceptions 274, 410, 773
 exec 304
 Exit Code 413
 Exponent 91
 Exponentialschreibweise 90
 Exponentialverteilung 329
 Extension 622, 769
 Eye-Candy 679

F

False 44, 92
 Farbverlauf 679
 Fenster 649
 Field Lookup 735
 File Transfer Protocol 537
 finally 281
 Flag 352
 for 65, 287

Formatierungsoperator 133
 Frame-Objekt 589
 from 218
 FTP 537
 Function Decorator 311
 Funktion 177
 Funktionsaufruf 178
 Funktionsiteratoren 303
 Funktionsobjekt 183
 Funktionsschnittstelle 178

G

Gammaverteilung 329
 Ganze Zahlen 40, 85
 Gaußverteilung 329, 330
 Generator Expression 294
 Generatoren 290
 Geplantes Sprachelement 306
 Geschwisterelement 454
 GET 757
 Getter-Methode 242
 Gleichverteilung 329
 Gleitkommazahlen 41, 90
 global 192
 Globale Referenz 191
 Globale Variable 429
 Globaler Namensraum 191
 GNU gettext API 366
 Grafische Benutzeroberfläche → GUI 649
 Gruppe 348
 Gtk 650
 GUI 649
 Guido van Rossum 27

H

Hash-Funktion 369
 Hash-Wert 147, 369
 Hauptdialog 655
 Hexadezimalsystem 86
 Hook 411
 HTTP 567

I

I18N 365
 Identität 74
 IDLE 31, 580, 799

if 56, 288
 Imaginärteil 97
 IMAP4 553
 immutable 77
 import 217, 224, 307
 in 104, 149, 158
 in place 115
 Index 106
 inf 91, 334
 Inline Markup 639
 Inlining 644
 Installation 31
 Installationsskript 624
 Instanz 69, 235
 Integer-Division 40
 Interaktiver Modus 31, 39
 Interface 240
 Internationalisierung 365
 Interpreter 28, 37
 IP-Adresse 507
 is 75
 ISO-Kalender 385
 Iterator 296
 Iterator-Protokoll 296
 Iterierbares Objekt 66
 Iterieren 67

J

Join 483

K

Keyword Arguments 184
 Kindelement 454
 Klasse 234
 Klassen-Member 247
 Knoten 453
 Kollision 370
 Kommandozeilen-Interpreter 791
 Kommandozeilenparameter 415
 Kommentar 51
 Kommunikationssocket 508
 Komodo IDE 801
 Kompilat 36
 Komplexe Zahlen 97
 Konsole 415
 Konsolenanwendung 34
 Konstruktor 237

Kontextmanager 308
 Kontrollstruktur 55
 Konvertierung 489
 Koordinatensystem 672
 Koordinierte Weltzeit 375
 Körperloses Tag 452

L

L10N 366
 lambda 193
 Laufzeitmessung 609
 Laufzeitmodell 69
 Laufzeitoptimierung 641
 Laufzeitverhalten 609
 Layout 658
 Lazy Evaluation 60
 Leichtgewichtprozess 429
 Library 217
 List Comprehension 287
 Listen 110
 Little Endian 408
 Locking 432, 439
 Lock-Objekt 434, 439
 Logarithmische Normalverteilung 330
 Logdatei 592
 Logger 592
 Logging Handler 596
 Logische Operatoren 93
 Logische Negierung 93
 Logisches ODER 93
 Logisches UND 93
 Logischer Ausdruck 44
 Lokale Funktionen 193
 Lokale Referenz 191
 Lokalen Variablen 429
 Lokaler Namensraum 191
 Lokalisierung 365
 Lokalzeit 375
 Lookup 645
 Loose Coupling 724

M

Magic Line 35
 Magic Members 257
 __abs__ 268
 __add__ 265
 __and__ 266

Magic Members (Forts.)

 __call__ 263, 312
 __cmp__ 262
 __complex__ 268
 __contains__ 271
 __del__ 238, 257
 __delattr__ 259
 __delitem__ 270
 __dict__ 258
 __div__ 266
 __doc__ 290
 __enter__ 309
 __eq__ 262
 __exit__ 309
 __float__ 268
 __floordiv__ 266
 __ge__ 262
 __getattr__ 258
 __getattribute__ 258
 __getitem__ 270, 302
 __gt__ 262
 __hash__ 263
 __hex__ 268
 __iadd__ 267
 __iand__ 267
 __idiv__ 267
 __ifloordiv__ 267
 __ilshift__ 267
 __imod__ 267
 __imul__ 267
 __index__ 268
 __init__ 237, 257
 __int__ 268
 __invert__ 268
 __ior__ 268
 __ipow__ 267
 __irshift__ 267
 __isub__ 267
 __iter__ 270, 296
 __ixor__ 268
 __le__ 262
 __len__ 270
 __long__ 268
 __lshift__ 266
 __lt__ 262
 __mod__ 266
 __mul__ 266
 __ne__ 262
 __neg__ 268

Magic Members (Forts.)

- `__nonzero__` 263
- `__oct__` 268
- `__or__` 266
- `__pos__` 268
- `__pow__` 266
- `__radd__` 266
- `__rand__` 267
- `__rdiv__` 266
- `__repr__` 257
- `__rfloordiv__` 266
- `__rshift__` 266
- `__rmod__` 266
- `__rmul__` 266
- `__ror__` 267
- `__rpow__` 266
- `__rrshift__` 267
- `__rshift__` 266
- `__rsub__` 266
- `__rxor__` 267
- `__setattr__` 259
- `__setitem__` 270
- `__slots__` 259
- `__str__` 258, 284
- `__sub__` 265
- `__unicode__` 263
- `__xor__` 266

Main Event Loop 664

Managed Attribute 244

Mantisse 91

Many-To-Many Relation 730

Matching 341, 362

Match-Objekt 354, 359

MD5 371

Mehrfachvererbung 253

Member 233

- Private* 241
- Protected* 241
- Public* 241

Memory Leak 777

Menge 155

Method Table 771

Methode 100, 233

Methodendefinition 236

MFC 649

MIME 559

Modaler Dialog 705

Model-API 731

Modell 723, 728

Modellklasse 683

Model-View-Konzept 654, 683, 723, 727

Modul 217, 623

Modularisierung 217

Multicall 574

Multiplexender Server 508, 522

Multitasking 427

mutable 77

MySQL 492

N

nan 92, 334

Netzwerk-Byte-Order 521

New-Style Classes 251

Nicht-blockierender Socket 515

Nicht-modaler Dialog 705

Node 453

None 82

NoneType

- Basisdatentypen* 81

Normalverteilung 330

not 44, 93

not in 104, 149, 158

Numerische Datentypen 82

O

object 250

Objekt 229, 233

Objektorientierung 229

Oktalsystem 86

Old-Style Classes 251

One-To-Many Relation 729

One-To-One Relation 730

Operand 79

Operator 40, 79

Operatorrangfolge 80, 803

Optimierung 641

Option 415

Optionale Parameter 100, 183

or 45

Ordnungsrelation 109

OSI-Schichtenmodell 505

Out-of-Band Data 523

P

Painter 671
 Painter Path 682
 Paket 221, 623
 Parallele Programmierung 427
 Paralleler Server 508
 Parameter 100, 178
 Parent 454
 Pareto-Verteilung 330
 Parser 452
 pass 68
 Passworteingabe 790
 PDB 579
 Pen 671
 Pfad 397
 Pipe 349
 Plattformunabhängigkeit 28
 POP3 548
 Port 507
 Positional Arguments 184
 POST 757
 Post Mortem Debugger 582
 Primzahl 437
 print 45, 165
 Privater Member 240
 Profiler 612
 Programm 427
 Programmdatei 34
 Programmierparadigma 28
 Protokollebene 505
 Prozess 427
 PSF-Lizenz 29
 PyDev 799
 PyGnome 650
 PyGtk 650
 PyQt 651
 Python 2.6 796
 Python 3000 795
 Python API 29, 770
 Python Shell 31
 Python Software Foundation 28
 PYTHONPATH 722
 Python-Website 31
 pyuic4 663

Q

QApplication 664
 QCheckBox 700
 QComboBox 701
 QDateEdit 702
 QDateTimeEdit 703
 QDial 704
 QDialog 664, 704
 QGLWidget 705
 QLineEdit 706
 QListView 707
 QListWidget 707
 QPainter 671
 QPen 671
 QProgressBar 708
 QPushButton 709
 QRadioButton 709
 QScrollArea 710
 QSlider 711
 Qt 651
 Qt Designer 656
 QTableView 711
 QTableWidget 712
 QTabWidget 713
 QTextEdit 713
 QTimeEdit 714
 QTreeView 715
 QTreeWidget 715
 Quantor 343
 Genügsamer Quantor 347
 Quelltext 34
 Query 474
 Queue 443, 509
 QWidget 716

R

raise 277
 Rapid Prototyping 29
 Raw-String 125
 Reader 465
 Realteil 97
 Reference Count 76, 413, 776
 Referenz 69
 Referenzzähler 76
 Regulärer Ausdruck 340
 Regular-Expression-Objekt 353, 356
 Rekursion 194

Rekursionstiefe 194
 Relationale Datenbank 474
 Relative Importanweisungen 224
 RE-Objekt 353, 356
 Request Handler 525
 Reservierte Wörter 803
 return 181
 ROT13 774
 RPM 629
 Rückgabewert 102, 178

S

SAX 464
 Schlafender Thread 428
 Schleife 60
 Schleifenkörper 60
 Schleifenzähler 65
 Schlüssel/Wert-Paar 146
 Schlüsselwörter 43
 Liste reverbierter Wörter 803
 Schlüsselwortparameter 101, 184
 Schnittmenge 159
 Schnittstelle 240
 Schrittweite 108
 Searching 341, 361
 Seiteneffekte 188
 self 236
 Semikolon 315
 Sequenzielle Datentypen 102
 Serialisieren 494
 Serieller Server 508
 Server 508
 Setter-Methode 243
 SHA-1 371
 SHA-224 371
 SHA-256 371
 SHA-384 371
 SHA-512 371
 Shared Object 762
 Shebang 35
 Shell 415
 Shortcut-Funktion 746
 Sibling 454
 Signal 653, 665
 Simple API for XML 464
 Slicing 107
 Slot 653, 665
 SMTP 545
 Socket 507
 Socket API 507
 Spacer 660
 Speicherzugriffsfehler 777
 Splitter 659
 Sprachkompilat 368
 SQL 474
 SQL Injection 480
 SQL-Statements
 CREATE 478
 INSERT 479
 SELECT 482
 SSH 564
 Stabiles Sortierverfahren 117
 Standardbibliothek 28, 317
 atexit 425
 cmath 319
 cmd 791
 codecs 787
 copy 420
 cProfile 612
 cStringIO 373
 csv 498
 ctypes 762
 datetime 381
 decimal 331
 distutils 621
 doctest 598
 email 559
 epydoc 633
 ftplib 537
 functools 789
 getpass 790
 gettext 365
 gzip 449
 hashlib 369
 imaplib 553
 inspect 582
 logging 592
 math 319
 optparse 415
 os 395
 os.path 403
 pickle 494
 platform 414
 poptlib 548
 pprint 590
 profile 612
 random 325

Standardbibliothek (Forts.)

- re* 340, 352
- select* 522
- shutil* 423
- SimpleXMLRPCServer* 568
- SMTP* 545
- socket* 507
- SocketServer* 525
- sqlite3* 477
- string* 337
- StringIO* 373
- sys* 407
- telnetlib* 564
- tempfile* 503
- thread* 430
- threading* 437
- time* 375
- timeit* 609
- Tkinter* 650
- trace* 616
- traceback* 606
- unittest* 602
- urllib* 529
- urlparse* 534
- webbrowser* 788
- xml* 451
- xml.dom* 456
- xml.etree.ElementTree* 469
- xmlrpclib* 571

Statischer Member 246

Steuerelement 649

Steuerzeichen 124

Stream 168

String 41

String Conversions 316

Strings 122

Subklasse 250

SVN-Repository 721

Symmetrische Differenzmenge 160

Syntax 47, 341

Syntaxanalyse 452

T

- Tabulatorreihenfolge 661
- Tag 451, 751
- Tag-Name 451
- Tastatureingaben 166
- TCP 512

- Teilmenge 158
- Telnet 564
- Template 747
- Template-System 339
- Templatevererbung 752
- Temporäre Datei 503
- Term 40
- Terminator 303
- Thread 429
- Threads 427
- Tk 650
- Toolkit 649
- Toplevel-Tag 461
- Traceback 274, 412
- Traceback-Objekt 278, 310, 410, 606
- Tracer 616
- Transformation 682
- Transformationsmatrix 682
- Transmission Control Protocol 512
- Transparenz 680
- Trolltech 651
- True 44, 92
- try 279
- Tupel 121
- Tuple Packing 121
- Tuple Unpacking 121
- type 72

U

- Überdeckungsanalyse 616
- Überladen 261
- UDP 510
- Umwandlungsflag 134
- Unicode 140
- Unit 598
- Unit Test 602
- Unix-Epoche 375
- Unix-Timestamp 375
- URL 529, 534, 788
- User Datagram Protocol 510
- UTC 376

V

- Variable 42
- Variablentyp 49
- Verbindungssocket 508
- Vereinigungsmenge 159

Vererbung 249
 Vergleich 44
 Vergleichsoperatoren 44, 84
 Viele-zu-viele-Relation 730
 View 723, 728, 741
 Viewklasse 683
 Virtuelle Maschine 37
 Von-Mises-Verteilung 330

W

Wahlfreier Zugriff 453
 Wahrheitswert 44
 Wallis'sches Produkt 430
 Weibull-Verteilung 330
 Wert einer Instanz 72
 while 60
 Whitespace 125, 345, 636
 Widget 649, 655, 700
 Widgets
 Button 709
 Check Box 700
 Combo Box 701
 DateEdit 702
 DateTimeEdit 703
 Dial 704
 Dialog 704
 Fortschrittsbalken 708
 LineEdit 706
 List 707
 ListView 707
 OpenGL 705
 Radio Button 709
 ScrollArea 710
 Slider 711
 Tab 713

Widgets (Forts.)
 Table 712
 TableView 711
 TextEdit 714
 TimeEdit 714
 Tree 715
 TreeView 715
 Widget 716
 Wing IDE 803
 with 307
 Worker-Thread 443
 Wrapper 366
 Wurzel 454

X

XML 451, 567
 XML-RPC 567

Y

yield 292

Z

Zahlensysteme 86
 Zählschleife 65
 Zeichen 123
 Zeichenkette 41
 Zeichenklasse 342, 345
 Zeichenliteral 341
 Zeichnen 670
 Zeilenkommentar 51
 Zeitscheibe 428
 Zope 531
 Zukunft 795
 Zuweisung 42