

Michael Weigend

Python

8. Auflage

GE-PACKT



- Schneller Zugriff auf Module, Klassen und Funktionen
- tkinter, Datenbanken, OOP und Internetprogrammierung
- Für die Version Python 3.8

Inhaltsverzeichnis

E	Einleitung	13
E.1	Was ist Python?	13
E.2	Einige besondere Merkmale von Python	13
E.3	Python 2 und 3	14
E.4	Hinweise zum Lesen dieses Buches	15
1	Basiskonzepte von Python	19
1.1	Python im interaktiven Modus	19
1.2	Ausführung von Python-Skripten	21
1.3	Die Zeilenstruktur	23
1.4	Deklaration der Codierung	26
1.5	Bezeichner (identifiers)	26
1.6	Objekte	28
1.7	Die Standard-Typ-Hierarchie	32
1.8	Literale für einfache Datentypen	33
1.9	Namensräume – lokale und globale Namen	39
2	Sequenzen	43
2.1	Gemeinsame Operationen für Sequenzen	43
2.2	Zeichenketten (Strings)	46
2.3	Tupel	52
2.4	Listen	53
2.5	Performance-Tipps	69

3	Dictionaries	73
4	Mengen	83
4.1	Der Typ set	83
4.2	Der Typ frozenset	84
4.3	Gemeinsame Operationen für set- und frozenset-Objekte	85
4.4	Mengen verändern	89
5	Operatoren	91
5.1	Unäre arithmetische Operatoren + - ~	92
5.2	Binäre arithmetische Operatoren + - * / % **	93
5.3	Bit-Operatoren << >> & ^ 	96
5.4	Vergleiche < <= > >= != ==	98
5.5	Zugehörigkeit (in, not in)	100
5.6	Identitätsvergleich (is, is not)	101
5.7	Logische Operatoren (not, and, or)	102
6	Einfache Anweisungen (Statements)	105
7	Kontrollstrukturen	123
7.1	Verzweigungen – die if-Anweisung	123
7.2	Bedingte Ausdrücke	125
7.3	Verzweigungen mit logischen Operatoren	125
7.4	Iterationen – die for-Anweisung	127
7.5	Schleifen mit Abbruchbedingung – while	131
7.6	Abfangen von Laufzeitfehlern – try	133
7.7	Kontrollierte Ausführung – with	137
8	Definition von Funktionen	141
8.1	Aufruf und Ausführung einer Funktion	142
8.2	Funktionsnamen als Parameter	144
8.3	Voreingestellte Parameterwerte	145

8.4	Schlüsselwort-Argumente	146
8.5	Funktionen mit beliebiger Anzahl von Parametern ...	147
8.6	Funktionsannotationen: Typen zuordnen	148
8.7	Positionsargumente und Schlüsselwortargumente erzwingen (/ , *)	150
8.8	Prozeduren.....	151
8.9	Rekursive Funktionen	152
8.10	Funktionen testen mit dem Profiler	152
8.11	Lokale Funktionen.....	154
8.12	Generatorfunktionen.....	155
8.13	Lambda-Formen.....	158
8.14	Decorators	159
9	Standardfunktionen (built in functions) und Standardtypen	163
10	Fehler und Ausnahmen	205
10.1	Syntaxfehler.....	205
10.2	Ausnahmen (Exceptions)	206
10.3	Erstellen einer eigenen Exception-Klasse	210
10.4	Testen von Vor- und Nachbedingungen mit assert ...	215
10.5	Selbstdokumentation im Debugging-Modus	216
10.6	Das Modul logging	218
11	Ein- und Ausgabe	229
11.1	Interaktive Eingabe über die Tastatur	229
11.2	Kommandozeilen-Argumente lesen.....	230
11.3	Formatierte Bildschirmausgabe	234
11.4	Lesbare Darstellung komplexer Objekte – das Modul pprint	237
11.5	Dateien	239
11.6	Objekte speichern – pickle	249
11.7	Zugriff auf beliebige Ressourcen über deren URL	255

12	Schnittstelle zum Laufzeitsystem – sys.....	257
13	Schnittstelle zum Betriebssystem – os und os.path.....	267
13.1	Das Modul os.....	267
13.2	Das Modul os.path	278
14	Datum und Zeit	285
14.1	Das Modul time	285
14.2	Das Modul datetime	291
15	Objektorientierte Programmierung mit Python	299
15.1	Definition von Klassen	300
15.2	Attribute	304
15.3	Methoden.....	307
15.4	Vererbung.....	317
15.5	Definition von Klassenbibliotheken.....	320
16	Verarbeitung von Zeichenketten	327
16.1	Standardmethoden für String-Objekte	327
16.2	Das Modul string	337
16.3	Formatierung mit dem %-Operator	340
16.4	Formatstrings.....	342
16.5	Reguläre Ausdrücke – das Modul re	347
16.6	Performance-Tipps zur Zeichenkettenbearbeitung ...	359
17	Mathematische Funktionen.....	361
17.1	array	361
17.2	cmath	364
17.3	decimal.....	365
17.4	math	374
17.5	random.....	376

17.6	statistics	384
17.7	Das Modul secrets	388
18	CGI und WSGI	393
18.1	CGI-Skripte erstellen	393
18.2	Kommunikation über HTML-Formulare	398
18.3	Die Klasse cgi.FieldStorage	401
18.4	Das Modul cgi – CGI-Skripte debuggen	406
18.5	Cookies	407
18.6	WSGI	410
18.7	Verarbeitung von Eingabedaten	412
18.8	mod_wsgi	414
19	Kommunikation im Internet	421
19.1	Das Modul ftplib	422
19.2	Erstellen eines CGI-Webserver	425
19.3	Das Modul imaplib	426
19.4	Das Modul poplib	428
19.5	Das Modul smtplib	431
19.6	Das Modul telnetlib	434
20	Datenbanken	437
20.1	Eine MySQL-Datenbank erstellen	438
20.2	Das Modul MySQLdb – Zugriff auf MySQL-Datenbanken	446
20.3	Das Modul sqlite3	451
21	Das Modul hashlib – Digitale Signaturen	455
21.1	Hashing-Objekte	456
21.2	Anwendung in der Sicherheitstechnik – Passwortgeschützte Online-Plattform	458

22	Grafische Benutzungsoberflächen	471
22.1	Widgets des Moduls tkinter	472
22.2	Die Benutzungsoberfläche als Aggregat von Widgets	473
22.3	Attribute der Widgets (Optionen)	476
22.4	Standard-Methoden der Widgets	486
22.5	Die Klasse Button	490
22.6	Die Klasse Canvas	491
22.7	Checkbox	504
22.8	Entry	508
22.9	Frame	509
22.10	Label	510
22.11	Listbox	510
22.12	Menu	513
22.13	Menubutton	522
22.14	Die Klasse PhotoImage	525
22.15	Radiobutton	526
22.16	Scale	528
22.17	Scrollbar	531
22.18	Die Klasse Text	533
22.19	Tk	541
22.20	Layout-Manager	542
22.21	Kontrollvariablen	553
22.22	Dialogboxen	554
22.23	Event-Verarbeitung	556
23	Bild und Ton	565
23.1	Der Python Package Index	565
23.2	Die Python Imaging Library (PIL)	566
23.3	Klänge mit Winsound	583
23.4	PlaySound()	585

24	Threads	587
24.1	Funktionen in einem Thread ausführen: start_new_thread()	588
24.2	Thread-Objekte erzeugen – die Klasse Thread	589
24.3	Die Klasse Timer	592
25	XML	595
25.1	Das Modul xml.dom.minidom	596
25.2	Verarbeitung eines XML-Objektes – Einführendes Beispiel.	597
25.3	Parsing – ein DOM-Objekt erstellen.	600
25.4	Knoten eines DOM-Objektes – die Basisklasse Node	601
25.5	Die Klasse Document	610
25.6	Die Klasse Element	611
25.7	Die Klasse Text.	615
A	Ressourcen im Internet	617
A.1	Usenet	617
A.2	Mailinglisten	617
A.3	WWW	618
B	Entwicklungsumgebungen	619
C	Python-Module	621
D	Von Python 2 zu Python 3	625
D.1	Unterschiede zwischen Python 2 und Python 3	625
E	Glossar.	629
	Stichwortverzeichnis	641

E

Einleitung

E.1 Was ist Python?

Python ist eine portable, interpretative, objektorientierte Programmiersprache. Ihre Entwicklung wurde 1989 von Guido van Rossum am Centrum voor Wiskunde en Informatica (CWI) in Amsterdam begonnen und wird nun durch die nichtkommerzielle Organisation »Python Software Foundation« (PSF, <http://www.python.org/psf>) koordiniert. Der Name soll an die britische Comedy-Gruppe Monty Python erinnern.

Ein Python-Programm – man bezeichnet es als Skript – ist ein Text, der von einem Interpreter ausgeführt werden kann. Weil Python-Skripte auf verschiedenen Systemplattformen (Unix, Windows, Mac OS) laufen können, bezeichnet man die Sprache als portabel. Die aktuelle Version von Python, auf die sich dieses Buch bezieht, ist Version 3.8. Sie kann von der Python-Homepage <http://www.python.org> heruntergeladen werden. Python ist kostenlos und kompatibel mit der GNU General Public License (GPL).

E.2 Einige besondere Merkmale von Python

- ▶ Die Python-Syntax ermöglicht sehr kompakte Programmtexte.
- ▶ Das Layout des Quelltextes dient nicht allein der besseren Lesbarkeit, sondern hat eine Bedeutung. So markiert das Zeilenende das Ende einer Anweisung. Anweisungsblöcke (wie z.B. das Innere einer Schleife) werden durch Einrückung festgelegt. Zeilen des Programmtextes, die

um die gleiche Anzahl von Stellen eingerückt sind, gehören zum gleichen Anweisungsblock.

- ▶ Mit Python kann man objektorientiert, imperativ und funktional programmieren.
- ▶ Im Unterschied etwa zu Pascal oder Java muss den Variablen kein Datentyp explizit zugeordnet werden. Es gibt keine Variablendeklarationen. Der Datentyp ergibt sich aus dem Kontext. Wenn es erforderlich ist, finden automatische Typkonvertierungen statt.
- ▶ Mehrere Variablen können zu Tupeln zusammengefasst werden, die in einer einzigen Zuweisung verarbeitet werden können. Die Komponenten eines Tupels werden durch Kommata getrennt. Durch diesen Mechanismus können Hilfsvariablen eingespart werden. So werden mit einer einzigen Anweisung $x, y = y, x$ die Inhalte der Variablen x und y vertauscht.
- ▶ In der Mathematik übliche Schreibweisen wurden in die Syntax übernommen. Ausdrücke wie $a < b < c$ oder verkettete Zuweisungen $a = b = c$ sind erlaubt.
- ▶ Python verwendet wenige, aber sehr mächtige Sprachkonstrukte. Auf alles Überflüssige wurde verzichtet. In dieser Hinsicht kann man Python als »minimalistisch« bezeichnen.
- ▶ Objekte besitzen einen Wahrheitswert. So haben alle Zahlen ungleich null und alle nicht leeren Zeichenketten den Wahrheitswert »wahr«.
- ▶ Langzahl-Arithmetik (Rechnen mit ganzen Zahlen beliebiger Länge) ist integriert.

E.3 Python 2 und 3

Es ist wichtig zu wissen, dass es zwei unterschiedliche Sprachvarianten von Python gibt: Python 2 und Python 3. Im Jahre 2007 erschien mit Python 3 erstmals eine Version, die mit den Vorgänger-Versionen nicht kompatibel ist. Man entschied sich zu diesem Bruch, um einige grundlegende Designschwächen von Python zu beseitigen. Python wurde noch

konsistenter, als es bereits war. Wie bei Java können nun sämtliche Unicode-Zeichen für Bezeichner und Strings verwendet werden. Man unterscheidet nun zwischen Strings als Zeichenketten und Bytestrings als Oktettenfolgen. Es gibt keine `print`-Anweisung mehr, sondern eine Funktion `print()`.

Ältere Python-2-Programme können in der Regel nicht von einem Python-3-Interpreter ausgeführt werden. Die letzte Python 2-Version ist Python 2.7. Man kann es immer noch herunterladen und auf vielen Computern findet man noch Python 2. Aber seit 2020 wird Python 2 nicht mehr weiterentwickelt und gepflegt.

Dieses Buch verwendet durchgehend die Syntax von Python 3.

E.4 Hinweise zum Lesen dieses Buches

Typographie

- Python-Quelltext, Funktions- und Variablennamen, Operatoren, Grammatikregeln, Zahlen und mathematische Ausdrücke werden in einem Zeichenformat mit fester Breite gesetzt. Beispiele:

```
x = y + 1
(x < y) and (len(liste) < 5)
```

- Bei der Darstellung des Formats eines Funktionsaufrufs sind die Argumente kursiv. Sie sind – im Unterschied zum Funktionsnamen – Metabezeichner, die nicht Buchstabe für Buchstabe so aufgeschrieben werden, wie es angegeben ist, sondern durch andere Bezeichner oder Literale ersetzt werden können. Beispiel:

```
range(zahl)
```

- Wichtige Passagen in Python-Listings, auf die im Text Bezug genommen wird, sind zuweilen fett gedruckt, um das Wiederfinden zu erleichtern.

Beispiele

Die Beispiele mit Python-Quelltext in diesem Buch kann man in drei Gruppen einteilen:

- ▶ Auszüge aus einer interaktiven Session erkennt man an dem Python-Prompt `>>>`. Diese Beispiele kann man im Shell-Fenster einer Python-Entwicklungsumgebung (z.B. IDLE) ausprobieren. Hinter dem Prompt ist die Benutzereingabe. Zeilen ohne Prompt enthalten eine System-Antwort. Beispiel:

```
>>> x = 1
>>> y = 2
>>> print(x + y)
3
```

- ▶ Beispiele für eigenständige Python-Skripte oder Module mit Funktionsdefinitionen enthalten überhaupt kein Prompt. Sie werden in einem Editorfenster erstellt und dann durch Aufruf des Interpreters oder nach Import in einer interaktiven Session getestet. Beispiel:

```
# Quicksort
def qsort(L):
    if len(L) <= 1: return L
    else:
        return qsort( [ x for x in L[1:] if x < L[0]]\
            + [L[0]] \
            + qsort( [y for y in L[1:] if y >= L[0]] )
```

- ▶ Für Aufrufe in einem Konsolenfenster des Betriebssystems (z.B. Unix-Shell oder MS-DOS-Eingabeaufforderung) verwenden wir das Prompt `>`. Beispiel:

```
> python addiere.py 1 27
28
```

Hinweise zum Aufbau der Kapitel

- ▶ Am Ende eines Unterkapitels mit einer Funktionsbeschreibung finden sich häufig Verweise auf andere Kapitel mit korrespondierendem Inhalt (»Siehe auch: ...«)
- ▶ Kapitel, in denen Module beschrieben werden, sind meist nach folgendem Schema aufgebaut:
 - ▶ Kurze Einleitung
 - ▶ Tabellarische Übersicht über die Objekte (Funktionen, Konstanten, Klassen, Methoden) des Moduls in alphabetischer Reihenfolge
 - ▶ Ausführliche Erklärung der wichtigsten Objekte mit Beispielen
 - ▶ Am Ende gelegentlich komplexere Anwendungsbeispiele
- ▶ Programmiertipps und wichtige Hinweise befinden sich in grauen Kästen.

1

Basiskonzepte von Python

1.1 Python im interaktiven Modus

Der Python-Interpreter kann in einem interaktiven Modus verwendet werden, in dem Sie einzelne Zeilen Programmtext eingeben und die Wirkung sofort beobachten können. Im interaktiven Modus können Sie mit Python experimentieren, etwa um sich in neue Programmiertechniken einzuarbeiten oder logische Fehler in einem Programm, das Sie gerade bearbeiten, aufzuspüren.

Der interaktive Python-Interpreter – die Python-Shell – kann auf verschiedene Weise gestartet werden:

1. In einem Konsolenfenster (z.B. Eingabeaufforderung bei Windows-Systemen) geben Sie das Kommando `python` ein. Damit das Betriebssystem den Python-Interpreter findet, muss der Systempfad richtig gesetzt sein. Bei einem Windows-System achten Sie bei der Installation von Python 3.8 darauf, dass auf der ersten Seite des Installationsprogramms unten die Checkbox `ADD PYTHON 3.8 TO PATH` gesetzt ist. Sie können natürlich auch nachträglich noch den Pfad setzen. Unter Windows 10 geben Sie in das Suchfeld links unten den Begriff `Systemeinstellungen` ein und drücken `ENTER`. Es erscheint eine Dialogbox mit mehreren Registerkarten. Wählen Sie die Registerkarte `ERWEITERT` und klicken Sie auf die Schaltfläche `UMGEBUNGSVARIABLEN`. Auf der nächsten Dialogseite wählen Sie die Variable `PATH` und klicken auf `BEARBEITEN`. Jetzt können Sie weitere Pfade hinzufügen.
2. Bei Windows-Rechnern klicken Sie auf den `START`-Button und klicken in der Liste Ihrer Apps im Ordner `Python` auf `Python3.8`. Es öffnet sich

ein Konsolenfenster (Eingabeaufforderung), in dem der Python-Interpreter läuft.

3. Sie öffnen die Entwicklungsumgebung IDLE, die zum Standardpaket gehört. Sie enthält neben einem Editorfenster ein eigenes Shell-Fenster, in dem man auf der Kommandozeile Python-Statements eingeben kann.

Die Python-Shell meldet sich immer mit einer kurzen Information über die Version und einigen weiteren Hinweisen. Dann folgt der charakteristische Promptstring aus drei spitzen Klammern `>>>` als Eingabeaufforderung. Hinter dem Prompt können Sie eine Anweisung eingeben und durch `[↵]` beenden. In den nächsten Zeilen kommt entweder eine Fehlermeldung, ein Funktionsergebnis oder (z.B. bei Zuweisungen) *keine* Systemantwort. Beispiel:

```
>>> 2+2  
4
```

Wichtige Tastenkombinationen

Es lohnt sich, einige wenige Tastenkombinationen zur effizienten Bedienung der Python-Shell auswendig zu lernen:

Vorige Anweisung. Mit der Tastenkombination `[Alt]+[p]` können Sie den vorigen Befehl (previous command) noch einmal in die Kommandozeile schreiben. Drücken Sie mehrmals diese Tastenkombination, werden die noch weiter zurückliegenden Anweisungen geholt.

Nächste Anweisung. Wenn Sie mehrmals auf `[Alt]+[p]` gedrückt haben, können Sie mit `[Alt]+[n]` wieder zum nächstneueren Kommando springen (next command).

Keyboard Interrupt. Mit der Tastenkombination `[Strg]+[C]` können Sie den Abbruch eines laufenden Programms erzwingen. Das ist z.B. für das Testen von Programmen mit `while`-Anweisungen wichtig, weil eventuell eine Endlosschleife vorliegt und das Programm von alleine nicht anhält.

1.2 Ausführung von Python-Skripten

Python-Programme – meist nennt man sie Skripte – sind Textdateien, die unter einem Namen mit der Extension `.py` oder unter Windows auch `.pyw` abgespeichert werden, z.B. `hello.py`. Ein Python-Skript wird von einem Python-Interpreter ausgeführt (interpretiert), der letztlich den Programmtext in maschinenbezogene Befehle überführt. Das heißt: Das Skript ist plattformunabhängig, aber für jedes Betriebssystem gibt es einen eigenen Interpreter. Um ein Python-Skript ausführen zu können, muss dem Betriebssystem auf irgendeine Weise mitgeteilt werden, welches Programm es zur Interpretation einsetzen soll. Voraussetzung für die Ausführung eines Skripts ist, dass Python installiert und Systempfade, Umgebungsvariablen usw. korrekt gesetzt sind.

Grundsätzlich kann ein Python-Skript von einer Entwicklungsumgebung (z.B. IDLE) aus gestartet werden. Darüber hinaus gibt es folgende plattformabhängigen Möglichkeiten.

Windows

Unter Windows können Sie ein Python-Skript auf zweierlei Weise ausführen:

1. Öffnen Sie ein Konsolenfenster (Eingabeaufforderung) und geben Sie das Kommando `python` gefolgt vom Pfad des Python-Skripts ein. Beispiel:

```
python meinSkript.py
```

2. Klicken Sie im Explorer-Fenster auf das Icon des Python-Skripts. Das Betriebssystem öffnet ein Konsolenfenster, in dem Ein- und Ausgaben erfolgen. Das funktioniert natürlich nur, wenn Programmnamen mit der Extension `.py` auch mit dem Python-Interpreter verbunden sind. Ist das nicht der Fall, klicken Sie das Programm-Icon mit der rechten Maustaste an und wählen im Kontextmenü die Option **EIGENSCHAFTEN**. Stellen Sie hinter **ÖFFNEN MIT:** den Verknüpfungspfad so ein, dass die Datei mit `python.exe` geöffnet wird. Nach Beendigung des

Programms wird das Fenster sofort wieder geschlossen. Das hat den Nachteil, dass die letzte Ausgabe des Programms nicht mehr gelesen werden kann. (Abhilfe bietet hier z.B. eine `input()`-Anweisung am Ende des Programms. Sie erzwingt, dass das DOS-Fenster geöffnet bleibt, bis die -Taste gedrückt ist.)

Bei Programmen mit grafischer Benutzungsoberfläche, die in einem eigenen Anwendungsfenster laufen, wird es als störend empfunden, wenn sich zuerst ein Konsolenfenster öffnet. Solche Skripte sollte man unter einem Namen mit der Extension `.pyw` abspeichern, z.B. `editor.pyw`. Dann erscheint nach dem Anklicken des Programmicons kein DOS-Fenster, sondern sofort die Applikation.

Unix

Unter Unix kann man (wie bei MS Windows) in einem Shell-Fenster (Konsole) den Python-Interpreter durch ein Kommando der folgenden Form starten:

```
python meinSkript.py
```

Außerdem gibt es bei Unix den Mechanismus der so genannten *magic line*. In der ersten Zeile des Skripts kann spezifiziert werden, mit welchem Interpreter das Skript ausgeführt werden soll. Die magic line beginnt mit der Zeichenfolge `#!/`, dahinter folgt entweder direkt der Pfad zum Python-Interpreter oder der Pfad zu einem Dictionary (`env`), in dem die Systemadministration den Pfad zum Python-Interpreter eingetragen hat. Wenn das Unix-System standardmäßig eingerichtet ist, müsste eine der beiden folgenden magic lines funktionieren:

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

Das Python-Skript mit einer magic line ist direkt ausführbar. Zum Start reicht z.B. die Eingabe des Dateinamens in der Konsole. Voraussetzung ist allerdings, dass die Zugriffsrechte entsprechend gesetzt sind, das

heißt, das executable-Bit (x) muss mit Hilfe des Unix-Kommandos `chmod` auf 1 gesetzt sein.

CGI-Skripte, die von jedermann über das Internet gestartet werden können, müssen eine magic line enthalten. Die Rechtevergabe erfolgt üblicherweise nach folgendem Muster:

```
chmod 711 meinSkript.py
```

Damit hat der Besitzer alle Rechte, die anderen dürfen die Datei nur ausführen, nicht aber lesen oder ändern.

1.3 Die Zeilenstruktur

Ein Python-Skript ist eine Folge von Anweisungen. Im Unterschied zu anderen Programmiersprachen muss bei Python eine Anweisung nicht durch ein besonderes Zeichen (wie das Semikolon bei Java) abgeschlossen werden. Ebenso gibt es keine Zeichen für Beginn und Ende eines Anweisungsblocks (wie z.B. geschweifte Klammern in Java oder `begin` und `end` in Pascal).

Das Ende einer Anweisung wird durch das Zeilenende markiert. Somit darf sich eine Anweisung nicht über mehrere Zeilen erstrecken.

Erlaubt ist:

```
summe = 1 + 2
```

Nicht erlaubt ist:

```
summe = 1
+ 2
```

Python unterscheidet aber zwischen »physischen« und »logischen« Zeilen. Eine physische Zeile endet mit einem (unsichtbaren) betriebssystemabhängigen Steuerungssymbol für den Zeilenwechsel. Bei Unix ist das das ASCII-Zeichen LF (linefeed), bei DOS/Windows-Systemen die ASCII-Zeichenfolge CR LF (carriage return und linefeed) und bei Mac OS das ASCII-Zeichen CR.

Explizites Verbinden von Zeilen

Mit Hilfe eines Backslashes \ kann man in einem Python-Skript mehrere physische Zeilen zu einer logischen Zeile verbinden. Damit ist folgender Programmtext eine gültige Anweisung:

```
summe = 1 \  
+ 2
```

Hinter dem Backslash darf aber in derselben Zeile kein Kommentarzeichen # stehen, denn ein Kommentarzeichen beendet eine logische Zeile. Nicht erlaubt ist also:

```
summe = 1 \ # Summenberechnung  
+ 2
```

Implizites Verbinden von Zeilen

Geklammerte Ausdrücke (mit normalen, eckigen oder geschweiften Klammern) sind häufig sehr lang. Sie dürfen bei Python auf mehrere physische Zeilen verteilt werden und werden implizit zu einer einzigen logischen Zeile verbunden. Beispiele:

```
wochentage = ["Sonntag", "Montag", "Dienstag",  
"Mittwoch", "Donnerstag", "Freitag", "Samstag"]  
def volumen(h,    # Höhe,  
            b,    # Breite und  
            t):   # Tiefe eines Quaders  
return h*b*t
```

Das zweite Beispiel zeigt auch Folgendes: Im Unterschied zu Zeilen, die mit einem Backslash \ explizit verbunden sind, dürfen implizit verbundene Zeilen Kommentare enthalten. Das ist auch sehr sinnvoll. Denn die Parameter einer Funktion möchte man häufig einzeln kommentieren.

Einrückungen – Anweisungsblöcke

Ein Anweisungsblock (in der Python-Dokumentation *suite* genannt) ist eine Folge von zusammengehörigen Anweisungen, z.B. das Innere einer Schleife, der Körper einer Funktionsdefinition oder ein Zweig einer if-else-Anweisung. Ein Block kann weitere Blöcke als Unterblöcke enthal-

ten. Beginn und Ende eines Blocks werden in einem Python-Skript nicht durch lesbare Symbole (geschweifte Klammern { } bei C oder Java), sondern durch eine Einrückung (indent) um eine bestimmte Anzahl von Stellen festgelegt. Beispiel:

```
a = 0
for i in range(5):
    a = a + i      # Beginn eines Blocks
    print(a)      # Ende des Blocks
print("Ende der Rechnung")
```

Die Anzahl der Leerzeichen vor dem ersten Nichtleerzeichen (Einrückungsgrad) ist beliebig. Wichtig ist allein, dass alle zusammengehörigen Anweisungen eines Blocks um exakt die gleiche Anzahl von Stellen eingerückt sind. Es ist gleichgültig, ob Sie beim Editieren Tabulatorzeichen (Tab) oder Leerzeichen verwenden. Empfohlen wird, Tabs und Leerzeichen nicht zu mischen. Der Python-Interpreter wandelt intern Tabulatorzeichen in die entsprechende Anzahl von Leerzeichen um. Am Ende eines Skripts werden alle begonnenen Blöcke vom Interpreter wieder beendet. Das folgende Beispielskript zeigt einige typische Einrückungsfehler:

```
def primzahl(n):                                #1
    teiler_gefunden = 0
    for i in range(2,n):
        if n%i == 0:                            #2
            print("keine Primzahl")
            teiler_gefunden = 1
            break                                #3
    if not teiler_gefunden:                      #4
        print("Primzahl")
```

#1: Fehler: Die erste Zeile darf nicht eingerückt sein.

#2: Fehler: Nach dem Doppelpunkt beginnt ein neuer Block, es muss eingerückt werden.

#3: Fehler: Unerwartete Einrückung. Die Zeile muss genauso weit eingerückt sein wie die Zeile davor.

#4: Fehler: Inkonsistente Einrückung. Die Anweisung gehört zum gleichen Block wie die `for`-Anweisung und muss ebenso weit eingerückt sein.

1.4 Deklaration der Codierung

Python-Skripte können in der ersten oder zweiten Zeile eine Zeile der Form

```
# -*- coding: <encoding-name> -*-
```

enthalten, die die Codierung des Textdokumentes angibt. Wenn die Deklaration der Codierung fehlt, wird `utf-8` angenommen. Beispiele:

```
# -*- coding: latin-1 -*-
import sys, os

...

#!/usr/bin/python
# -*- coding: iso-8859-15 -*-
import sys, os

...
```

Siehe auch: Kapitel 16 (String-Methoden `decode()` und `encode()`)

1.5 Bezeichner (identifiers)

Bezeichner (identifiers) sind Zeichenketten, die für die Namen von Funktionen, Klassen, Variablen usw. verwendet werden dürfen. Ein Bezeichner (häufig spricht man auch von Namen) kann beliebig lang sein, muss mit einem Buchstaben oder einem Unterstrich beginnen und ist ansonsten aus Buchstaben, Ziffern und Unterstrichen zusammengesetzt. Es wird zwischen Groß- und Kleinschreibung unterschieden. Im Unterschied zu den Vorgängerversionen lässt Python 3 auch Unicode-Zeichen zu, die Buchstaben repräsentieren, aber nicht zum ASCII-Zeichensatz gehören. Erlaubt sind also insbesondere auch deutsche Umlaute und ß. Gültige Bezeichner sind z.B. `x`, `x1`, `straßenname`, `__privat`, `Class_1`.

Empfohlen wird allerdings häufig, für Bezeichner ASCII-Zeichen zu verwenden.

Schlüsselwörter (keywords)

Die folgenden Bezeichner sind reservierte Wörter oder Schlüsselwörter von Python. Sie dürfen nicht als Bezeichner z.B. für Variablennamen verwendet werden.

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

Im Modul `keyword` gibt es eine Funktion, mit der getestet werden kann, ob ein String ein Schlüsselwort ist.

```
>>> import keyword
>>> keyword.iskeyword("with")
True
```

Verwendung von Unterstrichen

Unterstriche am Anfang und am Ende eines Bezeichners haben eine besondere Bedeutung.

Doppelte Unterstriche am Anfang und am Ende eines Bezeichners (z.B. `__init__`) kennzeichnen Objekte, die mit einem bestimmten »magischen« Verhalten verbunden sind. In Klassen z.B. gibt es immer eine Methode `__init__()`, die für die Initialisierung einer Instanz der Klasse zuständig ist (Konstruktor). Diese Methode wird aber nicht mit `__init__()` aufgerufen, sondern über den Namen der Klasse. Bezeichner mit doppelten Unterstrichen vorne und hinten verwendet man auch, um Operatoren zu überladen (siehe Kapitel 15.3).

Bezeichner, die mit zwei Unterstrichen beginnen, aber nicht mit Unterstrichen enden, werden für private Methoden und Attribute in Klassendefinitionen verwendet.

Ein einzelner Unterstrich am Anfang (z.B. `_irgendwas`) bewirkt, dass das bezeichnete Objekt durch die Anweisung `from modul import *` nicht in den Namensraum importiert wird. Bezeichner, die mit einem Unterstrich beginnen, sind also für interne Attribute und Methoden gedacht, die nur innerhalb eines Moduls verwendet werden.

Einen einzelnen Unterstrich am Ende verwendet man üblicherweise, um Namenskonflikte mit existierenden Schlüsselwörtern zu vermeiden.

Siehe auch: Kapitel 15.3

1.6 Objekte

Identität von Objekten

Python ist in einem sehr umfassenden Sinne eine objektorientierte Programmiersprache. Daten, Funktionen, Programmcode, Ausnahmen, Prozessframes und andere Sprachelemente werden durch Objekte repräsentiert. Jedes Objekt besitzt eine Identität, einen Wert und einen Typ.

Die Identität eines Objektes wird durch eine (einmalige) ganze Zahl repräsentiert, die mit der Standardfunktion `id()` abgefragt werden kann. Zwei Objekte mit gleichem Wert können unterschiedliche Identität besitzen. Mit dem Operator `is` kann ermittelt werden, ob zwei Objekte (angesprochen über Namen) dieselbe Identität besitzen.

Änderbare und unveränderbare Objekte

Bei manchen Objekten kann sich der Wert während ihrer Lebensdauer ändern. Man bezeichnet sie als änderbar (mutable). Zu den änderbaren Objekten gehören Listen. Objekte, die bei ihrer Erschaffung einen festen Wert erhalten, den sie bis zu ihrer Zerstörung behalten, bezeichnet man als unveränderbar (immutable). Zu den unveränderbaren Objekten gehö-

ren z.B. alle einfachen numerischen Datentypen (ganze Zahlen, Gleitkommazahlen etc.), Zeichenketten und Tupel.

Betrachten wir folgende Anweisungsfolge:

```
>>> a = 'ab'
>>> a = 'cd'
```

Hier wird nicht etwa der Wert eines Objektes geändert. In der ersten Zeile wird ein unveränderbares Objekt vom Typ String mit dem Wert 'ab' geschaffen und dem Namen a zugeordnet. In der zweiten Zeile wird ein *neues* String-Objekt mit dem Wert 'cd' erzeugt und (anstelle des ersten Objektes mit dem Wert 'ab') dem Namen a zugeordnet.

Das »alte« Objekt 'ab' wird übrigens durch die zweite Zuweisung nicht zerstört, sondern es ist jetzt einfach nicht mehr über den Namen a erreichbar. Erst die »garbage collection« (Müllabfuhr) des Laufzeitsystems sorgt dafür, dass nicht mehr erreichbare Objekte beseitigt werden.

Eine logische Schwierigkeit tritt bei Container-Objekten auf. Container-Objekte (z.B. Listen, Tupel) enthalten Referenzen auf andere Objekte. So enthält das Tupel ([1, 2], [3, 4]) Referenzen auf die Objekte [1, 2] und [3, 4]. Nun ist zwar ein Tupel unveränderbar, die Elemente des Tupels sind jedoch veränderbare Listen.

Typen und Klassen

Jedes Objekt gehört zu einem bestimmten Typ, der durch eine Klasse definiert ist. Der Typ kann mit der Standardfunktion `type()` ermittelt werden.

Beispiel:

```
>>> type(123)
<class 'int'>
>>> type('123')
<class 'str'>
```

Aus dem Typ ergeben sich bestimmte Eigenschaften eines Objektes. So können bestimmte Operatoren nur auf Objekte mit numerischem Typ (z.B. ganze Zahlen oder Gleitkommazahlen) angewendet werden. Der

Ausdruck $a*b + c/d$ macht nur Sinn, wenn a, b, c, d die Namen von numerischen Objekten sind.

Die Begriffe *Typ* und *Klasse* haben fast die gleiche Bedeutung. Ein Typ ist die Außenansicht einer Klasse. Seit Python 2.5 ist die Unterscheidung zwischen selbst definierten Klassen und vorgegebenen Standardtypen (built-in types) weitgehend aufgehoben.

Mit der Standardfunktion `isinstance()` kann man testen, ob ein Objekt (erstes Argument) zu einer bestimmten Klasse (zweites Argument) gehört:

```
>>> isinstance(1, int)
True
```

Wahrheitswert

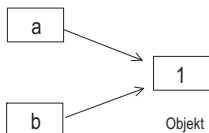
Daten-Objekte besitzen neben ihrem Wert auch einen Wahrheitswert (»wahr« oder »falsch«). Der Wahrheitswert ist eigentlich keine besondere Eigenschaft eines Objektes, sondern er ergibt sich unmittelbar aus dem Wert. Numerische Objekte mit dem Wert 0 und leere Container-Objekte haben den Wahrheitswert »falsch« und alle anderen Objekte dieser Kategorien tragen den Wahrheitswert »wahr«. Diese Konvention ermöglicht eine sehr kompakte Formulierung von Bedingungen (vgl. Kapitel 7). So ist z.B. folgende Anweisung gültiger Python-Programmtext:

```
while x:
    x = x-1
```

In der Bedingung hinter dem Schlüsselwort `while` wird der Wahrheitswert von `x` abgefragt, während im Schleifeninneren der numerische Wert verwendet wird.

Namen

Objekte sind über Namen erreichbar. Mit der Zuweisung `a = 1` wird dem Objekt 1 der Name `a` zugeordnet. Dasselbe Objekt kann mehrere Namen besitzen.



Namen

Abbildung 1.1: Variablen als Namen für Objekte

Namen für Daten werden häufig als Variablen bezeichnet. Eine Variable stellt man sich in der Informatik oft als Behälter vor, der als Inhalt ein Datenobjekt, etwa eine Zahl, besitzt (siehe Abbildung 1.2).

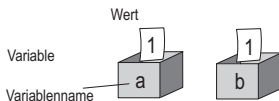


Abbildung 1.2: Variablen als Behälter für Werte

Im Falle einfacher unveränderbarer Datenobjekte (wie z.B. Zahlen) ist diese Metapher unproblematisch. Schwierigkeiten bekommt man mit dem Behälterkonzept unter Umständen bei Namen für veränderbare Objekte (z.B. Listen). Wenn nämlich der Wert eines Listenobjektes verändert wird, verweisen alle seine Namen auf das Objekt mit dem neuen Wert. Aus Sicht der Behältermetapher kann es sein, dass bei einer Änderung des Inhalts von Variable a sich »auf magische Weise« auch der Inhalt von Variable b ändert. Beispiel:

```

>>> a = [1, 2]
>>> b = a
>>> a[0] = 10
>>> print(b)
[10, 2]
  
```

Im Kapitel 2.4 gehen wir im Zusammenhang mit Listen detaillierter auf diesen Punkt ein.

1.7 Die Standard-Typ-Hierarchie

In der Standard-Typ-Hierarchie werden die Typen von Objekten folgendermaßen strukturiert (in Klammern die Python-Typbezeichnung):

- ▶ **None.** Von diesem Datentyp gibt es nur ein einziges Literal, nämlich den Wert `None`. Besitzt ein Objekt den Wert `None`, so wird damit zum Ausdruck gebracht, dass es keinen Wert besitzt.
- ▶ **Zahlen**
 - ▶ ganze Zahlen (`int`) beliebiger Länge, z.B. 12, 1234425562, -3
 - ▶ Gleitkommazahlen (`float`), z.B. 12.34, 1.3E-12
 - ▶ komplexe Zahlen (`complex`), z.B. $2.0 + 1.2j$
 - ▶ Wahrheitswerte (`True` und `False`) können auch numerisch interpretiert werden (1 und 0) und werden bei Python deshalb den Zahlen zugerechnet (`bool`)
- ▶ **Sequenzen** sind endliche Folgen von Objekten, deren Elemente durch nichtnegative ganze Zahlen indiziert sind. Wenn eine Sequenz die Länge n hat, so werden die Elemente mit den Indexen 0, 1, ..., als $(n-1)$ gekennzeichnet. Das i -te Item der Sequenz a wird mit $a[i]$ selektiert.
 - ▶ Zeichenketten (`str`), z.B. "Hallo", 'Python'
 - ▶ Bytefolge (`bytes`), z.B. `b'abc'`
 - ▶ Tupel (`tuple`), z.B. (1, 2, 3)
 - ▶ Listen (`list`), z.B. [1, "Hallo", [1, 2]]
- ▶ **Mengen** (`set`, `frozenset`), z.B. {1, 2}
- ▶ **Zuordnungen** (`Mappings`) sind endliche Mengen von Objekten, die durch (beinahe) willkürliche Werte indiziert werden. Das (momentan) einzige Mapping unter den Standard-Typen ist das Dictionary (`dict`). Beispiel: {'a':1, 'b':2, 'c':3}
- ▶ **Aufrufbare Typen** (`callable types`)
 - ▶ Funktionen (`function`)
 - ▶ Methoden (`instancemethod`)
 - ▶ Klassen (`class`)

- Module
- Klassen
- Instanzen von Klassen (instance)
- Dateien (file)

1.8 Literale für einfache Datentypen

Literale sind Zeichenfolgen, die Daten repräsentieren. Sie sind also potenzielle Inhalte von Variablen. Beispiele für Literale sind 124, 1.45, "Python". Literale kann man Datentypen zuordnen.

In vielen Programmiersprachen wie z.B. Java oder Pascal muss für eine Variable im Rahmen einer Variablendeklaration ein Datentyp festgelegt werden. Python kennt keine expliziten Typisierungen. Bei der Verarbeitung von Literalen muss das System aber wissen, um welchen Typ es sich handelt, denn manche Operatoren repräsentieren unterschiedliche Funktionen, je nachdem auf welche Datentypen sie angewandt werden. So bewirkt der Plusoperator + bei Zahlen eine Addition, bei Listen und Zeichenketten dagegen eine Konkatenation.

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> 'Kaffee' + 'pause'
'Kaffeepause'
```

Im Falle von Zahlen hängt der Typ des Rechenergebnisses davon ab, ob ganze Zahlen (int) oder Gleitkommazahlen (float) addiert werden.

```
>>> 1.0 + 2.0
3.0
>>> 1 + 2
3
```

Damit die Typzuordnung gelingen kann, muss man dem Literal »ansehen« können, um welchen Datentyp es sich handelt. Deshalb gibt es bei Python strenge Regeln für den syntaktischen Aufbau der Literale eines Datentyps.

None

None ist ein Literal, das das leere Objekt bezeichnet. Besitzt eine Variable den Wert None, wird damit zum Ausdruck gebracht, dass sie keinerlei Wert beinhaltet. Das klingt paradox, ist aber manchmal ganz praktisch. So ist es z.B. möglich, Funktionen mit eingeschränktem Definitionsbereich zu definieren, ohne Laufzeitfehler zu riskieren. Eine solche Funktion liefert den Wert None für alle Parameterwerte, die nicht in ihrem Definitionsbereich liegen.

None ist nicht etwa eine leere Menge oder die Zahl 0 oder ein leerer String ohne ein einziges Zeichen. Nein, None ist wirklich nichts. Wird einer Variablen der Wert None zugewiesen und anschließend der Inhalt ausgegeben, so erhält man keine Ausgabe.

```
>>> b = None
>>> b
>>>
```

Beim Versuch, zu None etwas hinzuzuaddieren, gibt es eine Fehlermeldung.

```
>>> b = None
>>> b = b + 1
Traceback (most recent call last):
  File "<pyshell#125>", line 1, in ?
    b = b + 1
TypeError: unsupported operand types for +: 'NoneType' and 'int'
```

Zahlen

Für die Darstellung von Zahlen gibt es bei Python vier Typen, nämlich ganze Zahlen beliebiger Länge (int), Gleitkommazahlen (float), komplexe Zahlen (complex) und Wahrheitswerte (bool). Weil die Wahrheitswerte True und False auch durch Zahlen 1 und 0 dargestellt werden können, rechnet man sie in der Python-Typhierarchie zu den ganzen Zahlen.

Ganze Zahlen

Die Syntax für Literale, die ganze Zahlen darstellen, lautet:

```
integer ::=
    decimalinteger | octinteger | hexinteger
decimalinteger ::= nonzerodigit digit* | "0"+
octinteger      ::= "0" ("o" | "O") octdigit+
hexinteger      ::= "0" ("x" | "X") hexdigit+
bininteger      ::= "0" ("b" | "B") bindigit+
nonzerodigit    ::= "1"..."9"
digit           ::= "0"..."9"
octdigit        ::= "0"..."7"
hexdigit        ::= digit | "a"..."f" | "A"..."F"
bindigit        ::= "0" | "1"
```

Die Literale für ganze Zahlen können beliebig lang sein. Die technische Grenze ist allein durch die Größe des Arbeitsspeichers gegeben. Eine ganze Zahl kann als Dezimalzahl, Oktalzahl, Hexadezimalzahl oder Binärzahl dargestellt werden. Das System gibt ganze Zahlen jedoch immer als Dezimalzahlen aus:

```
>>> 0b11001
25
>>> 0x1af
431
```

Das Vorzeichen ist übrigens nicht Teil des Zahliterals. Eine negative Zahl wie z.B. -123 wird als Term gesehen, der aus dem Minusoperator - und der Zahl 123 besteht.

Dezimalzahlen

Eine Dezimalzahl darf nicht mit einer führenden Null beginnen. Die Ziffernfolge 09 liefert eine Fehlermeldung. Allerdings sind Folgen beliebig vieler Nullen für die Zahl 0 erlaubt.

```
>>> 00000
0
```

Oktalzahlen

Oktalzahlen bestehen aus den Ziffern 0 bis 7 und müssen bei Python mit der Ziffer 0 (nicht zu verwechseln mit dem großen Buchstaben O!) gefolgt von dem Buchstaben 0 bzw. o beginnen. Beispiele für Oktalzahlen sind 0o123 und 00302330, nicht aber 0071 (zwei führende Nullen).

Hexadezimalzahlen

Das Hexadezimalsystem verwendet 16 Ziffern, die durch die üblichen Dezimalziffern 0 bis 9 und die sechs ersten Buchstaben des Alphabets dargestellt werden. Dabei repräsentiert A den numerischen Wert 10, B den Wert 11 usw. und schließlich F den Wert 15. Bei Python beginnen Hexadezimal-Literale mit dem Präfix 0x oder 0X, wobei das erste Zeichen wiederum die Ziffer null (und nicht der Buchstabe O) ist. Beispiele für Hexadezimalzahlen sind 0x10e3 oder 0XD423f2, nicht aber 0x3F (Buchstabe o am Anfang).

Im interaktiven Modus einer Python-Shell kann man sich die Dezimalwerte von Oktal- und Hexadezimalzahlen ausgeben lassen:

```
>>> 0o10
8
>>> 0x10
16
```

Binärzahlen

Analog zu Hexadezimal- und Oktalzahlen werden Literale für Binärzahlen gebildet. Sie beginnen mit einer Null, danach kommt der Buchstabe b bzw. B und eine Folge von Binärziffern, z.B. 0b101 oder 0b111.

Gleitkommazahlen

Gleitkommazahlen (floating point numbers) sind Literale zur Darstellung von Dezimalbrüchen. Die Grammatik für Gleitkommazahlen lautet:

```
floatnumber ::= pointfloat | exponentfloat
pointfloat  ::= [intpart] fraction | intpart "."
exponentfloat ::= (intpart | pointfloat) exponent
intpart     ::= digit+
```

```
fraction      ::= "." digit+
exponent      ::= ("e" | "E") ["+" | "-"] digit+
```

Eine Gleitkommazahl kann als Dezimalbruch mit einem Punkt oder in Exponentialschreibweise durch Angabe von Mantisse und Exponent dargestellt werden.

Gültige Gleitkommazahlen sind 3.14 oder .45 oder 0.23 oder 0.00012 oder 2., nicht aber 2 (Punkt fehlt).

Für rationale Zahlen, die sehr nahe bei 0 liegen oder sehr groß sind, wird die Exponentialschreibweise verwendet. Dabei wird die Zahl durch das Produkt einer rationalen Zahl m (Mantisse) mit einer Zehnerpotenz mit dem Exponenten e dargestellt:

$$z = m \cdot 10^e$$

Ein Gleitkomma-Literal in Exponentialschreibweise (`exponentfloat`) besteht aus einem Dezimalbruch oder einer ganzen Zahl (ohne Punkt) für die Mantisse, gefolgt von dem Buchstaben `e` oder `E`, einem Vorzeichen (+ oder -), das bei positiven Exponenten auch weggelassen werden kann, und schließlich einer ganzen Zahl als Exponenten.

Gültige Literale sind:

- 1.0e-10 entspricht der Zahl 0.0000000001
- 2.1E+7 entspricht der Zahl 21000000
- .2e0 entspricht der Zahl 0.2
- 001e2 entspricht der Zahl 100

Ungültig sind:

- 0x10E1 (Hexadezimalzahl als Mantisse)
- 0.1-E7 (Minuszeichen vor dem E)
- 1.2e0.3 (keine ganze Zahl als Exponent)

Man beachte, dass mehrere führende Nullen erlaubt sind. Mantisse und Exponent sind immer Dezimalzahlen (auch bei einer führenden Null) und niemals Oktal- oder Hexadezimalzahlen.

Die Genauigkeit der internen Darstellung von Gleitkommazahlen ist auf eine feste Anzahl von Stellen begrenzt. Gibt man längere Ziffernfolgen ein, so werden die letzten Stellen einfach abgetrennt.

```
>>> 1.2345678901234567890  
1.2345678901234567
```

Die begrenzte Genauigkeit der internen Darstellung erkennen Sie auch an folgendem Experiment:

```
>>> 0.6 / 3  
0.19999999999999999
```

Das Rechenergebnis, die rationale Zahl 0.2, kann durch eine Python-Gleitkommazahl nicht präziser angenähert werden.

Es gibt aber ein spezielles Modul für Dezimalarithmetik, mit der sich diese Ungenauigkeiten kontrollieren lassen (siehe Kapitel 17.3).

Imaginäre Literale

Komplexe Zahlen werden in der Mathematik als Summe aus einem Real- und einem Imaginärteil beschrieben:

```
c = a + b*i
```

Dabei bezeichnet der Buchstabe *i* die Wurzel aus -1. In der Technik verwendet man meist den Buchstaben *j* anstelle von *i*, um Verwechslungen mit dem Symbol für die Stromstärke zu vermeiden. Komplexe Zahlen spielen in der Elektrotechnik bei der mathematischen Beschreibung von Wellen eine große Rolle. Bei Python gibt es kein eigenes Literal für komplexe Zahlen, sondern nur für den Imaginärteil komplexer Zahlen. Oder anders ausgedrückt: Imaginäre Literale stellen komplexe Zahlen mit dem Realteil 0 dar. Für ihre Syntax gibt es die folgende Regel:

```
imagnumber ::= (floatnumber | intpart) ("j" | "J")
```

Das heißt, ein imaginäres Literal besteht aus einer ganzen Zahl oder Gleitkommazahl, der der Buchstabe *j* (oder *J*) angehängt ist. Beispiele sind:

```
0.3j 1.2e-3J 20j
```

Wahrheitswerte

Seit der Python-Version 2.3 gibt es einen eigenen Datentyp `bool` für logische Wahrheitswerte. Wahrheitswerte werden durch die Literale `True` (logisch »wahr«) und `False` (logisch »falsch«) repräsentiert. Achten Sie auf die Schreibweise! Beide Literale beginnen mit großen Anfangsbuchstaben. Mit Objekten vom Typ `bool` kann man logische Operationen durchführen (siehe Kapitel 5).

```
>>> a = True
>>> not a
False
>>> a or a
True
```

Darüber hinaus besitzen Objekte der Standard-Typen neben ihrem »eigentlichen« Wert auch einen Wahrheitswert. Leere Objekte tragen den Wahrheitswert `False` und nicht leere Objekte den Wahrheitswert `True`. Mit der Standardfunktion `bool()` können Sie den Wahrheitswert eines Objektes oder eines Ausdrucks ermitteln.

```
>>> bool(0)
False
>>> bool(100)
True
>>> bool(10+1-3)
True
```

1.9 Namensräume – lokale und globale Namen

In einem Namensraum werden Bindungen von Namen an Objekte definiert. Zu jedem Programmblock (Funktionskörper, Klassendefinition oder Modul) gibt es zwei Namensräume (name spaces), einen lokalen und einen globalen Namensraum. Jeder Namensraum wird durch ein Dictionary implementiert. Es ordnet Bezeichnern (Namen) Objekte zu. Die loka-

len und globalen Namensräume können mit den Kommandos `locals()` und `globals()` abgefragt werden.

Wenn Sie mit IDLE eine Python-Shell öffnen und als Erstes diese beiden Kommandos eingeben, erhalten Sie folgendes Ergebnis:

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, ...}

>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, ...}
```

Sie erkennen zunächst einmal, dass es auf dieser obersten Ebene des Hauptprogramms keinen Unterschied zwischen lokalen und globalen Objekten gibt. Die beiden Dictionaries beinhalten unter anderem folgende Namensbindungen:

- ▶ Die Variable `__name__` hat den Inhalt `'__main__'`. Das besagt, dass der Programmtext dieses Blocks direkt ausführbar ist (Hauptprogramm).
- ▶ Die Variablen `__doc__` und `__package__` besitzen keinen Inhalt.

Namensräume können auf verschiedene Weise erweitert werden:

- ▶ Eingabe einer Zuweisung, in der ein Variablenname vorkommt
- ▶ Definition einer Funktion
- ▶ Import von Modulen oder Objekten aus Modulen

Probieren Sie im interaktiven Modus Folgendes aus:

```
>>> a = 1
>>> import math
>>> globals()
{'__name__': '__main__', '__package__': None, ..., 'a': 1,
 'math': <module 'math' (built-in)>}
```

Sie sehen, dass die Namen `a` und `math` dem globalen Namensraum zugefügt worden sind. Der Aufruf `locals()` liefert genau das gleiche Dictionary.

Unterschiedliche globale und lokale Namensräume gibt es z.B. bei Funktionen. Beispiel:

```
>>> def test():
    x = 123
    print(locals())
    print(globals())
>>> test()
{'x': 123}
{'__name__': '__main__', '__package__': None, ..., 'test':
<function test at 0x03E196A8>}
```

Man erkennt, dass der lokale Namensraum der Funktion `test()` allein die Variable `x` mit dem Inhalt 123 enthält. Sie ist eine lokale Variable. Der Name der Funktion (`test`) gehört zum globalen Namensraum.

Die global-Anweisung

Programmblöcke und die zugehörigen Namensräume sind ineinander verschachtelt. Wenn in einem Programmblock ein Name verwendet wird, sucht das Laufzeitsystem zunächst im lokalen Namensraum des innersten Blocks nach dem zugehörigen Objekt.

Betrachten Sie folgendes Beispiel:

```
>>> def test():
    x = 1
    print(x)
>>> x = 2
>>> test()
1
>>> x
2
```

Offenbar gibt es zwei Variablen `x`. Die Funktion `test()` verwendet eine lokale Variable `x`, die sie auf den Wert 1 setzt. Die andere Variable `x`, der auf der Ausführungsebene der Wert 2 zugewiesen worden ist, wird durch `test()` nicht verändert. Beide Variablen haben nach außen den gleichen Namen, sind aber unterschiedliche Objekte in zwei unterschiedlichen lokalen Namensräumen.

Wenn die Funktion `test()` auf eine Variable des übergeordneten Blocks (damit ist der Programmtext gemeint, in dem die Funktion aufgerufen worden ist) zugreifen soll, muss sie als `global` deklariert werden. Das geschieht in einer `global`-Anweisung. Zu beachten ist, dass diese Anweisung innerhalb des Funktionskörpers steht und nicht etwa im übergeordneten Block. Beispiel:

```
>>> def test():
    global x
    x = 1
    print(x)
>>> x = 2
>>> test()
1
>>> x
1
```

Man sieht den Effekt der `global`-Anweisung: Es gibt nur noch eine (globale) Variable `x`. Die Funktion `test()` überschreibt nun den Inhalt der Variablen `x`, sie trägt jetzt am Ende den Wert 1.

Beachten Sie: Ob eine Variable `global` ist, wird innerhalb einer Funktion entschieden. Einer Funktion kann nicht »von außen« aufgezwungen werden, eine globale Variable zu übernehmen. Damit gibt es einen gewissen Schutz vor unbemerkten Seiteneffekten anderer Funktionen. Wenn Sie den Programmtext einer Funktion betrachten, können Sie sich durch Inspektion der `global`-Anweisungen einen Überblick über globale Variablen verschaffen.

S Stichwortverzeichnis

A

Abfangen von Laufzeitfehlern 133
abs() 163
abspath() 279
access() 270
addHandler() 223
Addition 93
Adjazenzliste 64
after_cancel() 488
after() 487
Aggregat 473
Aktueller Parameter 142
Aliasieren 57
Aliasierung 57
alpha_composite() 578
and 102
Anweisung 105
Anweisungsblock 141
Apache-Server
 konfigurieren 416
append() 58
appendChild() 603
Applikationsobjekt 411
argv 259
Array 65, 361
as 134
ASCII 333
ascii_letters 337
ascii_lowercase 337
ascii_uppercase 338
asctime() 286
assert 106, 215

Attribut 302, 304
 abgeleitetes Attribut 305
 Klassenattribut 304
 Objektattribut 305
 öffentlich 306
 privat 307
Ausdruck 107
Ausnahmen 205
Auswahlfeld 518
Authentifizierung 468

B

BaseHTTPServer 425
basename() 280
basicConfig() 220
Basisklasse 317
Beep() 584
Beliebige Anzahl von Parametern 147
Bezeichner 26
 Schlüsselwort 26
 Unterstrich 27
Bild 525
 einbinden 565
Binäre arithmetische Operatoren 93
Binärzahl 36
bind() 488
Binden 556
blend() 579
bool 39
bool() 165
Botschaft 299
break 107, 131–132

buffer () 167

Button 490

Bytestring 50

C

Canvas 491

capitalize() 331

center() 331

cgi.FieldStorage 401

CGIHTTPServer 425

CGI-Skript 393

Ausgabe 395

cgi.FieldStorage 401

debuggen 406

erste Zeile 394

GET-Methode 397

Installation 395

POST-Methode 398

cgibt 406

CGI-Webserver 425

chdir () 271

Checkbutton 504

childNodes 602

chmod() 271

choice 518

choice() 378, 389

chr() 167

clear() 76

Client-Server-System 394

close() 242

closed 242

cmath 364

cmp() 168

compile() 169, 355

complex() 170

connect() 447

Context 368

Context Management Protocol 139

Context Manager 139

continue 108, 131–132

Cookie 407

coords() 496

copy() 76, 570

count() 58, 331

create_arc() 497

create_bitmap() 499

create_image() 499

create_line() 500

create_oval() 502

create_polygon() 502

create_rectangle() 503

create_text() 503

create_window() 504

createElement() 611

createTextNode() 611

CRITICAL 219

crop() 570

ctime() 287

Cursorobjekt 448

Customizing 311

D

date 292

Datei siehe File

Datenbank 437

Cursorobjekt 448

Definition 440

erstellen 440

MySQLdb 446

SELECT 449

Tabelle erzeugen 441

Verbindung zum Datenbankserver
447

Verbindungsobjekt 448

Datentyp 33

datetime 291

Datum 285

DEBUG 219

decimal XE 365

decode() 50, 331

Decorator 159
def 141
DefaultContext 369
del 108
delattr() 170
Dezimale Gleitkomma-Arithmetik 365
Dezimalzahl 35
Dialogbox 554
dict() 171
Dictionary 73
 clear() 76
 copy() 76
 get() 77
 keys() 78
 popitem() 80
 Schlüssel 73
 setdefault() 80
 update() 81
 values() 81
difference() 87
digest() 458
Digitale Signatur 455
digits 338
dir() 172
dirname() 280
displayhook() 260
Division 94
divmod() 172
Docstring 141
Document 610
Document Object Model siehe DOM
documentElement 611
DOM 596
 createTextNode() 611
 Document 610
 documentElement 611
 Element 611
 getElementsByName() 611
 Knoten 596
 Knoten einfügen 607

Node 601
 tagName 612
 Text 615
DOM-Objekt 600
dump() 251
dumps() 252

E

Eingabefeld 399
Einrückung 24
Einrückungsfehler 25
Einwegfunktion 455
Element 611
elif 124
else 124
E-Mail 428, 431
Endlosschleife 133
endwith() 333
Entity-Typ 439
Entry 508
environ 272
Epoche 285
ER-Diagramm 439
ERROR 219
Erweiterte Zuweisung 120
Escape-Sequenz 47
EVA-Konzept 229
eval() 173
Event 556
 binden 556
 Taste 560
EventHandler 562
Event-Modifizierer 560
Event-Sequenz 558
Event-Typ 559
exc_info() 261
except 134, 210
Exception 206
exception 136
Exception-Klasse 210

execfile() 174
exists() 280
exit() 262
Exklusives Oder 97
expovariate() 378
extend() 59
ExtendedContext 371

F

False 39
Fehler 205
 Exception 206
 Syntaxfehler 205
Feldname 343
File
 close() 242
 closed 243
 flush() 243
 isatty() 243
 Iterationen 249
 mode 244
 read() 244
 readline() 245
 readlines() 245
 seek() 246
 tell() 247
 truncate() 247
 write() 248
FileHandler 226
Filter 580
filter() 175
finally 136
find() 334
findall() 355
firstChild 602
Flache Kopie 76
flash() 491
float() 175
flush() 243

for 127
Formaler Parameter 141
Format
 für Meldungen 227
format() 334
Formatierung 340
Formatierungsoperator % 234
Formatierungsstring 234
Formatspezifikation 344
Formatstring 342
Frame 509
from 112, 320
frozenset 84
frozenset() 177
f-String 51
ftplib 422
FTP-Server 422
Funktion 141
 lokale 154
 rekursive 152
 Standardfunktionen 163
Funktionskopf 141
Funktionskörper 141

G

Ganze Zahl 35
Ganzzahlige Division 94
gauss() 379
Generatorfunktion 117, 155
get() 77
getatime() 280
getattr() 177
getAttribute() 612
getcontext() 368
getcwd() 273
getElementsByTagName() 611–613
getenv() 273
getfirst() 404
getlist() 405

getLogger() 223
 getmtime() 281
 getopt() 231
 getpixel() 571
 getrecursionlimit() 264
 getrefcount() 263
 getsize() 281
 getstate() 381
 getvalue() 405
 Gierig 350
 Gleitkommazahlen 36
 global 42, 110
 Globale Variable 143
 Globaler Name 39
 globals() 178
 gmtime() 287
 Grafische Benutzungsoberfläche 471

- Dialogbox 554
- Kontrollvariable 553
- Layout-Manager 542
- Menü 513
- Tk 541
- Tk-Icon 542
- Widgets 472

 Grafische Benutzungsoberflächen

- Menü 519

 Graph 64
 grid() 549
 Grid-Layout-Manager 543
 GUI siehe Grafische Benutzungsoberfläche

H

Handler 225
 hasattr() 178
 hasAttribute() 612
 hasAttributes() 603
 hasChildNodes() 603
 hash() 178

Hashing-Objekt 456
 hashlib 455
 help() 179
 hex() 179
 Hexadezimalzahl 36
 hexdigits 338
 Hook 260
 HTML-Formular 398

- Checkbox 401
- Eingabefeld 399
- Radiobutton 400

 HTTP_COOKIE 408
 HTTP-Server 393

I

id() 180
 identifizier siehe Bezeichner
 Identität 28
 if 123
 Image 567

- alpha_composite() 578
- blend() 579–580
- copy() 570
- crop() 570
- Filter 580
- format 568
- getpixel() 571
- load() 571, 576
- mode 568
- paste() 571
- resize() 574
- save() 572
- show() 572
- size 568

 Image.open() 567
 ImageFilter 580
 IMAP4 426
 imaplib 426
 import 111, 320

in 44, 100
Index 536
index() 59
IndexError 343
Infinity 368
INFO 219
inorder 211
input() 180, 229
insert() 59
insertBefore() 603
Instanz 302
int() 181
Interaktiver Modus 19
Internet Message Protokoll 426
intersection() 87
Inversion 92
Inversionsoperator
 ~ 92
invoke() 491
is 101
is not 101
isatty() 243
isdir() 281
isfile() 281
isinstance() 182
islink() 282
issabs() 281
issubclass() 183
iter() 183
Iteration 127, 249
Iterator 155, 157
 Generatorfunktion 155
 in for-Anweisung 129
 iter() 183
 reversed() 195
 sorted() 198

J
join() 282, 335

jumpahead() 381

K

Kalenderdatum 291
key 62
key siehe Schlüssel
keys() 78
Klasse 300
Klassenattribut 304
Klassenbibliothek 320
Klassendiagramm 302
Klonen 58
Knoten 64, 596
Komplexe Zahl 38
Konstruktor 301
Kontext 368
 kontrollierter 138
Kontextmanager 138
Kontrollstruktur 123
 elif 124
 else 124
 Endlosschleife 133
 for 127
 if 123
 try 133
 while 131
Kontrollvariable 553
Konvertierungsspezifikator 340
Kurze Zeichenkette 46

L

Label 510
Lambda-Form 158
Lange Zeichenkette 47
lastChild 602
Latin-1 333
Layout-Manager 542
len() 184
List Comprehension 56

- list() 184
 - Listbox 510
 - listdir() 273
 - Liste 53
 - Aliasieren 57
 - append() 58
 - count() 58
 - erzeugen 54
 - extend() 59
 - index() 59
 - insert() 59
 - Klonen 58
 - List Comprehension 56
 - pop() 60
 - remove() 60
 - reverse() 60
 - sort() 60
 - sum() 200
 - Listen-Display 55
 - Literal 33
 - load() 576
 - loads() 252
 - locals() 185
 - localtime() 288
 - logging 218
 - basicConfig() 220
 - Format 227
 - getLogger() 223
 - Handler 225
 - logging level 223
 - Logische Fehler 205
 - Logischer Operator 102
 - Lokale Funktion 154
 - Lokale Variable 143
 - Lokaler Name 39
 - lstrip() 335
- M**
- Magic line 22
 - mainloop() 475, 542
 - map() 185
 - Marke 537
 - Master-Slave-Hierarchie 475
 - match() 356
 - Match-Objekt 358
 - math 374
 - max() 186
 - MD5 455
 - Mehrfachverzweigung 124
 - Menge 83, 100
 - frozenset() 177
 - set() 196
 - Menu 513
 - Menubutton 522
 - MessageBeep() 584
 - Methode 307
 - min() 187
 - mkdir() 273
 - makedirs() 274
 - mktime() 288
 - mod_wsgi 414
 - mode 244
 - Modul 320, 621
 - Modulo 95
 - Modulo-Operator 95
 - Monat 286
 - Multilisten-Struktur 55
 - Multiplikation 94
 - MySQL 438
 - MySQL Monitor 442
 - mysqlclient 446
 - MySQLdb 446
- N**
- Nachbedingungen 215
 - Name 30
 - Namensraum 39
 - NaN 368, 373
 - new() 580
 - next() 155

nextSibling 602
Node 601
nodeType 602
None 34
normcase() 282
not 102
not in 100

O

Oberklasse 317
Objekt 28
 änderbar 28
 unveränderbar 28
Objektattribut 305
Objektdiagramm 302
Objektorientierte Programmierung
 299
oct() 188
octdigits 338
Oktalzahl 36
OOA 299
OOD 299
OOP
 statische Methode 315
OOP siehe Objektorientierte Program-
 mierung
open() 188
Operator 91
 - 92
 ^ 97
 * 94
 ** 95
 / 94
 // 94
 & 97
 % 95
 + 92
 << 96
 >> 96
 | 98

binär 93
Boole'scher O. 102
in 44, 100
is 101
logisch 102
unär 92
Vergleich 98

or 102
ord() 190
os 267
os.name 274

P

pack() 543
Packer 542
Paket 323
Parameter
 aktueller 142
 beliebige Anzahl 147
 formaler 141
Parameterliste 141
Parameterübergabe 143
parentNode 603
parse() 601
parseString() 601
Parsing 600
pass 115
Passwort 459
paste() 571
path 265
Performance 359
 Sequenzen 69
Pfad
 hinzufügen 19
PILImage 525
pickle 249
Pickler 250, 253
PIL 566
PILImageTk 573
Pillow 566

pip 565
 place() 546
 Placer 543
 PlaySound() 585
 Polymorphismus 308
 pop() 60
 POP3-Protokoll 428
 popitem() 80
 poplib 428
 Positionargument 146
 pow() 190
 pprint() 237
 PrettyPrinter 238
 previousSibling 603
 Primärschlüssel 440
 printable 338
 Profiler 152
 Property 313
 Prozedur 151
 Prozess 587
 Prozess-Management 267
 Pseudozufall 389
 Pseudozufallszahl 389
 PSF siehe Python Software Foundation
 punctuation 338
 putenv() 274
 PyPI 565
 Python Image Library 566
 Python Package Index 565
 Python Software Foundation 13
 Python-Version
 abfragen 265

Q

Quicksort 217

R

r 51
 Radiobutton 526
 raise 116, 209

randbelow() 389
 randbits() 390
 randint() 382
 random 376
 random() 382
 range() 128, 194
 Raw-String 51
 re 347
 read() 244
 readline() 245
 readlines() 245
 Regulärer Ausdruck 347
 compile() 355
 findall() 355
 Funktionen 355
 match() 356
 Match-Objekt 358
 RE-Objekt 357
 search 356
 split() 357
 sub() 357
 Syntax 348
 Rekursive Funktion 152
 Relation 439
 remove() 60
 removeChild() 603, 606
 removedirs() 274
 rename() 275
 renames() 275
 RE-Objekt 357
 replace() 336
 repr() 195
 resize() 574
 return 116
 reverse() 60
 reversed() 195
 RFC 421
 rmdir() 275
 round() 196

S

- save() 572
- Scale 528
- Schablone 338
- Schleife 131
- Schleifenbedingung 132
- Schleifeninnere 131
- Schlüssel 73
- Schlüsselwort 27
- Scrollbar 531
- search() 356
- secrets (Modul) 388
- Seed 389
- seed() 383
- seek() 246
- Selbstdokumentation 216
- SELECT 449
- Sequenz
 - gemeinsame Operationen 43
 - in 44
 - indizieren 44
 - Konkatenation 44
 - Länge 44
 - not in 44
 - Slicing 45
 - sum() 200
- Serialisierung 249
- set 83
- set() 196
- setattr() 197
- setAttribute() 612
- setcontext() 370
- setdefault() 80
- setrecursionlimit() 264
- setstate() 384
- Shell 19
- show() 572
- shuffle() 382
- Signal 371
- sleep() 288
- slice() 198
- Sliceliste 45
- Slicing 45
- SMTP 431
- smtpplib 431
- Sommerzeit 287
- sort() 60
- sorted() 198
- Sortieren 62
- split() 283, 336, 357
- splitdrive() 283
- splitext() 283
- Standardfunktion 163
- Standard-Typ-Hierarchie 32
- start_new_thread() 588
- stat() 276
- staticmethod() 315
- Statische Methode 315
- stderr 266
- stdin 266
- stdout 266
- Stoßiteration 156
- str() 199
- StreamHandler 226
- strftime() 289
- String 46, 327
 - Escape-Sequenz 47
 - Formatierung 340
 - Konstanten 337
 - Konvertierungsspezifikator 340
 - kurze Zeichenkette 46
 - Lange Zeichenkette 47
 - Raw String 51
 - Unicode 49
- Strings
 - Performance 359
- sub() 357
- Subtraktion 93

Suite 24
 sum() 200
 Syntaxfehler 205
 sys 257
 sys.argv 230
 sys.version 264

T

Tabelle 441
 Tag 538
 tagName 612
 Tastenname 560
 Teilmenge 89
 tell() 247
 Telnet 434
 telnetlib 434
 Template 338
 Term 107
 Text 533
 Texteditor 539
 Thread 587
 threading 587
 time 294
 time() 290
 timedelta 291
 Timer 592
 Tk 541
 tkinter 555
 Tk-Icon ändern 542
 tkinter 471
 tkinter 554
 token_bytes() 390
 token_hex() 390
 token_urlsafe() 390
 Ton 565
 toprettyxml() 604, 609
 toxml() 604, 608
 Tracing 219
 Trap 373
 True 39

truncate() 247
 try 133, 209
 Tupel 52
 tuple() 201
 Typ 29, 163
 None 34
 Standard-Typ-Hierarchie 32
 type() 201–202

U

Überladen 308
 Umlaut 333
 unendlich 368
 Unicode 49
 uniform() 384
 union() 88
 Unix
 Programmausführung 22
 unlink() 604
 Unpickler 250, 253
 Unterklasse 317
 Unterstrich 27
 update() 81, 457
 urllib 255
 urlopen() 255
 UTC 285
 UTF-8 333

V

values() 81
 van Rossum 13
 Variable 31
 globale 143
 lokale 143
 vars() 202
 Verbinden von Zeilen 24
 Verbindungsobjekt 448
 Vererbung 317
 Vergleich 98
 Verschachtelte Liste 55

Verzeichnis

WSGI-Skripte 415

Vorbedingungen 215

Voreingestellter Parameterwert 145

W

Wahrheitswert 39

WARNING 219

Webcam 569

while 131, 210

Whitespace 338

Widget 472

Auswahlfeld 518

Bitmap 480

Button 490

Canvas 491

Checkbox 504

Cursor 481

Entry 508

Farben 479

Fokus 483

Fonts 479

Frame 509

Kontrollvariable 553

Koordinaten 478

Label 510

Längeneinheiten 477

Listbox 510

mainloop() 475

Master-Slave-Hierarchie 475

Menu 513

Menubutton 522

Option 476

Radiobutton 526

Rahmen 482

Scale 528

Scrollbar 531

Standard-Methoden 486

Text 533

Windows 21

Winsound 583

with 137

Wochentag 286

write() 248

writelines() 248

WSGI 410

WSGI-Skript 418

X

XML 595

xml.dom.minidom 595

xrange() 128

Y

yield 117, 155

Z

Zahl 34

ganze 35

Zeichenkette 46

Zeichenkette siehe String

Zeile

Einrückung 24

logische Zeile 23

physische Zeile 23

Verbinden von Zeilen 24

Zeilenstruktur 23

Zeit 285

Zeitabschnitt 291

zip() 203

Zufallsgenerator 377

Zugriffsrechte 267

Zusicherung 106

Zuweisung 118