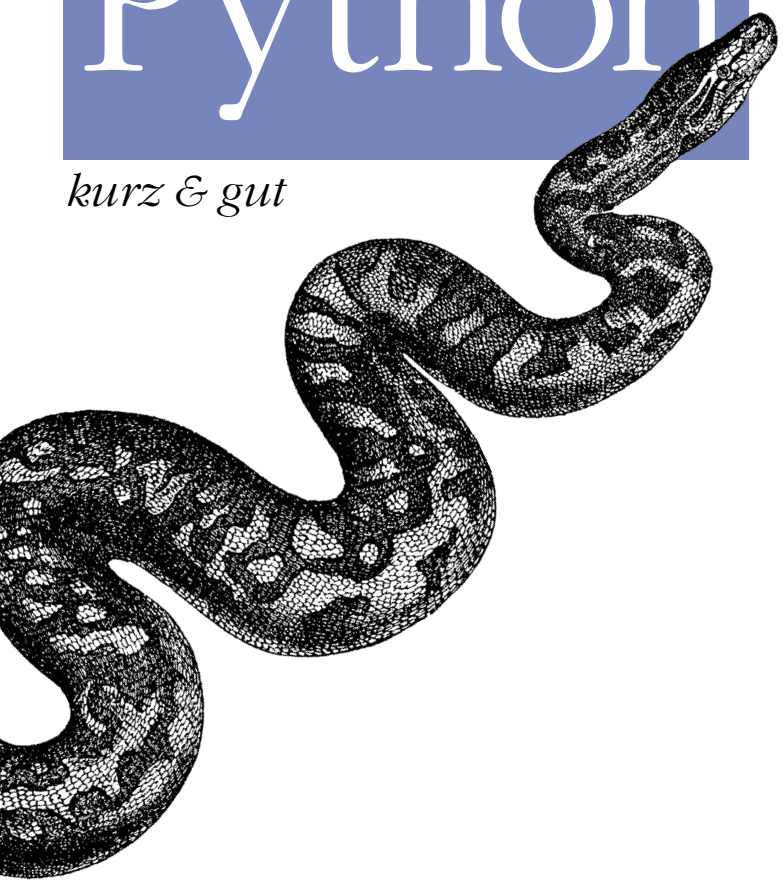
O'Reillys Taschenbibliothek

4. Auflage
Für Python 3.x und 2.6

Python

kurz & gut



O'REILLY®

Mark Lutz
Übersetzung von Lars Schulten

4. AUFLAGE

Python
kurz & gut

Mark Lutz

Deutsche Übersetzung von
Lars Schulten

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

1. Auflage 1999
2. Auflage 2002
3. Auflage 2005
4. Auflage 2010

Die Originalausgabe erschien 2010 unter dem Titel
PythonPocket Reference, Fourth Edition bei O'Reilly Media, Inc.

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

Übersetzung: Lars Schulten, Köln
Lektorat: Susanne Gerbert, Köln
Korrektorat: Sibylle Feldmann, Düsseldorf
Satz: III-satz, www.drei-satz.de
Umschlaggestaltung: Michael Oreal, Köln
Produktion: Karin Driesen, Köln
Druck: fgb freiburger graphische betriebe; www.fgb.de

ISBN 978-3-89721-556-6

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Python – kurz & gut

Einführung

Python ist eine universelle, objektorientierte Open Source-Programmiersprache, die sowohl für eigenständige Programme als auch für Skripten in verschiedensten Anwendungsbereichen eingesetzt wird und von Hunderttausenden von Entwicklern benutzt wird.

Python wurde entwickelt, um sowohl die Produktivität von Entwicklern als auch die Softwarequalität, die Programmportabilität und die Komponentenintegration zu optimieren. Python-Programme laufen auf den meisten gebräuchlichen Plattformen, inklusive Mainframes und Supercomputern, Unix und Linux, Windows und Macintosh, Java und .NET.

Diese Taschenreferenz fasst Python-Typen und -Anweisungen, spezielle Methodennamen, eingebaute Funktionen und Ausnahmen, häufig eingesetzte Module der Standardbibliothek und andere wichtige Python-Werkzeuge zusammen. Sie ist als kompakte Referenz für Entwickler und als Begleiter zu anderen Büchern gedacht, die Tutorien, Code, Beispiele und anderes Lehrmaterial enthalten.

In der vorliegenden vierten Auflage behandelt sie gleichermaßen Python 3.0 wie Python 2.6 und spätere Versionen in den 3.x- und 2.x-Zweigen. Sie befasst sich primär mit Python 3.0, dokumentiert aber auch Abweichungen in Python 2.6 und ist deswegen für beide Versionen verwendbar. Im Hinblick auf jüngste Sprach- und Bibliotheksänderungen wurde sie umfassend aktualisiert und um neue Sprachwerkzeuge und -themen erweitert.

Zusätzlich beinhaltet diese Auflage Anmerkungen zu wichtigen Erweiterungen im anstehenden Python 3.1, das Python 3.0 abrunden soll (der Begriff Python 3.0 bezeichnet in diesem Buch ganz allgemein Sprachänderungen, die in 3.0 eingeführt wurden, aber im gesamten 3.x-Zweig vorhanden sind). Große Teile dieser Auflage sind aber weiterhin mit früheren Python-Versionen nutzbar, die jüngsten Spracherweiterungen ausgenommen.

Typografische Konventionen

`[]`
: Eckige Klammern kennzeichnen normalerweise optionalen Code, außer wenn sie Teil der Python-Syntax sind.

`*`
: Etwas vor einem Sternchen kann null oder mehrmals wiederholt werden.

`a | b`
: Vertikale Balken kennzeichnen Alternativen.

Kursivschrift
: Wird für Dateinamen und URLs und zur Hervorhebung neuer Begriffe benutzt.

`Nichtproportionalschrift`
: Wird für Code, Befehle und Kommandozeilenoptionen benutzt und um Namen von Modulen, Funktionen, Attributen, Variablen und Methoden zu kennzeichnen.

`Nichtproportionalschrift kursiv`
: Wird für zu ersetzende Parameternamen in der Befehlssyntax benutzt.

Nutzung der Codebeispiele

Die Codebeispiele in diesem Buch sollen Ihnen bei der Bewältigung Ihrer Arbeit helfen. Im Allgemeinen können Sie den Code in diesem Buch für Ihre Programme und Ihre Dokumentationen nutzen. Sie müssen uns nicht um Erlaubnis bitten, es sei denn, Sie reproduzieren

erhebliche Teile des Codes. Einige Beispiele: Sie benötigen keine Genehmigung, wenn Sie ein Programm schreiben, das mehrere Codefragmente aus diesem Buch nutzt; Sie benötigen eine Genehmigung, wenn Sie vorhaben, eine CD-ROM mit Beispielen aus O'Reilly-Büchern zu vertreiben oder zu verteilen; Sie benötigen keine Genehmigung, wenn Sie eine Frage beantworten, indem Sie dieses Buch anführen und Beispielcode zitieren; Sie benötigen eine Genehmigung, wenn Sie erhebliche Teile der Codebeispiele in diesem Buch in der Dokumentation Ihres eigenen Produkts verwenden.

Wir begrüßen eine Quellenangabe, verlangen sie aber nicht. Eine Quellenangabe schließt üblicherweise Titel, Autor, Verlag und ISBN ein, zum Beispiel: »*Python kurz & gut*, 4. Auflage, von Mark Lutz. Copyright 2010 Mark Lutz, 3-89721-556-6«.

Sollten Sie den Eindruck haben, dass Ihr Gebrauch der Codebeispiele das Maß des Gewöhnlichen überschreitet oder von den oben erteilten Genehmigungen nicht gedeckt wird, sollten Sie sich unter *permissions@oreilly.com* mit uns in Verbindung setzen.

Kommandozeilenoptionen

Über Kommandozeilen werden Python-Programme von einer Shell-Eingabeaufforderung gestartet. Für Python selbst gedachte Kommandozeilenoptionen sollten vor der Angabe des auszuführenden Programmcodes erscheinen. Für den auszuführenden Code gedachte Optionen sollten nach der Programmangabe folgen. Kommandozeilen haben das folgende Format:

```
python [option*]
  [ dateiname | -c befehl | -m modul | - ] [arg*]
```

Python-Optionen

`-b`

Setzt eine Warnung ab, wenn `str()` mit einem `bytes`- oder `bytearray`-Objekt aufgerufen wird oder ein `bytes`- oder `bytearray`-Objekt mit einem `str`-Objekt verglichen wird. Option `-bb` meldet stattdessen einen Fehler.

`-B`
: Keine *.pyc*- oder *.pyo*-Bytecode-Dateien für Importe schreiben.

`-d`
: Schaltet die Debbuging-Ausgabe für den Parser an (gedacht für Entwickler von Python selbst).

`-E`
: Ignoriert die weiter unten beschriebenen Python-Umgebungsvariablen (wie `PYTHONPATH`).

`-h`
: Gibt eine Hilfemeldung aus und beendet dann die Ausführung.

`-i`
: Wechselt nach der Ausführung eines Skripts in den interaktiven Modus. Nützlich zur Fehlersuche nach Programmende.

`-O`
: Optimiert den generierten Bytecode (erzeugt und verwendet *.pyo*-Bytecode-Dateien). Bringt momentan eine kleine Leistungssteigerung.

`-OO`
: Wie `-O`, entfernt aber auch Docstrings aus dem Bytecode.

`-s`
: Das User-Siteverzeichnis (das benutzerspezifische Paketverzeichnis) nicht dem Modulsuchpfad `sys.path` hinzufügen.

`-S`
: Unterdrückt »import site« bei der Initialisierung.

`-u`
: Schaltet die Pufferung von `stdout` und `stderr` ganz aus.

`-v`
: Gibt bei jeder Initialisierung eines Moduls eine Meldung aus, die anzeigt, woher das Modul geladen wurde; wiederholen Sie diese Option für ausführlichere Meldungen.

`-V`
: Gibt die Python-Versionsnummer aus und beendet.

`-W` *`arg`*
: Warnungssteuerung; *`arg`* hat die Form *`Aktion:Nachricht:Kategorie:Modul:Zeilennummer`*. Siehe die Dokumentation zum Modul `warnings` im Python Library Reference-Handbuch (unter *http://www.python.org/doc/*).

`-x`
: Überspringt die erste Quellzeile. Dadurch ist es möglich, Nicht-Unix-Schreibweisen von `#!cmd` zu benutzen.

Angabe von Programmen

`dateiname`
: Der Name der Python-Skriptdatei, die als die Kerndatei eines Programmlaufs ausgeführt werden soll (z.B. `python main.py`). Der Name des Skripts wird in `sys.argv[0]` zur Verfügung gestellt.

`-c` *`befehl`*
: Gibt einen auszuführenden Python-Befehl (als String) an (z.B. führt `python -c "print('spam' * 8)"` einen `print`-Aufruf aus). `sys.argv[0]` wird auf `-c` gesetzt.

`-m` *`modul`*
: Führt ein Bibliotheksmodul als Skript aus: Sucht das Modul unter `sys.path` und führt es als oberste Datei aus (Beispiel: `python -m profile` startet den Python-Profiler, der sich im Standardbibliotheksverzeichnis befindet). `sys.argv[0]` wird auf den vollständigen Pfadnamen des Moduls gesetzt.

`-`
: Liest Python-Kommandos von `stdin` (Standard); geht in den interaktiven Modus, wenn `stdin` eine Konsole ist. `sys.argv[0]` wird auf - gesetzt.

`arg*`
: Gibt an, dass alles Weitere auf der Kommandozeile an das Skript oder Kommando weitergereicht wird (und in der String-Liste `sys.argv[1:]` erscheint).

Ohne *dateiname*, *kommando* oder *modul* wechselt Python in den interaktiven Modus (und nutzt GNU readline für die Eingabe, wenn es installiert ist).

Außer mit traditionellen Kommandozeilen in einer Shell-Eingabeaufforderung können Python-Programmen im Allgemeinen auch folgendermaßen ausgeführt werden: durch Anklicken ihrer Dateinamen in einem Dateibrowser, durch den Aufruf von Funktionen in der Python/C-API sowie durch Programmstart-Menüoptionen in IDEs wie IDLE, Komodo, Eclipse, NetBeans usw.

HINWEIS

Python 2.6 bietet keine Unterstützung für die Option `-b`, die sich auf Änderungen des String-Typs in Python 3.0 bezieht. Es unterstützt zusätzlich die folgenden Optionen:

- `-t` meldet eine Warnung bei inkonsistenter Verwendung von Tabs und Leerzeichen bei der Einrückung (`-tt` meldet stattdessen Fehler). Python 3.0 behandelt derartige Mischungen immer als Syntaxfehler.
- `-Q`, divisionsbezogene Optionen: `-Qold` (Standard), `-Qwarn`, `-Qwarnall` und `-Qnew`. Diese werden im neuen True-Division-Verhalten von Python 3.0 zusammengefasst.
- `-3` liefert Warnungen bei Python 3.x-Inkompatibilitäten im Code.

Umgebungsvariablen

Umgebungsvariablen sind systemweite, programmübergreifende Einstellungen, die zur globalen Konfiguration eingesetzt werden.

Operationale Variablen

`PYTHONPATH`
: Erweitert den Standardsuchpfad für importierte Moduldateien. Das Format ist dasselbe wie bei den `$PATH`-Variablen der jeweiligen Shell: Pfadnamen, die durch Doppelpunkte (Unix, Mac) oder Semikola (Windows) getrennt sind. Beim Modulimport

sucht Python in jedem aufgeführten Verzeichnis von links nach rechts nach der entsprechenden Datei bzw. dem entsprechenden Verzeichnis. Enthalten in `sys.path`.

`PYTHONSTARTUP`
: Falls auf den Namen einer lesbaren Datei gesetzt, werden die Python-Befehle in dieser Datei ausgeführt, bevor im interaktiven Modus die erste Meldung angezeigt wird.

`PYTHONHOME`
: Alternatives Präfixverzeichnis für Bibliotheksmodule (oder `sys.prefix`, `sys.exec_prefix`). Der Standard-Modulsuchpfad benutzt `sys.prefix/lib`.

`PYTHONCASEOK`
: Wenn nicht leer, wird (unter Windows) die Groß-/Kleinschreibung in importierten Namen ignoriert.

`PYTHONIOENCODING`
: *`Kodierung`*`[:`*`Fehler-Handler`*`]`-Überschreibung, die für die `stdin`-, `stdout`- und `stderr`-Streams verwendet wird.

Kommandozeilen-Optionsvariablen

`PYTHONDEBUG`
: Wenn nicht leer, gleichbedeutend mit Option `-d`.

`PYTHONDONTWRITEBYTECODE`
: Wenn nicht leer, gleichbedeutend mit Option `-B`.

`PYTHONINSPECT`
: Wenn nicht leer, gleichbedeutend mit Option `-i`.

`PYTHONNOUSERSITE`
: Wenn nicht leer, gleichbedeutend mit Option `-s`.

`PYTHONOPTIMIZE`
: Wenn nicht leer, gleichbedeutend mit Option `-O`.

`PYTHONUNBUFFERED`
: Wenn nicht leer, gleichbedeutend mit Option `-u`.

`PYTHONVERBOSE`
: Wenn nicht leer, gleichbedeutend mit Option `-v`.

Eingebaute Typen und Operatoren

Operatoren und Vorrang

Tabelle 1 führt Pythons Ausdrucksoperatoren auf. Je weiter unten die Operatoren in der Tabelle stehen, desto höher ist ihre Priorität, wenn sie in Ausdrücken mit mehreren unterschiedlichen Operatoren ohne Klammern eingesetzt werden.

Tabelle 1: Ausdrucksoperatoren und Vorrang in Python 3.0

Operator	Beschreibung
`yield X`	`send()`-Protokoll für Generator-Funktionen.
`lambda args: expr`	Anonyme Funktion.
`X if Y else Z`	Ternäre Auswahl: X wird nur ausgewertet, wenn Y wahr ist.
`X or Y`	Logisches ODER: Y wird nur ausgewertet, wenn X falsch ist.
`X and Y`	Logisches UND: Y wird nur ausgewertet, wenn X wahr ist.
`not X`	Logische Negation.
`X in Y, X not in Y`	Enthaltensein: iterierbare Objekte, Sets.
`X is Y, X is not Y`	Objektidentität testen.
`X < Y, X <= Y, X > Y, X >= Y`	Größenvergleich, Set-Teilmengen und -Obermengen.
`X == Y, X != Y`	Gleichheitsoperatoren.
`X | Y`	Bitweises OR, Set-Vereinigungsmenge.
`X ^ Y`	Bitweises exklusives OR, Set, symmetrische Differenz.
`X & Y`	Bitweises AND, Set-Schnittmenge.
`X << Y, X >> Y`	Schiebt X nach links bzw. rechts um Y Bits.
`X + Y, X – Y`	Addition/Verkettung, Subtraktion.
`X * Y, X % Y, X / Y, X // Y`	Multiplikation/Wiederholung, Rest/Formatieren, Division, restlose Division.
`-X, +X`	Unäre Negation, Identität.
`~X`	Bitweises Komplement (Umkehrung).
`X ** Y`	Potenzierung.
`X[i]`	Indizierung (Sequenzen, Abbildungen, andere).
`X[i:j:k]`	Slicing (alle Grenzen optional).

Tabelle 1: Ausdrucksoperatoren und Vorrang in Python 3.0 (Fortsetzung)

Operator	Beschreibung
`X(...)`	Aufruf (Funktion, Methode, Klasse und andere aufrufbare Objekte).
`X.attr`	Attributreferenz.
`(...)`	Tupel, Ausdruck, Generator-Ausdruck.
`[...]`	Liste, Listenkomprehension.
`{...}`	Dictionary, Set, Dictionary- und Set-Komprehension.

Hinweise zum Gebrauch von Operatoren

- In Python 2.6 kann Wertungleichheit mit `X != Y` oder `X <> Y` ausgedrückt werden. In Python 3.0 wurde die zweite dieser Optionen entfernt, da sie redundant ist.
- In Python 2.6 bedeutet ein Ausdruck in Backticks, `` `X` ``, das Gleiche wie `repr(X)`, es wandelt Objekte in Anzeigestrings um. Nutzen Sie in Python 3.0 die lesbareren eingebauten Funktionen `str()` und `repr()`.
- In Python 3.0 und 2.6 schneidet die *restlose Division* (`X // Y`) Restbruchteile immer ab und liefert als Ergebnis ganze Zahlen.
- In 3.0 führt der Ausdruck `X / Y` eine *echte Division* durch (die ein Fließkommaergebnis erzeugt und den Rest damit bewahrt), in 2.6 hingegen eine *klassische Division* (die bei ganzzahligen Operanden den Rest abschneidet).
- Die Syntax `[...]` wird für Listenliterale und Listenkomprehensionsausdrücke verwendet. Im zweiten Fall wird implizit eine Schleife ausgeführt, und die Ausdrucksergebnisse werden in einer neuen Liste gesammelt.
- Die Syntax `(...)` wird für Tupel, Ausdrücke und Generator-Ausdrücke – eine Form der Listenkomprehension, die Ergebnisse auf Anforderung liefert, statt eine Ergebnisliste zu generieren – verwendet. Bei allen drei Konstrukten können die Klammern gelegentlich weggelassen werden.
- Die Syntax `{...}` wird für Dictionary-Literale verwendet. In Python 3.0 wird sie auch für Set-Literale und Dictionary- sowie

Set-Komprehensionen genutzt; in 2.6 können Sie `set()` und Schleifenanweisungen verwenden.

- `yield` und der ternäre Auswahlausdruck `if/else` sind ab Python 2.5 verfügbar. `yield` liefert einen Wert aus einem Generator; der ternäre Auswahlausdruck ist eine Kurzform der mehrzeiligen `if`-Anweisung. `yield` verlangt Klammern, wenn es nicht allein auf der rechten Seite einer Zuweisung steht.
- Vergleichsoperatoren können verkettet werden: `X < Y < Z` liefert das gleiche Ergebnis wie `X < Y and Y < X`. In der verketteten Form wird `Y` nur einmal ausgewertet.
- Der Slice-Ausdruck `X[I:J:K]` entspricht der Indizierung mit einem Slice-Objekt: `X[slice(I, J, K)]`.
- In Python 2.6 sind Größenvergleiche verschiedener Typen erlaubt – Zahlen werden in einen gemeinsamen Typ umgewandelt, andere vermischte Typen nach Namen geordnet. In Python 3.0 sind Größenvergleiche mit verschiedenen Typen nur noch für Zahlen erlaubt. Alles andere löst eine Ausnahme aus, Sortieroperationen über Proxy-Objekte eingeschlossen.
- Größenvergleiche für Dictionaries werden in Python 3.0 ebenfalls nicht mehr unterstützt (Gleichheitsprüfungen hingegen schon); ein möglicher Ersatz unter 3.0 ist ein Vergleich von `sorted(dict.items())`.
- Aufrufausdrücke unterstützen positionelle und benannte Argumente in beliebiger Zahl; Informationen zur Aufrufsyntax finden Sie unter »Ausdrucksanweisungen« auf Seite 60 und »Die def-Anweisung« auf Seite 65.
- Python 3.0 gestattet die Verwendung von Ellipsen (d.h. `...`) als atomare Ausdrücke an beliebigen Stellen des Quellcodes. In einigen Kontexten (z. B. Funktions-Stubs, typunabhängige Variableninitialisierung) können sie als Alternativen zu `pass` oder `None` eingesetzt werden.

Operationen nach Kategorie

Alle Typen unterstützen die Vergleichsoperationen und Booleschen Operationen in Tabelle 2.

»Wahr« ist jede von null verschiedene Zahl sowie jedes nicht leere Sammlungsobjekt (Liste, Dictionary usw.). Die Namen `True` und `False` sind den Wahrheitswerten »wahr« und »falsch« zugeordnet und verhalten sich wie die Ganzzahlen 1 und 0. Das spezielle Objekt `None` hat den Wahrheitswert `False`.

Vergleiche geben `True` oder `False` zurück und werden bei zusammengesetzten Objekten rekursiv angewandt, um ein Ergebnis zu ermitteln.

Die Auswertung Boolescher `and`- und `or`-Operatoren wird abgebrochen (Kurzschluss), sobald das Ergebnis eindeutig ist, und liefert den Operanden zurück, bei dem angehalten wurde.

Tabelle 2: Vergleiche und Boolesche Operationen für alle Objekte

Operator	Beschreibung
`X < Y`	kleiner als[a]
`X <= Y`	kleiner oder gleich
`X > Y`	größer als
`X >= Y`	größer oder gleich
`X == Y`	gleich (gleicher Wert)
`X != Y`	ungleich (wie `X< >Y`, allerdings nur in Python 2.6)[b]
`X is Y`	gleiches Objekt
`X is not Y`	negierte Objektgleichheit
`X < Y < Z`	verkettete Vergleiche
`not X`	wenn `X` falsch ist, dann `True`, sonst `False`
`X or Y`	wenn `X` falsch ist, dann `Y`, sonst `X`
`X and Y`	wenn `X` falsch ist, dann `X`, sonst `Y`

[a] Vergleichen Sie für die Implementierung von Vergleichsausdrücken auch die Klassenmethoden für spezielle Vergleiche (z. B. `__lt__` für `<`) in 3.0 und 2.6 sowie die allgemeinere (`__cmp__`) in 2.6 im Abschnitt »Überladungsmethoden für Operatoren« auf Seite 90.

[b] `!=` und `<>` bedeuten in 2.6 Wertungleichheit, aber `!=` ist die in 2.6 bevorzugte und 3.0 allein unterstützte Syntax. `is` testet auf Identität, `==` auf Wertgleichheit und ist deswegen in der Regel wesentlich nützlicher.

Die Tabellen 3 bis 6 definieren gemeinsame Operationen für die drei Haupttypkategorien (Sequenzen, Abbildungen und Zahlen) sowie verfügbare Operationen für veränderliche Python-Datenty-

pen. Die meisten Typen exportieren dazu auch noch weitere, spezifischere Operationen, die weiter unten im Abschnitt »Besondere eingebaute Typen« auf Seite 15 beschrieben werden.

Tabelle 3: Sequenzoperationen (Strings, Listen, Tupel, Bytes, Bytearray)

Operation	Beschreibung	Klassenmethode
`X in S` `X not in S`	Test auf Enthaltensein.	`__contains__`, `__iter__`, `__getitem__`[a]
`S1 + S2`	Verkettung.	`__add__`
`S * N, N * S`	Repetition.	`__mul__`
`S[i]`	Indizierung über Position.	`__getitem__`
`S[i:j], S[i:j:k]`	Slicing: Elemente in `S` von Position `i` bis `j-1` mit optionalem Schritt `k`.	`__getitem__`[b]
`len(S)`	Länge.	`__len__`
`min(S), max(S)`	Minimales, maximales Element.	`__iter__`, `__getitem__`
`iter(S)`	Iterator-Objekt.	`__iter__`
`for X in S:`, `[expr for X in S]`, `map(func, S)` usw.	Iteration (jeder Kontext).	`__iter__`, `__getitem__`

[a] Siehe auch Iteratoren in Python 2.2, Generatoren und die Klassenmethode `__iter__` (siehe den Abschnitt »Die yield-Anweisung« auf Seite 70). Wenn definiert, wird `__contains__` der Methode `__iter__` und `__iter__` der Methode `__getitem__` vorgezogen.

[b] In Python 2.6 können Sie auch `__getslice__`, `__setslice__` und `__delslice__` definieren, um Slicing-Operationen abzuwickeln. In 3.0 wurden diese Methoden entfernt. Stattdessen sollten Sie Slice-Objekte an Ihre elementbasierten indizierten Gegenstücke übergeben. Slice-Objekte können in Indexausdrücken explizit anstelle der Grenzen `i:j:k` angegeben werden.

Tabelle 4: Operationen auf veränderlichen Sequenzen (Listen, Bytearray)

Operation	Beschreibung	Klassenmethode
`S[i] = X`	Zuweisung an Index: Ändert vorhandenen Eintrag.	`__setitem__`
`S[i:j] = S2`, `S[i:j:k] = S2`	Zuweisung an Teilbereich: `S` wird von `i` bis `j` durch `S2` ersetzt, mit einer optionalen Schrittweite `k`.	`__setitem__`
`del S[i]`	Eintrag in Index löschen.	`__delitem__`
`del S[i:j]`, `del S[i:j:k]`	Teilbereich löschen.	`__delitem__`

Tabelle 5: Operationen auf Abbildungen (Dictionaries)

Operation	Beschreibung	Klassenmethdoe
`D[k]`	Indizierung über Schlüssel.	`__getitem__`
`D[k] = X`	Zuweisung über Schlüssel: Ändert oder erstellt den Eintrag für den Schlüssel `k`.	`__setitem__`
`del D[k]`	Eintrag über Schlüssel löschen.	`__delitem__`
`len(D)`	Länge (Anzahl Schlüssel).	`__len__`
`k in D`	Prüft, ob Schlüssel enthalten.[a]	wie Tabelle 3
`k not in D`	Das Gegenteil von `k in D`.	wie Tabelle 3
`iter(S)`	Iterator-Objekt für Schlüssel.	wie Tabelle 3
`for k in D:` usw.	Schlüssel in `D` durchlaufen (alle Iterationskontexte).	wie Tabelle 3

[a] In Python 2.x kann die Schlüsselprüfung auch mit `D.has_key(K)` durchgeführt werden. Diese Methode wurde in Python 3.0 entfernt, es gibt also nur noch den `in`-Ausdruck, der auch in 2.6 vorzuziehen ist. Siehe »Dictionaries« auf Seite 40.

Tabelle 6: Numerische Operationen (alle numerischen Typen)

Operation	Beschreibung	Klassenmethode
`X + Y`, `X – Y`	Addition, Subtraktion.	`__add__`, `__sub__`
`X * Y`, `X / Y`, `X // Y`, `X % Y`	Multiplikation, Division, restlose Division, Rest.	`__mul__`, `__truediv__`, `__floordiv__`, `__mod__`
`-X`, `+X`	Negativ, Identität.	`__neg__`, `__pos__`
`X | Y`, `X & Y`, `X ^ Y`	Bitweises ODER, UND, exklusives ODER (Integer).	`__or__`, `__and__`, `__xor__`
`X << N`, `X >> N`	Bitweises Verschieben nach links/rechts (Integer).	`__lshift__`, `__rshift__`
`~X`	Bitweises Invertieren (Integer).	`__invert__`
`X ** Y`	`X` zur Potenz `Y`.	`__pow__`
`abs(X)`	Absolutwert (Betrag).	`__abs__`
`int(X)`	Umwandlung in Integer.[a]	`__int__`
`float(X)`	Umwandlung in Float.	`__float__`
`complex(X)`, `complex(re,im)`	Erstellt eine komplexe Zahl.	`__complex__`
`divmod(X, Y)`	Tupel: `(X/Y, X%Y)`.	`__divmod__`
`pow(X, Y [,Z])`	`X` zur Potenz `Y` [modulo `Z`].	`__pow__`

[a] In Python 2.6 ruft die eingebaute Funktion `long()` die Klassenmethode `__long__` auf. In Python 3.0 wurde `int` so geändert, dass er den Typ `long` subsumiert, der deswegen entfernt wurde.

Anmerkungen zu Sequenzoperationen

Indizierung: S[i]

- Wählt Komponenten an Positionen (erste Position ist 0).
- Negative Indizes zählen vom Ende (das letzte Element befindet sich an Position -1.
- `S[0]` wählt das erste Element.
- `S[-2]` wählt das zweitletzte Element (`S[len(S) - 2]`).

Einfache Teilbereiche (Slices): S[i:j]

- Extrahiert zusammenhängende Bereiche einer Sequenz.
- Die Standardgrenzen für Bereiche sind 0 und die Sequenzlänge.
- `S[1:3]` geht von Index 1 bis ausschließlich 3.
- `S[1:]` geht von Index 1 bis zum Ende.
- `S[:-1]` nimmt alles bis auf das letzte Element.
- `S[:]` macht eine flache Kopie der Sequenz `S`.
- Zuweisung an Teilbereiche entspricht Löschen und dann Einfügen.

Erweiterte Teilbereichsbildung: S[i:j:k]

- Das dritte Element `k`, falls vorhanden, ist eine Schrittweite: Sie wird zur Position jedes extrahierten Elements hinzugefügt.
- `S[::2]` liefert jedes zweite Element der Sequenz `S`.
- `S[::-1]` liefert die Umkehrung der Sequenz `S`.
- `S[4:1:-1]` holt Elemente von rechts nach links ab Position 4 (inklusive) bis 1 (exklusive).

Andere

- Verkettungs-, Wiederholungs- und Teilbereichsoperationen liefern neue Objekte zurück (bei Tupeln nicht immer).

Besondere eingebaute Typen

Dieser Abschnitt behandelt Zahlen, Strings, Listen, Dictionaries, Tupel, Dateien und andere zentrale eingebaute Typen. Zusammengesetzte Datentypen (z. B. Listen, Dictionaries und Tupel) können beliebig ineinander geschachtelt sein. Sets können ebenfalls an Schachtelungen teilhaben, aber nur unveränderliche Objekte enthalten.

Zahlen

Dieser Abschnitt behandelt Basistypen (Ganzzahlen, Fließkommazahlen) und fortgeschrittenere Typen (komplexe Zahlen, Dezimalzahlen und Brüche). Zahlen sind immer unveränderlich.

Literale und Erstellung

Zahlen werden in einer Vielzahl von Formen für numerische Konstanten geschrieben.

`1234, -24, 0`
: Ganzzahlen (unbeschränkte Genauigkeit).[1]

`1.23, 3.14e-10, 4E210, 4.0e+210, 1., .1`
: Fließkommazahlen (normalerweise implementiert als C `double` in CPython).

`0o177, 0x9ff, 0b1111`
: Oktale, hexadezimale und binäre Literale für Ganzzahlen.[2]

1 In Python 2.6 gibt es einen speziellen Typ namens `long` für Ganzzahlen unbegrenzter Genauigkeit; `int` dient gewöhnlichen Ganzzahlen mit einer Genauigkeit, die in der Regel auf 32 Bit beschränkt ist. Long-Literale können mit einem angehängten »L« (z. B. `99999L`) geschrieben werden, obwohl Integer automatisch in Longs umgewandelt werden, wenn zusätzliche Genauigkeit erforderlich ist. In 3.0 bietet der Typ `int` unbegrenzte Genauigkeit und fasst so die Typen `int` und `long` von 2.6 zusammen; die »L«-Literalsyntax wurde in 3.0 entfernt.

2 In Python 2.6 können oktale Literale auch einfach mit einer vorangestellten `0` geschrieben werden, `0777` und `0o777` sind also äquivalent. In 3.0 wird für oktale Literale nur noch die zweite Form unterstützt.

`3+4j`, `3.0+4.0j`, `3J`
: Komplexe Zahlen.

`decimal.Decimal('1.33')`, `fractions.Fraction(4, 3)`
: Modulbasierte Typen: Decimal, Fraction.

`int()`, `float()`, `complex()`
: Zahlen auf Basis anderer Objekte oder Strings erstellen und dabei eventuell Basisumwandlung durchführen; siehe »Eingebaute Funktionen« auf Seite 105.

Operationen

Numerische Typen unterstützen alle numerischen Operationen (siehe Tabelle 6). Kommen in einem Ausdruck verschiedene Typen vor, wandelt Python die Operanden in den Typ des Operanden mit dem »höchsten« Typ um (`int` ist kleiner als `float`, `float` kleiner als `complex`) Seit Python 3.0 und 2.6 bieten Ganzzahl- und Fließkommaobjekte eine Handvoll *Methoden* und andere *Attribute*; mehr Informationen finden Sie in Pythons Library Reference-Handbuch.

```
>>> (2.5).as_integer_ratio()           # float-Attribute
(5, 2)
>>> (2.5).is_integer()
False
>>> (2).numerator, (2).denominator     # int-Attribute
(2, 1)
>>> (255).bit_length(), bin(255)       # 3.1+ bit_length()
(8, '0b11111111')
```

Decimal und Fraction

Module der Python-Standardbibliothek bieten zwei weitere numerische Typen – *Decimal*, eine Fließkommazahl mit fester Genauigkeit, und *Fraction*, ein relationaler Typ, der explizit Nenner und Zähler festhält. Beide können eingesetzt werden, um Ungenauigkeiten in der Fließkommaarithmetik zu entgehen.

```
>>> 0.1 - 0.3
-0.19999999999999998
>>> from decimal import Decimal
>>> Decimal('0.1') - Decimal('0.3')
Decimal('-0.2')
```

```
>>> from fractions import Fraction
>>> Fraction(1, 10) - Fraction(3, 10)
Fraction(-1, 5)
>>> Fraction(1, 3) + Fraction(7, 6)
Fraction(3, 2)
```

Fraction vereinfacht Ergebnisse automatisch. Aufgrund der festgelegten Genauigkeit und der Unterstützung unterschiedlichster Beschneidungs- und Rundungsprotokolle sind Decimals für Finanzanwendungen nützlich. Mehr Informationen finden Sie in der Python Library Reference.

Weitere numerische Typen

Python bietet außerdem einen *set*-Typ (der in »Sets« auf Seite 50 beschrieben wird). Weitere numerische Typen wie Vektoren und Matrizen sind als externe Open Source-Erweiterungen verfügbar (siehe beispielsweise das Paket *NumPy*). Extern finden Sie außerdem noch Unterstützung für die Visualisierung, statistische Pakete und anderes.

Strings

Das gewöhnliche `str`-Objekt ist ein unveränderliches Zeichenarray, auf dessen Elemente über die Position zugegriffen wird. Seit Python 3.0 gibt es drei String-Typen mit sehr ähnlichen Schnittstellen:

- `str`, eine unveränderliche Zeichensequenz, die allgemein für Text verwendet wird, sowohl für ASCII als auch für Unicode.
- `bytes`, eine unveränderliche Sequenz kleiner Integer, die für binäre Bytedaten verwendet wird.
- `bytearray`, eine veränderliche Variante von `bytes`.

Python 2.x bietet hingegen zwei unveränderliche String-Typen: `str`, für 8-Bit-Text und Binärdaten sowie `unicode` für Unicode-Text, wie in »Unicode-Strings« auf Seite 32 beschrieben. Python 2.6 bietet als Back-Port von Python 3.0 auch den `bytearray`-Typ, stellt für diesen aber nicht die gleiche klare Trennung von Text- und Binärdaten auf (in 2.6 kann er beliebig mit Textstrings gemischt werden).

Die meisten Teile dieses Abschnitts gelten für alle String-Typen, weitere Informationen zu `bytes` und `bytearray` erhalten Sie in »String-Methoden« auf Seite 25, »Unicode-Strings« auf Seite 32 und »Eingebaute Funktionen« auf Seite 105.

Literale und Erstellung

Strings werden als eine Folge von Zeichen in Anführungszeichen geschrieben, denen optional ein Typindikator vorangestellt werden kann.

`"Python's"`, `'Python"s'`
: Doppelte und einfache Anführungszeichen funktionieren auf gleiche Weise, beide können in die andere Art unmaskiert eingebettet werden.

`"""Das ist ein`
`mehrzeiliger Block"""`
: Strings in drei aufeinanderfolgenden Anführungszeichen sammeln mehrere Zeilen zu einem einzigen String zusammen. Zwischen die ursprünglichen Zeilen werden Zeilenendezeichen (`\n`) eingefügt.

`'Python\'s\n'`
: Mit vorangestelltem Backslash angezeigte Escape-Sequenzen (siehe Tabelle 7) werden durch den Bytewert des Sonderzeichens ersetzt, das sie repräsentieren (z. B. ist `'\n'` ein Byte mit dem dezimalen Binärwert 10).

`"Dies" "ist" "zusammengesetzt"`
: Nebeneinanderstehende String-Konstanten werden zusammengefügt. Kann zeilenübergreifend erfolgen, wenn die Strings in Klammern stehen.

`r'ein roher\string'`, `R'ein\anderer'`
: Rohe Strings: Der Backslash wird nicht ausgewertet und bleibt im String erhalten (außer am Ende eines Strings), nützlich bei regulären Ausdrücken und DOS-Pfadangaben, zum Beispiel `r'c:\dir1\datei'`.

Die folgenden Literalformen bilden spezialisierte Strings, wie sie in »Unicode-Strings« auf Seite 32 beschrieben werden:

`b'...'`
: bytes-String-Literal: Sequenz von 8-Bit-Bytewerten, die rohe binäre Daten repräsentiert. Macht in Python 3.0 einen bytes-String, in Python 2.6 einen gewöhnlichen str-String (im Dienste der 3.0-Kompatibilität). Siehe »String-Methoden« auf Seite 25, »Unicode-Strings« auf Seite 32 und »Eingebaute Funktionen« auf Seite 105.

`bytearray(...)`
: bytearray-String-Objekt: eine veränderliche Variante von bytes. Verfügbar in Python 2.6 und 3.0. Siehe »String-Methoden« auf Seite 25, »Unicode-Strings« auf Seite 32 und »Eingebaute Funktionen« auf Seite 105.

`u'...'`
: Unicode-String-Literal in Python 2.x (in Python 3 unterstützen gewöhnliche str-Strings Unicode-Text). Siehe »Unicode-Strings« auf Seite 32.

`str()`, `bytes()`, `bytearray()`
: Erstellt Strings aus Objekten, in Python 3.0 mit eventueller Unicode-Kodierung/-Dekodierung. Siehe »Eingebaute Funktionen« auf Seite 105.

`hex()`, `oct()`, `bin()`
: Erstellt aus Zahlen Strings mit hexadezimalen/oktalen/binären Ziffern. Siehe »Eingebaute Funktionen« auf Seite 105.

String-Literale können die Escape-Sequenzen aus Tabelle 7 nutzen, um spezielle Zeichen einzubinden.

Tabelle 7: Escape-Sequenzen für String-Konstanten

Escape	Bedeutung	Escape	Bedeutung
neuezeile	Zeilenende ignorieren	\t	Horizontaltabulator
\\	Backslash (\)	\v	Vertikaltabulator
\'	einfaches Anführungszeichen	\N{*id*}	Unicode-Datenbank-ID
\"	doppeltes Anführungszeichen	\u*hhhh*	Unicode 16-Bit, hexadezimal
\a	Glocke (bell)	\U*hhhhhhhh*	Unicode 32-Bit, hexadezimal[a]

Tabelle 7: Escape-Sequenzen für String-Konstanten (Fortsetzung)

Escape	Bedeutung	Escape	Bedeutung
\b	Backspace	\x*hh*	hexadezimal (maximal zwei Ziffern)
\f	Seitenvorschub	*ooo*	oktal (maximal drei Ziffern)
\n	Zeilenvorschub	\0	null (kein String-Ende!)
\r	Wagenrücklauf	*andere*	kein Escape

[a] \U*hhhhhhhh* hat genau acht hexadezimale Ziffern (h); \u und \U dürfen nur in Unicode-String-Literalen benutzt werden.

Operationen

Alle String-Typen unterstützen alle Operationen für unveränderliche Sequenzen (siehe Tabelle 3) und zusätzlich die String-Methodenaufrufe (siehe nachfolgenden Abschnitt). Zusätzlich unterstützt der Typ `str` %-Formatierungsausdrücke und Template-Substitution, und der Typ `bytearray` unterstützt alle Operationen für veränderliche Sequenzen (Tabelle 4 plus zusätzliche listenartige Methoden). Werfen Sie auch einen Blick in das `re`-Modul für Mustervergleiche mit Strings in »Das Mustervergleichsmodul re« auf Seite 162 und die String-bezogenen eingebauten Funktionen im Abschnitt »Eingebaute Funktionen« auf Seite 105.

String-Formatierung

In Python 2.6 und 3.0 unterstützen gewöhnlich `str`-Strings zwei verschiedene Arten der String-Formatierungsoperationen, die Objekte anhand von Formatbeschreibungsstrings formatieren:

- den ursprünglichen Ausdruck mit dem %-Operator: `fmt % (werte)` und
- die neue Methode, die mit `fmt.format(werte)` aufgerufen wird.

Beide erzeugen neue Strings, die eventuell auf typspezifischen Substitutionscodes basieren. Ihre Ergebnisse können ausgegeben oder zur späteren Verwendung Variablen zugewiesen werden:

```
>>> '%s, %s, %.2f' % (42, 'spam', 1 / 3.0)
'42, spam, 0.33'
```

```
>>> '{0}, {1}, {2:.2f}'.format(42, 'spam', 1 / 3.0)
'42, spam, 0.33'
```

Obgleich sich aktuell der Methodenaufruf aktiver zu entwickeln scheint, wird der Ausdruck in vorhandenem Code weiterhin intensiv genutzt. Beide Formen werden immer noch vollständig unterstützt. Und obwohl einige der Ansicht sind, die Methodenform sei einprägsamer und stimmiger, ist der Ausdruck häufig einfacher und kompakter. Da diese beiden Formen eigentlich nur kleinere Variationen eines Themas mit äquivalenter Funktionalität und Komplexität sind, gibt es zurzeit keinen überzeugenden Grund, die eine statt der anderen zu empfehlen.

Der String-Formatierungsausdruck

String-Formatierungsausdrücke ersetzen spezielle, mit `%` beginnende Zielfelder in Formatstrings auf der linken Seite des `%`-Operators mit Werten auf der rechten Seite (ähnlich wie `sprintf` in C). Wenn mehrere Werte ersetzt werden sollen, müssen diese als Tupel rechts vom Operator `%` geschrieben werden. Soll nur ein Element ersetzt werden, kann es rechts als einzelner Wert oder als einstelliges Tupel formuliert werden (schachteln Sie Tupel, um ein Tupel selbst zu formatieren). Wenn links Schlüsselnamen verwendet werden, muss rechts ein Dictionary angegeben werden. * erlaubt es, die Breite und die Genauigkeit dynamisch anzugeben.

`'The knights who say %s!' % 'Ni'`
Ergebnis: `'The knights who say Ni!'`

`"%d %s %d you" % (1, 'spam', 4.0)`
Ergebnis: `'1 spam 4 you'`

`"%(n)d %(x)s" % {"n":1, "x":"spam"}`
Ergebnis: `'1 spam'`

`'%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)`
Ergebnis: `'0.333333, 0.33, 0.3333'`

Die Substitutionsziele im Formatstring links des `%`-Operators folgen diesem allgmeinen Format:

`%[(schlüssel`*name*`)][`*flags*`][`*breite*`][.`*genauigkeit*`]`*typcode*

schlüsselname referenziert ein Element im erwarteten Dictionary. *flags* kann – (links ausrichten), + (numerisches Vorzeichen), ein Leerzeichen (lässt ein Leerzeichen vor einer positiven Zahl stehen) und 0 (mit Nullen füllen) sein; *breite* ist die gesamte Feldbreite; *genauigkeit* bezeichnet die Stellen nach dem ».«. *typcode* ist ein Zeichen aus Tabelle 8. Sowohl *breite* als auch *genauigkeit* können mit * angegeben werden, um zu erzwingen, dass deren Werte vom nächsten Element der Werte rechts vom Operator % genommen werden, falls die Größen erst zur Laufzeit bekannt sind. Hinweis: %s konvertiert jedes Objekt in seinen druckbaren Repräsentationsstring.

Tabelle 8: Typcodes für die %-String-Formatierung

Code	Bedeutung	Code	Bedeutung
s	String (oder jegliches Objekt, benutzt str())	X	wie x mit Großbuchstaben
r	wie s, benutzt jedoch repr() statt str()	e	Fließkomma-Exponent
c	Zeichen (int oder str)	E	wie e, Großbuchstaben
d	dezimal (Integer zur Basis 10)	f	Fließkomma dezimal
i	Integer	F	wie f, Großbuchstaben
u	wie d (obsolet)	g	Fließkomma, e oder f
o	oktal (Integer zur Basis 8)	G	Fließkomma, E oder F
x	hexadezimal (Integer zur Basis 16)	%	das Zeichenliteral '%'

Die String-Formatierungsmethode

Der Formatierungsmethodenaufruf funktioniert ähnlich wie der Ausdruck aus dem vorangegangenen Abschnitt, wird aber mit der gewöhnlichen Methodenaufrufsyntax auf dem Formatstring-Objekt aufgerufen und identifiziert Substitutionsziele mit der {}-Syntax statt mit %. Substitutionsziele im Formatstring können Argumente für den Methodenaufruf über die Position oder über einen Schlüsselwortnamen angeben, können außerdem Argumentattribute, Schlüssel oder Positionen referenzieren, können eine Standardformatierung akzeptieren oder explizite Typcodes stellen und die Zielsyntax schachteln, um Werte aus der Argumentliste herauszuziehen:

```
>>> '{0}, {food}'.format(42, food='spam')
'42, spam'
>>> import sys
>>> fmt = '{0.platform} {1[x]} {2[0]}'  # attr,key,index
>>> fmt.format(sys, {'x': 'schinken', 'y': 'eier'}, 'AB')
'win32 schinken A'
>>> '{0} {1:+.2f}'.format(1 / 3.0, 1 / 3.0)
'0.333333333333 +0.33'
>>> '{0:.{1}f}'.format(1 / 3.0, 4)
'0.3333'
```

Zu den meisten davon gibt es Gegenstücke unter den %-Ausdrucksmustern (z.B. Dictionary-Schlüssel und *-Wertreferenzen). Beim Ausdruck kann es allerdings erforderlich sein, einige der Operationen außerhalb des Formatstrings zu formulieren. Substitutionsziele in Strings, die für Formatierungsmethodenaufrufe genutzt werden, folgen dieser allgemeinen Form:

{*feldname*!*umwandlungs-flag*:*formatangabe*}

Die Elemente in der Syntax für Substitutionsziele bedeuten:

- `feldname` ist eine Zahl oder ein Schlüssel, die/der ein Argument bezeichnet, auf die/den optional ein `.name`-Attribut- oder eine `[index]`-Komponentenreferenz folgen kann.
- `umwandlungs-flag` ist `r`, `s` oder `a` und gibt an, ob auf dem Wert `repr()`, `str()` bzw. `ascii()` aufgerufen werden soll.
- `formatangabe` gibt an, wie der Wert dargestellt werden soll, enthält Informationen wie die Feldbreite, die Ausrichtung, die Auffüllung, die dezimale Genauigkeit und so weiter und schließt mit einem optionalen Datentypcode.

Die `formatangabe`-Komponente nach dem Doppelpunkt wird formell folgenderweise beschrieben (eckige Klammern zeigen optionale Komponenten an und werden nicht geschrieben):

[[*füllung*]*ausrichtung*][*vorzeichen*][#][0][*breite*][.*genauigkeit*][*typcode*]

`ausrichtung` kann <, >, = oder ^ sein und bedeutet links ausrichten, rechts ausrichten, Auffüllen nach Vorzeichen oder zentriert ausrichten. `vorzeichen` kann +, - oder ein Leerzeichen sein, und

typecode entspricht im Allgemeinen den Typcodes für den %-Ausdruck in Tabelle 8 mit folgenden erwähnenswerten Ausnahmen:

- Die Typcodes i und u gibt es nicht; nutzen Sie d, um Integer im Dezimalformat (Basis 10) anzuzeigen.
- Der zusätzliche Typcode b zeigt Integer im Binärformat an (wie bei Verwendung der eingebauten Funktion bin()).
- Der zusätzliche Typcode % zeigt eine Zahl als Prozentwert an.

Ein einzelnes Objekt kann auch mit der eingebauten Funktion format(*objekt*, *formatangabe*) formatiert werden (siehe »Eingebaute Funktionen« auf Seite 105). Formatierungen können mit der __format__-Methode zur Operatorenüberladung in Klassen angepasst werden (siehe »Überladungsmethoden für Operatoren« auf Seite 90).

HINWEIS

In Python 3.1 und neuer fügt ein einem Integer- oder Fließkomma-typecode vorangestelltes »,« Kommata zur Abgrenzung von Tausendern ein:

```
'{0:,d}'.format(1000000)
```

erzeugt '1,000,000' und:

```
'{0:13,.2f}'.format(1000000)
```

ist ' 1,000,000.00'.

Außerdem werden Feldnummern ab Python 3.1 automatisch sequenziell nummeriert, wenn sie in feldname nicht angegeben werden. Die folgenden drei Formulierungen bewirken alle das Gleiche, obwohl automatisch nummerierte Felder weniger gut lesbar sind, wenn es viele Felder gibt:

```
'{0}/{1}/{2}'.format(x, y, z)' # explizite Nummerierung
'{}/{}/{}'.format(x, y, z)     # 3.1 automatische
                                 Nummerierung
'%s/%s/%s' % (x, y, z)         # Ausdruck
```

Template-String-Substitution

In Python 2.4 und höher gibt es eine Form der einfachen String-Substitution, die eine Alternative zum Formatierungsausdruck und

zur Formatierungsmethode ist, die in den letzten Abschnitten beschrieben wurden. Variablen werden üblicherweise über den %-Operator interpoliert:

```
>>> '%(page)i: %(title)s' % {'page':2, 'title': 'PyRef4E'}
'2: PyRef4E'
```

Für einfache Formatierungsaufgaben wurde eine `Template`-Klasse dem `string`-Modul hinzugefügt, in der Substitutionen mit `$` angezeigt werden:

```
>>> import string
>>> t = string.Template('$page: $title')
>>> t.substitute({'page':2, 'title': 'PyRef4E'})
'2: PyRef4E'
```

Die zu ersetzenden Werte können als Schlüsselwortargumente oder als Schlüssel in Dictionaries angegeben werden:

```
>>> s = string.Template('$who likes $what')
>>> s.substitute(who='bob', what=3.14)
'bob likes 3.14'
>>> s.substitute(dict(who='bob', what='pie'))
'bob likes pie'
```

Die Methode `safe_substitute` ignoriert fehlende Schlüssel, statt eine Ausnahme auszulösen:

```
>>> t = string.Template('$page: $title')
>>> t.safe_substitute({'page':3})
'3: $title'
```

String-Methoden

Neben der zuvor beschriebenen `format()`-Methode gibt es weitere String-Methoden, die erweiterte, über String-Ausdrücke hinausgehende Möglichkeiten zur Textverarbeitung bieten. Tabelle 9 führt die verfügbaren String-Methoden auf; `S` ist dabei ein beliebiges String-Objekt. String-Methoden, die Text verändern, liefern immer einen neuen String zurück. Da Strings unveränderlich sind, wird das Objekt nie vor Ort verändert. Siehe auch das `re`-Modul im Abschnitt »Das Mustervergleichsmodul re« auf Seite 162. Dort finden Sie Informationen zu musterbasierten Gegenstücken für einige der String-Methoden.

Tabelle 9: String-Methoden in Python 3.0

```
S.capitalize()
S.center(breite, [, füllzeichen])
S.count(sub [, start [, ende]])
S.encode([encoding [,fehler]])
S.endswith(suffix [, start [, ende]])
S.expandtabs([tabgröße])
S.find(sub [, start [, ende]])
S.format(fmtstr, *args, **kwargs)
S.index(sub [, start [, ende]])
S.isalnum()
S.isalpha()
S.isdecimal()
S.isdigit()
S.isidentifier()
S.islower()
S.isnumeric()
S.isprintable()
S.isspace()
S.istitle(
S.isupper()
S.join(iterable)
S.ljust(breite [, füllzeichen])
S.lower()
S.lstrip([zeichen])
S.maketrans(x[, y[, z]])
S.partition(sep)
S.replace(ald, neu [, anzahl])
S.rfind(sub [,start [,ende]])
S.rindex(sub [, start [, ende]])
S.rjust(breite [, füllzeichen])
```

Tabelle 9: String-Methoden in Python 3.0 (Fortsetzung)

`S.rpartition(sep)`
`S.rsplit([sep[, maxsplits]])`
`S.rstrip([chars])`
`S.split([sep [,maxsplits]])`
`S.splitlines([zeilenenden_bewahren])`
`S.startswith(präfix [, start [, ende]])`
`S.strip([zeichen])`
`S.swapcase()`
`S.title()`
`S.translate(abbildung)`
`S.upper()`
`S.zfill(breite)`

bytes- und bytearray-Methoden

Die bytes- und bytearray-String-Typen von Python 3.0 bieten einen ähnlichen, aber nicht vollkommen identischen, Methodensatz wie der gewöhnliche str-String-Typ (str ist Unicode-Text, bytes sind rohe Binärdaten, und bytearray ist veränderlich). Im Folgenden berechnet `set(X) - set(Y)` Elemente in `X`, die nicht in `Y` sind:

- bytes und bytearray unterstützen keine Unicode-Kodierung (sie repräsentieren rohe Bytes, keinen dekodierten Text) oder String-Formatierung (`str.format` und den mit `__mod__` implementierten `%`-Operator).
- str unterstützt keine Unicode-Dekodierung (es ist bereits dekodierter Text).
- Nur bytearray hat wie Listen Methoden, die das Objekt vor Ort verändern:

```
>>> set(dir(str)) - set(dir(bytes))
{'isprintable', 'format', '__mod__', 'encode',
'isidentifier', '_formatter_field_name_split',
'isnumeric', '__rmod__', 'isdecimal',
'_formatter_parser', 'maketrans'}
```

```
>>> set(dir(bytes)) - set(dir(str))
{'decode', 'fromhex'}

>>> set(dir(bytearray)) - set(dir(bytes))
{'insert', '__alloc__', 'reverse', 'extend',
'__delitem__', 'pop', '__setitem__',
'__iadd__', 'remove', 'append', '__imul__'}
```

Außer Methoden unterstützen `bytes` und `bytearray` die üblichen Sequenzoperationen, die in Tabelle 3 aufgeführt wurden, und `bytearray` unterstützt die Operationen für veränderliche Sequenzen aus Tabelle 4. Mehr Informationen finden Sie in »Unicode-Strings« auf Seite 32 und in »Eingebaute Funktionen« auf Seite 105.

HINWEIS

Die in Python 2.6 verfügbaren String-Methoden sehen etwas anders aus (z.B. gibt es eine `decode`-Methode für das andere Unicode-Typmodell von Python 2.6). Die Schnittstelle des `unicode`-String-Typs von Python 2.6 ist fast mit der von `str`-Objekten unter Python 2.6 identisch. Werfen Sie einen Blick in die Python 2.6 Library Reference oder führen Sie interaktiv `dir(str)` und `help(str.`*`methode`*`)` aus, wenn Sie weitere Informationen benötigen.

Die folgenden Abschnitte befassen sich ausführlicher mit einigen der in Tabelle 9 aufgeführten Methoden. Alle nachfolgenden Operationen, die ein String-Ergebnis liefern, liefern einen neuen String. (Da Strings unveränderlich sind, werden sie nie vor Ort geändert.) Leerraum umfasst Leerzeichen, Tabulatoren und Zeilenendezeichen (alles in `string.whitespace`).

Suchen

`s.find(sub, [, start [, ende]])`
: Gibt die Position des ersten Vorkommens des Strings `sub` in `s` zwischen den Positionen `start` und `ende` (mit den Standardwerten `0` und `len(s)`, also dem ganzen String) zurück. Gibt `-1` zurück, wenn nichts gefunden wird. Siehe auch den auf Enthaltensein prüfenden `in`-Operator, der ebenfalls eingesetzt werden kann, um zu prüfen, ob ein Substring in einem String enthalten ist.

`s.rfind(sub, [, start [, ende]])`
: Wie `find`, sucht jedoch vom Ende her (von rechts nach links).

`s.index(sub [, start [, ende]])`
: Wie `find`, löst aber `ValueError` aus, wenn nichts gefunden wird, statt `-1` zurückzugeben.

`s.rindex(sub [, start [, ende]])`
: Wie `rfind`, löst aber `ValueError` aus, wenn nichts gefunden wird, statt `-1` zurückzugeben.

`s.count(sub [, start [, ende]])`
: Zählt die Anzahl nicht-überlappender Vorkommen von `sub` in `s` zwischen den Positionen `start` und `ende` (Standardwerte: `0` und `len(s)`).

`s.startswith(sub [, start [, ende]])`
: `True`, falls der String `s` mit dem Substring `sub` anfängt. `start` und `ende` sind die optionalen Anfangs- und Endpositionen beim Vergleich mit `sub`.

`s.endswith(sub [, start [, ende]])`
: `True`, falls der String `s` mit dem Substring `sub` endet. `start` und `ende` sind die optionalen Anfangs- und Endpositionen beim Vergleich mit `sub`.

Aufteilen und Zusammenfügen

`s.split([sep [, maxsplits]])`
: Gibt eine Liste von Wörtern im String `s` zurück mit `sep` als Trennstring. Wenn angegeben, bezeichnet `maxsplit` die maximale Anzahl von Aufteilungen. Wenn `sep` nicht angegeben wird oder gleich `None` ist, fungieren alle Whitespace-Strings als Trennstrings. `'a*b'.split('*')` ergibt `['a','b']`. Verwenden Sie `list(s)`, um einen String in eine Liste von Zeichen umzuwandeln, z.B. `['a','*','b']`.

`sep.join(iterierbare Objekte)`
: Verkettet ein iterierbares Objekt (z.B. eine Liste oder ein Tupel) mit Strings zu einem einzigen String mit `sep` dazwischen. `sep` kann `''` (ein leerer String) sein, um eine Liste von Zeichen in einen String umzuwandeln (`'*'.join(['a','b'])` ergibt `'a*b'`).

`s.replace(alt, neu [, anzahl])`
: Gibt eine Kopie des Strings `s` zurück, in der alle Vorkommen des Strings `alt` mit `neu` ersetzt sind. Wenn `anzahl` angegeben wird, werden nur die ersten `anzahl` Vorkommen ersetzt. Das funktioniert wie eine Kombination aus `x=s.split(alt)` und `new.join(x)`.

`s.splitlines([zeilenenden_bewahren])`
: Teilt den String `s` an Zeilenenden auf und gibt eine Liste von Zeilen zurück. Das Ergebnis enthält keine Zeilenendezeichen, außer wenn `zeilenenden_bewahren` wahr ist.

Formatieren

`s.capitalize()`
: Wandelt das erste Zeichen des Strings `s` in einen Großbuchstaben um.

`s.expandtabs([tabgroesse])`
: Ersetzt Tabulatoren in String `s` mit `tabgroesse` Leerzeichen (Standardwert ist 8).

`s.strip([zeichen])`
: Entfernt Whitespace (bzw. die Zeichen in `zeichen`, falls angegeben) am Anfang und Ende des Strings `s`.

`s.lstrip([zeichen])`
: Entfernt Whitespace (bzw. die Zeichen in `zeichen`, falls angegeben) am Anfang des Strings `s`.

`s.rstrip([zeichen])`
: Entfernt Whitespace (bzw. die Zeichen in `zeichen`, falls angegeben) am Ende des Strings `s`.

`s.swapcase()`
: Wandelt alle Klein- in Großbuchstaben um und umgekehrt.

`s.upper()`
: Wandelt alle Buchstaben in Großbuchstaben um.

`s.lower()`
: Wandelt alle Buchstaben in Kleinbuchstaben um.

`s.ljust(breite [, füllzeichen])`
: Richtet den String `s` links in einem Feld mit der Breite `breite` aus und füllt rechts mit `füllzeichen` auf (standardmäßig ein Leerzeichen). Mit dem Ausdruck oder der Methode zur String-Formatierung kann man Ähnliches erreichen.

`s.rjust(breite [, füllzeichen])`
: Richtet den String `s` rechts in einem Feld mit der Breite `breite` aus und füllt links mit `füllzeichen` auf (standardmäßig ein Leerzeichen). Mit dem Ausdruck oder der Methode zur String-Formatierung kann man Ähnliches erreichen.

`s.center(breite [, füllzeichen])`
: Zentriert den String `s` in einem Feld mit der Breite `breite` aus und füllt links sowie rechts mit `füllzeichen` auf (standardmäßig ein Leerzeichen). Mit dem Ausdruck oder der Methode zur String-Formatierung kann man Ähnliches erreichen.

`s.zfill(breite)`
: Füllt String `s` links mit Nullen auf, um einen String der gewünschten `breite` zu erhalten (kann man auch mit dem String-Formatierungsausdruck `%` erreichen).

`s.translate(tabelle [, zu_loeschende_zeichen])`
: Löscht in String `s` alle Zeichen, die in `zu_loeschende_zeichen` vorkommen (falls vorhanden), und übersetzt dann die Zeichen mit `tabelle`, einem String von 256 Zeichen, der für jedes Zeichen an der Position seiner Ordnungszahl ein Ersetzungszeichen enthält.

`s.title()`
: Gibt eine Version des Strings zurück, die den englischen Überschriften entspricht: Wörter beginnen mit Großbuchstaben, während alle weiteren Buchstaben kleingeschrieben werden.

Tests auf Enthaltensein

`s.is*()`
: Die Booleschen `is*()`-Testfunktionen arbeiten auf Strings beliebiger Länge. Sie testen den Inhalt von Strings in Bezug auf verschiedene Kategorien (und geben bei leeren Strings immer `False` zurück).

Das ursprüngliche string-Modul

Seit Python 2.0 sind die meisten der zuvor im Standardmodul `string` enthaltenen Funktionen auch als Methoden von String-Objekten verfügbar. Wenn `X` ein String-Objekt referenziert, ist der Aufruf einer Funktion im Modul `string` wie

```
import string
res = string.replace(X, 'span', 'spam')
```

in Python 2.0 normalerweise einem Aufruf einer String-Methode äquivalent:

```
res = X.replace('span', 'spam')
```

String-Methodenaufrufe sind die bevorzugte und schnellere Form, und für String-Methoden muss kein Modul importiert werden. Beachten Sie, dass die Operation `string.join(liste, grenze)` zu einer Methode des Begrenzerstrings `grenze.join(liste)` wird. In Python 3.0 wurden diese Funktionen vollständig aus dem `string`-Modul entfernt; nutzen Sie stattdessen die äquivalenten Methoden des String-Objekts.

Unicode-Strings

Technisch gesehen ist Text immer Unicode-Text, auch Text im billigen 8-Bit-ASCII-Kodierungsschema. Python unterstützt Unicode-Zeichenfolgen, die alle Zeichen mit 16 oder mehr Bits repräsentieren und nicht nur mit 8. Diese Unterstützung ist in den verschiedenen Python-Zweigen nicht identisch. Python 3.0 behandelt Text grundsätzlich als Unicode und repräsentiert binäre Daten separat, während Python 2.x 8-Bit-Text (und Daten) von Unicode-Text unterscheidet. Im Einzelnen bedeutet das:

- In *Python 2.6* repräsentiert der Typ `str` sowohl 8-Bit-Text als auch binäre Daten, während ein separater `unicode`-Typ für Unicode-Text vorhanden ist. Die `u'ccc'`-Literalform unterstützt die Formulierung binärer Daten, ein `codecs`-Modul bietet Unterstützung für das Lesen und Schreiben von Dateien, die Unicode-Text enthalten.
- In *Python 3.0* kümmert sich der gewöhnliche String-Typ `str` um alle Textformen (8-Bit-Text und auch Unicode-Text), während

ein separater bytes-Typ 8-Bit-Binärdaten repräsentiert. bytes können in der b'ccc'-Literalform formuliert werden. bytes-Objekte enthalten unveränderliche Sequenzen von 8-Bit-Integer-Werten, unterstützen aber die meisten str-Operationen und lassen sich, wenn möglich, als ASCII-Text ausgeben. Außerdem gehen Dateien in 3.0 im Text- und Binärmodus von str- bzw. bytes-Objekten aus; im Textmodus kodieren und dekodieren Dateien automatisch Text. Ein zusätzlicher bytearray-Typ ist eine veränderlich Variante von bytes und bietet zusätzliche listenartige Methoden für Vor-Ort-Änderungen. bytearray gibt es auch in 2.6, bytes hingegen nicht (b'ccc' erzeugt in 2.6 aus Gründen der Kompatibilität einen str).

Unicode-Unterstützung in Python 3.0

Python 3.0 ermöglicht die Kodierung von Nicht-ASCII-Zeichen in Strings mit hexadezimalen (\x) und 16- sowie 32-Bit-Unicode-Escape-Sequenzen (\u, \U). Zusätzlich unterstützt chr() Unicode-Zeichencodes:

```
>>> 'A\xE4B'
'AäB'
>>> 'A\u00E4B'
'AäB'
>>> 'A\U000000E4B'
'AäB'
>>> chr(0xe4)
'ä'
```

Gewöhnliche Strings können in rohen Bytes kodiert werden, und rohe Bytes können zu gewöhnlichen Strings dekodiert werden, wobei entweder eine Standardkodierung oder eine explizit angegebene Kodierung verwendet wird:

```
>>> 'A\xE4B'.encode('latin-1')
b'A\xe4B'
>>> 'A\xE4B'.encode()
b'A\xc3\xa4B'
>>> 'A\xE4B'.encode('utf-8')
b'A\xc3\xa4B'
>>>
>>> b'A\xC3\xA4B'.decode('utf-8')
'AäB'
```

Auch Dateiobjekte kodieren und dekodieren bei Eingabe und Ausgabe automatisch und akzeptieren einen Kodierungsnamen, um die Standardkodierung zu überschreiben:

```
>>> S = 'A\xE4B'
>>> open('uni.txt', 'w', encoding='utf-8').write(S)
3
>>> open('uni.txt', 'rb').read()
b'A\xc3\xa4B'
>>>
>>> open('uni.txt', 'r', encoding='utf-8').read()
'AäB'
```

bytes- und bytearray-Strings

In Python 3.0 repräsentieren bytes- und bytearray-String-Objekte 8-Bit-Binärdaten (einschließlich kodiertem Unicode-Text). Sie werden, wenn möglich, in ASCII ausgegeben und unterstützen die meisten gewöhnlichen str-String-Operationen einschließlich Methoden und Sequenzoperationen:

```
>>> B = b'spam'
>>> B
b'spam'
>>> B[0]                # Sequenzoperatoren
115
>>> B + b'abc'
b'spamabc'
>>> B.split(b'a')       # Methoden
[b'sp', b'm']
>>> list(B)             # int-Sequenz
[115, 112, 97, 109]
```

bytearray unterstützt zusätzlich listentypische veränderliche Operationen:

```
>>> BA = bytearray(b'spam')
>>> BA
bytearray(b'spam')
>>> BA[0]
115
>>> BA + b'abc'
bytearray(b'spamabc')
>>> BA[0] = 116          # Veränderlichkeit
>>> BA.append(115)       # Listenmethode
hods
```

```
>>> BA
bytearray(b'tpams')
```

Siehe auch die Behandlung der `byte`- und `bytearray`-Methoden in »String-Methoden« auf Seite 25 und die Typkonstruktoraufrufe in »Eingebaute Funktionen« auf Seite 105. Python 2.6 bietet `bytearray`, aber nicht `bytes` (in 2.6 ist `b'ccc'` ein Synonym für `'ccc'` und erstellt ein gewöhnliches `str`-Objekt).

Unicode-Unterstützung in Python 2.x

In Python 2.x werden Unicode-Strings mit `u'string'` formuliert (in Python 3.0 werden der gewöhnliche String-Typ und das gewöhnliche String-Literal für Unicode genutzt). Beliebige Unicode-Zeichen können mit der speziellen Escape-Sequenz `\uHHHH` geschrieben werden, wobei `HHHH` eine Zahl mit vier hexadezimalen Ziffern zwischen `0000` und `FFFF` ist. Die traditionelle Escape-Sequenz *`\xHH`* kann ebenfalls genutzt werden, und oktale Escape-Sequenzen können für Zeichen bis zu `+01FF` verwendet werden, was mit `\777` dargestellt wird.

Für Unicode-Strings gelten die gleichen Operationen für unveränderliche Sequenzen wie für gewöhnliche Strings. Normale und Unicode-String-Objekte können in Python 2.x beliebig gemischt werden, eine Kombination aus 8-Bit- und Unicode-Strings liefert immer einen Unicode-String, wobei die Standard-ASCII-Kodierung verwendet wird (z. B. ist das Ergebnis von `'a' + u'bc'` gleich `u'abc'`). Operationen mit gemischten Typen erwarten, dass der 8-Bit-String aus 7-Bit-US-ASCII-Daten besteht (und lösen bei Nicht-ASCII-Zeichen einen Fehler aus). Mit den eingebauten Funktionen `str()` und `unicode()` kann zwischen normalen und Unicode-Strings konvertiert werden. Die String-Methode `encode` wendet das gewünschte Kodierungsschema auf Unicode-Strings an. Einige verwandte Module (z. B. `codecs`) und eingebaute Funktionen sind ebenfalls verfügbar.

Listen

Listen sind veränderliche Sequenzen mit Objektreferenzen, auf die über einen Index zugegriffen wird.

Erzeugung von Literalen

Listen werden als durch Kommata getrennte Werte in eckigen Klammern geschrieben.

`[]`
: Eine leere Liste.

`[0, 1, 2, 3]`
: Eine Liste mit vier Elementen: Indizes 0 bis 3.

`liste = ['spam', [42, 3.1415], 1.23, {}]`
: Verschachtelte Unterliste: `liste[1][0]` holt das Element 42.

`liste = list('spam')`
: Erzeugt eine Liste aller Elemente eines iterierbaren Objekts durch Aufruf des Typkonstruktors.

`liste = [x**2 for x in range(9)]`
: Erzeugt eine Liste durch Sammlung von Ausdrucksergebnissen während der Iteration (Listenkomprehension).

Operationen

Zu den Operationen zählen alle Sequenzoperationen (siehe Tabelle 3), alle Operationen veränderlicher Sequenzen (siehe Tabelle 4) sowie folgende Listenmethoden:

`liste.append(x)`
: Hängt das Objekt `x` ans Ende von `liste` an; ändert die Liste vor Ort.

`liste.extend(x)`
: Fügt alle Elemente im Iterable `x` am Ende von `liste` ein (ein Vor-Ort-+). Ähnlich wie `liste[len(liste):] = list(x)`.

`liste.sort(key=None, reverse=False)`
: Sortiert `liste` vor Ort, standardmäßig in aufsteigender Folge. Wird als `key` eine Funktion übergeben, die ein Argument erwartet, wird diese genutzt, um einen Vergleichswert für jedes Listenelement zu berechnen oder abzurufen. Wird für `reverse` `True` übergeben, werden die Elemente der Liste sortiert, als wären die einzelnen Vergleiche umgekehrt worden, zum Bei-

spiel: `liste.sort(key=str.lower, reverse=True)`. Siehe auch die eingebaute Funktion `sorted()`.

`liste.reverse()`
: Dreht die Reihenfolge der Listenelemente um (ändert Liste).

`liste.index(x [, i [, j]])`
: Liefert den Index des ersten Auftretens von Objekt `x` in der Liste oder löst eine Ausnahme aus, wenn `x` nicht gefunden wird. Das ist eine Suchmethode. Werden `i` und `j` übergeben, wird das kleinste `k` zurückgegeben, für das gilt: `s[k] == x` und `i <= k < j`.

`liste.insert(i, x)`
: Fügt Objekt `x` an Position `i` in die Liste ein (wie `liste[i:i] = [x]` für `i > 0`).

`liste.count(x)`
: Liefert die Anzahl an Vorkommen von `x` in der Liste.

`liste.remove(x)`
: Löscht das erste Auftreten von Objekt `x` aus der Liste; löst eine Ausnahme aus, wenn `x` nicht gefunden werden kann, wie `del liste[liste.index(x)]`.

`liste.pop([i])`
: Löscht das letzte Element (oder das an Position `i`) aus der Liste und gibt es zurück. Formt gemeinsam mit `append` eine einfache Stack-Implementierung. Dasselbe wie `x=liste[i], del liste[i], return x`. Standardwert für `i` ist `-1` (das letzte Element).

HINWEIS

In Python 2.x lautete die Signatur der `sort`-Methode für Listen `liste.sort(cmp=None, key=None, reverse=False)`. `cmp` ist dabei eine Vergleichsfunktion mit zwei Argumenten, die einen Rückgabewert kleiner, gleich oder größer null liefert, um als Vergleichsergebnisse kleiner, gleich bzw. größer anzuzeigen. Diese Vergleichsfunktion wurde in 3.0 entfernt, weil sie üblicherweise genutzt wurde, um Sortierwerte zuzuordnen und die Sortierfolge umzukehren – zwei Anwendungsfälle, die durch die anderen beiden Argumente bereits gedeckt werden.

Listenkomprehensionsausdrücke

Ein Listenliteral in eckigen Klammern ([...]) kann eine einfache Liste von Ausdrücken oder ein Listenkomprehensionsausdruck der folgenden Form sein:

```
[ausdruck for ausdruck1 in iterierbares1 [if bedingung1]
          for ausdruck2 in iterierbares2 [if bedinungen2] ...
          for ausdruckN in iterierbaresN [if bedingungN] ]
```

Listenkomprehensionen sammeln die Werte für *ausdruck* über alle Iterationen über alle geschachtelten `for`-Schleifen, für die jede der optionalen Bedingungen wahr ist. Die zweite bis *n*-te `for`-Schleife und alle `if`-Klauseln sind optional, und *ausdruck* und *bedingungX* können Variablen enthalten, die in geschachtelten `for`-Schleifen zugewiesen werden. Namen, die innerhalb der Komprehensionen gebunden sind, werden in jenem Geltungsbereich erzeugt, in dem sich die Komprehension befindet.

Listenkomprehensionen ähneln der eingebauten Funktion `map()`:

```
>>> [ord(x) for x in 'spam']
[115, 112, 97, 109]
>>> map(ord, 'spam')
[115, 112, 97, 109]
```

Sie können jedoch helfen, eine temporäre Funktion zu vermeiden:

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
>>> map((lambda x: x**2), range(5))
[0, 1, 4, 9, 16]
```

Komprehensionen mit Bedingungen ähneln `filter`:

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]
>>> filter((lambda x: x % 2 == 0), range(5))
[0, 2, 4]
```

Komprehensionen mit geschachtelten `for`-Schleifen ähneln der normalen `for`-Anweisung:

```
>>> [y for x in range(3) for y in range(3)]
[0, 1, 2, 0, 1, 2, 0, 1, 2]
>>> res = []
>>> for x in range(3):
```

```
...     for y in range(3):
...         res.append(y)
>>> res
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

Generator-Ausdrücke

Ab Python 2.4 leisten Generator-Ausdrücke Ähnliches wie Listenkomprehensionen, ohne aber eine physische Liste zu erzeugen, die die Ergebnisse aufnimmt. Generator-Ausdrücke sparen Speicherplatz, da sie eine Ergebnismenge definieren, aber nicht die gesamte Liste erzeugen, sondern einen Generator, der die Elemente in Iterationskontexten nach und nach erzeugt. Ein Beispiel:

```
ords = (ord(x) for x in aString if x not in skipThese)
for o in ords:
    ...
```

Generator-Ausdrücke werden in runden statt in eckigen Klammern formuliert, unterstützen aber ansonsten die gesamte Syntax von Listenkomprehensionen. Zum Erstellen eines Iterators, der an eine Funktion übergeben wird, genügen die runden Klammern einer Funktion mit einem einzigen Argument:

```
sum(ord(x) for x in aString)
```

Auf die Schleifenvariablen von Generator-Ausdrücken (`x` im vorangehenden Beispiel) kann außerhalb des Generator-Ausdrucks nicht zugegriffen werden. In Python 2.x bleibt der Variablen bei Listenkomprehensionen der letzte Wert zugewiesen; Python 3.0 ändert das, damit sich der Geltungsbereich von Listenkomprehensionen verhält wie der von Generator-Ausdrücken.

Nutzen Sie die `__next__()`-Methode des Iterator-Protokolls (`next()` in Python 2.x), um die Ergebnisse außerhalb eines Iterationskontexts wie in `for`-Schleifen zu durchlaufen, und nutzen Sie den `list`-Aufruf, um gegebenenfalls eine Liste aller Ergebnisse zu erzeugen:

```
>>> squares = (x ** 2 for x in range(5))
>>> squares
<generator object <genexpr> at 0x027C1AF8>
>>> squares.__next__()
0
>>> squares.__next__()              # oder next(squares)
```

```
1
>>> list(x ** 2 for x in range(5))
[0, 1, 4, 9, 16]
```

Mehr Informationen zu Generatoren und Iteratoren, einschließlich zusätzlicher Generator-Methoden, finden Sie in »Die yield-Anweisung« auf Seite 70.

Andere Generatoren und Komprehensionen

Über die verwandte `yield`-Anweisung sowie Dictionary-Komprehensionen und Set-Komprehensionen gibt es an anderen Stellen dieses Buchs mehr. Die beiden letzten sind ähnliche Ausdrücke, die Dictionaries und Sets erzeugen; sie unterstützen die gleiche Syntax wie Listenkomprehensionen und Generator-Ausdrücke, werden aber in {} gepackt und Dictionary-Komprehensionen beginnen mit einem `key:value`-Ausdruckspaar:

```
>>> [x * x for x in range(10)]      # Listenkomprehension
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> (x * x for x in range(10))      # Generator-Ausdruck
<generator object at 0x009E7328>
>>> {x * x for x in range(10)}      # Set-Komp. (3.0)
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> {x: x * x for x in range(10)}   # Dict-Komp. (3.0)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,
8: 64, 9: 81}
```

Dictionaries

Dictionaries sind veränderliche Tabellen mit Objektreferenzen. Der Zugriff erfolgt über den Schlüssel, nicht über die Position. Es sind ungeordnete Sammlungen, die intern als dynamisch erweiterbare Hashtabellen implementiert sind. Dictionaries haben sich in Python 3.0 erheblich verändert:

- In Python 2.x liefern die Methoden `keys()`/`values()`/`items()` Listen. Es gibt eine `has_key()`-Suchmethode, und es gibt eigene Iterator-Methoden, `iterkeys()`/`itervalues()`/`iteritems()`; außerdem können Dictionaries direkt verglichen werden.
- In Python 3.0 liefern die Methoden `keys()`/`values()`/`items()` *View*-Objekte statt Listen. `has_key()` wurde entfernt, und statt-

dessen sollen in-Ausdrücke verwendet werden. Die Iterator-Methoden von Python 2.x wurden entfernt, stattdessen soll die View-Objekt-Iteration genutzt werden. Dictionaries können nicht mehr direkt verglichen werden, aber ihr `sorted(D.items())`-Resultat; außerdem gibt es einen neuen Dictionary-Komprehensionsausdruck.

- Die *View*-Objekte von Python 3.0 erzeugen ihre Ergebnisse auf Anforderung, bewahren die ursprüngliche Reihenfolge im Dictionary, spiegeln spätere Änderungen am Dictionary und können Set-Operationen unterstützen. Schlüssel-Views sind immer Set-artig, Wert-Views nie, Element-Views sind es, wenn alle ihre Elemente eindeutig und hashbar (unveränderlich) sind. Unter »Sets« auf Seite 50 finden Sie einige Set-Ausdrücke, die auf einige Views angewandt werden können. Übergeben Sie Views an einen `list()`-Aufruf, um eine sofortige Generierung all ihrer Ergebnisse zu erzwingen (z.B. zur Anzeige oder für `list.sort()`).

Literale und Erzeugung

Dictionary-Literale werden als kommaseparierte Folge von *`Schlüssel:Wert`*-Paaren in geschweiften Klammern angegeben, die eingebaute Funktion `dict()` unterstützt andere Erzeugungsmuster, und die Komprehensionen von Python 3.0 nutzen Iteration. Zuweisungen an neue Schlüssel erzeugen neue Einträge. Schlüssel können beliebige unveränderliche Objekte (z.B. Strings, Zahlen, Tupel) sowie Klasseninstanzen sein, wenn sie die Methoden des Hashing-Protokolls geerbt haben. Tupel-Schlüssel unterstützen zusammengesetzte Werte (z.B. `eindict[(M,D,Y)]`, wobei die Klammern optional sind).

`{}`
: Ein leeres Dictionary.

`{'spam': 2, 'eggs': 3}`
: Ein Dictionary mit zwei Elementen: Schlüssel `'spam'` und `'eggs'`.

`eindict = { 'info': { 42: 1, type(''): 2 }, 'spam': [] }`
: Verschachtelte Dictionaries: `dict['info'][42]` ergibt `1`.

`eindict = dict(name='Bob', age=45, job=('mgr', 'dev'))`
: Erstellt ein Dictionary durch Übergabe von Schlüsselwortargumenten an den Typkonstruktor.

`eindict = dict(zip('abc', [1, 2, 3]))`
: Erzeugt ein Dictionary, indem eine Schlüssel/Wert-Tupel-Liste an den Typkonstruktor übergeben wird.

`eindict = dict([['a', 1], ['b', 2], ['c', 3]])`
: Bewirkt das Gleiche wie die vorangehende Zeile: alle iterierbaren Sammlungen mit Schlüsseln und Werten.

`eindict = {c: ord(c) for c in 'spam'}`
: Dictionary-Komprehensionsausdruck (Python 3.x). Die Syntax finden Sie unter »Listenkomprehensionsausdrücke« auf Seite 38.

Operationen

Dictionaries unterstützen alle Operationen für Abbildungen (siehe Tabelle 5) und die folgenden Dictionary-spezifischen Methoden.

`eindict.keys()`
: Alle Schlüssel in `eindict`. Liefert in Python 2.x eine Liste, in Python 3.0 eins der zuvor beschriebenen iterierbaren View-Objekte. `for k in adict:` unterstützt ebenfalls die Iteration über Schlüssel.

`eindict.values()`
: Alle in `eindict` gespeicherten Werte. Liefert in Python 2.x eine Liste, in Python 3.0 eins der oben erwähnten iterierbaren View-Objekte.

`eindict.items()`
: Liefert Tupel-Paare (`Schlüssel, Wert`) für jeden Eintrag in `eindict`. Liefert in Python 2.x eine Liste, in Python 3.0 ebenfalls ein iterierbares View-Objekt.

`eindict.clear()`
: Löscht alle Einträge aus `eindict`.

`eindict.copy()`
: Liefert eine flache Kopie (oberste Ebene) von `eindict`.

`dict1.update(dict2)`
: Mischt zwei Dictionaries (ändert `dict1` an Ort und Stelle). Ähnlich wie `for (k, v) in dict2.items(): dict1[k] = v`. Akzeptiert in Python 2.4 auch ein iterierbares Objekt mit Schlüssel/Wert-Paaren ebenso wie Schlüsselwortargumente (`dict.update (k1=v1, k2=v2)`).

`eindict.get(schlüssel [, standard])`
: Ähnlich wie `eindict[schlüssel]`, liefert jedoch `standard` (`None`, falls nicht angegeben), statt eine Ausnahme auszulösen, wenn `schlüssel` nicht existiert.

`eindict.setdefault(schlüssel, [, standard])`
: Identisch mit `eindict.get(schlüssel, standard)`, weist aber `schlüssel` den Wert `standard` zu, wenn `schlüssel` nicht existiert.

`eindict.popitem()`
: Löscht Schlüssel/Wert-Paar und liefert es zurück.

`eindict.pop(k [, standard])`
: Liefert `eindict[k]`, wenn `k` in `eindict` enthalten ist (und entfernt `k` aus `eindict`); liefert andernfalls `standard`.

`eindict.fromkeys(seq [, wert])`
: Erzeugt ein neues Dictionary mit Schlüsseln aus `seq` mit dem jeweils gleichen `wert`.

Die folgenden Methoden sind nur in Python 2.x verfügbar:

`eindict.has_key(k)`
: Liefert `True`, wenn `eindict` den Schlüssel `k` enthält, andernfalls `False`. In Python 2.x entspricht diese Methode `k in eindict`. Von ihrer Verwendung wird allerdings abgeraten. In Python 3.0 wurde sie entfernt.

`eindict.iteritems()`, `eindict.iterkeys()`, `eindict.itervalues()`
: Liefert einen Iterator über Schlüssel/Wert-Paare, Schlüssel oder Werte. Diese Methoden wurden in Python 3.0 entfernt, weil `keys()`, `values()` und `items()` iterierbare View-Objekte liefern.

Die folgenden Operationen wurden in Tabelle 5 beschrieben, beziehen sich aber auf die oben aufgeführten Methoden:

`k in eindict`
: Liefert `True`, wenn `eindict` den Schlüssel `k` enthält, andernfalls `False`. Ersetzt in Python 3.0 `has_key()`.

`for k in eindict:`
: Iteriert über alle Schlüssel in `eindict` (alle Iterationskontexte). Dictionaries unterstützen die direkte Iteration. `for key in dict` ähnelt `for key in dict.keys()`. Die erste Variante nutzt den Schlüssel-Iterator des Dictionary-Objekts. In Python 2.x liefert `keys()` eine neue Liste und führt damit einen kleinen Overhead ein. In Python 3.0 liefert `keys()` ein iterierbares View-Objekt anstelle einer physisch gespeicherten Liste, was beide Formen äquivalent macht.

Tupel

Tupel sind unveränderliche Sequenzen mit Objektreferenzen, die numerisch indiziert werden.

Literale

Tupel werden als durch Kommata getrennte Folgen von Werten in runden Klammern eingeschlossen. Die umschließenden Klammern können manchmal weggelassen werden, etwa in Schleifenköpfen oder Zuweisungen mit »=«.

`()`
: Ein leeres Tupel.

`(0,)`
: Tupel mit einem Element (kein einfacher Ausdruck, man beachte das Komma).

`(0, 1, 2, 3)`
: Tupel mit vier Elementen.

`0, 1, 2, 3`
: Das gleiche Vierer-Tupel in anderer Schreibweise; in Funktionsaufrufen nicht erlaubt.

`tupel = ('spam', (42, 'eggs'))`
: Verschachtelte Tupel: `tupel[1][1]` holt `'eggs'`.

`tupel = tuple('spam')`
: Erstellt ein Tupel mit allen Elementen eines iterierbaren Objekts durch Aufruf des Typkonstruktors.

Operationen

Tupel unterstützen alle Sequenzoperationen (siehe Tabelle 3) plus die folgenden Tupel-spezifischen Methoden in Python 2.6, 3.0 und später:

`tupel.index(x [, i [, j]])`
: Liefert den Index des ersten Vorkommens von Objekt `x` in `tupel`; löst eine Ausnahme aus, wenn `x` nicht gefunden wird. Eine Suchmethode. Werden `i` und `j` übergeben, wird das kleinste `k` geliefert, für das `s[k] == x` und `i <= k < j` gilt.

`tupel.count(x)`
: Liefert die Anzahl an Vorkommen von `x` in `tupel`.

Dateien

Die eingebaute Funktion `open()` erstellt ein Dateiobjekt, die gebräuchlichste Dateischnittstelle. Dateiobjekte exportieren die Datentransfermethoden im nächsten Abschnitt. In Python 2.x kann `file()` als Synonym für `open()` genutzt werden, wenn ein Dateiobjekt erstellt wird. `open()` ist allerdings die allgemein empfohlene Variante; in Python 3.0 ist `file()` nicht mehr verfügbar.

Vollständige Informationen zur Erstellung von Dateien finden Sie bei der Funktion `open()` im Abschnitt »Eingebaute Funktionen« auf Seite 105. Der Unterschied zwischen Text- und Binärdateien und den damit korrespondierenden String-Typunterschieden in Python 3.0 wird in »Unicode-Strings« auf Seite 32 erläutert.

Verwandte dateiartige Werzkzeuge werden weiter unten in diesem Buch behandelt; siehe die Module `dbm`, `shelve` und `pickle` im Abschnitt »Module zur Objekt-Persistenz« auf Seite 169, außerdem die deskriptorbasierten Dateifunktionen des `os`-Moduls und die Verzeichnispfadwerkzeuge von `os.path` im Abschnitt »Das Systemmodul os« auf Seite 146 sowie die Python SQL-Datenbank-API im Abschnitt »Die portable SQL-Datenbank-API« auf Seite 186.

Eingabedateien

`infile = open('data.txt', 'r')`
: Erzeugt ein Dateiobjekt für die Eingabe (`'r'` steht für »read«, lesen als Text, während `'rb'` binär liest, ohne Zeilenenden zu interpretieren). Der Dateiname (z.B. `'data.txt'`) bezieht sich auf das aktuelle Arbeitsverzeichnis, falls ihm kein Verzeichnispfad vorangestellt ist (z.B. `'c:\\dir\\data.txt'`). Der Modus ist optional und standardmäßig `'r'`.

`infile.read()`
: Liest die gesamte Datei als einen einzigen String ein. Im Textmodus (`'r'`) werden Zeilenenden in `'\n'` übersetzt. Im Binärmodus (`'rb'`) kann der gelesene String nicht-druckbare Zeichen enthalten (z.B. `'\0'`). In Python 3.0 wird Unicode-Text im Textmodus zu einem `str`-String dekodiert, während der Inhalt im Binärmodus unverändert in einem `bytes`-Objekt bereitgestellt wird.

`infile.read(N)`
: Liest maximal `N` weitere Bytes (1 oder mehr), am Dateiende ist ein leerer String.

`infile.readline()`
: Liest die nächste Zeile (bis einschließlich Zeilenendezeichen), am Dateiende ist eine leerer String.

`infile.readlines()`
: Liest die ganze Datei als Liste von Zeilenstrings mit Endezeichen. Siehe auch Datei-Iteratoren, die gleich behandelt werden.

`for zeile in infile:`
: Nutzt Datei-Iteratoren, um die Zeilen in einer Datei automatisch zu durchlaufen. In allen anderen Iterationskontexten ebenfalls verfügbar (z.B. `[zeile[:-1] for zeile in open('datei-name')]`). Die Iterationen für `for zeile in dateiobj:` hat eine ähnliche Wirkung wie `for zeile in dateiobj.readlines():`, aber die Iterator-Version lädt nicht die gesamte Datei in den Speicher und ist deswegen effizienter.

Ausgabedateien

`outfile = open('/tmp/spam', 'w')`
: Erzeugt ein Dateiobjekt zur Ausgabe (`'w'` steht für »write«, schreiben, `'wb'` schreibt binäre Daten, ohne Zeilenenden zu interpretieren).

`outfile.write(S)`
: Schreibt String `S` in die Datei (alle Bytes in `S`, ohne eine Formatierung vorzunehmen). Im Textmodus wird `'\n'` in das plattformspezifische Zeilenendezeichen übersetzt. Im Binärmodus kann der String nichtdruckbare Bytes enthalten (zum Beispiel `'a\0b\0c'`, um fünf Bytes zu schreiben, von denen zwei Null-Bytes sind). In Python 3.0 wird `str`-Unicode-Text im Textmodus kodiert, während `bytes` im Binärmodus unverändert geschrieben werden.

`outfile.writelines(L)`
: Schreibt alle Strings in Liste `L` in die Datei.

Alle Dateien

`datei.close()`
: Manuelles Schließen zur Freigabe von Ressourcen (aktuell schließt Python Dateien automatisch, wenn sie bei der Garbage-Collection immer noch offen sind). Siehe auch die später folgende Diskussion zum Kontextmanager von Dateiobjekten.

`datei.tell()`
: Liefert die aktuelle Position in der Datei.

`datei.seek(verschiebung [, von_wo])`
: Verschiebt die aktuelle Dateiposition für wahlfreien Zugriff um `verschiebung`. `von_wo` kann 0 (Verschiebung vom Anfang), 1 (positive oder negative Verschiebung von der aktuellen Position) oder 2 (Verschiebung vom Ende) sein. Der Standard für `von_wo` ist 0.

`file.isatty()`
: Ergibt 1, wenn die Datei mit einem konsolenähnlichen interaktiven Gerät (tty) verbunden ist.

`file.flush()`
: Leert den `stdio`-Puffer der Datei (wie `fflush` in C). Nützlich für gepufferte Pipes, wenn ein anderer Prozess (oder Mensch) sie zu lesen versucht. Ebenfalls nützlich bei Dateien, die vom gleichen Prozess gelesen und geschrieben werden.

`file.truncate([größe])`
: Schneidet die Datei bis auf maximal `größe` Bytes ab (oder an der aktuellen Position, wenn `größe` nicht angegeben wurde). Nicht auf allen Plattformen verfügbar.

`file.fileno()`
: Gibt die Dateinummer (Integer-Zahl des internen Datei-Deskriptors) für die Datei zurück. Konvertiert quasi Dateiobjekte in Deskriptoren, die an die Werkzeuge des `os`-Moduls übergeben werden können. Tipp: Nutzen Sie `os.fdopen` und `socket-obj.makefile`, um Deskriptoren bzw. Sockets in Dateiobjekte umzuwandeln; `io.StringIO` (`StringIO.StringIO` in Python 2.x) wandelt einen String in ein Objekt mit einer dateiähnlichen Schnittstelle um.

Attribute (alle nur lesbar)

`datei.closed`
: `True`, wenn Datei geschlossen wurde.

`datei.mode`
: Modusstring (z. B. `'r'`), mit dem die Datei geöffnet wurde.

`datei.name`
: String mit dem Namen der entsprechenden externen Datei.

Datei-Kontextmanager

In Standard-Python (CPython) schließen sich Dateiobjekte automatisch, wenn sie bei der Garbage-Collection noch offen sind. Deswegen müssen temporäre Dateien (z. B. `open('name').read()`) nicht explizit geschlossen werden. Um zu sichern, dass Dateien, unabhängig davon, ob eine Ausnahme ausgelöst wurde, geschlossen werden, nachdem ein Codeblock ausgeführt wurde, können Sie die `try/finally`-Anweisung einsetzen und die Dateien manuell schließen:

```
datei = open(r'C:\misc\script', 'w')
try:
    ...Datei nutzen...
finally:
    datei.close()
```

Oder Sie können die seit Python 2.6 und 3.0 verfügbare `with/as`-Anweisung nutzen:

```
with open(r'C:\misc\script', 'w') as datei:
    ...Datei nutzen...
```

Die erste Variante setzt den `close()`-Aufruf als Abschlussaktion ein. Die zweite nutzt Dateiobjekt-Kontextmanager, die garantieren, dass eine Datei automatisch geschlossen wird, nachdem die Ausführung des eingeschlossenen Codeblocks abgeschlossen ist. Siehe die `try`- und `with`-Anweisungen in »Anweisungen und Syntax« auf Seite 54.

Anmerkungen

- Einige Modi (z.B. `'r+'`) gestatten, dass eine Datei für die Ein- und Ausgabe geöffnet wird, andere (z.B. `'rb'`) geben einen Transfer im Binärmodus an und unterdrücken die Umwandlung von Zeilenendezeichen (und in Python 3.0 auch Unicode-Kodierungen). Siehe `open()` im Abschnitt »Eingebaute Funktionen« auf Seite 105.
- Dateioperationen erfolgen an der aktuellen Dateiposition, aber Aufrufe der `seek`-Methode repositionieren die Datei für den wahlfreien Zugriff.
- Dateioperationen können *ungepuffert* sein; siehe dazu `open()` im Abschnitt »Eingebaute Funktionen« auf Seite 105 und den Kommandozeilenschalter `-u` im Abschnitt »Kommandozeilenoptionen« auf Seite 3.
- In Python 2.x bieten Dateiobjekte außerdem eine `xreadlines()`-Methode, die wie der automatische Zeilen-Iterator des Dateiobjekts arbeitet. Da sie redundant ist, wurde sie in Python 3 entfernt.

Sets

Sets sind veränderliche und ungeordnete Sammlungen eindeutiger und unveränderlicher Objekte. Sets unterstützen mathematische Mengenoperationen wie Vereinigungsmengen und Schnittmengen. Es sind keine Sequenzen (da sie ungeordnet sind) und keine Abbildungen (da sie keine Schlüssel/Wert-Zuordnung vornehmen), sie unterstützen aber die Iteration und verhalten sich ähnlich wie wertlose Dictionaries (Dictionaries, die nur Schlüssel enthalten).

Literale und Erstellung

In Python 2.x und 3.0 können Sets erstellt werden, indem die eingebaute Funktion `set()` mit einem iterierbaren Objekt aufgerufen wird, dessen Elemente zu Mitgliedern des resultierenden Sets werden. In Python 3.0 können Sets auch mit dem Set-Literal `{...}` und der Set-Komprehensionsausdruckssyntax erstellt werden, obwohl `set()` immer noch genutzt wird, um ein leeres Set zu erstellen ({} ist das leere Dictionary) und Sets auf bestehenden Objekten zu erstellen. Sets sind veränderlich, aber die Elemente eines Sets müssen unveränderlich sein; die eingebaute Funktion `frozenset()` erstellt ein unveränderliches Set, das in ein anderes Set geschachtelt werden kann.

`set()`
: Ein leeres Set.

`menge = set('spam')`
: Ein Set mit vier Elementen, Werte: `'s'`, `'p'`, `'a'`, `'m'` (es werden beliebige iterierbare Objekte akzeptiert).

`menge = {'s', 'p', 'a', 'm'}`
: Ein Set mit vier Elementen, identisch mit der vorangegangenen Zeile (Python 3).

`menge = {ord(c) for c in 'spam'}`
: Set-Komprehensionsausdruck (Python 3); mehr zur Syntax finden Sie unter »Listenkomprehensionsausdrücke« auf Seite 38.

`menge = frozenset(range(-5, 5))`
: Ein fixiertes (gefrorenes, d.h. unveränderliches) Set mit zehn Integern von `-5...4`.

Operationen

Der folgende Abschnitt beschreibt die relevantesten Set-Operationen, ist aber nicht vollständig; eine erschöpfende Liste der Set-Ausdrücke und verfügbaren Methoden finden Sie in Pythons Library Reference. Die meisten Ausdrucksoperatoren erfordern zwei Sets, während ihre methodenbasierten Äquivalente mit jedem iterierbaren Objekt zufrieden sind (was im Folgenden mit »anderes« bezeichnet wird), z. B. schlägt `{1, 2} | [2, 3]` fehl, während `{1, 2}.union([2, 3])` funktioniert:

`wert in menge`
: Enthaltensein: Liefert `True`, wenn `menge` `wert` enthält.

`menge1 – menge2`, `menge1.difference(anderes)`
: Differenzmenge: Neues Set, das alle Elemente in `menge1` enthält, die nicht in `menge2` enthalten sind.

`menge1 | menge2`, `menge1.union(andres)`
: Vereinigungsmenge: Neues Set, das alle Elemente aus den Mengen `menge1` und `menge2` enthält; ohne Duplikate.

`menge1 & menge2`, `menge1.intersection(anderes)`
: Schnittmenge: Neues Set, das alle Elemente enthält, die in `menge1` und `menge2` enthalten sind.

`menge1 <= menge2`, `menge1.issubset(anderes)`
: Teilmenge: Prüft, ob alle Elemente in `menge1` auch in `menge2` enthalten sind.

`menge1 >= menge2`, `menge2.issubset(anderes)`
: Obermenge: Prüft, ob alle Elemente in `menge2` auch in `menge1` enthalten sind.

`menge1 < menge2`, `menge1 > menge2`
: Echte Teil- und Obermenge: Stellt ebenfalls sicher, dass `menge1` und `menge2` nicht identisch sind.

`menge1 ^ menge2`, `menge1.symmetric_difference(anderes)`
: Symmetrische Differenz: Neues Set mit allen Elementen, die entweder in `menge1` oder in `menge2` enthalten sind, aber nicht in beiden.

`menge1 |= menge2`, `menge1.update(anderes)`
: Aktualisierung (nicht bei fixierten Sets): Fügt Elemente aus `menge2` zu `menge1` hinzu.

`menge1.add(X)`, `menge1.remove(X)`, `menge1.discard(X)`, `menge1.pop()`, `menge1.clear()`
: Aktualisierung (nicht bei fixierten Sets): Fügt ein Element hinzu, entfernt ein Element, entfernt ein vorhandenes Element, entfernt ein beliebiges Element und liefert es zurück, entfernt alle Elemente.

`len(menge)`
: Länge: Zählt die Elemente in einem Set.

`for x in menge:`
: Iteration: alle Iterationskontexte.

`menge.copy()`
: Erstellt eine Kopie von `menge`; entspricht `set(menge)`.

Andere wichtige Typen

Zu Pythons zentralen Typen zählen weiterhin *Boolesche Werte* (die im Anschluss beschrieben werden), `None` (ein Platzhalterobjekt), *Typen* (auf die mit der eingebauten Funktion `type()` zugegriffen wird und die in Python 3.x immer Klassen sind) und *Typen für Programmeinheiten* wie Funktionen, Module und Klassen (in Python Laufzeitobjekte).

Boolesche Werte

Der Boolesche Typ namens `bool` verfügt über zwei vordefinierte Konstanten für die Wahrheitswerte wahr und falsch namens `True` und `False` (seit Python 2.3). In den meisten Fällen können diese Konstanten so behandelt werden, als hätten sie jeweils die Werte 1 und 0 (so ergibt z.B. `True + 3` den Wert `4`). Allerdings ist der Typ `bool` eine Unterklasse der Klasse `int` für Ganzzahlen und passt sie so an, dass Instanzen anders ausgegeben werden (`True` wird als »True« ausgeben, nicht als `1`, und kann in Booleschen Tests als eingebauter Name verwendet werden).

Typumwandlungen

Die Tabellen Tabelle 10 und Tabelle 11 definieren wichtige eingebaute Werkzeuge, mit denen Typen in andere Typen umgewandelt

werden können. (Python 2 unterstützt zusätzlich die Umwandlung in Long- und String-Werte mit `long(S)` und `` `X` ``, die in Python 3.0 entfernt wurden.)

Tabelle 10: Sequenzumwandlungen

Wandler	von	nach
`list(X)`, `[n for n in X]`[a]	String, Tupel, iterierbare Objekte	Liste
`tuple(X)`	String, Liste, iterierbare Objekte	Tupel
`''.join(X)`	iterierbare Objekte mit Strings	String

[a] Die Listenkomprehensionsform kann langsamer sein als `list()`. In Python 2.x bewirkt `map(None, X)` in diesem Kontext das Gleiche wie `list(X)`. In Python 3.0 wurde diese Form von `map()` entfernt.

Tabelle 11: String-/Objektumwandlungen

Wandler	von	nach
`eval(S)`	String	jedes Objekt mit einer Syntax (Ausdruck)
`int(S [, basis])`,[a] `float(S)`	String oder Zahl	Integer, Fließkommazahl
`repr(X)`, `str(X)`	jedes Python-Objekt	String (`repr` in Codeform, `str` benutzerfreundlich)
`F % X`, `F.format(X)`	Objekte mit Formatcodes	String
`hex(X)`, `oct(X)`, `bin(X)`, `str(X)`	Ganzzahltypen	hexadezimale, oktale, binäre, dezimale Strings
`ord(C)`, `chr(I)`	Zeichen, ganzzahlige Codes	ganzzahlige Codes, Zeichen

[a] In Version 2.2 und höher dienen die Umwandlungsfunktionen (z. B. `int`, `float`, `str`) auch als Klassenkonstruktoren und können erweitert werden. In Python 3.0 sind diese Typen sämtlich Klassen und alle Klassen Instanzen der Klasse `type`.

Anweisungen und Syntax

Dieser Abschnitt beschreibt die Regeln für Syntax und Variablennamen.

Syntaxregeln

Dies sind die Grundregeln beim Schreiben von Python-Programmen:

Kontrollfluss
: Anweisungen werden der Reihe nach ausgeführt, außer es werden den Kontrollfluss ändernde, zusammengesetzte Anweisungen verwendet (`if`, `while`, `for`, `raise`, Aufrufe usw.).

Blöcke
: Blöcke werden dadurch abgegrenzt, dass alle ihre Anweisungen um die gleiche Anzahl Leerzeichen oder Tabulatoren eingerückt werden. Ein Tabulator entspricht der Anzahl von Leerzeichen, mit der die Spaltennummer das nächste Vielfache von acht erreicht. Blöcke dürfen in der gleichen Zeile stehen wie die Kopfzeile der Anweisung, wenn sie aus einer einfachen Anweisung bestehen.

Anweisungen
: Anweisungen gehen normalerweise bis zum Zeilenende, können sich aber über mehrere Zeilen fortsetzen, wenn die Zeile mit `\` endet, ein Klammernpaar wie `()`, `[]` oder `{}` nicht geschlossen wurde oder aber ein dreifach quotierter String offen ist. Mehrere Anweisungen in einer Zeile werden durch Semikola (`;`) getrennt.

Kommentare
: Kommentare beginnen mit einem Doppelkreuz `#` (aber nicht innerhalb einer String-Konstanten) und reichen bis zum Zeilenende.

Dokumentationsstrings
: Wenn eine Funktion, Moduldatei oder eine Klasse mit einem String-Literal beginnt, wird diese im `__doc__`-Attribut des Objekts gespeichert. Siehe die `help()`-Funktion sowie das Modul und Skript `pydoc` in der Python Library Reference, das

Werkzeuge enthält, mit denen diese automatisch herausgezogen und dargestellt werden können.

Leerraum
: Im Allgemeinen nur links vom Code relevant, wo Blöcke mittels Einrückung gruppiert werden. Leerzeilen und Leerraum werden ansonsten ignoriert, außer als Token-Begrenzer und in String-Literalen.

Namensregeln

Dieser Abschnitt enthält die Regeln für benutzerdefinierte Namen (z.B. Variablen) in Programmen.

Namensformat

Struktur
: Benutzerdefinierte Namen beginnen mit einem Buchstaben oder Unterstrich (_), gefolgt von beliebig vielen Buchstaben, Unterstrichen oder Ziffern.

Reservierte Wörter
: Benutzerdefinierte Namen dürfen keinem von Python reservierten Wort (siehe Tabelle 12) entsprechen.[3]

Groß-/Kleinschreibung
: Groß-/Kleinschreibung ist bei benutzerdefinierten Namen und reservierten Wörtern immer relevant: *SPAM*, *spam* und *Spam* sind unterschiedliche Namen.

Nicht verwendete Zeichen (Token)
: Die Zeichen $ und ? werden in der Python-Syntax nicht genutzt; sie können aber in String-Literalen und Kommentaren auftauchen, und $ hat eine spezielle Bedeutung bei der Template-Substitution in Strings.

Erzeugung
: Benutzerdefinierte Namen werden erzeugt, wenn ihnen etwas zugewiesen wird; sie müssen aber existieren, wenn sie referen-

3 In Jython, der Java-basierten Implementierung von Python, können benutzerdefinierte Namen manchmal identisch mit reservierten Wörtern sein.

ziert werden (z.B. müssen Zähler explizit mit null initialisiert werden). Siehe den Abschnitt »Namensraum und Gültigkeitsregeln« auf Seite 84.

Tabelle 12: Reservierte Wörter in Python 3.0

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

HINWEIS

In Python 2.6 sind `print` und `exec` ebenfalls reserviert, da sie dort Anweisungen und keine eingebauten Funktionen sind. Außerdem sind `nonlocal`, `True` und `False` in Python 2.6 keine reservierten Wörter. Das erste steht nicht zur Verfügung, die beiden anderen sind einfach eingebaute Namen. `with` und `as` sind in Python 2.6 und 3.0 reserviert, in früheren 2.x-Versionen jedoch nicht, es sei denn, Kontextmanager wurden explizit aktiviert. `yield` ist seit 2.3 reserviert. Später hat es sich von einer Anweisung zu einem Ausdruck verändert, ist aber immer noch ein reserviertes Word.

Namenskonventionen

- Namen, die mit zwei Unterstrichen beginnen und enden (zum Beispiel `__init__`), haben eine besondere Bedeutung für den Interpreter, sind aber keine reservierten Wörter.
- Namen, die mit einem Unterstrich beginnen (z.B. `_X`) und die auf oberster Ebene eines Moduls zugewiesen werden, werden bei einem Import der Form `from...*` nicht kopiert (siehe auch die Modulexport-Namensliste `__all__` in den Abschnitten »Die from-Anweisung« auf Seite 74 und »Pseudoprivate Attribute«

auf Seite 88). In anderen Kontexten ist das die informelle Konvention für interne Namen.

- Namen in einer `class`-Anweisung, die mit zwei Unterstrichen beginnen, aber nicht damit enden (z. B. `__X`), wird der Name der umgebenden Klasse vorangestellt (siehe den Abschnitt »Pseudoprivate Attribute« auf Seite 88).
- Der nur aus einem Unterstrich bestehende Name (`_`) wird (nur) im interaktiven Interpreter benutzt und steht für das Ergebnis der letzten Auswertung.
- Namen von eingebauten Funktionen und Ausnahmen (z. B. `open`, `SyntaxError`) sind nicht reserviert. Sie existieren im zuletzt durchsuchten Geltungsbereich und dürfen neu zugewiesen werden, wodurch ihre eingebaute Bedeutung im aktuellen Geltungsbereich verborgen wird, z. B. `open=funktionX`).
- Klassennamen beginnen üblicherweise mit einem Großbuchstaben (z. B. `DieKlasse`) und Module mit einem Kleinbuchstaben (z. B. `dasmodul`).
- Das erste (am weitesten links stehende) Argument der Methodendefinition von Klassen wird normalerweise `self` genannt.

Spezifische Anweisungen

Die folgenden Abschnitte beschreiben alle Python-Anweisungen. Bei zusammengesetzten Anweisungen steht *folge* für eine oder mehrere Anweisungen (Anweisungsfolge). Zusammengesetzte Anweisungen (`if`, `while` usw.) bestehen aus Kopfzeilen und Anweisungsfolgen, die direkt die Kopfzeile erweitern oder einen eigenen eingerückten Block bilden (der Normalfall). Anweisungsfolgen mit eigenen zusammengesetzten Anweisungen (`if`, `while`, usw.) müssen in jedem Fall auf einer neuen Zeile stehen und eingerückt werden. Zwei Beispiele für gültige Konstrukte:

```
if x < 42:
    print(x)
    while x: x = x - 1
if x < 42: print(x)
```

Zuweisungsanweisungen

```
ziel = ausdruck
ziel = ziel2 = ausdruck
ziel1, ziel2 = ausdruck1, ausdruck
ziel1 += ausdruck
ziel1, ziel2,  ...  = iterierbares-gleicher-länge
(ziel1, ziel2, ...) = iterierbares-gleicher-länge
[ziel1, ziel2, ...] = iterierbares-gleicher-länge
ziel1, *ziel2, ...  = iterierbares-passender-länge
```

Zuweisungen speichern Referenzen auf Objekte in Zielen. Ausdrücke geben Objekte zurück. Ziele können einfache Namen sein (`X`), qualifizierte Attribute (`X.attr`) oder Indizes und Teilbereiche (`X[i]`, `X[i:j]`).

Die zweite Form weist jedem Ziel dasselbe Objekt *ausdruck* zu. Die dritte Form weist von links nach rechts paarweise zu. Die letzten drei Formen weisen Komponenten einer beliebigen Sequenz (bzw. eines iterierbaren Objekts) an Ziele in Tupeln oder Listen zu, ebenfalls von links nach rechts. Die Sequenz (das iterierbare Objekt) rechts kann jeden Typ haben, muss aber die gleiche Länge wie die Sequenz mit den Zielen haben, es sei denn, in den Zielen auf der linken Seite erscheint ein Name mit Stern, der beliebig viele Elemente akzeptieren kann (die erweiterte Sequenzzuweisung von Python 3.0 wird weiter unten behandelt):

```
ziel1, *ziel2, ... = iterierbares-objekt
```

Erweiterte Zuweisung

Python verfügt über einen Satz weiterer Zuweisungsformen, siehe Tabelle 13. Diese als *erweiterte Zuweisungen* bekannten Formen bündeln einen binären Ausdruck und eine Zuweisung. Die folgenden beiden Formen z. B. sind ungefähr äquivalent:

```
X = X + Y
X += Y
```

Die Referenz auf das Ziel `X` in der zweiten Form muss jedoch nur einmal ausgewertet werden, und Operationen an Ort und Stelle können bei veränderlichen Objekten als Optimierung angewendet werden (z. B. ruft `liste1 += liste2` automatisch `liste1.extend(liste2)`

auf statt die langsamere Verkettungsoperation mit +). Klassen dürfen solche Zuweisungen mit Methodennamen überladen, die mit einem `i` (von »in-place«) beginnen, z. B. `__iadd__` für `+=` und `__add__` für `+`. Die Form `X //= Y` (restlose Division) gibt es seit Python 2.2.

Tabelle 13: Erweiterte Zuweisungsanweisungen

`X += Y`	`X &= Y`	`X -= Y`	`X |= Y`
`X *= Y`	`X ^= Y`	`X /= Y`	`X >>= Y`
`X %= Y`	`X <<= Y`	`X **= Y`	`X //= Y`

Gewöhnliche und erweiterte Sequenzzuweisung

In Python 2.x und 3.0 können Sequenzen (oder andere iterierbare Objekte) von Werten einer beliebigen Sequenz von Namen zugewiesen werden, vorausgesetzt, die Längen sind identisch. Diese elementare Form der Sequenzzuweisung funktioniert in den meisten Zuweisungskontexten:

```
>>> a, b, c, d = [1, 2, 3, 4]
>>> a, d
(1, 4)
>>> for (a, b, c) in [[1, 2, 3], [4, 5, 6]]:
...     print(a, b, c)
...
1 2 3
4 5 6
```

In Python 3.0 wurde die Sequenzzuweisung erweitert, um die Zusammenstellung beliebig vieler Elemente zu unterstützen, indem eine der Variablen im Zuweisungsziel mit einem Stern markiert wird; wird diese Syntax genutzt, muss die Länge der Sequenzen nicht übereinstimmen:

```
>>> a, *b = [1, 2, 3, 4]
>>> a, b
(1, [2, 3, 4])
>>> a, *b, c = [1, 2, 3, 4]
>>> a, b, c
(1, [2, 3], 4)
>>> *a, b = [1, 2, 3, 4]
>>> a, b
```

```
([1, 2, 3], 4)
>>> for (a, *b) in [[1, 2, 3], [4, 5, 6]]: print(a, b)
...
1 [2, 3]
4 [5, 6]
```

Ausdrucksanweisungen

```
ausdruck
funktion([wert, name=wert, *name, **name...])
objekt.methode([wert, name=wert, *name, **name...])
```

Ausdrücke können allein auf einer Zeile als eigenständige Anweisungen stehen (während Anweisungen nicht als Ausdrücke auftauchen können). Ausdrücke werden gewöhnlich zum Aufruf von Funktionen und Methoden sowie, im interaktiven Modus, zur Ausgabe genutzt. Ausdrucksanweisungen sind außerdem die häufigste Formulierungsform für `yield`-Ausdrücke und den Aufruf der in Python 3.0 eingebauten Funktion `print()` (obwohl beide in diesem Buch als spezifische Anweisungen beschrieben werden).

Aufrufsyntax

In Funktions- und Methodenaufrufen werden die konkreten Argumente durch Kommata getrennt und den Argumenten im Header der Funktionsdefinition über die Position zugeordnet. Optional können Aufrufe bestimmte Argumentnamen in Funktionen mit der Schlüsselwortargumentsyntax *`name=wert`* angeben, um die übergebenen Werte wieder zurückzuerhalten. Schlüsselwortargumente sind benannte Argumente, die über den Namen statt über die Position zugeordnet werden.

Aufrufsyntax für beliebig viele Argumente

In den Argumentlisten für Funktions- und Methodenaufrufe kann eine spezielle Syntax genutzt werden, um beliebig viele Argumente anzugeben. Sind *`pargs`* und *`kargs`* eine Sequenz (oder ein anderes iterierbares Objekt) und ein Dictionary:

```
f(*pargs, **kargs)
```

ruft dieser Aufruf die Funktion `f` mit den positionellen Argumenten aus dem iterierbaren Objekt *pargs* und den benannten Argumenten aus dem Dictionary *kargs* auf, zum Beispiel:

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

Diese Syntax soll das Gegenstück zur Syntax für beliebig viele Argumente bei der Funktionsdefinition sein, z. B. `def f(*pargs, **kargs)`. Sie ist außerdem flexibel, da sie leicht mit anderen positionellen oder benannten Argumenten kombiniert werden kann (z. B. `g(1, 2, foo=3, bar=4, *args, **kw)`).

In Python 2.x lässt sich mit der eingebauten (in Python 3.0 nicht mehr vorhandenen) Funktion `apply()` Ähnliches erreichen:

```
apply(f, pargs, kargs)
```

Weitere Informationen zum Aufrufsystem finden Sie im Abschnitt »Die def-Anweisung« auf Seite 65.

Die print-Anweisung

In Python erfolgt die Ausgabe über einen Aufruf einer eingebauten Funktion, was üblicherweise als Ausdrucksanweisung (eigenständig auf einer Zeile) geschrieben wird. Die Aufrufsignatur hat folgende Form:

```
print([Wert [, Wert]*]
   [, sep=String] [, end=String] [, file=Datei])
```

`wert` zeigt dabei jeweils ein auszugebendes Objekt an. Konfiguriert wird dieser Aufruf über die drei nur benannt verfügbaren Argumente:

- `sep` ist ein String, der zwischen den Werten eingefügt wird (standardmäßig ein Leerzeichen: `' '`).
- `end` ist ein String, der ans Ende des ausgegebenen Texts angehängt wird (standardmäßig ein Zeilenumbruch: `'\n'`).
- `file` ist das dateiartige Objekt, in das der Text ausgegeben wird (standardmäßig die Standardausgabe: `sys.stdout`).

Übergeben Sie leere oder angepasste Strings, um Trennzeichen und Zeilenumbrüche zu unterdrücken oder zu ändern, übergeben Sie ein dateiähnliches Objekt, um die Ausgabe in Ihrem Skript umzuleiten:

```
>>> print(2 ** 32, 'spam')
4294967296 spam
>>> print(2 ** 32, 'spam', sep='')
4294967296spam
>>> print(2 ** 32, 'spam', end=' '); print(1, 2, 3)
4294967296 spam 1 2 3
>>> print(2 ** 32, 'spam', sep='', file=open('out', 'w'))
>>> open('out').read()
'4294967296spam\n'
```

Da `print` standardmäßig einfach die `write`-Methode des aktuell von `sys.stdout` referenzierten Objekts aufruft, ist der folgende Ausdruck `print(X)` äquivalent:

```
import sys
sys.stdout.write(str(X) + '\n')
```

Um die `print`-Ausgabe in Dateien oder Klassenobjekte umzuleiten, können Sie entweder, wie bereits gezeigt, ein Objekt mit einer `write`-Methode für das benannte Argument `file` übergeben oder `sys.stdout` ein derartiges Objekt zuweisen:

```
sys.stdout = open('log', 'a')  # beliebiges Objekt mit
                               # write()-Methode
print('Warning-bad spam!')     # geht an die write()-
                               # Methode des Objekts
```

Da `sys.stdout` geändert werden kann, ist das benannte Argument `file` eigentlich nicht zwingend erforderlich, es kann jedoch helfen, explizite `write`-Aufrufe zu vermeiden und den ursprünglichen `sys.stdout`-Wert um einen umgeleiteten `print`-Aufruf zu speichern und wiederherzustellen, wenn der ursprüngliche Stream noch benötigt wird (siehe »Eingebaute Funktionen« auf Seite 105).

Die print-Anweisung von Python 2.x

In Python 2.x erfolgt die Ausgabe über eine spezielle Anweisung statt über eine eingebaute Funktion. Diese hat folgende Form:

```
print [Wert [, Wert]* [,]]
print >> Datei [, Wert [, Wert]* [,]]
```

Die print-Anweisung von Python 2.x gibt eine druckbare Darstellung von Objekten auf der Standardausgabe aus (aktueller Wert von sys.stdout). Zwischen den Werten werden Leerzeichen eingefügt. Ein nachstehendes Komma unterbindet den Zeilenvorschub am Ende und entspricht der end=' '-Einstellung in Python 3.0:

```
>>> print 2 ** 32, 'spam'
4294967296 spam
>>> print 2 ** 32, 'spam',; print 1, 2, 3
4294967296 spam 1 2 3
```

Bei der print-Anweisung von Python 2.x kann auch ein geöffnetes dateiartiges Ausgabeobjekt als Ziel des auszugebenden Texts angegeben werden (statt sys.stdout):

```
fileobj = open('log', 'a')
print >> fileobj, "Warning-bad spam!"
```

Ist das Dateiobjekt None, wird sys.stdout genutzt. Die >>-Syntax von Python 2.x entspricht dem benannten file=F-Argument von Python 3.0. Für sep=S gibt es bei der Python 2.x-Anweisung kein Gegenstück.

Nutzen Sie folgende Anweisung, um die Ausgabefunktion von Python 3.0 unter Python 2.x zu nutzen:

```
from __future__ import print_function
```

Die if-Anweisung

```
if test:
    folge
[elif test:
    folge]*
[else:
    folge]
```

Die if-Anweisung wählt aus einer oder mehreren Aktionen (Anweisungsblöcken) aus. Ausgeführt wird die erste Folge, die zum ersten if- oder elif-Test mit einem wahren Ergebnis gehört, oder die else-Folge, wenn alle anderen falsch ergeben.

Die while-Anweisung

```
while test:
    folge
[else:
    folge]
```

Die `while`-Schleife ist eine allgemeine Schleife. Solange der Test am Anfang wahr ergibt, führt sie die erste Folge aus. Die `else`-Folge wird ausgeführt, wenn die Schleife normal ohne `break`-Anweisung verlassen wird.

Die for-Anweisung

```
for ziel in iterierbares:
    folge
[else:
    folge]
```

`for` ist eine Schleife über Sequenzen (und andere iterierbare Objekte). Sie weist die Elemente in *`iterierbares`* nacheinander *`ziel`* zu und führt dann jedes Mal die erste Folge aus. Sie führt die `else`-Folge aus, wenn die Schleife normal ohne `break`-Anweisung beendet wird. *`ziel`* darf alles sein, was links von einer Zuweisung mit `=` stehen darf (zum Beispiel `for (x, y) in tupelliste:`).

In Python 2.2 folgt das, indem zunächst versucht wird, mit `iter(`*`iterierbares`*`)` ein *Iterator*-Objekt `I` zu erhalten und anschließend die `I.__next__()`-Methode aufzurufen, bis `StopIteration` abgesetzt wird. (In Python 2 heißt `I.__next__()` `I.next()`.) Mehr zu Iterationen finden Sie im Abschnitt »Die yield-Anweisung« auf Seite 70. In früheren Versionen oder wenn kein Iterator-Objekt existiert (z. B. wenn die Methode `__iter__` nicht definiert ist), wird stattdessen *`iterierbares`* wiederholt mit steigenden Positionen indiziert, bis ein `IndexError` ausgelöst wird.

Die pass-Anweisung

```
pass
```

Das ist eine Tu-nichts-Platzhalteranweisung. Sie wird eingesetzt, wenn syntaktisch eine Anweisung erforderlich ist. In Python 3.x kann man mit einer Ellipse (...) Gleiches erreichen.

Die break-Anweisung

```
break
```

Beendet unmittelbar die innerste umgebende `while`- oder `for`-Schleife, wobei ein eventuell dazugehöriges `else` übersprungen wird.

Die continue-Anweisung

```
continue
```

Geht unmittelbar zum Anfang der innersten umgebenden `while`- oder `for`-Schleife und setzt die Abarbeitung in der Kopfzeile fort.

Die del-Anweisung

```
del name
del name[i]
del name[i:j:k]
del name.attribut
```

Die `del`-Anweisung löscht Namen, Elemente, Slices und Attribute und entfernt Bindungen. In den letzten drei Formen kann *name* ein beliebiger Ausdruck sein (mit Klammern, falls aus Vorranggründen erforderlich), zum Beispiel: `del a.b()[1].c.d`.

Die def-Anweisung

```
def name([arg,... arg=wert,... *arg, **arg]):
    folge
```

Die `def`-Anweisung bildet neue Funktionen. Ein Funktionsobjekt wird erzeugt und *name* zugewiesen. Jeder Funktionsaufruf erzeugt einen neuen, lokalen Geltungsbereich, wobei zugeordnete Namen lokal für den Funktionsaufruf gelten (außer man deklariert sie global). Lesen Sie auch den Abschnitt »Namensraum und Gültigkeitsregeln« auf Seite 84. Argumente werden per Zuweisung übergeben und können eine der vier Formen in Tabelle 14 haben.

Die Argumentformen in Tabelle 14 können auch in Funktionsaufrufen verwendet werden, werden dort aber so interpretiert, wie in

Tabelle 15 gezeigt (mehr zur Syntax für Funktionsaufrufe finden Sie im Abschnitt »Ausdrucksanweisungen« auf Seite 60).

Tabelle 14: Argumentformate in Definitionen

Argumentformat	Interpretation
`arg`	Zuordnung über Name oder Position.
`arg=wert`	Standardwert, wenn `arg` nicht übergeben wird.
`*arg`	Sammelt zusätzliche positionelle `arg` als neues Tupel.
`**arg`	Sammelt zusätzliche benannte `arg` als neues Dictionary.
`*name, arg[=wert]`	Benannte Argumente nach * in Python 3.0.
`*, arg[=wert]`	Identisch mit der letzten Zeile.

Tabelle 15: Argumentformate in Aufrufen

Argumentformat	Interpretation
`arg`	Positionelles Argument.
`arg=wert`	Benanntes Argument (Schlüsselwortargument).
`*arg`	Sequenz (oder anderes iterierbares Objekt) mit positionellen Argumenten.
`**arg`	Dictionary mit benannten Argumenten.

Rein benannte Argumente in Python 3.0

Python 3.0 verallgemeinert Funktionsdefinitionen durch die Einführung von rein benannten Argumenten, die nur benannt übergeben werden können und erforderlich sind, wenn kein Standardwert definiert ist. Rein benannte Argumente werden nach dem * angegeben, der ohne einen Namen stehen kann, wenn es rein benannte Argumente gibt, aber keine beliebige Anzahl positioneller Argumente:

```
>>> def f(a, *b, c): print(a, b, c)  # c erforderliches
...                                  # rein benanntes Arg
>>> f(1, 2, c=3)
1 (2,) 3

>>> def f(a, *, c=None): print(a, c) # c optionales
...                                  # rein benanntes Arg
```

```
>>> f(1)
1 None
>>> f(1, c='spam')
1 spam
```

Funktionsannotationen in Python 3.0

Eine weitere Erweiterung von Funktionsdefinitionen in Python 3.0 ist die Möglichkeit, Argumente und Rückgabewerte mit Objektwerten zu annotieren, die in Erweiterungen genutzt werden können. Annotationen werden in der Form `:wert` nach dem Argumentnamen, aber vor einem eventuellen Standardwert, und in der Form `->wert` nach der Argumentliste angegeben. Sie werden in einem `__annotations__`-Attribut gesammelt, von Python selbst ansonsten aber nicht speziell behandelt:

```
>>> def f(a:99, b:'spam'=None) -> float:
...     print(a, b)
...
>>> f(88)
88 None
>>> f.__annotations__
{'a': 99, 'b': 'spam', 'return': <class 'float'>}
```

lambda-Ausdrücke

Funktionen können auch mit dem `lambda`-Ausdruck erstellt werden, der ein neues Funktionsobjekt erzeugt und zur späteren Verwendung zurückliefert, anstatt ihm einen Namen zuzuweisen:

```
lambda arg, arg,...: ausdruck
```

arg bedeutet bei `lambda` das Gleiche wie bei `def`, und *ausdruck* ist der implizierte Rückgabewert späterer Aufrufe. Da `lambda` ein Ausdruck und keine Anweisung ist, kann es an Stellen verwendet werden, an denen `def` nicht verwendet werden kann (z.B. in einem Dictionary-Literalausdruck oder der Argumentliste eines Funktionsaufrufs). Weil `lambda` einen einzelnen Ausdruck berechnet, statt Anweisungen auszuführen, ist es nicht für komplexe Funktionen geeignet.

Standardwerte und Attribute

Veränderliche Standardwerte für Attribute werden nur bei Ausführung der `def`-Anweisung berechnet, nicht bei jedem Aufruf, und

behalten ihren Wert deswegen zwischen Aufrufen. Es gibt allerdings manche, die dieses Verhalten als Fehler betrachten. Klassen und Referenzen in den einschließenden Geltungsbereich können häufig geeignetere Mittel sein, Zustände zu bewahren. Nutzen Sie None-Standardwerte für veränderliche Objekte und explizite Tests, um Veränderungen zu vermeiden, wie es die Kommentare im folgenden Code illustrieren:

```
>>> def grow(a, b=[]):       # ..., b=None)
...     b.append(a)          # if b == None: b = []
...     print(b)
...
>>> grow(1); grow(2)
[1]
[1, 2]
```

Python 2.x und 3.0 gestatten es, Funktionen frei wählbare Attribute zuzuweisen. Das ist ein anderes Mittel zur Zustandsbewahrung (obwohl Attribute nur funktionsbezogene Objektzustände unterstützen, die nur aufrufgebunden sind, wenn jeder Aufruf eine neue Funktion erzeugt):

```
>>> grow.food = 'spam'
>>> grow.food
'spam'
```

Funktions- und Methodendekoratoren

Seit Python 2.4 kann Funktionsdefinitionen eine Deklarationssyntax vorangestellt werden, die die darauffolgende Funktion beschreibt. Diese Deklarationen werden als *Dekoratoren* bezeichnet und durch ein vorangestelltes @ angezeigt. Sie bieten eine explizite Syntax für funktionale Techniken. Die Funktionsdekoratorsyntax

```
@dekorator
def F():
    ...
```

ist dieser manuellen Namensneubindung äquivalent:

```
def F():
    ...
F = dekorator(F)
```

Das bewirkt, dass der Funktionsname an das Ergebnis des durch den Dekorator angegebenen aufrufbaren Objekts gebunden wird, dem die Funktion selbst übergeben wird. Funktionsdekoratoren können zur Verwaltung von Funktionen oder später an sie gerichtete Aufrufe (über Proxy-Objekte) genutzt werden. Dekoratoren können an beliebige Funktionsdefinitionen geheftet werden, auch an die Methoden in einer Klasse:

```
class C:
    @dekorator
    def M():            # entspricht M = dekorator(M)
        ...
```

Allgemeiner: Die folgende geschachtelte Dekoration:

```
@A
@B
@C
def f(): ...
```

entspricht diesem nicht dekoratorbasierten Code:

```
def f(): ...
f = A(B(C(f)))
```

Dekoratoren können Argumentlisten haben:

```
@spam(1,2,3)
def f(): ...
```

`spam` muss dann eine Funktion sein, die selbst eine Funktion liefert (was man auch als Fabrikmethode bezeichnet). Ihr Ergebnis wird als der eigentlich Dekorator genutzt. Dekoratoren müssen auf der Zeile vor der Funktionsdefinition stehen und dürfen nicht auf der gleichen Zeile stehen (z.B. wäre `@A def f(): ...` illegal).

Da sie aufrufbare Objekte akzeptieren und zurückliefern, können einige eingebaute Funktionen wie beispielsweise `property()`, `staticmethod()` und `classmethod()` als Funktionsdekoratoren genutzt werden (siehe »Eingebaute Funktionen« auf Seite 105). In Python 2.6, 3.0 und später wird die Dekoratorsyntax auch für Klassen unterstützt; siehe »Die class-Anweisung« auf Seite 75.

Die return-Anweisung

```
return [ausdruck]
```

Die return-Anweisung verlässt die umgebende Funktion und gibt den Wert von *ausdruck* als Ergebnis des Funktionsaufrufs zurück. Falls nicht angegeben, hat *ausdruck* den Standardwert None. Tipp: Nutzen Sie Tupel, wenn Funktionen mehrere Werte liefern sollen.

Die yield-Anweisung

```
yield ausdruck
```

Der yield-Ausdruck, der gewöhnlich als Ausdrucksanweisung geschrieben wird (eigenständig auf einer Zeile), unterbricht die Ausführung der Funktion, speichert ihren Zustand und gibt *ausdruck* zurück. Bei der nächsten Iteration wird der vorherige Zustand der Funktion wiederhergestellt, und die Ausführung wird unmittelbar nach der yield-Anweisung fortgesetzt. Die Iteration wird durch eine return-Anweisung ohne Wert oder das Erreichen des Endes des Funktionsrumpfs beendet:

```
def quadrateErzeugen(N):
    for i in range(N):
        yield i ** 2
>>> G =  quadrateErzeugen(5)
>>> list(G)                  # erzwingt Ergebnisgenerierung
[0, 1, 4, 9, 16]
```

Wird yield als Ausdruck genutzt, liefert es das der send()-Methode des Generators übergebene Objekt an den Aufrufer (z. B. A = yield X). Der Ausdruck muss in Klammern eingeschlossen werden, wenn er nicht das einzige Element auf der rechten Seite von = ist (z. B. A = (yield X) + 42). In diesem Modus werden dem Generator Werte übergeben, indem send(wert) aufgerufen wird. Der Generator nimmt die Arbeit wieder auf, und der yield-Ausdruck liefert wert. Wird die eigentliche __next__()-Methode oder die eingebaute Funktion next() aufgerufen, um den Generator vorzurücken, liefert yield None.

Generatoren bieten außerdem eine throw(typ)-Methode, um beim letzten yield innerhalb des Generators eine Ausnahme auszulösen, und eine close()-Methode, die im Generator ein neue Generator-

Exit-Ausnahme auslöst, um die Iteration zu beenden. yield ist seit Version 2.3 Standard; die Generator-Methoden send(), throw() und close() sind seit Python 2.5 verfügbar.

Generatoren und Iteratoren

Funktionen, die eine yield-Anweisung enthalten, werden als *Generatoren* kompiliert. Werden sie aufgerufen, liefern sie ein Generator-Objekt, das das Iterator-Protokoll unterstützt und Ergebnisse auf Anforderung erzeugt. *Iteratoren* sind Objekte, die von der eingebauten Funktion iter(X) zurückgeliefert werden. Sie definieren eine __next__()-Methode, die das nächste Elemente in der Iterationsfolge liefert oder eine StopIteration-Ausnahme auslöst, um die Iteration zu beenden.

Alle Iterationskontexte einschließlich for-Schleifen und Komprehensionen nutzen automatisch das Iterator-Protokoll, um Sammlungen zu durchlaufen. Außerdem ruft die eingebaute Funktion next(I) automatisch I.__next__() auf, um manuelle Iterationsschleifen zu vereinfachen.

Klassen können eine __iter__()-Methode bereitstellen, um Aufrufe der eingebauten Funktion iter(X) abzufangen. Ist diese definiert, hat ihr Ergebnis eine __next__()-Methode, die genutzt wird, um die Ergebnisse in Iterationskontexten zu durchlaufen. Ist __iter__() nicht definiert, wird als Ausweichmethode die Indizierungsmethode __getitem__() genutzt, um zu iterieren, bis ein IndexError ausgelöst wird.

In Python 2.x heißt die Methode I.__next__() I.next(), aber ansonsten funktionieren Iterationen auf genau die gleiche Weise. Die next()-Funktion ruft in 2.6 die Methode I.next() auf. Der Abschnitt »Generator-Ausdrücke« auf Seite 39 liefert Ihnen Informationen zu verwandten Werkzeugen.

Die global-Anweisung

```
global name [, name]*
```

Die global-Anweisung ist eine Namensraumdeklaration. Wird sie in einer Klassen- oder Funktionsdefinitionsanweisung verwendet,

bewirkt sie, dass alle Vorkommen von *name* in diesem Kontext als Referenzen auf eine globale (modulbasierte) Variable dieses Namens behandelt werden – unabhängig davon, ob `name` ein Wert zugewiesen ist oder nicht, und unabhängig davon, ob `name` existiert oder nicht.

Diese Anweisung ermöglicht es, globale Variablen in Funktionen oder Klassen zu erstellen oder zu ändern. Pythons Geltungsbereichsregeln sorgen dafür, dass Sie nur globale Namen deklarieren müssen, denen ein Wert zugewiesen ist. Nicht deklarierte Namen werden bei einer Zuweisung lokal gemacht, aber globale Referenzen werden automatisch im enthaltenden Modul angelegt (siehe auch »Namensraum und Gültigkeitsregeln« auf Seite 84).

Die nonlocal-Anweisung

```
nonlocal name [, name]*
```

Nur in Python 3.0 verfügbar.

Die `nonlocal`-Anweisung ist eine Namensraumdeklaration. Wird sie in einer eingebetteten Funktion verwendet, bewirkt sie, dass alle Vorkommen von *name* in diesem Kontext als Referenzen auf eine lokale Variable im unmittelbar übergeordneten Geltungsbereich behandelt werden.

name muss in jenem Geltungsbereich gebunden sein. Diese Anweisung ermöglicht, ihn in einem eingebetteten Geltungsbereich zu verändern. Pythons Geltungsbereichsregeln bewirken, dass Sie nur nicht lokale Namen deklarieren müssen, die zugewiesen sind. Nicht deklarierte Namen werden bei Zuweisung lokal erstellt, aber nicht lokale Referenzen werden automatisch auf den übergeordneten Geltungsbereich bezogen (siehe auch »Namensraum und Gültigkeitsregeln« auf Seite 84).

Die import-Anweisung

```
import modul [, modul]*
import [paket.]* modul [, [paket.]* modul]*
import [paket.]* modul as name
                  [, [paket.]*modul as name]*
```

Die `import`-Anweisung ermöglicht den Modulzugriff: Sie importiert ein Modul als Ganzes. Module ihrerseits enthalten Namen, die mittels Qualifikation verfügbar sind (z.B. *`modul.attribut`*). Zuweisungen auf oberster Ebene einer Python-Datei erzeugen Objektattribute im Modul. Die `as`-Klausel weist das importierte Modulobjekt einer Variablen *`name`* zu – nützlich als kurze Synonyme für längere Modulnamen.

`modul` bezeichnet das Zielmodul – in der Regel eine Python-Datei oder ein übersetztes Modul –, das sich in einem Verzeichnis in `sys.path` befinden muss. `modul` wird ohne Dateinamenserweiterung angegeben (*.py* und andere Erweiterungen werden weggelassen). Der Modulimportsuchpfad `sys.path` ist eine Verzeichnisaufstellung, die auf Basis des obersten Verzeichnisses des Programms, der `PYTHONPATH`-Einstellungen, der `.pth`-Pfaddateiinhalte und der Python-Standards aufgebaut wird.

Importoperationen übersetzen eine Quelldatei in Bytecode, wenn nötig (und speichern ihn in einer *.pyc*-Datei, wenn möglich). Dann führen sie den übersetzten Code von oben nach unten aus, um per Zuweisung Objektattribute im Modul zu erzeugen. Benutzen Sie die eingebaute Funktion `imp.reload()`, um eine erneute Übersetzung und Ausführung von bereits geladenen Modulen zu erzwingen (siehe auch das von `import` benutzte `__import__` im Abschnitt »Eingebaute Funktionen« auf Seite 105).

In der Jython-Implementierung können auch Java-Klassenbibliotheken importiert werden; Jython erzeugt einen Python-Modul-Wrapper als Schnittstelle zur Java-Bibliothek. Im Standard-CPython können Importe auch übersetzte C- und C++-Erweiterungen laden.

Import von Paketen

Wenn angegeben, bezeichnen *`paket`*-Präfixe umgebende Verzeichnisnamen, und punktierte Modulpfade reflektieren Verzeichnishierarchien. Ein Import der Form `import dir1.dir2.mod` lädt normalerweise die Moduldatei im Verzeichnispfad *dir1/dir2/mod.py*, wobei *dir1* in einem Verzeichnis enthalten sein muss, das im Modulsuchpfad `sys.path` aufgeführt wird.

Jedes Verzeichnis, das in einer Importanweisung vorkommt, muss eine (eventuell leere) Datei `__init__.py` enthalten, die als Modulnamensraum auf Verzeichnisebene fungiert. Diese Datei wird beim ersten Import über das Verzeichnis ausgeführt. Alle in `__init__.py`-Dateien zugewiesenen Namen werden zu Attributen des Verzeichnismodulobjekts. Verzeichnispakete können Konflikte lösen, die durch die lineare Natur von `PYTHONPATH` bedingt sind.

Siehe auch »Paketbezogene import-Syntax« auf Seite 75.

Die from-Anweisung

```
from [paket.]* modul import name [,name]*
from [paket.]* modul import *
from [paket.]* modul import name as neuername
from [paket.]* modul import (name1, name2, name3, name4)
```

Die `from`-Anweisung importiert Namen aus einem Modul, damit sie anschließend ohne Qualifizierung genutzt werden können. Die Variante `from modul import *` kopiert *alle* Namen, die auf oberster Ebene des Moduls definiert sind, abgesehen von Namen, die mit einem Unterstrich beginnen, und Namen, die nicht im Modulattribut `__all__` (einer Liste von Strings) vorkommen (sofern definiert).

Die optionale `as`-Klausel erzeugt ein Namenssynonym. Beim optionalen Präfix *paket* funktionieren Importpfade genau so wie bei `import`-Anweisungen (z. B. `from dir1.dir2.mod import X`), aber der Pfad des Pakets muss nur bei `from` selbst aufgeführt werden. Wegen der neuen Gültigkeitsregeln erzeugt die Form mit * ab Version 2.2 Warnungen, wenn sie innerhalb einer Funktion oder Klasse auftaucht (in Python 3.0 führt das zu einem Fehler).

Ab Python 2.4 dürfen die aus einem Modul importierten Namen in runden Klammern stehen, damit sie ohne Backslashs auf mehrere Zeilen verteilt werden können. Seit Python 3.0 ist die Form `from module import *` innerhalb einer Funktion ungültig, weil sie es unmöglich macht, die Namensgeltung bei der Definition zu klassifizieren.

Die `from`-Anweisung wird auch zur Aktivierung von zukünftigen (aber noch experimentellen) Spracherweiterungen genutzt, nämlich

mit `from __future__ import featurename`. Diese Form darf nur zu Beginn einer Moduldatei erscheinen (ihr darf lediglich ein optionaler Dokumentationsstring vorausgehen).

Paketbezogene import-Syntax

In Python 3.0 kann `from` (aber nicht `import`) vorangestellte Punkte in Modulnamen nutzen, um anzugeben, dass sie relativ zu dem Paketverzeichnis importiert werden, in dem sich das importierende Modul befindet:

```
from modul import name [, name]*    # sys.path: abs
from . import module [, module]*     # nur Paketverz.: rel
from .modul import name [, name]*   # nur Paketverz.: rel
from .. import module [, module]*    # Elternverz. in Paket
from ..modul import name [, name]*  # Elternverz. in Paket
```

Die Syntax mit den führenden Punkten bewirkt, dass Importe in Python 2.6 und 3.0 explizit paketrelativ gemacht werden. Bei Importen ohne Punktsyntax wird in Python 2.6 zunächst das Verzeichnis des Pakets durchsucht. In Python 3.0 passiert das nicht. Nutzen Sie Folgendes, um die Importsemantik von Python 3.0 unter Python zu aktivieren:

```
from __future__ import  absolute_import
```

Absolute Paketimporte in Bezug auf ein Verzeichnis in `sys.path` werden den impliziten paketrelativen Importen von Python 2.x und der expliziten paketrelativen Paketimportsyntax von Python 2.x und 3.0 im Allgemeinen vorgezogen.

Die class-Anweisung

```
class name [ ( super [, super]* [, metaclass=M] ) ]:
    folge
```

Die `class`-Anweisung bildet neue Klassen, die Fabriken für Instanzobjekte sind. Das neue Klassenobjekt erbt von allen aufgeführten `super`-Klassen in der angegebenen Reihenfolge und wird *name* zugewiesen. Die `class`-Anweisung führt einen neuen lokalen Namensraum ein, und alle in der `class`-Anweisung zugewiesenen Namen erzeugen Klassenobjektattribute, die sich alle Instanzen der Klasse gemeinsam teilen.

Zu den wichtigen Klasseneigenschaften gehören folgende (siehe auch die Abschnitte »Objektorientierte Programmierung« und »Überladungsmethoden von Operatoren«):

- Superklassen (auch Oberklassen), von denen eine neue Klasse erbt, werden im Kopf der Klasse in Klammern aufgelistet (z.B. `class Sub(Super1, Super2):`).
- Zuweisungen in der Anweisungsfolge erzeugen Klassenattribute, die Instanzen zur Verfügung stehen: `def`-Anweisungen werden zu Methoden, während Zuweisungen einfache Instanzvariablen erzeugen usw.
- Aufrufe der Klasse erzeugen Instanzobjekte, wobei jedes seine eigenen sowie die von der Klasse und allen ihren Oberklassen geerbten Attribute besitzt.
- Methodenfunktionen erhalten ein spezielles erstes Argument, das üblicherweise `self` heißt und das Instanzobjekt ist, das das implizierte Subjekt des Methodenaufrufs ist und Zugriff auf die Instanzattribute mit den Zustandsinformationen gewährt.
- Speziell benannte `__X__`-Methodendefinitionen fangen eingebaute Operationen ab.

Klassendekoratoren in Python 2.6 und 3.0

In Python 2.6, 3.0 und später kann die Dekoratorsyntax neben Funktionsdefinitionen auch auf Klassendefinitionen angewandt werden. Sie hat dann diese Form:

```
@dekorator
class C:
   def meth():
       ...
```

Das entspricht folgender Namensneubindung:

```
class C:
     def meth():
       ...
C = dekorator(C)
```

Es bewirkt, dass der Dekorater mit der Klasse als Argument aufgerufen und sein Rückgabewert an den Klassennamen gebunden

wird. Wie Funktionsdekoratoren können Klassendekoratoren geschachtelt werden und unterstützen Dekoratorargumente. Klassendekoratoren können zur Verwaltung von Klassen oder später an sie gerichteten Instanzerzeugungsaufrufen (über Proxy-Objekte) eingesetzt werden.

Metaklassen

Metaklassen sind Klassen, die grundsätzlich von der Klasse type abgeleitet sind, um die Erzeugung des Klassenobjekts selbst anzupassen:

```
class Meta(type):
    def __new__(meta, cname, supers, cdict):
        # geerbte type.__call__-Methode aufrufen
        return type.__new__(meta, cname, supers, cdict)
```

In Python 3.0 definieren Klassen ihre Metaklassen mit einem benannten Argument im `class`-Header:

```
class C(metaclass=Meta): ...
```

In Python 2.x nutzen Sie stattdessen Klassenattribute:

```
class C:
    __metaclass__ = Meta
    ...
```

Siehe auch `type()` in »Eingebaute Funktionen« auf Seite 105, dort finden Sie Informationen zur Abbildung von `class`-Anweisungen.

Die try-Anweisung

```
try:
    folge
except [typ [as wert]]:       # [, wert] in Python 2
    folge
[except [typ [as wert]]:
    folge]*
[else:
    folge]
[finally:
    folge]
try:
```

```
        folge
    finally:
        folge
```

Die try-Anweisung fängt Ausnahmen ab. try-Anweisungen können except-Klauseln, deren Inhalt als Handler für die Ausnahmen dient, die während der Ausführung des try-Blocks ausgelöst werden, else-Klauseln, die ausgeführt werden, wenn im try keine Ausnahmen auftreten, sowie finally-Klauseln nutzen, die unabhängig davon ausgeführt werden, ob eine Ausnahme auftrat oder nicht. except-Klauseln fangen Ausnahmen ab und verarbeiten sie, während finally-Klausen Abschlussaktionen definieren.

Ausnahmen können von Python oder explizit ausgelöst werden (siehe auch die im Abschnitt »Die raise-Anweisung« auf Seite 80 behandelte raise-Anweisung). In except-Klauseln ist *typ* ein Ausdruck, der die abzufangende Ausnahmeklasse angibt. Zusätzlich kann eine weitere Variable, *wert*, genutzt werden, um die Instanz der Ausnahmeklasse abzufangen, die ausgelöst wurde. Tabelle 16 führt alle Klauseln auf, die in einer try-Anweisung auftauchen können.

try muss entweder ein except oder ein finally oder beides enthalten. Die Reihenfolge der Teile muss try→except→else→finally sein. else und finally sind optional. Es kann null oder mehr excepts geben, es muss aber mindestens eins vorhanden sein, wenn es ein else gibt. finally arbeitet vernünftig mit return, break und continue zusammen (wird mit einem davon die Kontrolle aus dem try-Block herausgegeben, wird die finally-Klausel auf dem Weg nach draußen ausgeführt).

Tabelle 16: Klauselformate der try-Anweisung

Klauselformate	Interpretation
except:	Alle oder alle anderen Ausnahmen abfangen.
except *typ*:	Eine bestimmte Ausnahme abfangen.
except *typ* as *wert*:	Ausnahme und Instanz abfangen.
except (*typ1*, *typ2*):	Die angegebenen Ausnahmetypen abfangen.
except (*typ1*, *typ2*) as *wert*:	Die angegebenen Ausnahmetypen und die entsprechende Instanz abfangen.

Tabelle 16: Klauselformate der try-Anweisung (Fortsetzung)

Klauselformate	Interpretation
`else:`	Ausführen, wenn keine Ausnahmen ausgelöst werden.
`finally:`	Immer ausführen, wenn `try`-Block verlassen wird.

Übliche Varianten sind:

`except` *`typ`* `as X:`
: Ausnahme der entsprechenden Ausnahmeklasse abfangen und die ausgelöste Instanz `X` zuweisen. `X` bietet Zugriff auf alle Zustandsattribute, Ausgabestrings und auf der ausgelösten Instanz aufrufbaren Methoden. Bei älteren String-Ausnahmen werden `X` alle zusätzlichen Daten zugewiesen, die mit dem String weitergegeben werden (String-Ausnahmen wurden in Python 3.0 und 2.6 entfernt).

`except (`*`typ1, typ2, typ3`*`) as X:`
: Alle im Tupel aufgeführten Ausnahmen abfangen und `X` die Instanz zuweisen.

Beim `sys.exc_info()`-Aufruf in »Das Modul sys« auf Seite 138 finden Sie Informationen zum allgemeinen Zugriff auf die Ausnahmeklasse und -instanz (d.h. Typ und Wert), nachdem eine Ausnahme ausgelöst wurde.

Formen der try-Anweisung in Python 2.x

In Python 2.x funktionieren `try`-Anweisungen wie oben beschrieben, aber die in `except`-Handlern für den Zugriff auf die ausgelöste Instanz genutzte `as`-Klausel wird stattdessen durch ein Komma angegeben:

`except` *`klassenname`*`, X:`
: Ausnahmen der angegebenen Klasse abfangen und die ausgelöste Instanz `X` zuweisen.

`except (`*`name1`*`, `*`name2`*`, `*`name2`*`), `*`X`*`:`
: Alle angegebenen Ausnahmen abfangen und `X` die Instanz zuweisen.

Die raise-Anweisung

In Python 3.0 hat die raise-Anweisung die folgenden Formen:

`raise` *`instanz`* [`from` *`andereausnahme`*]
: Löst eine manuell erstellte Instanz einer Ausnahmeklasse aus (z. B. `Error(args)`).

`raise` *`klasse`* [`from` *`andereausnahmen`*]
: Neue Instanz von *`klasse`* erstellen und auslösen (äquivalent zu `raise` *`klasse`*`()`).

`raise`
: Löst die letzte Ausnahme erneut aus.

Die raise-Anweisung löst Ausnahmen aus. Sie können sie nutzen, um eingebaute oder benutzerdefinierte Ausnahmen explizit auszulösen. Ohne Argumente löst raise die letzte Ausnahme erneut aus. Die von Python selbst ausgelösten Ausnahmen werden unter »Eingebaute Ausnahmen« auf Seite 129 beschrieben.

Bei einem raise springt die Ausführung zur passenden except-Klausel der zuletzt betretenen passenden try-Anweisung oder zur obersten Ebene des Prozesses (wo sie das Programm beendet und eine Fehlermeldung auf der Standardausgabe ausgibt). Das ausgelöste Instanzobjekt wird der as-Variablen in der passenden except-Klausel zugewiesen (falls es eine gibt).

Die optionale from-Klausel erlaubt in Python 3.0 das Verketten von Ausnahmen: *`andereausnahme`* ist eine andere Ausnahmeklasse oder -instanz und wird an das `__cause__`-Attribut der ausgelösten Ausnahme gebunden. Wird die ausgelöste Ausnahme nicht abgefangen, gibt Python beide Ausnahmen als Teil der Standardfehlermeldung aus.

Klassenausnahmen

In Python 3.0 und 2.6 werden alle Ausnahmen durch Klassen identifiziert, die von der eingebauten Klasse `Exception` abgeleitet sein müssen (in 2.6 ist diese Ableitung nur bei Klassen neuen Stils erforderlich). Die Superklasse `Exception` bietet eine Standard-String-Darstellung und die Speicherung von Konstruktorargumenten im Tupel-Attribut `args`.

Klassenausnahmen unterstützen Ausnahmek*ategorien*, die leicht erweitert werden können. Weil except-Klauseln alle Unterklassen abfangen, wenn sie eine Superklasse angeben, können Ausnahmekategorien durch Änderung der Menge der Unterklassen verändert werden, ohne dass sich das auf die bestehende Ausnahmebehandlung auswirkt. Die ausgelöste Instanz bietet ebenfalls Speicher für zusätzliche Informationen zur Ausnahme:

```
class Allgemein(Exception):
    def __init__(self, x):
        self.data = x
class Spezifisch1(General): pass
class Spezifisch2(General): pass
try:
    raise Spezifisch1('spam')
except General as X:
    print(X.data)             # prints 'spam'
```

Formen der raise-Anweisung in Python 2.x

Vor Python 2.6 erlaubte Python 2.x, dass Ausnahmen durch Klassen und Strings identifiziert werden. Deswegen kann die raise-Anweisung die folgenden Formen annehmen, von denen es viele aus Gründen der Rückwärtskompatibilität gibt:

raise *string*
: Wird durch eine except-Klausel abgefangen, die das angegebene String-Objekt angibt.

raise *string*, *daten*
: Gibt einer Ausnahme ein zusätzliches *daten*-Objekt mit (der Standard ist None). Es wird in einer except string, X:-Klausel der Variablen X zugewiesen.

raise *Instanz*
: Entspricht raise instanz.__class__, instanz.

raise *klasse*, *instanz*
: Entspricht einem except, das diese *klasse* oder eine ihrer Superklassen angibt. Übergibt das Objekt *instanz* als zusätzliche Ausnahmedaten, die in einer except *klasse*, *X*:-Klausel *X* zugewiesen werden.

`raise` *`klasse`*
: Entspricht `raise` *`klasse`*`()` (erstellt eine Instanz der Klasse).

`raise` *`klasse`*`,` *`arg`*
: Entspricht `raise` *`klasse`*`(`*`arg`*`)` (erstellt eine Instanz der Klasse auf Nicht-Instanz arg).

`raise` *`klasse`*`, (`*`arg`* `[,` *`arg`*`]*)`
: Entspricht `raise` *`class`*`(`*`arg`*`,` *`arg`*`, ...)` (erstellt eine Instanz der Klasse).

`raise`
: Löst die aktuelle Ausnahme erneut aus.

String-Ausnahmen gelten seit Python 2.5 als veraltet (und führen zu Warnmeldungen). Python 2.x erlaubt außerdem ein drittes Element in `raise`-Anweisungen, das ein Traceback-Objekt sein muss, das anstelle des aktuellen Orts als der Ort genutzt wird, an dem die Ausnahme auftrat.

Die assert-Anweisung

```
assert ausdruck [, meldung]
```

Die `assert`-Anweisung führt Prüfungen zur Fehlersuche (Debugging) durch. Wenn *`ausdruck`* den Wahrheitswert falsch ergibt, wird `AssertionError` ausgelöst mit *`meldung`* als Zusatzinformation, sofern angegeben. Die Kommandozeilenoption `-O` entfernt `assert` aus dem übersetzten Code (die Tests werden nicht ausgeführt).

Die with-Anweisung

```
with ausdruck [as variable]:      # Python 2.6 und 3.0
    folge
with ausdruck [as variable]
        [, ausdruck [as variable]]*:  # 3.1
    folge
```

Die `with`-Anweisung hüllt einen eingebetteten Codeblock in einen Kontextmanager, der sichert, dass bei Eintritt in den und/oder Austritt aus dem Block bestimmte Aktionen ausgeführt werden. Das ist eine Alternative zu `try`/`finally` bei Objekten mit Kontextmana-

gern, die unabhängig davon, ob Ausnahmen ausgelöst wurden oder nicht, spezifische Abschlussaktionen vornehmen.

Es wird erwartet, dass *ausdruck* ein Objekt liefert, das das Kontextmanager-Protokoll unterstützt. Dieses Objekt kann zusätzlich einen Wert zurückliefern, der *variable* zugewiesen wird, wenn die optionale as-Klausel vorhanden ist. Klassen können eigene Kontextmanager definieren, und einige eingebaute Klassen wie die für Dateien und Threads bieten Kontextmanager mit Abschlussaktionen, die Dateien schließen, Thread-Sperren freigeben usw.:

```
with open(r'C:\misc\script', 'w') as datei:
    ...datei verarbeiten, wird automatisch bei Verlassen
                          des Blocks geschlossen ...
```

Mehr Informationen zur Verwendung von Datei-Kontextmanagern finden Sie im Abschnitt »Dateien« auf Seite 45. Die Python-Referenz bietet Informationen zu weiteren eingebauten Typen, die dieses Protokoll und diese Anweisung unterstützen, sowie zum Protokoll selbst.

Diese Anweisung wird seit Python 2.6 und 3.0 unterstützt und kann unter 2.5 folgendermaßen aktiviert werden:

```
from __future__ import with_statement
```

Mehrere Kontextmanager in Python 3.1

In Python 3.1 kann diese Anweisung auch mehrere (d.h. geschachtelte) Kontextmanager angeben. Eine beliebige Anzahl von Kontextmanager-Elementen kann durch Kommata getrennt angegeben werden, und mehrere Elemente bewirken das Gleiche wie geschachtelte with-Anweisungen. In Allgemeinen entspricht der 3.1-Code

```
with A() as a, B() as b:
    ...anweisungen...
```

dem folgenden in 3.1, 3.0 und 2.6 funktionierenden Code:

```
with A() as a:
    with B() as b:
        ...anweisungen...
```

Beispielsweise bewirkt der folgende Code, dass die Abschlussaktionen für beide Dateien automatisch ausgeführt werden, wenn der Anweisungsblock beendet wird, unabhängig davon, ob Ausnahmen auftraten oder nicht:

```
with open('data') as fin, open('results', 'w') as fout:
    for line in fin:
        fout.write(transform(line))
```

Python 2.x-Anweisungen

Python 2.x unterstützt die zuvor beschriebene `print`-Anweisung, unterstützt hingegen `nonlocal` nicht und `with` erst ab 2.6. Außerdem haben `raise`, `try` und `def` in Python 2.x, wie bereits bemerkt, eine etwas andere Syntax.

Die folgende Anweisung ist ebenfalls nur in Python 2.x verfügbar:

```
exec codestring [in globaldict [, localdict]]
```

Die `exec`-Anweisung kompiliert Codestrings und führt sie aus. *`codestring`* ist eine beliebige Python-Anweisung (oder mehrere durch Zeilenumbrüche getrennte Anweisungen) in Form eines Strings. Die Anweisung wird im gleichen Namensraum ausgeführt wie der Code, der `exec` ausführt, oder in den globalen/lokalen Namensraum-Dictionaries, wenn diese angegeben sind (*`localdict`* ist standardmäßig *`globaldict`*). *`codestring`* kann auch ein kompiliertes Codeobjekt sein. Siehe `compile()`, `eval()` und die Python 2.x-Funktion `execfile()` in »Eingebaute Funktionen« auf Seite 105.

In Python 3.0 wird diese Anweisung zur Funktion `exec()` (siehe »Eingebaute Funktionen« auf Seite 105). Die rückwärts- und vorwärtskompatible Syntax `exec(a, b, c)` wird auch in Python 2 akzeptiert.

Namensraum und Gültigkeitsregeln

Dieser Abschnitt behandelt die Regeln für Namensbindung und -suche (siehe auch »Namensformat« auf Seite 55 und »Namenskonventionen« auf Seite 56). Namen werden immer dann erzeugt, wenn ihnen erstmalig etwas zugewiesen wird, sie müssen aber

bereits existieren, wenn sie referenziert werden. Qualifizierte und unqualifizierte Namen werden unterschiedlich aufgelöst.

Qualifizierte Namen: Namensräume von Objekten

Qualifizierte Namen (`X`, in `object.X`) werden als *Attribute* bezeichnet und bilden Objektnamensräume. Zuweisungen in bestimmten lexikalischen Geltungsbereichen[4] initialisieren Objektnamensräume (Module, Klassen).

`Zuweisung: objekt.X = wert`
: Erzeugt oder ändert den Attributnamen `X` im Namensraum des referenzierten Objekts `objekt`.

`Referenz: objekt.X`
: Sucht nach dem Attributnamen `X` im Objekt `objekt` und dann in allen Oberklassen (bei Instanzen und Klassen), wie es das Prinzip der *Vererbung* vorsieht.

Unqualifizierte Namen: Lexikalische Geltungsbereiche

Unqualifizierte Namen (`X`) unterliegen lexikalischen Gültigkeitsregeln. Zuweisungen binden solche Namen an den lokalen Geltungsbereich, außer sie sind als global deklariert.

`Zuweisung: X = wert`
: Macht den Namen `X` standardmäßig lokal: Erzeugt oder ändert den Namen `X` im aktuellen lokalen Geltungsbereich. Ist `X` als `global` deklariert, wird der Name `X` im Geltungsbereich des einschließenden Moduls erzeugt oder geändert. Ist `X` unter Python 3.0 als `nonlocal` deklariert, wird der Name `X` im Geltungsbereich der einschließenden Funktion geändert. Lokale Variablen werden zur Laufzeit auf dem Aufruf-Stack gespeichert, damit schnell auf sie zugegriffen werden kann.

`Referenz: X`
: Sucht nach dem Namen `X` in mindestens vier Geltungsbereichskategorien: im aktuellen *lokalen* Geltungsbereich (Funktion);

4 Lexikalische Geltungsbereiche beziehen sich auf physisch (syntaktisch) geschachtelte Codestrukturen im Quellcode eines Programms.

dann in den lokalen Geltungsbereichen aller lexikalisch *einschließenden Funktionen* (falls es welche gibt und von innen nach außen), dann im aktuellen *globalen* Geltungsbereich (Modul), dann im *eingebauten* Geltungsbereich (der in Python 3.0 dem Modul `builtins` entspricht und in Python 2.x dem Modul `__builtin__`). Lokale und globale Geltungsbereichskontexte werden in Tabelle 17 definiert. `global`-Deklarationen bewirken, dass die Suche stattdessen im globalen Geltungsbereich beginnt, `nonlocal`-Deklarationen beschränken die Suche auf einschließende Funktionen.

Tabelle 17: Geltungsbereiche für unqualifizierte Namen

Kontext	Globaler Bereich	Lokaler Bereich
Modul	wie lokal	das Modul selbst
Funktion, Methode	umgebendes Modul	Funktionsaufruf
Klasse	umgebendes Modul	`class`-Anweisung
Skript, interaktiver Modus	wie lokal	`modul__main__`
`exec`, `eval`	globaler Bereich des Aufrufers (oder übergeben)	lokaler Bereich des Aufrufers (oder übergeben)

Statisch geschachtelte Geltungsbereiche

Die Suche in *einschließenden Funktionen* der letzten Regel des vorangegangenen Abschnitts (`Referenz: X`) bezeichnet man als *statisch geschachtelte Geltung*. In Version 2.2 wurde sie zum Standard gemacht. Beispielsweise funktioniert die folgende Funktion jetzt, weil die Referenz auf `x` in `f2` Zugriff auf den Geltungsbereich von `f1` hat:

```
def f1():
    x = 42
    def f2():
        print(x)
    f2()
```

Vor Python 2.2 versagt diese Funktion, da der Name `x` weder lokal (im Bereich von `f2`) noch global (im umgebenden Modul von `f1`)

noch eingebaut ist. Solche Fälle werden vor Version 2.2 mit Argumentvorgabewerten behandelt, um Werte aus dem unmittelbar umgebenden Geltungsbereichs zu erhalten (Vorgabewerte werden noch vor Eintritt in ein `def` ausgewertet):

```
def f1():
    x = 42
    def f2(x=x):
        print(x)
    f2()
```

Diese Regel gilt auch für `lambda`-Ausdrücke, die wie `def` einen verschachtelten Geltungsbereich enthalten und in der Praxis öfter verschachtelt auftreten:

```
def func(x):
    action = (lambda n: x ** n)        # ab 2.2
    return action
def func(x):
    action = (lambda n, x=x: x ** n)   # vor 2.2
    return action
```

Standardwerte sind gelegentlich immer noch erforderlich, um Schleifenvariablen zu referenzieren, wenn Funktionen innerhalb von Schleifen definiert werden (sie geben den letzten Wert der Schleifenvariablen wieder). Geltungsbereiche können beliebig geschachtelt werden, aber nur die einschließenden Funktionen (nicht Klassen) werden durchsucht:

```
def f1():
    x = 42
    def f2():
        def f3():
            print(x)   # im Geltungsbereich von f1 gefunden
        f3()
    f2()
```

Objektorientierte Programmierung

Klassen sind das wichtigste OOP-Konstrukt in Python. Sie erlauben mehrere Objektinstanzen einer Klasse, die Vererbung von Attributen sowie das Überladen von Operatoren.

Klassen und Instanzen

Klassenobjekte bieten Standardverhalten

- Die `class`-Anweisung erzeugt ein *Klassen*objekt und weist es einem Namen zu.
- Zuweisungen in `class`-Anweisungen erzeugen Klassena*ttribute*, die vererbbare Zustände und Verhalten von Objekten sind.
- Klassenm*ethoden* sind eingebettete `def`s mit einem speziellen ersten Argument, das das implizite Instanzobjekt aufnimmt.

Instanzobjekte werden auf Klassen erzeugt

- Wird eine Klasse wie eine Funktion aufgerufen, wird ein neues *Instanz*objekt erstellt.
- Jedes Instanzobjekt erhält Klassenattribute und bekommt seinen eigenen *Namensraum* für Attribute.
- Zuweisungen an Attribute des ersten Arguments (zum Beispiel `self.X=obj`) in Methoden erzeugen instanzspezifische *Attribute*.

Vererbungsregeln

- Klassen erben Attribute von allen in der Kopfzeile ihrer Klassendefinition aufgelisteten Klassen (Oberklassen). Die Angabe mehrerer Klassen bewirkt *Mehrfachvererbung*.
- Instanzen erhalten Attribute jener Klasse, die sie erzeugt hat, sowie die von ihr geerbten Attribute aller Superklassen.
- Der Vererbungsmechanismus durchsucht zuerst die Instanz, dann die Klasse, dann alle zugreifbaren Superklassen und nutzt die erste gefundene Version eines Attributnamens. Superklassen werden in der Hierarchie aufsteigend durchsucht, dann in der Breite (bei Klassen neuen Stils wird bei Hierarchien mit Mehrfachvererbung jedoch zuerst in der Breite gesucht, bevor in der Hierarchie aufgestiegen wird).

Pseudoprivate Attribute

Standardmäßig sind alle Attributnamen in Modulen und Klassen überall sichtbar. Spezielle Konventionen erlauben eine begrenzte

Datenkapselung, dienen jedoch überwiegend der Vermeidung von Namenskollisionen (siehe auch »Namensregeln« auf Seite 55).

Private Daten in Modulen

Namen in Modulen, die mit einem einzelnen Unterstrich beginnen (z.B. `_X`), und jene, die nicht in der `__all__`-Liste des Moduls vorkommen, werden bei einem Import der Form `from module import *` nicht übertragen. Das ist jedoch keine echte Kapselung, da auf solche Namen immer noch mit anderen Formen der `import`-Anweisung zugegriffen werden kann.

Private Daten in Klassen

Namen in `class`-Anweisungen, denen zwei Unterstriche vorangestellt sind (z.B. `__X`), werden beim Übersetzen verschleiert (Name-Mangling) und enthalten den Namen der umgebenden Klasse als Präfix (z.B. `_Klasse__X`). Dieses Präfix lokalisiert solche Namen für die umgebende Klasse und macht sie dadurch sowohl im Instanzobjekt `self` als auch in der Klassenhierarchie unterscheidbar.

Das ist hilfreich, um Namenskonflikte zu vermeiden, die bei Methoden gleichen Namens und Attributen in einem Instanzobjekt am Ende der Vererbungskette auftreten können (alle Zuweisungen an `self.attr` irgendwo in einer Hierarchie verändern den Namensraum dieses einen Instanzobjekts). Es ist dennoch keine strenge Kapselung, da solche Attribute weiterhin über den verschleierten Namen zugänglich sind.

Klassen neuen Stils

In Python 3.0 gibt es nur noch ein einziges Klassenmodell: Alle Klassen werden als Klassen neuen Stils betrachtet, ob sie von `object` abgeleitet sind oder nicht. In Python 2.x gibt es zwei Klassenmodelle: *das klassische Klassenmodell* (der Standard) und *das neue Klassenmodell* ab Version 2.2 (das durch Ableitung eines eingebauten Typs oder von `object` – `class A(object)` – genutzt wird).

Klassen neuen Stils (und alle Klassen in Python 3.0) unterscheiden sich von klassischen Klassen in folgender Weise:

- Hierarchien mit aus Mehrfachvererbung resultierendem Diamantmuster werden in etwas anderer Reihenfolge durchsucht – grob gesagt, werden sie zunächst in der Breite, dann in der Tiefe durchsucht.
- Klassen sind jetzt Typen und Typen Klassen. Die eingebaute Funktion `type(I)` liefert die Klasse, auf deren Basis eine Instanz erstellt wurde, statt eines allgemeinen Instanztyps. Normalerweise entspricht dieser `I.__class__`. Die Klasse `type` kann erweitert werden, um die Erzeugung von Klassen anzupassen und alle Klassen erben von `object`.
- Die Methoden `__getattr__` und `__getattribute__` werden für Attribute, die implizit von eingebauten Operationen abgerufen werden, nicht mehr aufgerufen. Sie werden nicht mehr für Operatoren-überladende Methodennamen, `__X__`, aufgerufen; die Suche nach Namen beginnt bei der Klasse, nicht bei der Instanz. Um den Zugriff auf derartige Methodennamen abzufangen und zu delegieren, müssen sie in Wrapper-/Proxy-Klassen neu definiert werden.
- Klassen neuen Stils bieten einen neuen Satz an Klassenwerkzeugen, einschließlich Slots, Eigenschaften, Deskriptoren und der Methode `__getattribute__`. Die meisten davon sind für den Aufbau von Codewerkzeugen gedacht. Der kommende Abschnitt, »Überladungsmethoden für Operatoren« auf Seite 90, liefert Informationen zu `__slots__`, `__getattribute__`, den Deskriptormethoden `__get__`, `__set__` und der `__delete__`-Methode; »Eingebaute Funktionen« auf Seite 105 liefert Informationen zu `property()`.

Überladungsmethoden für Operatoren

Klassen fangen eingebaute Operationen ab und implementieren sie, indem sie Methoden mit speziellen Namen definieren, die alle mit zwei Unterstrichen beginnen und enden. Diese Namen sind nicht reserviert und können ganz normal von Oberklassen geerbt werden. Pro Operation wird höchstens eine Methode gesucht und aufgerufen.

Python ruft automatisch die Überladungsmethoden einer Klasse auf, wenn Instanzen in Ausdrücken und anderen Kontexten vorkommen. Definiert eine Klasse z. B. eine Methode namens `__getitem__` und ist `X` eine Instanz dieser Klasse, ist der Ausdruck `X[i]` äquivalent zum Methodenaufruf `X.__getitem__(i)`.

Die Namen von Überladungsmethoden sind gelegentlich recht willkürlich: Die `__add__`-Methode einer Klasse muss keine Addition (oder Verkettung) implementieren. Außerdem können Klassen numerische Methoden und Methoden für Sammlungen sowie Operationen für veränderliche und unveränderliche Objekte vermischen. Für die meisten Namen zur Operatorenüberladung gibt es keine Standarddefinition. Es wird also die entsprechende Ausnahme ausgelöst, wenn die Methode nicht definiert ist.

Für alle Typen

`__new__(klasse [, args...])`
: Wird aufgerufen, um eine neue Instanz von `klasse` zu erstellen und zurückzuliefern. Erhält die an die Klasse übergebenen Konstruktorargumente. Wird eine Instanz der Klasse geliefert, wird die `__init__`-Methode der Instanz mit den gleichen Argumenten aufgerufen. Wird in gewöhnlichen Klassen nicht verwendet; ist dafür gedacht, Unterklassen unveränderlicher Typen die Anpassung der Instanzerzeugung zu ermöglichen und benutzerdefinierten Metaklassen die Anpassung der Klassenerstellung.

`__init__(self [, arg]*)`
: Wird bei `klasse(args...)` aufgerufen. Das ist der Konstruktor, der die neue Instanz, `self`, initialisiert. Bei einem Aufruf an einen Klassennamen wird `self` automatisch bereitgestellt; `arg` sind die Argumente für den Konstruktor. Sie können jede der Formen haben, die in einer Funktionsdefinition erlaubt sind (siehe »Ausdrucksanweisungen« auf Seite 60 und »Die def-Anweisung« auf Seite 65). Darf keinen Wert zurückliefern. Kann bei Bedarf manuell die `__init__`-Methode der Superklasse aufrufen und dabei die Instanz an `self` übergeben.

`__del__(self)`
: Wird bei Garbage-Collection der Instanz aufgerufen. Diese Destruktormethode räumt auf, wenn eine Instanz freigegeben wird. Eingebettete Objekte werden automatisch freigegeben, wenn das einschließende Objekt freigegeben wird (es sei denn, es gibt andere Referenzen darauf). In dieser Methode auftretende Ausnahmen werden ignoriert und geben nur eine Fehlermeldung auf `sys.stderr` aus. Mit der `try/finally`-Anweisung können verlässlichere Abschlussaktionen für einen Codeblock vorgegeben werden.

`__repr__(self)`
: Wird für `repr(self)` und interaktive `echo`-Befehle (und in Python 2.0 bei `` `self` ``) aufgerufen. Wird auch bei `str(self)` und `print(self)` aufgerufen, wenn `__str__` nicht definiert ist. Diese Methode liefert eine elementare »codemäßige« String-Darstellung von `self`.

`__str__(self)`
: Wird bei `str(self)` und `print(self)` aufgerufen (oder nutzt als Reserve `__repr__`, falls definiert). Diese Methode liefert eine vollständige »benutzerfreundliche« String-Darstellung von `self`.

`__format__(self, formatangabe)`
: Wird von der eingebauten Funktion `format()` aufgerufen (und entsprechend auch bei der `str.format()`-Methode von `str`-Strings), um eine formatierte String-Darstellung eines Objekts abzurufen. Siehe »Strings« auf Seite 17 und »Eingebaute Funktionen« auf Seite 105. Neu in Python 2.6 und 3.0.

`__hash__(self)`
: Wird bei `dictionary[self]` und `hash(self)` sowie anderen Operationen für hashbasierte Sammlungen aufgerufen, einschließlich denen, die den Objekttyp setzen. Diese Methode liefert einen eindeutigen und sich nie ändernden ganzzahligen Hashschlüssel.

`__bool__(self)`
: Wird bei Wahrheitswerttests und der eingebauten Funktion `bool()` aufgerufen; liefert `False` oder `True`. Ist `__bool__()` nicht

definiert, wird `__len__()` aufgerufen, falls das definiert ist und einen wahren Wert mit einer Länge ungleich null anzeigt. Sind weder `__len__()` noch `__bool__()` definiert, werden alle Instanzen als wahr betrachtet. In Python 3.0 neu eingeführt; in Python 2.x heißt diese Methode `__nonzero__` statt `__bool__`, verhält sich aber ansonsten identisch.

`__call__(self [, arg]*)`
: Wird auf `self(args...)` aufgerufen, wenn eine Instanz wie eine Funktion aufgerufen wird. `arg` kann alle Formen haben, die in der Argumentliste einer Funktionsdefinition zulässig sind (siehe »Ausdrucksanweisungen« auf Seite 60 und »Die def-Anweisung« auf Seite 65). Zum Beispiel entsprechen `__call__(self, a, b, c, d=5)` und `__call__(self, *pargs, **kargs)` entsprechen beide den Aufrufen `self(1, 2, 3, 4)` und `self(1, *(2,), c=3, **dict(d=4))`.

`__getattr__(self, name)`
: Wird bei `self.name` aufgerufen, wenn `name` ein undefiniertes Attribut ist (wird nicht aufgerufen, wenn `name` existiert oder von `self` geerbt wird). `name` ist ein String. Diese Methode liefert ein Objekt oder löst einen `AttributeError` aus.

  Wird in Python 3.0 nicht mehr bei `__X__`-Attributen aufgerufen, die implizit von einer eingebauten Operation angefordert werden; definieren Sie derartige Namen in Wrapper-/Proxy-Klassen um.

`__setattr__(self, name, wert)`
: Wird bei `self.name=wert` aufgerufen (alle Attributzuweisungen). Tipp: Weisen Sie über einen `__dict__`-Schlüssel zu, um rekursive Schleifen zu vermeiden: Eine `self.attr=x`-Anweisung in `__setattr__` ruft `__setattr__` erneut auf, `self.__dict__['attr']=x` hingegen nicht. Rekursion kann auch durch expliziten Aufruf der Superklassenversion vermieden werden: `object.__setattr__(self, name, wert)`.

`__delattr__(self, name)`
: Wird bei `del self.name` aufgerufen (alle Attributlöschungen). Tipp: Sie müssen rekursive Schleifen vermeiden, indem Sie die

Attributlöschung über `__dict__` oder eine Superklasse abwickeln, ähnlich wie bei `__setattr__`.

`__getattribute__(self, name)`
: Wird ohne Bedingungen aufgerufen, um den Attributzugriff für Instanzen der Klasse zu implementieren. Wird nie aufgerufen, wenn die Klasse auch `__getattr__` definiert (es sei denn, der Aufruf erfolgt explizit). Diese Methode sollte den (berechneten) Attributwert zurückliefern oder eine `AttributeError`-Ausnahme auslösen. Um eine endlose Rekursion in dieser Methode zu vermeiden, sollte die Implementierung immer die Superklassenmethode gleichen Namens aufrufen, um auf die Attribute zuzugreifen, die sie benötigt (z.B. `object.__getattribute__(self, name)`.

  Wird in Python 3.0 nicht mehr für `__X__`-Attribute ausgeführt, die implizit von eingebauten Operationen angefordert werden; definieren Sie derartige Namen in Wrapper-/Proxy-Klassen neu.

`__lt__(self, anderes)`,

`__le__(self, anderes)`,

`__eq__(self, anderes)`,

`__ne__(self, anderes)`,

`__gt__(self, anderes)`,

`__ge__(self, anderes)`
: Werden bei `self < anderes`, `self <= anderes`, `self == anderes`, `self != anderes`, `self > anderes` bzw. `self >= anderes` aufgerufen. Diese in Version 2.1 eingeführten Methoden werden als *Rich-Comparison*-Methoden bezeichnet und in Python 3.0 bei allen Vergleichsoperationen aufgerufen. `X < Y` ruft beispielsweise `X.__lt__(Y)` auf, falls diese Methode definiert ist. In Python 2.x werden diese Methoden `__cmp__` vorgezogen, wenn sie vorhanden sind, und `__ne__` wird auch bei `self <> anderes` aufgerufen.

  Diese Methoden können einen beliebigen Wert liefern. Wird der Vergleichsoperator in einem Booleschen Kontext verwendet, wird der Rückgabewert als das Boolesche Ergebnis des Operators interpretiert. Diese Methoden können auch das spe-

zielle Objekt `NotImplemented` liefern (nicht auslösen), wenn sie für die Operanden nicht definiert sind (was dazu führt, dass verfahren wird, als wäre die Methode überhaupt nicht definiert, und Python 2.x dazu zwingt, auf die allgemeinere `__cmp__`-Methode zurückzugreifen, falls sie definiert ist).

Es gibt keine implizierten Verhältnisse zwischen den Vergleichsoperatoren. Dass `x==y` wahr ist, muss nicht notwendigerweise heißen, dass `x!=y` falsch ist: `__ne__` sollte in Bezug auf `__eq__` definiert werden, wenn sich diese Operatoren symmetrisch verhalten sollen. Es gibt keine rechtsassoziativen Varianten (mit vertauschten Argumenten), wenn das linke Argument die Operation nicht unterstützt, das rechte aber schon. `__lt__` und `__gt__`, `__le__` und `__ge__` sind jeweils das Gegenstück des anderen, `__eq__` und `__ne__` sind jeweils ihr eigenes Gegenstück. In Python 3.0 sollten Sie für Sortierungen `__lt__` einsetzen.

`__slots__`
: Diesem Klassenattribut kann ein String, ein iterierbares Objekt oder eine Sequenz mit String zugewiesen sein, die die Namen von Attributen von Instanzen der Klasse angeben. Ist es in einer Klasse neuen Stils definiert (einschließlich allen Klassen unter Python 3), reserviert `__slots__` Platz für die deklarierten Attribute und verhindert die automatische Erzeugung von `__dict__` für jede Instanz (es sei denn, der String `'__dict__'` ist in `__slots__` eingeschlossen, was dazu führt, dass alle Instanzen auch ein `__dict__` haben und alle nicht in `__slots__` aufgeführten Attribute dynamisch eingefügt werden können).

  Zur Unterstützung von Klassen mit `__slots__` müssen Werkzeuge, die allgemein Attribute aufführen oder auf sie über den String-Namen zugreifen, speicherungsneutrale Werkzeuge wie `getattr()`, `setattr()` und `dir()` nutzen, die sich auf die `__dict__`- und die `__slots__`-Attributspeicherung beziehen. Eventuell müssen beide Attributquellen abgefragt werden.

`__dir__(self)`
: Wird bei `dir(self)` aufgerufen. Liefert eine Liste der Attributnamen. Neu in Python 3.0.

Für Sammlungen (Sequenzen, Abbildungen)

`__len__(self)`
: Wird bei `len(self)` und eventuell bei Wahrheitswerttests aufgerufen. Diese Methode liefert die Größe einer Sammlung. Bei Booleschen Tests sucht Python zunächst nach `__bool__`, dann nach `__len__` und behandelt das Objekt dann als wahr (in Python 2 heißt `__bool__ __nonzero__`). Die Länge null bedeutet falsch.

`__contains__(self, Element)`
: Wird für benutzerdefinierte Tests auf Enthaltensein bei `item in self` aufgerufen (andernfalls nutzen Tests auf Enthaltensein `__iter__`, falls definiert, sonst `__getitem__`). Diese Methode liefert einen wahren oder einen falschen Wert.

`__iter__(self)`
: Wird bei `iter(self)` aufgerufen. Diese in Version 2.2 eingeführte Methode ist Teil des Iterationsprotokolls. Sie liefert ein Objekt mit einer `__next__()`-Methode (eventuell `self`). Das Ergebnis der `__next__()`-Methode eines Objekts wird dann in allen Iterationskontexten (z.B. `for`-Schleifen) wiederholt aufgerufen und sollte das nächste Ergebnis liefern oder `StopIteration` auslösen, um das Fortschreiten in den Ergebnissen zu beenden (siehe auch die Abschnitte »Die for-Anweisung« auf Seite 64 und »Die yield-Anweisung« auf Seite 70). Ist `__iter__` nicht definiert, weicht die Iteration auf `__getitem__` aus. In Python 2.x heißt `__next__()` `next()`.

`__next__(self)`
: Wird von der eingebauten Funktion `next(self)` und allen Iterationskontexten aufgerufen, die Ergebnisse durchlaufen. Die Verwendung wird bei `__iter__()` beschrieben. Neu in Python 3.0; in Python 2.x heißt diese Methode `next()`, verhält sich ansonsten aber identisch.

`__getitem__(self, schlüssel)`
: Wird bei `self[key]`, `self[i:j:k]`, `x in self` und `for x in self` (und in allen Iterationskontexten) aufgerufen. Diese Methode implementiert alle positionsbasierten Operationen. Iterationskontexte (z.B. `in` und `for`) durchlaufen die Indizes von 0 bis zur Auslösung eines `IndexError`, es sei denn, `__iter__` ist definiert.

In Python 3.0 werden diese und die folgenden beiden Methoden auch bei Ausschnittoperationen (Slicing) aufgerufen. `schlüssel` ist dann das Slice-Objekt. Slice-Objekte können an einen anderen Slice-Ausdruck weitergegeben werden und haben die Attribute `start`, `stop` und `step`, die alle `None` sein können. Siehe auch `slice()` in »Eingebaute Funktionen« auf Seite 105.

`__setitem__(self, schlüssel, wert)`
: Wird bei `self[schlüssel]=wert`, `self[i:j:k]=wert` aufgerufen. Diese Methode wird bei Zuweisungen an einen Sammlungsschlüssel oder -index oder einen Ausschnitt einer Sequenz aufgerufen.

`__delitem__(self, key)`
: Wird bei `del self[key]`, `del self[i:j:k]` aufgerufen. Diese Methode wird aufgerufen, wenn ein Schlüssel, ein Index oder ein Ausschnitt einer Sequenz gelöscht wird.

`__reversed__(self)`
: Wird, falls definiert, von der eingebauten Funktion `reversed()` aufgerufen, um eine benutzerdefinierte Rückwärts-Iteration zu implementieren. Liefert ein neues Iterator-Objekt, das alle Objekte im Container in umgekehrter Reihenfolge durchläuft. Ist `__reversed__` nicht definiert, erwartet und nutzt `reversed()` das Sequenzprotokoll (die Methoden `__len__()` und `__getitem__()`).

Für Zahlen (binäre Operationen)

Wenn eine dieser Methoden die Operation für die übergebenen Argumente nicht unterstützt, sollte sie das eingebaute Objekt `NotImplemented` liefern (nicht auslösen), was bewirkt, dass so verfahren wird, als wäre die Methode überhaupt nicht definiert.

Einfache binäre Methoden

`__add__(self, anderes)`
: Bei `self + anderes`. Numerische Addition oder Sequenzverkettung.

`__sub__(self, anderes)`
: Bei `self - anderes`.

`__mul__(self, anderes)`
: Bei `self * anderes`. Numerische Multiplikation oder Sequenzwiederholung.

`__truediv__(self, anderes)`
: Wird in Python 3.0 bei `self / anderes` für alle Divisionen aufgerufen (bewahrt Rest). In Python 2.x wird `__div__` für die klassische Division aufgerufen (bei Ganzzahlen wird Rest abgeschnitten).

`__floordiv__(self, anderes)`
: Bei `self // anderes` für die (grundsätzlich) abschneidende Division aufgerufen.

`__mod__(self, anderes)`
: Bei `self % anderes`.

`__divmod__(self, anderes)`
: Bei `divmod(self, anderes)`.

`__pow__(self, anderes [, modulo])`
: Bei `pow(self, anderes [, modulo])` und `self ** anderes`.

`__lshift__(self, anderes)`
: Bei `self << anderes`.

`__rshift__(self, anderes)`
: Bei `self >> anderes`.

`__and__(self, anderes)`
: Bei `self & anderes`.

`__xor__(self, anderes)`
: Bei `self ^ anderes`.

`__or__(self, anderes)`
: Bei `self | anderes`.

Rechtsseitige binäre Methoden

`__radd__(self, anderes)`,

`__rsub__(self, anderes)`,

`__rmul__(self, anderes)`,

`__rtruediv__(self, anderes)`,

`__rfloordiv__(self, anderes)`,

`__rmod__(self, anderes)`,
`__rdivmod__(self, anderes)`,
`__rpow__(self, anderes)`,
`__rlshift__(self, anderes)`,
`__rrshift__(self, anderes)`,
`__rand__(self, anderes)`,
`__rxor__(self, anderes)`,
`__ror__(self, anderes)`

Das sind die rechtsseitigen Gegenstücke zu den binären Operatoren aus dem vorangegangenen Abschnitt. Binäre Operatormethoden haben eine rechtsseitige Variante, die mit einem `r`-Präfix beginnt, z. B. `__add__` und `__radd__`. Rechtsseitige Varianten haben die gleichen Argumentlisten, aber `self` ist auf der rechten Seite des Operators. `self + anderes` zum Beispiel ruft `self.__add__(anderes)` auf, aber `other + self` ruft `self.__radd__(anderes)` auf.

Die `r`-Form wird nur dann aufgerufen, wenn die Instanz rechts steht und der linke Operand keine Instanz einer Klasse ist, die die Operation implementiert:

Instanz + Nichtinstanz `__add__`
Instanz + Instanz `__add__`
Nichtinstanz + Instanz `__radd__`

Wenn zwei Instanzen verschiedener Klassen auftreten, die die Operation überschreiben, wird die Klasse auf der linken Seite vorgezogen. `__radd__` konvertiert oftmals oder tauscht die Reihenfolge und addiert erneut, um `__add__` aufzurufen.

Erweiterte binäre Methoden

`__iadd__(self, anderes)`,
`__isub__(self, anderes)`,
`__imul__(self, anderes)`,
`__itruediv__(self, anderes)`,
`__ifloordiv__(self, anderes)`,
`__imod__(self, anderes)`,

`__ipow__(self, anderes[, modulo])`,
`__ilshift__(self, anderes)`,
`__irshift__(self, anderes)`,
`__iand__(self, anderes)`,
`__ixor__(self, anderes)`,
`__ior__(self, anderes)`
: Das sind Methoden zur erweiterten Zuweisung (das Objekt wird verändert). Sie werden jeweils für folgende Zuweisungsformen aufgerufen: `+=`, `-=`, `*=`, `/=`, `//=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=` und `|=`. Diese Methoden sollten versuchen, die Operation auf dem Objekt selbst auszuführen (so dass `self` modifiziert wird), und das Ergebnis (eventuell `self`) liefern. Wenn eine Methode nicht definiert ist, fällt die erweiterte Operation auf die normalen Methoden zurück. Um `X += Y` auszuwerten, wobei `X` eine Instanz einer Klasse ist, die `__iadd__` definiert, wird `x.__iadd__(y)` aufgerufen. Sonst werden `__add__` und `__radd__` berücksichtigt.

Für Zahlen (andere Operationen)

`__neg__(self)`
: Bei `-self`.

`__pos__(self)`
: Bei `+self`.

`__abs__(self)`
: Bei `abs(self)`.

`__invert__(self)`
: Bei `~self`.

`__complex__(self)`
: Bei `complex(self)`.

`__int__(self)`
: Bei `int(self)`.

`__long__(self)`
: Bei `long(self)`.

`__float__(self)`
: Bei `float(self)`.

`__round__(self [, n])`
: Wird bei `round(self [, n])` aufgerufen. In Python 3.0 neu eingeführt.

`__index__(self)`
: Wird aufgerufen, um `operator.index()` zu implementieren. Wird auch in anderen Kontexten aufgerufen, in denen Python ein Integer-Objekt benötigt. Dazu zählen Instanzen, die als Indizes, Slice-Grenzen und als Argumente für die eingebauten Funktionen `bin()`, `hex()` und `oct()` verwendet werden. Muss einen Integer liefern.

  In Python 3.0 und 2.6 ähnlich, wird in 2.6 aber nicht für `hex()` und `oct()` aufgerufen (diese erfordern dort die Methoden `__hex__` und `__oct__`). In Python 3.0 ersetzt `__index__` die `__oct__`- und `__hex__`-Methoden von Python 2.x und der gelieferte Integer wird automatisch formatiert.

Für Deskriptoren

Die folgenden Methoden gelten nur, wenn eine Instanz einer Klasse, die eine der Methoden definiert (eine Deskriptorklasse), einem Klassenattribut einer anderen Klasse (die als Halterklasse, *Owner-Klasse*, bezeichnet wird) zugewiesen ist. Diese Methoden werden für den Zugriff auf das Attribut der Halterklasse aufgerufen, dessen Name der Instanz der Deskriptorklassen zugewiesen ist.

`__get__(self, instanz, halter)`
: Wird aufgerufen, um das Attribut der Halterklasse oder einer Instanz jener Klasse abzurufen. `halter` ist immer die Halterklasse; `instanz` ist die Instanz, über die auf das Attribut zugegriffen wird, oder `None`, wenn auf das Attribut direkt über die Halterklasse zugegriffen wird. Liefert den Attributwert oder löst einen `AttributeError` aus.

`__set__(self, instanz, wert)`
: Wird aufgerufen, um das Attribut auf einer Instanz der Halterklasse auf einen neuen Wert zu setzen.

`__delete__(self, instanz)`
: Wird aufgerufen, um das Attribut auf einer Instanz der Halterklasse zu löschen.

Für Kontextmanager

Die folgenden Methoden implementieren das Kontextmanager-Protokoll, das von der `with`-Anweisung genutzt wird (siehe »Die with-Anweisung« auf Seite 82).

`__enter__(self)`
: In den Laufzeitkontext für dieses Objekt eintreten. Die `with`-Anweisung weist den Rückgabewert dieser Methode dem Ziel zu, das in der `as`-Klausel der Anweisung angegeben wird (falls es eins gibt).

`__exit__(self, exc_typ, exc_wert, traceback)`
: Den Laufzeitkontext für dieses Objekt verlassen. Die Parameter beschreiben die Ausnahme, die veranlasst hat, dass der Kontext verlassen wird. Wird der Kontext ohne eine Ausnahme verlassen, sind alle drei Argumente `None`. Liefert einen Wert zurück, um zu verhindern, dass eine ausgelöste Ausnahme an den Aufrufer weitergereicht wird.

Methoden zur Operatorüberladung in Python 2.x

Methoden von Python 3.0

Die folgenden Methoden werden in Python 3.0 unterstützt, aber nicht in Python 2.x:

- `__bool__` (nutzen Sie in Python 2.x den Methodennamen `__nonzero__` oder `__len__`)
- `__next__` (nutzen Sie in Python 2.x den Methodennamen `next`)
- `__truediv__` (in Python 2.x nur verfügbar, wenn die echte Division aktiviert ist)
- `__dir__`
- `__round__`
- `__index__` für `oct()`, `hex()` (nutzen Sie Python 2.x `__oct__`, `__hex__`)

Methoden in Python 2.x

Die folgenden Methoden werden in Python 2.x unterstützt, aber nicht in Python 3.0:

`__cmp__(self, anderes)` *(und `__rcmp__`)*
: Wird bei `self > x`, `x == self`, `cmp(self, x)` usw. aufgerufen. Diese Methode wird für alle Vergleiche aufgerufen, für die keine der spezifischeren Methoden (wie `__lt__`) definiert ist oder geerbt wird. Sie liefert `-1`, `0` oder `1`, wenn `self` kleiner, gleich oder größer `anderes` ist. Ist keine der Rich-Comparison-Methoden oder `__cmp__` definiert, werden Klasseninstanzen auf Identität verglichen (Adresse im Speicher). `__rcmp__`, die rechtsseitige Variante der Methode, wird seit Version 2.1 nicht mehr unterstützt.

  Nutzen Sie in Python 3.0 die zuvor beschriebenen Vergleichsmethoden: `__lt__`, `__ge__`, `__eq__` usw. Nutzen Sie `__lt__` zum Sortieren.

`__nonzero__(self)`
: Wird bei einem Wahrheitswert aufgerufen (nutzt ansonsten `__len__`, falls definiert).

  In Python 3.0 wurde diese Methode in `__bool__` umbenannt.

`__getslice__(self, unten, oben)`
: Wird bei `self[unten:oben]` aufgerufen, um Ausschnitte aus Sequenzen abzurufen. Ist `__getslice__` nicht definiert und bei erweiterten dreielementigen Slicing-Operationen wird stattdessen ein *Slice-Objekt* erstellt und an die Methode `__getitem__` übergeben.

  In Python 2.x gelten diese und die beiden folgenden Methoden als veraltet, werden aber immer noch unterstützt – falls definiert, werden sie bei Slicing-Ausdrücken ihren elementbasierten Gegenstücken vorgezogen. In Python 3.0 wurden sie entfernt – beim Slicing wird stattdessen immer `__getitem__`, `__setitem__` oder `__delitem__` aufgerufen, und ein Slice-Objekt wird als Argument übergeben. Siehe `slice()` in »Eingebaute Funktionen« auf Seite 105.

`__setslice__(self, unten, oben, seq)`
: Wird bei `self[unten:oben]=seq` für eine Zuweisung an einen Sequenzausschnitt aufgerufen.

`__delslice__(self, unten, oben)`
: Wird bei `del self[unten:oben]` zum Löschen eines Sequenzausschnitts aufgerufen.

`__div__(self, anderes)` *(und* `__rdiv__`*,* `__idiv__`*)*
: Wird bei `self / anderes` aufgerufen, es sei denn, echte Division wurde mit `from` aktiviert (dann wird `__truediv__` genutzt). In Python 3.0 werden diese Methoden von `__truediv__`, `__rtruediv__` und `__itruediv__` ersetzt, weil `/` immer eine echte Division ist.

`__long__(self)`
: Wird bei `long(self)` aufgerufen. In Python 3.0 ersetzt der Typ `int` den Typ `long` vollständig.

`__oct__(self)`
: Wird bei `oct(self)` aufgerufen. Diese Methode liefert eine oktale String-Darstellung. Lassen Sie in Python 3.0 stattdessen `__index__` einen ganzzahligen Wert liefern.

`__hex__(self)`
: Wird bei `hex(self)` aufgerufen. Diese Methode liefert eine hexadezimale String-Darstellung. Lassen Sie in Python 3.0 stattdessen `__index__` einen ganzzahligen Wert liefern.

`__coerce__(self, anderes)`
: Wird bei arithmetischen Ausdrücken gemischten Typs sowie `coerce()` aufgerufen. Ergibt Tupel aus (`self, other`) nach der Konvertierung zu einem gemeinsamen Typ. Wenn `__coerce__` definiert ist, wird diese Methode aufgerufen, bevor echte Operatormethoden ausprobiert werden (z. B. vor `__add__`). Sollte ein Tupel mit Operanden ergeben, die in einen gemeinsamen Typ konvertiert wurden (oder `None`, wenn sie nicht konvertiert werden konnten). Siehe die Python Language Reference (*http://www.python.org/doc/*) für weitere Details zu den Konvertierungsregeln.

`__metaclass__`
: Klassenattribut, das Metaklassen zugewiesen ist. Nutzen Sie in Python 3.0 das benannte Argument `metaclass=M` in der Klassendeklaration.

Eingebaute Funktionen

Alle eingebauten Namen (Funktionen, Ausnahmen usw.) befinden sich im äußeren Geltungsbereich, der dem Modul `__builtins__` (das in Python 2 `__buildin__` heißt) entspricht. Da in diesem Bereich immer zuletzt gesucht wird, sind diese Funktionen in Programmen auch ohne Import stets verfügbar. Ihre Namen sind jedoch keine reservierten Wörter und können durch Zuweisung an die gleichen Namen in globalen oder lokalen Geltungsbereichen verdeckt werden.

`abs(N)`
: Ergibt den Absolutwert (Betrag) einer Zahl `N`.

`all(iterierbares)`
: Liefert `True`, wenn alle Elemente von `iterierbares` wahr sind.

`any(iterierbares)`
: Liefert `True`, wenn mindestens ein Element von `iterierbares` wahr ist.

`ascii(objekt)`
: Wie `repr()`, liefert einen String, der eine druckbare Darstellung des Objekts enthält, maskiert aber die Nicht-ASCII-Zeichen im `repr()`-String-Ergebnis mit `\x`-, `\u`- oder `\U`-Escape-Sequenzen. Dieses Ergebnis entspricht dem, das `repr()` unter Python 2.x lieferte.

`bin(N)`
: Wandelt einen ganzzahligen Wert in einen String mit binären Ziffern (Basis 2) um. Das Ergebnis ist ein gültiger Python-Ausdruck. Ist das Argument `N` kein Python-`int`-Objekt, muss es eine `__index__()`-Methode definieren, die einen `int` liefert. Siehe auch `int(x, 2)` zur Umwandlung von Binärzahlen, `0bNNN`-Binärliterale und den Typcode `b` in `str.format()`.

`bool([x])`
: Wandelt einen Wert in einen Booleschen Wert um und benutzt dabei die normale Prozedur zum Testen auf Wahrheitswerte. Wenn `x` falsch ist oder weggelassen wird, wird `False` zurückgegeben, sonst `True`. Aber `bool` ist auch eine Unterklasse von `int`. Von `bool` kann keine Unterklasse abgeleitet werden. Sie hat nur die zwei Instanzen `False` und `True`.

`bytearray([arg [, kodierung [, fehler]]])`
: Liefert ein neues Array mit Bytes. Der Typ `bytearray` ist eine veränderliche Sequenz von Integer-Werten im Bereich 0...255, der, wenn möglich, als ASCII-Text ausgegeben wird. Im Grunde ist das lediglich eine veränderliche Variante von `bytes`, die die meisten Operationen veränderbarer Sequenzen sowie die meisten Methoden des `str`-String-Typs unterstützt. `arg` kann folgende Werte haben: ein `str`-String mit der angegebenen `kodierung` (und optionalen `fehlern`) wie bei `str()`, eine durch einen Integer angegebene Größe (erstellt ein `bytearray` dieser Größe, dessen Elemente mit NULL-Bytes initialisiert sind); ein iterierbares Objekt mit Integern im Bereich 0..255 (`bytearray` wird mit den Elementen des Objekts initialisiert), ein `bytes`-String oder ein anderes `bytearray`-Objekt oder ein Objekt, das der Memory-View-Schnittstelle genügt (die zuvor als die Buffer-Schnittstelle bezeichnet wurde), das genutzt wird, um das Array zu initialisieren. Wird kein Argument angegeben, wird ein Array der Länge null erstellt.

`bytes([arg [, kodierung [, fehler]]])`
: Liefert ein neues `bytes`-Objekt, eine unveränderliche Sequenz mit Integern im Bereich 0...255. `bytes` ist eine unveränderliche Version von `bytearray`. Es hat die gleichen nicht verändernden String-Methoden und Sequenzoperationen. Normalerweise wird es genutzt, um 8-Bit-Byte-Strings mit Binärdaten darzustellen. Die Konstruktorargumente haben die gleiche Bedeutung wie bei `bytearray()`. `bytes`-Objekte können auch mit dem `b'ccc'`-Literal erstellt werden.

`chr(I)`
: Liefert einen Ein-Zeichen-String, dessen Unicode-Codepoint der Integer `I` ist. Die Umkehrung von `ord()` (z. B. ist `chr(97)` `'a'` und `ord('a')` `97`).

`classmethod(funktion)`
: Gibt eine Klassenmethode zu einer Funktion zurück. Eine Klassenmethode erhält eine Klasse als implizites erstes Argument, so wie eine Instanzmethode eine Instanz erhält. Verwenden Sie in Python 2.4 und höher den Funktionsdekorator `@classmethod` (siehe »Die def-Anweisung« auf Seite 65).

`compile(string, dateiname, art [, flags[, nicht_erben]])`
: Übersetzt `string` in ein Codeobjekt. `string` ist ein Python-String, der Python-Programmcode enthält. `dateiname` ist ein String, der für Fehlermeldungen benutzt wird und normalerweise der Name der Quelldatei oder bei interaktiver Eingabe `<string>` ist. `art` muss gleich `exec` sein, wenn `string` Anweisungen enthält, `eval` bei Ausdrücken oder `single` für eine einzelne interaktive Anweisung (im letzteren Fall werden Ausdrücke ausgegeben, die nicht `None` ergeben). Das resultierende Codeobjekt kann mit Aufrufen der eingebauten Funktionen `exec()` oder `eval()` ausgeführt werden. Die beiden optionalen letzten Argumente steuern, welche Future-Anweisungen sich auf die Kompilierung des Strings auswirken; fehlen sie, wird der String unter Verwendung der Future-Anweisungen kompiliert, die beim `compile()`-Aufruf aktiv waren (mehr Informationen liefert das Python-Handbuch).

`complex([real [, imag]])`
: Erzeugt ein komplexes Zahlenobjekt (kann auch mit dem Suffix `J` oder `j` erreicht werden: `real+imagJ`). `imag` ist mit 0 voreingestellt. Liefert `0j`, wenn beide Argumente fehlen.

`delattr(objekt, name)`
: Löscht das Attribut `name` (einen String) in `objekt`. Ähnlich wie `del objekt.name`, aber `name` ist ein String, keine Variable (`delattr (a,'b')` zum Beispiel ist identisch mit `del a.b`).

`dict([abbildung | iterierbares | benargs])`
: Gibt ein neues Dictionary zurück, das mit einer Abbildung, einer Sequenz (oder einem anderen iterierbaren Objekt) von Schlüssel/Wert-Paaren oder einer Menge von Schlüsselwortargumenten initialisiert wird. Ohne Argument wird ein leeres Dictionary zurückgegeben. Gleichzeitig der Klassenname eines erweiterbaren Typs.

`dir([objekt])`
: Ohne Argumente wird eine Liste aller Namen im aktuellen lokalen Geltungsbereich (Namensraum) zurückgegeben. Bei einem Objekt, das Attribute besitzt, ist das Ergebnis eine Liste mit Attributnamen dieses Objekts. Das funktioniert bei Modulen, Klassen und Klasseninstanzen ebenso wie bei eingebauten Objekten mit Attributen (Listen, Dictionaries usw.). Sie enthält geerbte Attribute und sortiert das Ergebnis. Für einfache Attributlisten eines einzigen Objekts verwenden Sie Attribute in `__dict__` (und bei einigen Klassen eventuell auch `__slots__`).

`divmod(X, Y)`
: Liefert ein Tupel aus `(X / Y, X % Y)`.

`enumerate(iterierbar, start=0)`
: Gibt ein iterierbares `enumerate`-Objekt zurück. `iterierbar` muss eine Sequenz, ein Iterator oder ein anderes Objekt sein, das Iteration unterstützt. Die `next()`-Methode des von `enumerate()` zurückgegebenen Iterators gibt ein Tupel mit einer Anzahl (beginnt bei `start`, standardmäßig null) und dem entsprechenden Objekt bei der Iteration über `iterierbar` zurück. Nützlich bei indizierten Folgen, wenn sowohl die Position als auch die Elemente in `for`-Schleifen benötigt werden: `(0, sequenz[0])`, `(1, sequenz[1])`, `(2, sequenz[2])`... Verfügbar seit Version 2.3.

`eval(ausdruck [, globals [, locals]])`
: Wertet `ausdruck` unter der Annahme aus, dass es sich um einen String mit einem gültigen Python-Ausdruck oder um ein übersetztes Codeobjekt handelt. `ausdruck` wird in den Namensräumen des `eval`-Aufrufs ausgewertet, außer es werden über die Argumente `globals` und/oder `locals` Namensraum-Dictio-

naries angegeben. locals hat globals als Voreinstellung, wenn nur globals übergeben wird. Gibt das Ergebnis von ausdruck zurück. Siehe auch die ebenfalls in diesem Abschnitt betrachteten eingebauten Funktionen compile() und exec() zur dynamischen Ausführung von Anweisungen.

exec(anweisungen [, globals [, locals]])
: Werten anweisungen aus. anweisungen muss entweder ein Python-String mit Python-Anweisungen oder ein kompiliertes Codeobjekt sein. Ist anweisungen ein String, wird dieser als Folge von Python-Anweisungen geparst und dann ausgeführt, bis ein Syntaxfehler auftritt. Ist es ein Codeobjekt, wird dieses einfach ausgeführt. globals und locals bewirken das Gleiche wie eval(), compile() kann eingesetzt werden, um Codeobjekte vorzukompilieren. Seit Python 2.x in Anweisungsform verfügbar (siehe »Spezifische Anweisungen« auf Seite 57).

filter(funktion, iterierbares)
: Liefert die Elemente von iterierbares, für die funktion True liefert. funktion muss einen Parameter erwarten. Ist funktion None, werden alle wahren Elemente geliefert.

  In Python 2.6 liefert dieser Aufruf eine Liste, in Python 3.0 ein iterierbares Objekt, das Werte auf Anforderung generiert und nur einmal durchlaufen werden kann (packen Sie es in einen list()-Aufruf, um die Generierung der Ergebnisse zu erzwingen, falls das erforderlich ist).

float([X])
: Konvertiert eine Zahl oder einen String X in eine Fließkommazahl (oder 0.0, falls kein Argument übergeben wird). Gleichzeitig der Klassenname eines erweiterbaren Typs.

format(wert [, formatangabe])
: Wandelt das Objekt wert in eine formatierte Darstellung um, die vom String formatangabe gesteuert wird. Die Interpretation von formatangabe ist vom Typ des Arguments wert abhängig (von den meisten eingebauten Typen wird die Standardsyntax zur Formatierung genutzt, die weiter oben in diesem Buch bei der Methode zur String-Formatierung beschrieben wurde). format(wert, formatangabe) ruft wert.__format__(formatspec)

auf und ist eine Basisoperation der Methode `str.format` (z.B. entspricht `format(1.3333, '.2f')` `'{0:.2f}'.format(1.3333)`).

`frozenset([iterierbares])`
: Gibt ein *fixiertes* Set mit Elementen aus `iterierbares` zurück. Fixierte Sets sind unveränderliche Sets, die keine Methoden zur Aktualisierung haben und in andere Sets eingebettet sein können.

`getattr(objekt, name [, standard])`
: Liefert den Wert des als String angegebenen Attributs `name` von `objekt`. Ähnlich wie `objekt.name`, aber `name` ist ein String, keine Variable (`getattr(a,'b')` zum Beispiel ist identisch mit `a.b`). Wenn das Attribut nicht existiert, wird `standard` zurückgegeben, sofern vorhanden, andernfalls wird ein `AttributeError` ausgelöst.

`globals()`
: Liefert ein Dictionary mit allen globalen Variablen des Aufrufers, d.h. den Namen und Werten des umgebenden Moduls.

`hasattr(objekt, name)`
: Liefert `True`, wenn `objekt` ein Attribut `name` (ein String) hat, sonst `False`.

`hash(objekt)`
: Liefert den Hashwert für `objekt`, sofern es einen besitzt. Hashwerte sind ganzzahlige Werte, die beim Suchen in Dictionaries zum schnelleren Vergleich von Dictionary-Schlüsseln genutzt werden.

`help([objekt])`
: Ruft das eingebaute Hilfesystem auf. (Für den interaktiven Einsatz gedacht.) Wird kein Argument angegeben, wird in der Interpreter-Console eine interaktive Hilfesitzung gestartet. Ist das Argument ein String, wird geprüft, ob es sich um den Namen eines Moduls, einer Funktion, einer Klasse oder einer Methode, sein Schlüsselwort oder ein Dokumentationsthema handelt, und gegebenenfalls wird der entsprechende Hilfetext angezeigt. Ist das Argument eine andere Art von Objekt, wird eine Hilfe für dieses Objekt generiert.

`hex(N)`
: Wandelt die Integer-Zahl `N` in einen String mit hexadezimalen Ziffern (Basis 16) um. Ist `N` kein Python-`int`-Objekt, muss es eine `__index__()`-Methode definieren, die einen `int` liefert.

`id(objekt)`
: Gibt die eindeutige Identität von `objekt` (seine Adresse im Speicher) als Ganzzahl zurück.

`__import__(name [, globals [, locals [, ausliste [, stufe] ]]])`
: Importiert und liefert zur Laufzeit das Modul zum übergebenen String `name` (z. B. `mod = __import__("dasmod")`). Dieser Aufruf ist meist schneller als die Konstruktion eines Strings mit einer `import`-Anweisung, der dann mit `exec()` ausgeführt wird. Diese Funktion wird intern von `import`- und `from`-Anweisungen aufgerufen und kann überschrieben werden, um Importoperationen anzupassen. Alle Argumente mit Ausnahme des ersten haben fortgeschrittenere Rollen (siehe Python Library Reference). Informationen zu verwandten Werkzeugen finden Sie auch unter dem `imp`-Modul der Standardbibliothek.

`input([prompt])`
: Gibt `prompt` aus, falls angegeben, liest dann eine Zeile vom `stdin`-Eingabestream (`sys.stdin`) und liefert diese als String. Das `\n` am Ende der Zeile wird abgeschnitten. Wird das Ende des Eingabestreams erreicht, wird ein `EOFError` ausgelöst. Auf Plattformen mit entsprechender Unterstützung nutzt `input()` GNU `readline`. In Python 2.x heißt diese Funktion `raw_input()`.

`int([zahl | string [, basis]])`
: Wandelt eine Zahl oder einen String in einen `int` um. Bei der Umwandlung von Fließkommazahlen wird der Nachkommaanteil abgeschnitten. `basis` darf nur angegeben werden, wenn das erste Argument ein String ist. Der Standardwert ist 10. Wird `0` für `basis` übergeben, wird die Basis auf Grundlage des String-Inhalts ermittelt. Andernfalls wird der übergebene Wert als Basis für die Umwandlung des Strings genutzt. `basis` kann `0` sowie `2...36` sein. Einem String kann ein Vorzeichen vorangehen, und er kann von Leerraumzeichen umrahmt sein. Liefert

0, wenn kein Argument angegeben wird. Ebenfalls der Klassenname eines erweiterbaren Typs.

`isinstance(objekt, klassenangabe)`
: Liefert `True`, wenn `objekt` eine Instanz von `klassenangabe` oder eine Subklasse von `klassenangabe` ist. `klassenangabe` kann auch ein Tupel mit Klassen und/oder Typen sein. In Python 3.0 sind Typen Klassen. Es gibt also keinen besonderen Fall für Typen mehr. In Python 2.x kann das zweite Argument auch ein Typobjekt sein, was diese Funktion als alternatives Werkzeug zur Typprüfung nutzbar macht (`isinstance(X, Typ)` vs. `type(X) is Type`).

`issubclass(klasse1, klasse2)`
: Liefert `True`, wenn `klasse1` von `klasse2` abgeleitet ist. `klasse2` darf auch ein Tupel von Klassen sein.

`iter(objekt [, wächter])`
: Liefert ein Iterator-Objekt, das eingesetzt werden kann, um die Elemente von `objekt` zu durchlaufen. Die zurückgelieferten Iterator-Objekte haben eine `__next__()`-Methode, die das nächste Element liefert oder `StopIteration` auslöst, um die Iteration zu beenden. Alle Iterationskontexte in Python nutzen dieses Protokoll, wenn `objekt` es unterstützt. Die eingebaute Funktion `next(I)` ruft `I.__next__()` automatisch auf. Wird nur ein Argument angegeben, wird erwartet, dass `objekt` einen eigenen Iterator bereitstellt oder eine Sequenz ist. Werden zwei Argumente angegeben, muss `objekt` ein aufrufbares Objekt sein, das aufgerufen wird, bis es `wächter` liefert. In Klassen kann `iter()` über `__iter__` überladen werden.

  In Python 2.x haben iterierbare Objekte statt einer `__next__()`-Methode eine `next()`-Methode. Aus Gründen der Vorwärtskompatibilität ist die eingebaute Funktion `next()` in 2.6 verfügbar und ruft `I.next()` statt `I.__next__()` auf (vor 2.6 kann `I.next()` stattdessen manuell aufgerufen werden).

`len(objekt)`
: Liefert die Anzahl von Elementen (die Länge) in der Sammlung `objekt`, die eine Sequenz oder eine Abbildung sein kann.

`list([iterierbares])`
: Liefert eine neue Liste, die alle Elemente im Objekt `iterierbares` enthält. Ist `iterierbares` bereits eine Liste, wird eine Kopie der Liste geliefert. Wird kein Argument angegeben, wird eine leere Liste geliefert. Ebenfalls der Klassenname eines erweiterbaren Typs.

`locals()`
: Ergibt ein Dictionary mit allen lokalen Variablen des Aufrufers (mit einem *`key:wert`*-Eintrag pro Variable).

`map(funktion, iterierbares [, iterierbares]*)`
: Wendet `funktion` auf alle Elemente einer Sequenz oder eines iterierbaren Objekts an und liefert die einzelnen Werte. `map(abs, (1, -2))` liefert beispielsweise `1` und `2`. Werden weitere iterierbare Argumente übergeben, muss `funktion` entsprechend viele Argumente akzeptieren und erhält bei jedem Aufruf ein Element aus jedem der Argumente; die Ausführung endet, wenn das Ende des kürzesten iterierbaren Objekts erreicht ist.

  Liefert in Python 2.6 eine Liste mit den Ergebnissen der einzelnen Aufrufe. Liefert in Python 3.0 ein iterierbares Objekt, das seine Ergebnisse auf Anforderung generiert und nur einmal durchlaufen werden kann (sofortige Ergebnisgenerierung erzwingen Sie durch Einbettung in einen `list()`-Aufruf). Außerdem werden in Python 2.x (aber nicht Python 3) alle Elemente in einer Ergebnisliste zusammengestellt, wenn `funktion` `None` ist; haben die Sequenzen unterschiedliche Länge, werden alle bis zur Länge der längsten mit `None` aufgefüllt. Ähnliche Mittel stellt in Python 3.0 das Modul `itertools` bereit.

`max(iterierbares [, arg]* [, key])`
: Wird nur das Argument `iterierbares` angegeben, wird das größte Element aus einem nicht leeren iterierbaren Objekt (z. B. String, Tupel oder Liste) geliefert. Werden mehrere Argumente angegeben, wird das größte unter allen Argumenten geliefert. Das optionale nur benannt verwendbare Argument `key` gibt eine Funktion an, die ein Argument akzeptiert und der Werttransformierung dient, wie die für `list.sort()` und `sorted()`.

`memoryview(objekt)`
: Liefert ein auf dem angegebenen Argument erstelltes Memory-View-Objekt. Memory-Views ermöglichen Python-Code den Zugriff auf die internen Daten von Objekten, die das entsprechende Protokoll unterstützen, ohne das Objekt zu kopieren. Memory kann als einfache Bytefolge oder komplexere Datenstruktur interpretiert werden. Eingebaute Objekte, die das Memory-View-Protokoll unterstützen, sind beispielsweise `bytes` und `bytearray` (siehe das Python-Handbuch). Memory-Views sind im Wesentlichen ein Ersatz für das *Buffer*-Protokoll und die entsprechende eingebaute Funktion in Python 2.x.

`min(iterierbares [, arg]* [, key])`
: Wird nur das Argument `iterierbares` angegeben, wird das kleinste Element aus einem nicht leeren iterierbaren Objekte (z.B. String, Tupel oder Liste) geliefert. Werden mehrere Argumente angegeben, wird das kleinste unter allen Argumenten geliefert. Das Schlüsselwortargument `key` verhält sich wie bei `max()`.

`next(iterator [, standard])`
: Ruft das nächste Element von `iterator` durch Aufruf der `__next__()`-Methode ab. Ist der `iterator` erschöpft, wird `standard` geliefert, wenn angegeben, andernfalls wird `StopIteration` ausgelöst.

  In Python 2.6 ist das aus Gründen der Vorwärtskompatibilität verfügbar, ruft aber `iterator.next()` statt `iterator.__next__()` auf. In den Python-Versionen vor 2.6 gibt es diesen Aufruf nicht; nutzen Sie stattdessen manuell `iterator.next()`.

`object()`
: Liefert ein neues, eigenschaftsloses Objekt. `object` ist die Basisklasse aller Klassen neuen Stils, d.h. in Python 2.x alle explizit von `object` abgeleiteten Klassen und alle Klassen in Python 3.0.

`oct(N)`
: Wandelt `N` in einen oktalen String (Basis 8) um. Ist `N` kein `int`-Objekt, muss es eine `__index__()`-Methode definieren, die eine ganze Zahl liefert.

`open(…)`

```
open(datei [, mode='r'
   [, buffering=None
   [, encoding=None        # nur Textmodus
   [, errors=None          # nur Textmodus
   [, newline=None         # nur Textmodus
   [, closefd=True] ]]]]]) # nur Deskriptoren
```

Liefert ein neues Dateiobjekt, das mit der über den Namen `datei` angegebenen externen Datei verbunden ist, oder löst einen `IOError` aus, wenn das Öffnen der Datei fehlschlägt. `datei` ist üblicherweise ein String- oder Bytes-Objekt, das den Namen (und, falls sich die Datei nicht im aktuellen Arbeitsverzeichnis befindet, den Pfad) der zu öffnenden Datei angibt. `datei` kann auch ein ganzzahliger Dateideskriptor für eine zu umhüllende Datei sein. Wird ein Dateideskriptor angegeben, wird er geschlossen, wenn das zurückgelieferte I/O-Objekt geschlossen wird, es sei denn, `closefd` ist auf `False` gesetzt. Alle Optionen können als Schlüsselwortargumente angegeben werden.

`mode` ist ein optionaler String, der den Modus angibt, in dem die Datei geöffnet wird. Der Standardwert ist `'r'`, Lesen im Textmodus. Andere wichtige Werte sind `'w'`, Schreiben (unter Leeren einer bereits bestehenden Datei) und `'a'` (Anhängen an eine bereits bestehende Datei). Wird im Textmodus `encoding` nicht angegeben, ist die genutzte Kodierung plattformabhängig, und `'\n'` wird in das plattformspezifische Zeilenendezeichen umgewandelt. Zum Lesen und Schreiben roher Bytes nutzen Sie die Binärmodi `'rb'`, `'wb'` oder `'ab'` und geben kein `encoding` an.

Verfügbare und kombinierbare Modi: `'r'`, lesen (Standard), `'w'`, Schreiben und Datei zuerst leeren, `'a'`, schreiben und an bestehende Datei anhängen, `'b'`, Binärmodus, `'t'`, Textmodus (Standard), `'+'`, Aktualisierung (lesen und schreiben), `'U'`, universeller Zeilenumbruchmodus (für die Rückwärtskompatibilität gedacht und in neuem Code nicht erforderlich). Der Standard `'r'` entspricht `'rt'` (zum Lesen von Text öffnen). `'w+b'` öffnet und leert eine bestehende Datei für wahlfreien binären Zugriff, während `'r+b'` die Datei ohne Leeren öffnet.

Python unterscheidet im Binär- und Textmodus geöffnete Dateien auch dann, wenn das zugrunde liegende Betriebssystem das nicht macht.[5]

- Bei der *Eingabe* liefern Dateien, die im Binärmodus geöffnet wurden (indem `'b'` an `mode` angehängt wurde), ihren Inhalt als `bytes`-Objekte, ohne dass Unicode dekodiert oder Zeilenenden übersetzt werden. In Textmodus (Standard oder wenn `'t'` an `mode` angehängt wurde) werden die Inhalte von Dateien als `str`-Objekte geliefert, nachdem die Bytes dekodiert wurden, indem entweder eine mit `encoding` explizit angegebene Kodierung oder eine plattformabhängig Standardkodierung eingesetzt wird und Zeilenenden mit `newline` übersetzt wurden.
- Bei der *Ausgabe* erwartet der Binärmodus ein `bytes`- oder ein `bytearray`-Objekt und schreibt es unverändert. Der Textmodus erwartet ein `str`-Objekt, kodiert es mit `encoding` und führt mit `newline` eine Übersetzung der Zeilenenden durch, bevor die Datei geschrieben wird.

`buffering` ist ein optionaler Integer-Wert, über den das Pufferungsverhalten bestimmt wird. Standardmäßig ist die Pufferung eingeschaltet. Übergeben Sie 0, um die Pufferung abzuschalten (nur im Binärmodus zulässig), 1, um die Zeilenpufferung einzuschalten, oder einen Integer >1 für vollständige Pufferung. Ist die Datenübertragung gepuffert, kann es sein, dass sie nicht sofort ausgeführt wird (mit `file.flush` erzwingbar).

`encoding` ist der Name der Kodierung, die zur Kodierung oder Dekodierung des Inhalts von Textdateien bei der Übertragung verwendet wird. Sollte nur im Textmodus verwendet werden. Die

5 Da der Dateimodus gleichermaßen Konfigurationsoptionen wie String-Datentypen impliziert, sollten Sie `open()` am besten so betrachten, als gäbe es dieses in zwei vollkommen verschiedenen Formen – Text und binär –, die über das Modusargument angegeben werden. Die Entwickler von Python entschieden sich, eine einzige Funktion so zu überladen, dass sie zwei Dateitypen mit modusspezifischen Argumenten und unterschiedlichen Inhaltstypen unterstützt, anstatt zwei separate Funktionen zum Öffnen von Dateien und zwei separate Objekttypen für Dateien bereitzustellen.

Standardkodierung ist plattformabhängig, aber es kann jede von Python unterstützte Kodierung angegeben werden. Das Modul `codecs` bietet eine Liste der unterstützten Kodierungen.

`errors` ist ein optionaler String, der angibt, wie Kodierungsfehler behandelt werden sollen. Sollte nur im Textmodus genutzt werden. Übergeben Sie `'strict'`, um eine `ValueError`-Ausnahme auszulösen, wenn es einen Kodierungsfehler gibt (der Standard `None` bewirkt das Gleiche), oder `'ignore'`, damit Fehler ignoriert werden. Das Ignorieren von Kodierungsfehlern kann zu Datenverlust führen. Die erlaubten Werte finden Sie bei `codecs.register()`.

`newline` steuert, wie universelle Zeilenumbrüche funktionieren, und gilt nur für den Textmodus. Kann `None` (Standard), `''`, `'\n'`, `'\r'` oder `'\r\n'` sein.

- Bei der *Eingabe* werden universelle Zeilenumbrüche aktiviert, wenn `newline` `None` ist: Zeilen können mit `'\n'`, `'\r'` oder `'\r\n'` enden, und diese Werte werden vor Rückgabe an den Aufrufer in `'\n'` übersetzt. Ist `newline` gleich `''`, werden universelle Zeilenumbrüche aktiviert, Zeilenenden aber unverändert an den Aufrufer geliefert. Bei einem der anderen zulässigen Werte werden Zeilen nur durch den angegebenen String begrenzt; das Zeilenende wird unübersetzt an den Aufrufer geliefert.
- Bei der *Ausgabe* werden `'\n'`-Zeichen in das Standardzeilenendezeichen der Plattform, `os.linesep`, umgewandelt, wenn `newline` gleich `None` ist. Ist `newline` gleich `''`, erfolgt keine Übersetzung. Bei einem der anderen zulässigen Werte, werden `'\n'`-Zeichen in den angegebenen String übersetzt.

Ist `closefd` gleich `False`, wird der zugrunde liegende Dateideskriptor offen gehalten, wenn die Datei geschlossen wird. Das funktioniert nicht, wenn ein Dateiname als String angegeben wird. Das Argument muss dann `True` sein (der Standard).

`ord(C)`
: Liefert den ganzzahligen Codepoint-Wert für den Ein-Zeichen-String `C`. Bei ASCII-Zeichen ist das der 7-Bit-ASCII-Code für `C`; bei Unicode ist es der Unicode-Codepoint des Ein-Zeichen-Unicode-Strings.

`pow(X, Y [, Z])`
: Liefert `X` zur Potenz `Y` [modulo `Z`]. Funktioniert wie der **-Operator.

`print([objekt,...][, sep=' '] [, end='n'] [, file=sys.stdout])`
: Gibt `objekt`(e) getrennt von `sep` und gefolgt von `end` in den Stream `file` aus. `sep`, `end` und `file` müssen, falls sie vorhanden sind, als Schlüsselwortargumente angegeben werden. Sie haben die gezeigten Standardwerte.

  Alle nicht benannten Argumente werden in Strings umgewandelt, wie es `str()` macht, und in den Stream geschrieben. `sep` und `end` müssen entweder Strings oder `None` sein (was zur Verwendung der Standardwerte führt). Wird kein `objekt` angegeben, wird `end` geschrieben. `file` muss ein Objekt mit einer `write(string)`-Methode sein, aber nicht notwendigerweise eine Datei; wird nichts oder `None` übergeben, wird `sys.stdout` verwendet. In Python 2.x ist die `print()`-Funktion als Anweisung verfügbar (siehe »Spezifische Anweisungen« auf Seite 57).

`property([fget[, fset[, fdel[, doc]]]])`
: Liefert ein Eigenschaftsattribut für Klassen neuen Stils (Klassen, die von `object` abgeleitet sind). `fget` ist eine Funktion zum Abrufen, `fset` eine Funktion zum Setzen eines Attributwerts, `fdel` ist eine Funktion zum Löschen eines Attribut. Dieser Aufruf kann selbst als Funktionsdekorator genutzt werden und liefert ein Objekt mit den Methoden `getter`, `setter` und `deleter`, die ebenfalls in dieser Rolle als Dekoratoren genutzt werden können (siehe »Die def-Anweisung« auf Seite 65).

`range([start,] stop [, step])`
: Ergibt eine Liste von aufeinanderfolgenden Integern zwischen `start` und `stop`. Mit einem Argument ergeben sich die Zahlen von `0` bis `stop-1`. Mit zwei Argumenten ergeben sich die Zahlen von `start` bis `stop-1`. Bei drei Argumenten ergeben sich die Zahlen von `start` bis `stop-1`, wobei `step` zu jedem Vorgänger im Ergebnis addiert wird. `start` und `step` haben die Standardwerte 0 und 1. `range(0,20,2)` ist eine Liste mit geraden Zahlen von 0 bis inklusive 18. Dies wird oft zur Bildung von Positionslisten oder wiederholten Zählungen für `for`-Schleifen benutzt.

In Python 2.6 liefert dieser Aufruf eine Liste, in Python 3.0 ein iterierbares Objekt, das Ergebniswerte auf Anforderung generiert und mehrfach durchlaufen werden kann (bei Bedarf erzwingen Sie die Ergebnisgenerierung mit einem Aufruf von `list()`).

`repr(object)`
: Liefert einen String mit einer druckbaren und eventuell als Code parsebaren Darstellung von `objekt`. In Python 2.x (aber nicht Python 3.0) entspricht das `` `objekt` `` (Backticks-Ausdruck).

`reversed(seq)`
: Liefert einen umgekehrten Iterator. `seq` muss ein Objekt mit einer `__reversed__()`-Methode sein oder das Sequenzprotokoll unterstützen (die Methoden `__len__()` und `__getitem__()` mit bei 0 beginnenden Argumenten).

`round(X [, N])`
: Liefert den Fließkommawert `X` auf `N`-Stellen nach dem Dezimaltrenner gerundet. Standardmäßig ist `N` null. Negative Werte können angegeben werden, um Stellen links des Dezimaltrenners anzuzeigen. Bei einem Aufruf mit nur einem Argument ist der Rückgabewert ein Integer, andernfalls ein Wert des gleichen Typs wie `X`. In Python 2.x ist das Ergebnis immer eine Fließkommazahl. Ruft in Python 3.0 `X.__round__()` auf.

`set([iterierbares])`
: Liefert ein Set, dessen Elemente aus `iterierbares` entnommen werden. Die Werte müssen unveränderlich sein. Sets von Sets können Sie aufbauen, indem Sie als innere Sets `frozenset`-Objekte verwenden. Ist kein `iterierbares` angegeben, wird ein leeres Set geliefert. Verfügbar seit Version 2.4. Siehe auch den Abschnitt »Sets« auf Seite 50 und das `{...}`-Set-Literal von Python 3.0.

`setattr(objekt, name, wert)`
: Weist `wert` dem Attribut `name` (ein String) in `objekt` zu. Wie `objekt.name = wert`, aber `name` ist ein Laufzeitstring, kein festgelegter Variablenname (z.B. ist `setattr(a,'b',c)` gleich `a.b=c`).

`slice([start ,] stop [, step])`
: Liefert ein Slice-Objekt, das einen Ausschnitt darstellt, mit den nur-lesbaren Attributen `start`, `stop` und `step`, von denen jedes `None` sein kann. Die Argumente sind dieselben wie bei `range`. Slice-Objekte können anstelle der Slice-Notation `i:j:k` verwendet werden (`X[i:j]` ist z. B. äquivalent zu `X[slice(i, j)]`).

`sorted(iterierbares, key=None, reverse=False)`
: Liefert eine neue sortierte Liste mit den Elementen aus `iterierbares`. Die optionalen Schlüsselwortargumente `key` und `reverse` haben die gleiche Bedeutung wie die für die zuvor bereits beschriebene Methode `list.sort()`; `key` ist eine ein Argument erwartende Transformationsfunktion. Funktioniert mit jedem iterierbaren Objekt und liefert ein neues Objekt, anstatt die Liste vor Ort zu ändern. Ist in `for`-Schleifen nützlich, um zu vermeiden, dass man `sort()`-Aufrufe in separate Anweisungen auslagern muss, weil sie `None` liefern. Verfügbar ab Python 2.4.

  In Python 2.x hat diese Funktion die Signatur `sorted(iterierbares, cmp=None, key=None, reverse=False)`, wobei die optionalen Argumente `cmp`, `key` und `reverse` die gleiche Bedeutung haben wie bei der bereits beschriebenen `list.sort()`-Methode unter Python 2.x.

`staticmethod(funktion)`
: Gibt eine statische Methode für `funktion` zurück. Eine statische Methode erhält kein implizites erstes Argument und ist daher hilfreich bei der Verarbeitung von Klassenattributen über Instanzen hinweg. Verwenden Sie in Version 2.4 den Funktionsdekorator `@staticmethod` (siehe den Abschnitt »Die def-Anweisung« auf Seite 65).

`str([objekt [, kodierung [, fehler]]])`
: Liefert eine »benutzerfreundliche« und druckbare String-Version eines Objekts. Das ist auch der Klassenname einer erweiterbaren Klasse. Operiert in einem der folgenden Modi:

- Ist nur `objekt` gegeben, wird eine druckbare Darstellung geliefert. Bei Strings ist das der String selbst. Der Unterschied zu `repr(objekt)` ist, dass `str(objekt)` nicht immer versucht, einen String zu liefern, den `eval()` akzeptieren würde. Ziel ist, eine

druckbare Version zu liefern. Liefert den leeren String, wenn gar keine Argumente angegeben werden.

- Werden `kodierung` und/oder `fehler` angegeben, wird das Objekt, das entweder ein Byte oder ein Bytearray sein kann, unter Verwendung des Codes für `kodierung` kodiert. `kodierung` ist ein String, der den Namen einer Kodierung liefert. Ist die Kodierung nicht bekannt, wird ein `LookupError` ausgelöst. Die Fehlerbehandlung wird mit `fehler` gesteuert. Bei `'strict'` (Standard) wird bei Kodierungsfehlern ein `ValueError` ausgelöst, bei `'ignore'` werden Fehler stillschweigend ignoriert, und bei `'replace'` wird das offizielle Unicode-Ersatzzeichen `U+FFFD` als Ersatz für Zeichen der Eingabe verwendet, die nicht dekodiert werden können. Siehe auch das Modul `codecs` und die vergleichbare Methode `bytes.decode()` (`b'a\xe4'.decode('latin-1')` entspricht `str(b'a\xe4', 'latin-1')`).

  In Python 2.x ist die Signatur dieser Methode einfacher, `str([object])`. Der Aufruf liefert einfach einen String mit einer druckbaren Darstellung von `objekt` (die erste Verwendung in Python 3.0).

`sum(iterierbares [, start])`
: Bildet die Summe von `start` und den Elementen eines iterierbaren Objekts von links nach rechts und gibt sie zurück. `start` hat den Standardwert 0. Die Elemente des iterierbaren Objekts sind normalerweise Zahlen und dürfen keine Strings sein (benutzen Sie `''.join(iterierares)` zum Zusammenfügen von iterierbaren Objekten mit Strings).

`super(typ [, objektOderTyp])`
: Gibt die Oberklasse von `typ` zurück. Wenn das zweite Argument weggelassen wird, ist das zurückgegebene Objekt unbeschränkt. Ist es ein Objekt, muss `isinstance(obj, typ)` wahr sein, ist es ein Typ, muss `issubclass(typ2, typ)` wahr sein. Ein Aufruf ohne Argumente entspricht `super(this_class, first_arg)`. In einer auf Einfachvererbung basierenden Hierarchie kann dieser Aufruf genutzt werden, um auf Elternklassen zu verweisen, ohne sie explizit zu benennen. Kann auch einge-

setzt werden, um kooperative Mehrfachvererbung in dynamischen Ausführungsumgebungen zu implementieren.

Funktioniert nur für Klassen neuen Stils in Python 2.x (`typ` ist dann nicht optional) und alle Klassen in Python 3.0.

`tuple([iterierbares])`
: Liefert ein neues Tupel mit den gleichen Elementen wie das übergebene `iterierbares`. Wenn `iterierbares` bereits ein Tupel ist, wird es direkt zurückgeliefert (keine Kopie). Liefert ohne Argument ein neues leeres Tupel. Ist gleichzeitig der Klassenname eines erweiterbaren Typs.

`type(objekt | (name, basis, dict))`
: Dieser Aufruf wird auf zwei unterschiedliche Weisen genutzt, die durch das Aufrufmuster festgelegt werden:

- Bei einem Argument wird ein Typobjekt geliefert, das den Typ von `objekt` repräsentiert. Nützlich für Typprüfungen in `if`-Anweisungen (z.B. `type(X)==type([])`) und für Dictionary-Schlüssel. Siehe auch das Modul `types` zu vordefinierten Typobjekten, die keine eingebauten Namen sind, und `isinstance()` weiter oben in diesem Abschnitt. Aufgrund der jüngeren Zusammenführung von Typen und Klassen ist `type(objekt)` im Grunde das Gleiche wie `objekt.__class__`. In Python 2.x schließt das Modul `types` auch eingebaute Typen ein.
- Dient bei drei Argumenten als Konstruktor und liefert ein neues Typobjekt. Bildet so eine dynamische Form der `class`-Anweisung. Der String `name` ist der Klassenname und wird zum `__name__`-Attribut, das Tupel `basis` führt die Basisklassen auf und wird zum `__bases__`-Attribut, und das Dictionary `dict` ist der Namensraum mit Definitionen für den Klasseninhalt und wird zum `__dict__`-Attribut. `class X(object): a = 1` entspricht `X = type('X', (object,), dict(a=1))`. Diese Abbildung wird üblicherweise zur Konstruktion von Metaklassen genutzt.

`vars([objekt])`
: Liefert ohne Argumente ein Dictionary mit den Namen des aktuellen lokalen Geltungsbereichs. Wird als Argument ein Modul, eine Klasse oder ein Instanzobjekt angegeben, wird ein Dictionary geliefert, das dem Attributnamensraum des `objekt`

entspricht (d.h. `__dict__`). Das Ergebnis sollte nicht verändert werden. Nützlich zur %-String-Formatierung.

`zip([iterierbares [, iterierbares]*])`
: Liefert eine Folge von Tupeln, die jeweils die korrespondierenden Elemente der als Argumente angegebenen iterierbaren Objekte enthalten. `zip('ab', 'cd')` liefert beispielsweise `('a', 'c')` und `('b', 'd')`]. Mindestens ein iterierbares Objekt ist erforderlich, ansonsten wird ein `TypeError` ausgelöst. Die resultierende Folge wird auf die Länge des kürzesten Arguments gekürzt. Wird nur ein iterierbares Objekt angegeben, wird eine Folge von Tupeln mit einem Element geliefert. Kann auch eingesetzt werden, um verpackte Tupel zu entpacken: `X, Y = zip(*zip(T1, T2))`.

  Liefert in Python 2.6 eine Liste, in Python 3.0 ein iterierbares Objekt, das die Ergebniswerte auf Anforderung generiert und nur einmal durchlaufen werden kann (erzwingen Sie bei Bedarf die Generierung der Ergebnisse mit einem `list()`-Aufruf). In Python 2.x (nicht jedoch in Python 3) entspricht `zip`, wenn es mehrere iterierbare Objekte mit der gleichen Länge gibt, `map` mit dem ersten Argument `None`.

Eingebaute Funktionen von Python 2.x

Die vorangegangene Liste gilt für Python 3. Semantische Unterschiede zwischen eingebauten Funktionen, die in Python 3.0 and 2.x verfügbar sind, wurden in jenem Abschnitt angemerkt.

Eingebaute Funktionen von Python 3.0, die Python 2.6 nicht unterstützt

Python 2.x bietet keine Unterstützung für die folgenden eingebauten Funktionen von Python 3.0:

- `ascii()` (arbeitet wie `repr()` in Python 2)
- `exec()` (in Python 2.x gibt es eine Anweisungsform mit ähnlicher Semantik)
- `memoryview()`

- `print()` (im `__builtin__`-Modul von Python 2 vorhanden, aber syntaktisch nicht direkt verwendbar, da die Ausgabe in Anweisungsform erfolgt und `print` in Python 2.x ein reserviertes Wort ist)

Eingebaute Funktionen von Python 2.6, die Python 3.0 nicht unterstützt

Python 2.x bietet zusätzlich die folgenden eingebauten Funktionen, von denen einige in Python 3.0 in anderen Formen verfügbar sind:

`apply(func, pargs [, kargs])`
: Ruft das aufrufbare Objekt `funktion` (eine Funktion, Methode, Klasse usw.) auf und übergibt dabei die positionellen Argumente im Tupel `pargs` und die benannten Argumente im Dictionary `kargs`. Liefert das Ergebnis des Funktionsaufrufs zurück.

  In Python 3.0 nicht mehr unterstützt. Nutzen Sie stattdessen die argumententpackende Aufrufsyntax `func(*pargs, **kargs)`. Diese Form wird auch in Python 2.6 vorgezogen, da sie allgemeiner und enger an Funktionsdefinitionen angelegt ist.

`basestring()`
: Die Basisklasse für gewöhnliche Strings und Unicode-Strings (nützlich für `isinstance`-Tests).

  In Python 3.0 wird Text jeder Form durch den Typ `str` repräsentiert (Unicode wie anderer Text).

`buffer(objekt [, position [, größe]])`
: Liefert ein neues Buffer-Objekt für ein unterstütztes `objekt` (siehe Python Library Reference).

  In Python 3.0 nicht mehr enthalten. Die eingebaute Funktion `memoryview()` bietet eine ähnliche Funktionalität.

`callable(objekt)`
: Liefert `1`, wenn `objekt` aufrufbar ist, andernfalls `0`.

  In Python 3.0 nicht mehr vorhanden. Nutzen Sie stattdessen `hasattr(f, '__call__')`.

`cmp(X, Y)`
: Liefert einen negativen Integer, null oder einen positiven Integer, um anzuzeigen, dass `X < Y`, `X == Y` respektive `X > Y` ist.

  In Python 3.0 entfernt, kann aber folgendermaßen simuliert werden: `(X > Y) - (X < Y)`. Allerdings wurden in Python 3.0 auch die meisten Anwendungsfälle für `cmp()` entfernt (Vergleichsfunktionen zum Sortieren und die `__cmp__`-Methode von Klassen).

`coerce(X, Y)`
: Liefert ein Tupel mit den in einen gemeinsamen Typ umgewandelten numerischen Argumenten `X` und `Y`.

  In Python 3.0 entfernt (wichtiger Anwendungsfall waren die klassischen Klassen von Python 2.x).

`execfile(dateiname [, globals [, locals]])`
: Wie `eval`, führt aber den Code in einer Datei aus, deren Name über `dateiname` als String angegeben wird (statt als Ausdruck). Anders als bei Importen wird dabei kein neues Modulobjekt für die Datei erstellt. Liefert `None`. Die Namensräume für den Code in `dateiname` entsprechen `eval`.

  Kann in Python 3.0 folgendermaßen simuliert werden: `exec(open(dateiname).read())`.

`file(filename [, mode[, bufsize]])`
: Ein Alias für die eingebaute Funktion `open()` und der erweiterbare Klassenname des eingebauten Typs `file`.

  In Python 3.0 wurde der Name `file` entfernt: Nutzen Sie `open()`, um Dateiobjekte zu erstellen, und die Klassen des Moduls `io`, um Dateioperationen anzupassen.

`input([prompt])` *(ursprüngliche Form)*
: Gibt `prompt` aus, falls angegeben, und liest dann eine Zeile mit Eingaben aus dem `stdin`-Stream (`sys.stdin`), wertet sie als Python-Code aus und liefert das Ergebnis. Entspricht `eval(raw_input(prompt))`.

  Da in Python 3.0 `raw_input()` in `input()` umbenannt wurde, steht die `input()`-Methode von Python 2.x nicht mehr zur

Verfügung, kann aber folgendermaßen simuliert werden: `eval(input(prompt))`.

`intern(string)`
: Fügt `string` in die Tabelle »internierter Strings« ein und liefert ihn zurück. Derartige Strings sind »unsterblich« und dienen der Leistungsoptimierung (sie können über das schnellere `is` auf Identität geprüft werden statt über `==` auf Gleichheit).

  In Python 3.0 wurde dieser Aufruf nach `sys.intern()` verschoben. Importieren Sie das Modul `sys`, um ihn zu nutzen.

`long(X [, basis])`
: Wandelt die Zahl oder den String `X` in einen `long`-Integer um. `basis` kann nur übergeben werden, wenn `X` ein String ist. Ist der Wert des Arguments 0, wird die Basis auf Basis des String-Inhalts ermittelt; andernfalls wird der Wert als Basis für die Umwandlung genutzt. Ist gleichzeitig der Klassenname eines erweiterbaren Typs.

  In Python 3.0 unterstützt der Ganzzahltyp `int` beliebige Genauigkeit und subsummiert deswegen den `long`-Typ von Python 2.x. Nutzen Sie in Python 3.0 `int()`.

`raw_input([prompt])`
: Das ist in Python 2.x der Name der Funktion, die in Python 3.0 `input()` heißt, und im vorangegangenen Abschnitt beschrieben wurde.

  Nutzen Sie in Python 3.0 `input()`.

`reduce(funktion, iterierbares [, anfang])`
: Wendet die zwei Argumente erwartende Funktion `funktion` nacheinander auf die Werte in `iterierbares` an, um die Sammlung auf einen einzigen Wert zu reduzieren. Ist `anfang` angegeben, wird er `iterierbares` vorangestellt.

  Ist in Python 3.0 über `functools.reduce()` weiterhin verfügbar. Importieren Sie das Modul `functools`, wenn Sie sie nutzen wollen.

`reload(modul)`
: Lädt, parst und führt ein bereits in den aktuellen Namensraum des Moduls importiertes `modul` erneut aus. Durch die Neuaus-

führung werden bestehende Werte der Attribute des Moduls vor Ort geändert. `modul` muss eine Referenz auf ein bestehendes Modulobjekt sein, es darf kein neuer Name oder String sein. Nützlich im interaktiven Modus, wenn Sie ein Modul neu laden wollen, nachdem Sie es repariert haben, ohne dazu Python neu starten zu müssen. Liefert das `modul`-Objekt zurück.

In Python 3.0 weiterhin über `imp.reload()` verfügbar. Importieren Sie das Modul `imp`, wenn sie diese Funktion nutzen wollen.

`unichr(i)`
: Liefert einen Unicode-String mit dem Zeichen, dessen Unicode-Code der Integer `i` ist (`unichr(97)` liefert beispielsweise den String `u'a'`). Das ist die Umkehrung von `ord` für Unicode-Strings and die Unicode-Version von `chr()`. Das Argument muss im Bereich 0...65535 liegen, andernfalls wird ein `ValueError` ausgelöst.

  In Python 3.0 repräsentieren gewöhnliche Strings Unicode-Zeichen: Nutzen Sie stattdessen `chr()` (`ord('\xe4')` ist beispielsweise `228`, und `chr(228)` sowie `chr(0xe4)` sind beide `'ä'`).

`unicode(string [, kodierung [, fehler]])`
: Dekodiert `string` unter Verwendung des Codecs für `kodierung`. Die Fehlerbehandlung erfolgt gemäß `fehler`. Standardmäßig wird UTF-8 im strengen Modus dekodiert, d.h., Fehler führen zu einem `ValueError`. Siehe auch das `codecs`-Modul in der Python Library Reference.

  In Python 3.0 gibt es keinen separaten Typ für Unicode mehr – der Typ `str` repräsentiert alle Textformen (Unicode und andere), und der Typ `bytes` repräsentiert binäre 8-Bit-Byte-Daten. Nutzen Sie `str` für Unicode-Text, `bytes.decode()` oder `str()`, um gemäß einer Kodierung rohe Bytes zu Unicode zu dekodieren, und gewöhnliche Dateiobjekte, um Unicode-Textdateien zu verarbeiten.

`xrange([start,] stop [, step])`
: Wie `range`, speichert aber nicht die gesamte Liste auf einmal (sondern generiert sie jeweils einzeln). Nützlich bei `for`-Schlei-

fen, wenn der Bereich groß, der verfügbare Speicherplatz aber gering ist. Optimiert den Speicherplatz, bietet in der Regel aber keine Geschwindigkeitsvorteile.

In Python 3.0 wurde die ursprüngliche `range()`-Funktion so geändert, dass sie ein iterierbares Objekt generiert, statt im Speicher eine Liste aufzubauen, und umfasst so die `xrange()`-Funktion von Python 2.x. Nutzen Sie in Python 3.0 `range()`.

Außerdem hat sich der `open`-Aufruf für Dateien in Python so radikal geändert, dass eine separate Betrachtung der Variante von Python 2.0 angebracht ist (in Python 2.x bietet `codecs.open` viele der Features, die `open` in Python 3 bietet):

`open(filename [, mode, [bufsize]])`

Liefert ein neues Dateiobjekt, das mit der über `dateiname` (ein String) angegebenen externen Datei verknüpft ist, oder löst `IOError` aus, wenn das Öffnen fehlschlägt. Der Dateiname wird dem aktuellen Arbeitsverzeichnis zugeordnet, wenn er keinen vorangestellten Pfad enthält. Die ersten beiden Argumente sind im Wesentlichen die gleichen wie für die C-Funktion `fopen`, und die Datei wird über das `stdio`-System verwaltet. Bei `open()` werden Dateidaten in Ihrem Skript immer als gewöhnliche `str`-Strings wiedergegeben, die Bytes aus der Datei enthalten (`codecs.open()` interpretiert Dateiinhalte als kodierten Unicode-Text, der mit `unicode`-Objekten dargestellt wird).

`modus` hat den Standardwert `'r'`. Mögliche Angaben sind `'r'` für Eingabe, `'w'` für Ausgabe, `'a'` für Anfügen sowie `'rb'`, `'wb'` oder `'ab'` für Binärdateien (ohne Umwandlung von Zeilenenden). Auf den meisten Systemen können diese Modi auch ein angehängtes `+` zum Öffnen im Schreib-/Lesemodus haben (z. B. `'r+'` zum Lesen/Schreiben und `'w+'` zum Lesen/Schreiben, wobei die Datei aber erst geleert wird).

`bufsize` (Puffergröße) hat einen implementierungsspezifischen Standardwert. Angaben sind 0 für ungepuffert, 1 für Zeilenpufferung und negativ für die Systemeinstellung, oder es ist eine spezifische Angabe. Ein gepufferter Datentransfer wird möglicherweise nicht sofort ausgeführt (kann mit `flush` erzwungen werden).

Eingebaute Ausnahmen

Dieser Abschnitt beschreibt die Ausnahmen, die Python während der Ausführung eines Programms auslösen kann. Seit Python 1.5 sind alle eingebauten Ausnahmen Klassenobjekte (vor 1.5 waren es Strings). Eingebaute Ausnahmen werden im Built-in-Geltungsnamensbereich bereitgestellt. Mit vielen Ausnahmen sind Zustandsinformationen verknüpft, die Informationen zur Ausnahme liefern.

Superklassen (Kategorien)

Die folgenden Ausnahmen werden nur als Superklassen für andere Ausnahmen genutzt.

`BaseException`
: Die Basisklasse für alle eingebauten Ausnahmen. Sie sollte von benutzerdefinierten Ausnahmen nicht direkt erweitert werden; nutzen Sie dazu `Exception`. Wird `str()` auf einer Instanz dieser Klasse aufgerufen, wird eine Darstellung der Kontruktorargumente geliefert, die bei Erstellung der Instanz übergeben wurden (oder der leere String, wenn keine Argumente angegeben wurden). Diese Konstruktorargumente werden in der Instanz gespeichert und über ihr `args`-Attribut als Tupel zur Verfügung gestellt. Subklassen erben dieses Protokoll.

`Exception`
: Die Basisklasse für alle eingebauten und nicht zur Systembeendung führenden Ausnahmen. Ist eine direkte Subklasse von `BaseException`.

  Benutzerdefinierte Klassen sollten diese Klasse erweitern. Diese Ableitung ist in Python 3.0 für alle benutzerdefinierten Ausnahmen erforderlich; Python 2.6 verlangt dies bei Klassen neuen Stils, erlaubt aber auch eigenständige Ausnahmeklassen.

  `try`-Anweisungen, die diese Ausnahme abfangen, fangen alle Ereignisse ab, die nicht zu einem Systemabbruch führen, weil diese Klasse die Basisklasse aller Ausnahmen außer `SystemExit`, `KeyboardInterrupt` und `GeneratorExit` ist (diese drei sind direkt von `BaseException` abgeleitet).

`ArithmeticError`
: Ausnahmekategorie für arithmetische Fehler: Superklasse von `OverflowError`, `ZeroDivisionError` und `FloatingPointError` und eine Subklasse von `Exception`.

`LookupError`
: Fehler bei Indizes von Sequenzen und Abbildungen: die Superklasse für `IndexError` und `KeyError` und eine Subklasse von `Exception`.

`EnvironmentError`
: Kategorie für Fehler, die außerhalb von Python eintreten: die Superklasse für `IOError` und `OSError` und eine Subklasse von `Exception`. Die ausgelöste Instanz enthält die informativen Attribute `errno` und `strerror` (und eventuell `filename` bei Ausnahmen in Bezug auf Pfade), die auch zu `args` zählen.

Spezifische ausgelöste Ausnahmen

Die folgenden Klassen sind Ausnahmen, die tatsächlich ausgelöst werden. `NameError`, `RuntimeError`, `SyntaxError`, `ValueError` und `Warning` sind spezifische Ausnahmen und zugleich Superklassen anderer eingebauter Ausnahmen.

`AssertionError`
: Wird ausgelöst, wenn der Test einer `assert`-Anweisung `False` ist.

`AttributeError`
: Wird bei einem Attributzugriffs- oder -zuweisungsfehler ausgelöst.

`EOFError`
: Wird bei unmittelbarem Dateiende bei `input()` (oder in Python 2 `raw_input()`) ausgelöst. Lesemethoden für Dateiobjekte liefern am Ende der Datei stattdessen ein leeres Objekt.

`FloatingPointError`
: Wird ausgelöst, wenn eine Fließkommaoperation fehlschlägt.

`GeneratorExit`
: Wird ausgelöst, wenn die `close()`-Methode eines Generators aufgerufen wird. Erbt direkt von `BaseException` statt von `Exception`, da das kein Fehler ist.

`IOError`
: Wird ausgelöst, wenn eine ein-/ausgabe- oder dateibezogene Operation fehlschlägt. Von `EnvironmentError` abgeleitet. Enthält die oben genannten Zustandsinformationen.

`ImportError`
: Wird ausgelöst, wenn `import` oder `from` ein Modul oder Attribut nicht finden kann.

`IndentationError`
: Wird ausgelöst, wenn im Quellcode eine fehlerhafte Einrückung aufgefunden wird. Abgeleitet von `SyntaxError`.

`IndexError`
: Wird ausgelöst bei Sequenzpositionen außerhalb des Bereichs (Abruf und Zuweisung). Slice-Indizes werden stillschweigend so angepasst, dass sie in den zulässigen Bereich fallen; ist ein Index ein Integer, wird ein `TypeError` ausgelöst.

`KeyError`
: Wird ausgelöst bei Referenzen auf nicht existierende Abbildungsschlüssel (Abruf). Zuweisungen an einen nicht existierenden Schlüssel erstellen den Schlüssel.

`KeyboardInterrupt`
: Wird ausgelöst, wenn der Benutzer die Unterbrechungstaste drückt (üblicherweise Strg-C oder Entf). Während der Ausführung wird regelmäßig auf Unterbrechungen geprüft. Diese Ausnahme ist direkt von `BaseException` abgeleitet, damit sie nicht versehentlich von Code abgefangen wird, der `Exception` abfängt und so verhindert, dass die Ausführung beendet wird.

`MemoryError`
: Wird ausgelöst bei zu wenig Speicherplatz. Führt dazu, dass ein Stacktrace ausgegeben wird, wenn ein durchgegangenes Programm zu dem Problem geführt hat.

`NameError`
: Wird ausgelöst, wenn ein lokal oder global unqualifizierter Name nicht gefunden werden kann.

`NotImplementedError`
: Wird ausgelöst, wenn erwartete Protokolle nicht definiert werden. Abstrakte Klassenmethoden können diesen Fehler auslö-

sen, wenn sie erfordern, dass eine Methode neu definiert wird. Von `RuntimeError` abgeleitet. (Sollte nicht mit `NotImplemented` verwechselt werden, einem speziellen eingebauten Objekt, das von einigen Methoden zur Operatorüberladung ausgelöst wird, wenn Operandentypen nicht unterstützt werden.)

`OSError`
: Wird ausgelöst bei Fehlern im `os`-Modul (seine `os.error`-Ausnahme). Abgeleitet von `EnvironmentError`. Mit den zuvor beschriebenen Zustandsinformationen.

`OverflowError`
: Wird ausgelöst bei zu großen arithmetischen Operationen. Kann bei Integern nicht auftreten, da diese beliebige Genauigkeit unterstützen, und auch die meisten Fließkommaoperationen werden nicht geprüft.

`ReferenceError`
: In Verbindung mit schwachen Referenzen ausgelöst. Siehe `weakref`-Modul.

`RuntimeError`
: Eine selten genutzte alles abfangende Ausnahme.

`StopIteration`
: Wird ausgelöst, wenn ein Iterator keine weiteren Werte liefert. Ausgelöst von der eingebauten Funktion `next(X)` und der eingebauten Methode `X.__next__()` (`X.next()` in Python 2).

`SyntaxError`
: Wird ausgelöst, wenn der Parser auf einen Syntaxfehler stößt. Das kann während Importoperationen, Aufrufen von `eval()` und `exec()` sowie beim Lesen von Code in einer Skriptdatei der obersten Ebene oder der Standardeingabe geschehen. Instanzen dieser Klasse haben die Attribute `filename`, `lineno`, `offset` und `text` für Zugriff auf Details; die `str()`-Methode der Ausnahmeinstanz liefert nur die Nachricht.

`SystemError`
: Wird ausgelöst bei Interpreter-internen Fehlern, die nicht ernsthaft genug sind, um den Interpreter herunterzufahren (diese sollten gemeldet werden).

`SystemExit`
: Wird ausgelöst bei einem Aufruf von `sys.exit(N)`. Wird diese Ausnahme nicht behandelt, beendet der Python-Interpreter die Ausführung, ohne einen Stacktrace auszugeben. Ist der übergebene Wert ein Integer, gibt er den Beendigungsstatus des Systems an (der an die C-Funktion `exit()` übergeben wird); ist der Wert `None`, ist der Beendigungsstatus null; hat er einen anderen Wert, wird dieser ausgegeben, und der Beendigungsstatus ist eins. Abgeleitet direkt von `BaseException`, um zu verhindern, dass diese Ausnahme versehentlich von Code abgefangen wird, der `Exception` abfängt und so verhindert, dass der Interpreter beendet wird.

  `sys.exit()` löst diese Ausnahme aus, damit Handler für Aufräumarbeiten (`finally`-Klauseln von `try`-Anweisungen) ausgeführt werden und der Debugger ein Skript ausführen kann, ohne die Steuerung zu verlieren. Die Funktion `os._exit()` beendet bei Bedarf sofort (z.B. in einem Kindprozess nach einem Aufruf von `fork()`). Siehe auch das Standardbibliotheksmodul `atexit` zur Angabe von Exit-Funktionen.

`TabError`
: Wird ausgelöst, wenn Quellcode eine fehlerhafte Mischung von Leerzeichen und Tabulatoren enthält. Abgeleitet von `IndentationError`.

`TypeError`
: Wird ausgelöst, wenn eine Operation oder Funktion auf ein Objekt eines ungeeigneten Typs angewandt wird.

`UnboundLocalError`
: Wird ausgelöst, wenn lokale Namen referenziert werden, denen noch kein Wert zugewiesen wurde. Abgeleitet von `NameError`.

`UnicodeError`
: Wird ausgelöst bei Unicode-bezogenen Kodierungs- und Dekodierungsfehlern. Eine Superklassenkategorie und eine Subklasse von `ValueError`.

`UnicodeEncodeError`,
`UnicodeDecodeError`,
`UnicodeTranslateError`
: Wird ausgelöst bei Fehlern bei der Unicode-Verarbeitung. Subklassen von `UnicodeError`.

`ValueError`
: Wird ausgelöst, wenn eine eingebaute Operation oder Funktion ein Argument des richtigen Typs, aber mit falschem Wert erhält und die Situation nicht durch einen spezifischeren Fehler wie `IndexError` beschrieben wird.

`WindowsError`
: Wird ausgelöst bei Windows-spezifischen Fehlern; Subklasse von `OSError`.

`ZeroDivisionError`
: Wird ausgelöst bei Divisions- oder Modulo-Operationen mit 0 auf der rechten Seite.

Ausnahmen der Kategorie Warnung

Die folgenden Ausnahmen fallen in die Kategorie Warnung:

`Warning`
: Superklasse für alle Warnungskategorien, Unterklasse von `Exception`.

`UserWarning`
: Vom Code des Benutzers generierte Warnungen.

`DeprecationWarning`
: Warnungen zu veralteten Funktionalitäten.

`PendingDeprecationWarning`
: Warnungen zu Funktionalitäten, deren Verwendung in Zukunft veraltet sein könnte.

`SyntaxWarning`
: Warnungen bei unklarer Syntax.

`RuntimeWarning`
: Warnungen bei unklarem Laufzeitverhalten.

`FutureWarning`
: Warnungen zu Konstrukten, deren Semantik sich in der Zukunft ändern wird.

`ImportWarning`
: Warnungen zu wahrscheinlichen Fehlern beim Import von Modulen.

`UnicodeWarning`
: Warnungen in Bezug auf Unicode.

`BytesWarning`
: Warnungen zu `bytes` und Buffer-Objekten (Memory-View).

Warnungssystem

Warnungen werden gemeldet, wenn zukünftige Sprachänderungen die Funktionalität von bestehendem Python-Code in zukünftigen Versionen beeinträchtigen könnte. Warnungen können so konfiguriert werden, dass sie zur Ausgabe von Meldungen führen, Ausnahmen auslösen oder ignoriert werden. Das Warnungssystem kann genutzt werden, um Warnungen über die Funktion `warnings.warn` zu melden:

```
warnings.warn("Funktion veraltet", DeprecationWarning)
```

Darüber hinaus können Sie Filter hinzufügen, um bestimmte Warnungen zu unterdrücken. Sie können einen regulären Ausdruck auf eine Meldung oder einen Modulnamen anwenden, um Warnungen verschiedener Stufen auszublenden. Beispielsweise lässt sich eine Warnung bei der Benutzung des veralteten Moduls `regex` folgendermaßen vermeiden:

```
import warnings
warnings.filterwarnings(action = 'ignore',
                        message='.*regex module*',
                        category=DeprecationWarning,
                        module = '__main__')
```

Der hinzugefügte Filter betrifft nur Warnungen der Klasse `DeprecationWarning`, die im Modul `__main__` auftreten, und wendet einen regulären Ausdruck an, um nur die entsprechenden Warnungen zu filtern, deren Meldung das `regex`-Modul anführt. Warnungen kön-

nen einmalig oder bei jeder Ausführung des beanstandeten Codes ausgegeben werden oder in Ausnahmen umgewandelt werden, die das Programm anhalten (wenn die Ausnahme nicht abgefangen wird). Für weitere Informationen lesen Sie die Dokumentation des Moduls `warnings` in Version 2.1 und später. Siehe ebenso das Argument `-W` im Abschnitt »Kommandozeilenoptionen« auf Seite 3.

Eingebaute Ausnahmen von Python 2.x

Der Satz und die Form der Klassenhierarchie der verfügbaren Ausnahmen sind in Python 2.6 etwas anders als in der Beschreibung für 3.0 im vorangegangenen Abschnitt. Unter anderem gilt in Python 2.x:

- `Exception` ist die oberste Basisklasse (nicht die Klasse `BaseException`, die es in Python 2 nicht gibt).
- `StandardError` ist eine zusätzliche `Exception`-Subklasse und eine Superklasse über allen eingebauten Ausnahmen außer `SystemExit`.

Vollständige Informationen liefert das Python 2.6-Handbuch.

Eingebaute Attribute

Manche Objekte exportieren spezielle Attribute, die durch Python vordefiniert werden. Die folgende Liste ist unvollständig, da viele Typen ihre eigenen Attribute haben (siehe dazu die Eintragungen für spezifische Typen in der Python Library Reference).[6]

`X.__dict__`
: Dictionary für die schreibbaren Attribute von Objekt `X`.

6 Ab Python 2.1 können Sie auch selbst definierte Attribute durch einfaches Zuweisen *Funktionsobjekten* hinzufügen. Python 2.x unterstützt außerdem die speziellen Attribute `I.__methods__` und `I.__members__`: Listen beispielsweise mit den Namen von Methoden- und Datenmembern einiger eingebauter Typen. Diese Methoden wurden in Python 3 entfernt; nutzen Sie die eingebaute Funktion `dir()`.

`I.__class__`
: Das Klassenobjekt, auf dem Instanz `I` erzeugt wurde. In Version 2.2 und höher gilt das auch für Objekttypen; die meisten Objekte haben ein `__class__`-Attribut (z.B. `[].__class__ == list == type([])`).

`C.__bases__`
: Tupel aus den Basisklassen von `C`, wie im Kopf der Klassenanweisung für `C` angegeben.

`X.__name__`
: Der Name von Objekt `X` als String: bei Klassen der Name im Anweisungskopf, bei Modulen der zum Importieren verwendete Name, oder aber `"__main__"` für das Modul, das das Hauptprogramm darstellt (z.B. die Hauptdatei zum Starten des Programms).

Module der Standardbibliothek

Standardbibliotheksmodule sind immer vorhanden, müssen aber zur Benutzung in Clientmodulen importiert werden. Um auf sie zuzugreifen, benutzt man eine der folgenden Formen:

- `import modul`, Namen aus dem Modul werden qualifiziert genutzt (`module.name`)
- `from modul import name`, um spezifische Namen aus dem Modul unqualifiziert zu nutzen (`name`)
- `from module import *`, um alle Namen aus dem Modul unqualifiziert zu nutzen (`name`)

Um z.B. den Namen `argv` im Modul `sys` zu verwenden, benutzen Sie entweder `import sys` und den Namen `sys.argv` oder stattdessen `from sys import argv` und den Namen `argv` ohne Präfix.

Die Standardbibliothek birgt viele Module; der nachfolgende Abschnitt ist deswegen nicht erschöpfend. Ziel ist, die am häufigsten verwendeten Namen aus den am häufigsten verwendeten Modulen zu dokumentieren. Eine vollständigere Referenz zu den Modulen der Standardbibliothek bietet Pythons Library Reference.

Für sämtliche Module im nachfolgenden Abschnitt gilt:

- Aufgelistete exportierte Namen, gefolgt von Klammern, sind Funktionen, die aufzurufen sind. Die anderen sind einfache Attribute wie zum Beispiel Variablennamen in Modulen.
- Modulbeschreibungen dokumentieren den Status des Moduls unter Python 3.0; Informationen zu Attributen, die es nur in Python 2 oder 3 gibt, finden Sie in den Python-Handbüchern.

Das Modul sys

Das Modul sys enthält Interpreter-spezifische Exporte. Außerdem bietet es Zugriff auf einige Umgebungskomponenten wie die Kommandozeile, Standard-Streams usw.

`argv`
: Liste mit den Strings der Kommandozeile: [`Kommando`, `Argumente` ...]. Wie das Array `argv` in C.

`byteorder`
: Zeigt die native Byteordnung an (zum Beispiel `big` für Big-Endian).

`builtin_module_names`
: Tupel aus den Namensstrings der in diesen Interpreter einkompilierten C-Module.

`copyright`
: String mit dem Copyright des Python-Interpreters.

`dllhandle`
: Das Handle der Python-DLL als Integer; nur Windows (siehe Python Library Reference).

`displayhook(funktoin)`
: Wird von Python zur Anzeige von Ergebniswerten in interaktiven Sitzungen aufgerufen. Weisen Sie eine Funktion mit einem Argument `sys.displayhook` zu, um die Ausgabe anzupassen.

`excepthook(typ, wert, traceback)`
: Wird von Python zur Anzeige von Informationen zu nicht abgefangenen Ausnahmen auf `stderr` aufgerufen. Für ange-

passte Ausnahmeanzeigen weisen Sie eine dreiargumentige Funktion `sys.excepthook` zu.

`exc_info()`
: Liefert ein Tupel mit drei Werten, die die Ausnahme beschreiben, die aktuell behandelt wird (`typ`, `wert`, `traceback`). `typ` ist dabei die Ausnahmeklasse, `wert` die ausgelöste Instanz und `traceback` ein Objekt, das Zugriff auf den Laufzeit-Aufruf-Stack in dem Zustand bietet, den er hatte, als die Ausnahme auftrat. Auf den aktuellen Thread bezogen. Subsumiert `exc_type`, `exc_value` und `exc_traceback` aus Python 1.5 und später (die in Python 3 alle drei entfernt wurden). Informationen zur Verarbeitung von Tracebacks finden Sie unter dem Modul `traceback` in der Python Library Reference. Der Abschnitt »Die try-Anweisung« auf Seite 77 liefert weitere Informationen zu Ausnahmen.

`exec_prefix`
: Liefert einen String mit dem sitespezifischen Verzeichnispräfix, das den Ort angibt, an dem die plattformabhängigen Python-Dateien installiert sind. Der Standardwert ist */usr/local* bzw. der Wert, der bei der Kompilierung über ein Argument festgelegt wurde. Wird genutzt, um Shared Library-Module (in *<exec_prefix>/lib/python<version>/lib-dynload*) sowie Konfigurationsdateien zu finden.

`executable`
: Vollständiger Dateipfadname der ausführbaren Datei des laufenden Python-Interpreters.

`exit([N])`
: Beendet einen Python-Prozess mit Status `N` (Standardwert 0) durch Auslösen der eingebauten Ausnahme `SystemExit` (kann in einer `try`-Anweisung abgefangen und ignoriert werden). Siehe auch `SystemExit` (in »Eingebaute Ausnahmen« auf Seite 129) und die Funktion `os._exit()` (im Abschnitt »Das Systemmodul os«), die ohne Ausnahmebehandlung augenblicklich beendet (nützlich in Kindprozessen nach einem `os.fork()`). Beim Modul `atexit` finden Sie Informatioen zur Definition von Funktionen zum Programmabbruch.

`getcheckinterval()`
: Gibt das »Prüfintervall« des Interpreters zurück. Siehe `setcheckinterval` weiter unten in dieser Liste.

`getdefaultencoding()`
: Liefert den Namen der aktuell von der Unicode-Implementierung verwendeten Standardkodierung.

`getfilesystemencoding()`
: Liefert den Namen der Kodierung, die genutzt wird, um Unicode-Dateinamen in Systemdateinnamen umzuwandeln, oder `None`, wenn die Standardkodierung des Systems verwendet wird.

`getrefcount(objekt)`
: Liefert den aktuellen Referenzzähler von `objekt` (+1 für das Argument des Aufrufs).

`getrecursionlimit()`
: Die maximal erlaubte Schachtelungstiefe des Python-Callstacks. Siehe auch `setrecursionlimit` weiter unten in dieser Liste.

`getsizeof(objekt [, standard])`
: Liefert die Größe eines Objekts in Bytes. Das Objekt kann ein Objekt beliebigen Typs sein. Alle eingebauten Objekte liefern korrekte Ergebnisse, aber die Ergebnisse bei benutzerdefinierten Erweiterungen sind implementierungsspezifisch. `standard` stellt einen Wert, der geliefert wird, wenn der Objekttyp das Protokoll zum Abrufen der Größe nicht unterstützt.

`_getframe([tiefe])`
: Gibt ein Frame-Objekt vom Python-Callstack zurück (siehe die Python Library Reference).

`hexversion`
: Die Versionsnummer von Python als einzelner Integer (am besten mit der eingebauten Funktion `hex` zu betrachten). Wächst mit jeder neuen Ausgabe.

`intern(string)`
: Fügt `string` in die Tabelle »internierter« Strings ein und liefert den so gecachten String – den String selbst oder ein Kopie. Das

Caching von Strings bietet kleine Leistungsverbesserungen beim Suchen in Dictionaries: Wenn die Schlüssel im Dictionary und der zur Suche verwendete Schlüssel gecacht sind, kann der Vergleich der Schlüssel (nach Berechnung der Hashwerte) über Referenzen statt über Werte erfolgen. Üblicherweise werden die in Python-Programmen genutzten Namen automatisch gecacht, ebenso die Schlüssel der Dictionaries, die Module, Klassen und Instanzattribute festhalten.

`last_type`
`last_value`
`last_traceback`
: Typ, Wert und Traceback-Objekt der letzten unbehandelten Ausnahme (vornehmlich für das Debugging nach Programmabbrüchen).

`maxsize`
: Ein Integer, der den maximalen Wert angibt, den eine Variable des Typs `Py_ssize_t` aufnehmen kann. Üblicherweise ist das auf einer 32-Bit-Plattform `2**31 - 1` und auf einer 64-Bit-Plattform `2**36 - 1`.

`maxunicode`
: Ein Integer, der den höchsten unterstützten Codepoint für ein Unicode-Zeichen angibt. Dieser Wert ist von der Konfigurationsoption abhängig, die vorgibt, ob Unicode-Zeichen in UCS-2 oder in UCS-4 gespeichert werden.

`modules`
: Dictionary der bereits geladenen Module mit einem `name: object`-Eintrag je Modul. Schreibbar (d.h., `del sys.modules['name']` erzwingt das erneute Laden eines Moduls beim nächsten Import).

`path`
: String-Liste, die den Importsuchpfad für Module angibt. Initialisiert aus der Umgebungsvariablen `PYTHONPATH`, *.pth*-Pfaddateien und den jeweiligen installationsabhängigen Standardwerten. Schreibbar (zum Beispiel fügt `sys.path.append('C:\\dir')` während der Skriptausführung ein Verzeichnis zum Suchpfad hinzu).

Das erste Element, `path[0]`, ist das Verzeichnis mit dem Skript, das genutzt wurde, um den Python-Interpreter zu starten. Ist das Skriptverzeichnis nicht verfügbar (weil der Interpreter interaktiv gestartet wurde oder das Skript von der Standardeingabe gelesen wird), ist `path[0]` der leere String. Das weist Python an, zuerst im aktuellen Verzeichnis zu suchen. Das Skriptverzeichnis wird vor den Einträgen aus `PYTHONPATH` eingefügt.

`platform`
: Ein String, der das System angibt, auf dem Python läuft: z.B. `'sunos5'`, `'darwin'`, `'linux2'`, `'win32'`, `'cygwin'`, `'PalmOS3'`. Nützlich für Abfragen in plattformabhängigem Code. Tipp: `'win32'` meint alle aktuellen Versionen von Windows. Kann auch mit `sys.platform[:3]=='win'` oder `sys.platform.startswith('win')` getestet werden.

`prefix`
: Liefert einen String mit dem sitespezifischen Verzeichnispräfix, das den Ort angibt, an dem die plattformabhängigen Python-Dateien installiert sind. Der Standardwert ist */usr/local* bzw. der Wert, der bei der Kompilierung über ein Argument festgelegt wurde. Python-Bibliotheksmodule werden im Verzeichnis *<prefix>/lib/python<version>* gespeichert, plattformunabhängige Header-Dateien in *<prefix>/include/python<version>*.

`ps1`
: String mit dem primären Prompt-Zeichen für den interaktiven Modus. Standardmäßig >>>, wenn kein anderer Wert zugewiesen wird.

`ps2`
: String mit dem sekundären Prompt-Zeichen für Fortsetzungen zusammengesetzter Anweisungen. Standardmäßig ..., wenn kein anderer Wert zugewiesen wird.

`dont_write_bytecode`
: Ist es `True`, versucht Python nicht, beim Import von Quellmodulen *.pyc*- oder *.pyo*-Dateien zu schreiben (siehe auch die Kommandozeilenoption *-B*).

`setcheckinterval(wiederholungen)`
: Legt fest, wie oft der Interpreter prüft, ob periodische Aufgaben zu erledigen sind (z. B. Thread-Umschaltungen, Signalbehandlung). Gemessen in Instruktionen der virtuellen Maschine, der Standardwert ist 10. Im Allgemeinen wird eine Python-Anweisung in mehrere Instruktionen der virtuellen Maschine übersetzt. Kleinere Werte erhöhen die Ansprechbarkeit von Threads, aber gleichzeitig auch die Zusatzkosten für die Thread-Umschaltung.

`setdefaultencoding(name)`
: Setzt die aktuelle Standardkodierung für Strings, die von der Unicode-Implementierung genutzt wird. Zur Verwendung durch des Modul `site` gedacht und nur beim Start verfügbar.

`setprofile(funktion)`
: Setzt die Systemprofilfunktion auf `funktion`: der Profiler-Hook (wird nicht für jede Zeile ausgeführt). Weitere Informationen finden Sie in der Python Library Reference.

`setrecursionlimit(tiefe)`
: Setzt die maximale Verschachtelungstiefe des Python-Callstacks auf `tiefe`. Diese Grenze verhindert einen Überlauf des C-Stacks und damit den Absturz von Python bei unendlicher Rekursion. Auf Windows ist der Standard 1000, der Wert kann aber auch anders lauten.

`settrace(funktion)`
: Setzt die System-Trace-Funktion auf `funktion`: der Callback-Hook für Änderungen des Programmorts oder -zustands, der von Debuggern usw. genutzt wird. Weitere Informationen liefert die Python Library Reference.

`stdin`
: Standardeingabe-Stream: ein vorab geöffnetes Dateiobjekt. In einem Skript kann die Eingabe umgeleitet werden, indem dem Namen ein Objekt zugewiesen wird, das `read`-Methoden besitzt (zum Beispiel `sys.stdin=MyObj()`). Wird für die Interpreter-Eingabe benutzt, einschließlich der eingebauten Funktion `input()` (und `raw_input()` in Python 2).

`stdout`
: Standardausgabe-Stream: ein vorab geöffnetes Dateiobjekt. In einem Skript kann die Ausgabe umgeleitet werden, indem dem Namen eine Objekt zugewiesen wird, das `write`-Methoden besitzt (zum Beispiel `sys.stdout=open('log', 'a')`). Wird für einige Prompts und die eingebaute Funktion `print()` (und die `print`-Anweisung von Python 2) genutzt.

`stderr`
: Standardfehler-Stream: ein vorab geöffnetes Dateiobjekt. Kann in einem Skript umgeleitet werden, indem dem Namen ein Objekt zugewiesen wird, das `write`-Methoden besitzt (zum Beispiel `sys.stderr=wrappedsocket`). Wird für Interpreter-Prompts und Fehler benutzt.

`__stdin__`
`__stdout__`
`__stderr__`
: Originalwerte von `stdin`, `stderr` und `stdout` beim Programmstart (etwa zum Zurücksetzen als letzte Rettung; bei der Zuweisung an `sys.stdout` usw. speichern Sie üblicherweise den alten Wert und stellen ihn in der `finally`-Klausel wieder her). Kann unter Windows bei GUI-Anwendungen ohne Konsole `None` sein.

`tracebacklimit`
: Maximale Anzahl von Traceback-Stufen, die bei nicht abgefangenen Ausnahmen ausgegeben werden; Standardwert ist 1000, wenn kein anderer Wert zugewiesen wird.

`version`
: String mit der Versionsnummer des Python-Interpreters.

`version_info`
: Tupel mit den fünf Komponenten der Versionsangabe: Major, Minor, Micro, Release Level und Serial. Ist für Python 3.0.1 `(3, 0, 1, 'final', 0)` (siehe die Python Library Reference).

`winver`
: Versionsnummer zum Bilden von Registry-Schlüsseln auf Windows-Plattformen (siehe Python Library Reference).

Das Modul string

Das Modul `string` definiert Konstanten und Variablen zur Bearbeitung von String-Objekten. Lesen Sie auch den Abschnitt »Strings« auf Seite 17, eine Beschreibung der Template-basierten String-Substitution mithilfe von `Template` und `SafeTemplate`, die in diesem Modul definiert werden.

Modulfunktionen

Seit Python 2.0 sind die meisten Funktionen in diesem Modul auch als Methoden von String-Objekten verfügbar. Diese Methodenaufrufe werden allgemein bevorzugt und sind auch effizienter. Im Abschnitt »Strings« auf Seite 17 finden Sie weitere Details und eine Liste aller String-Methoden, die hier nicht wiederholt werden. In diesem Abschnitt werden nur jene Elemente aufgelistet, die lediglich im `string`-Modul vorkommen.

`capwords(s)`
: Teilt das Argument mithilfe von `split` in einzelne Wörter auf, schreibt mit `capitalize` den Anfangsbuchstaben groß und verbindet mit `join` die so entstandenen Wörter. Ersetzt mehrere Whitespace-Zeichen durch ein einziges Leerzeichen und entfernt Whitespace vorn und hinten.

`maketrans(von, nach)`
: Liefert eine für die Übergabe an `bytes.translate` geeignete Übersetzungstabelle, die alle Zeichen in `von` auf das Zeichen an der gleichen Position in `nach` übersetzt; `von` und `nach` müssen die gleiche Länge haben.

`Formatter`
: Eine Klasse, die die Erstellung benutzerdefinierter Formatierer ermöglicht. Dazu wird der gleiche Mechanismus genutzt wie von der Methode `str.format()`, die im Abschnitt »Strings« auf Seite 17 beschrieben wurde.

`Template`
: Klasse für String-Template-Substitutionen (siehe den Abschnitt »Strings« auf Seite 17).

Konstanten

`ascii_letters`
: Der String `ascii_lowercase + ascii_uppercase`.

`ascii_lowercase`
: Der String `'abcdefghijklmnopqrstuvwxyz'`, unabhängig von den lokalen Landeseinstellungen, bleibt auch in Zukunft gleich.

`ascii_uppercase`
: Der String `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`, unabhängig von den lokalen Landeseinstellungen, bleibt auch in Zukunft gleich.

`digits`
: Der String `'0123456789'`.

`hexdigits`
: Der String `'0123456789abcdefABCDEF'`.

`octdigits`
: Der String `'01234567'`.

`printable`
: Kombination von `digits`, `ascii_letters`, `punctuation` und `whitespace`.

`punctuation`
: Die Menge der Zeichen, die im aktuellen Locale als Interpunktionszeichen betrachtet werden.

`whitespace`
: Ein String, der Leerzeichen, Tabulator, Zeilenvorschub, Wagenrücklauf und den vertikalen Tabulator enthält: `' \t\n\r\v\f'`.

Das Systemmodul os

Das Modul `os` ist die primäre Schnittstelle zum Betriebssystem (Operating System, OS). Es bietet allgemeine OS-Unterstützung und eine standardisierte, plattformunabhängige Schnittstelle. Es enthält Werkzeuge für Umgebungen, Prozesse, Dateien, Shell-Befehle und vieles mehr. Es bietet außerdem das Untermodul `os.path` mit einer portablen Schnittstelle zur Bearbeitung von Verzeichnispfaden.

Skripten, die zur Systemprogrammierung os und os.path nutzen, sind im Allgemeinen zwischen den meisten Python-Plattformen portabel. Allerdings sind manche Exporte von os nicht auf allen Plattformen implementiert (z.B. gibt es fork unter Unix und Cygwin, aber nicht in der Standard-Python-Version für Windows). Da sich die Portabilität derartiger Aufrufe mit der Zeit ändern kann, sollten Sie die Details der Python Library Reference entnehmen.

Siehe auch verwandte Systemmodule: glob (Dateinamenserweiterung), tempfile (temporäre Dateien), signal (Signal-Handling), socket (Netzwerk und IPC), threading (Multi-Threading), queue (Thread-Kommunikation), subprocess (zur Steuerung gestartet Prozess), multiprocessing (threading-ähnliche API für Prozesse), getopt sowie optparse (Kommandozeilenverarbeitung) und weitere.

Administrationswerkzeuge

Im Folgenden finden Sie unterschiedliche modulbezogene Exporte:

error
: Ein Alias für die eingebaute Ausnahme OSError. Wird ausgelöst bei Fehlern, die das os-Modul betreffen. Der mitgelieferte Wert ist ein Tupel mit der Fehlernummer aus errno und dem korrespondierenden String, den die C-Funktion perror() ausgeben würde. Siehe das Modul errno in der Python Library Reference für die Namen der Fehlercodes, die das zugrunde liegende Betriebssystem definiert.

  Wenn Ausnahmen Klassen sind, hat diese Ausnahme zwei Attribute: errno, der Wert der C-Variablen errno, und strerror, die entsprechende Meldung aus strerror(). Bei Ausnahmen, die Dateipfadnamen betreffen (wie chdir(), unlink()), hat die Ausnahmeinstanz zusätzlich das Attribut filename, das den übergebenen Dateinamen festhält.

name
: Name des plattformspezifischen Moduls, der in die Hauptebene von os kopiert wurde (zum Beispiel posix, nt, dos, mac, os2, ce oder java). Siehe auch platform im Abschnitt »Das Modul sys« auf Seite 138.

`path`
: Eingebettetes Modul mit portablen Werkzeugen für Pfadnamen. `os.path.split` ist z.B. ein plattformunabhängiges Werkzeug für Verzeichnisnamen, das intern einen geeigneten plattformspezifischen Aufruf benutzt.

Portabilitätskonstanten

Dieser Abschnitt beschreibt Werkzeuge zum portablen Parsen und Konstruieren von Verzeichnis- und Suchpfaden. Sie werden automatisch auf geeignete Werte für die Plattform gesetzt, auf der das Skript läuft.

`curdir`
: String, der für das aktuelle Verzeichnis steht (zum Beispiel `.` für Windows und POSIX).

`pardir`
: String, der für das übergeordnete Verzeichnis steht (zum Beispiel `..` für POSIX).

`sep`
: String mit dem Trennzeichen für Verzeichnisse (zum Beispiel `/` für POSIX, `\` für Windows).

`altsep`
: Alternativer Trennzeichenstring oder `None` (zum Beispiel `/` für Windows).

`extsep`
: Das Trennzeichen zwischen dem Dateinamen und der Dateierweiterung (zum Beispiel `.`).

`pathsep`
: Zeichen zum Trennen von Suchpfadkomponenten, wie etwa in den Shell-Variablen `PATH` und `PYTHONPATH` (zum Beispiel `;` für Windows, `:` für POSIX).

`defpath`
: Vorgegebener Suchpfad, den `os.exec*p*` benutzt, wenn die Shell keine Einstellung für `PATH` hat.

`linesep`
: Der String, der auf der aktuellen Plattform ein Zeilenende anzeigt (zum Beispiel `\n` für POSIX und `\r\n` für Windows). Sollte nicht verwendet werden, wenn im Textmodus Zeilen in Dateien geschrieben werden – nutzen Sie stattdessen die automatische Übersetzung von `'\n'`.

Shell-Befehle

Folgende Funktionen führen Programme im zugrunde liegenden Betriebssystem aus. In Python 2.x bietet dieses Modul die Aufrufe `os.popen2/3/4`, die in Python 3.0 durch `subprocess.Popen` ersetzt wurden.

`system(befehl)`
: Führt den als String angegebenen Befehl in einem Subshell-Prozess aus. Liefert den Rückgabecode des erzeugten Prozesses zurück. Nutzt, anders als `popen`, keine Pipes, um eine Verbindung mit den Standard-Streams des gestarteten Prozesses herzustellen. Tipp: Hängen Sie unter Unix ein `&` ans Ende von `befehl` an, um den Befehl im Hintergrund ausführen zu lassen (z.B. `os.system('python main.py &')`), oder nutzen Sie unter Windows den DOS-Befehl `start` (etwa `os.system('start file.html')`).

`startfile(dateipfadname)`
: Führt eine Datei mit der mit ihrem Dateityp verknüpften Anwendung aus. Entspricht einem Doppelklicks auf die Datei im Windows Explorer oder der Angabe der Datei als Argument für den DOS-Befehl `start` (z.B. mit `os.system ('start` *`pfad`*`')`). Die Datei wird in der mit ihrer Dateierweiterung verknüpften Anwendung geöffnet. Der Aufruf wartet nicht das Ende ab und macht normalerweise kein eigenes DOS-Fenster dafür auf. Neu in Version 2.0 und nur unter Windows.

`popen(befehl, mode='r', buffering=None)`
: Öffnet eine Pipe zum oder vom als String angegebenen Shell-Befehl `befehl`, um Daten zu senden oder zu empfangen. Liefert ein geöffnetes Dateiobjekt, das genutzt werden kann, um die

Standardausgabe von `befehl` zu lesen (`modus 'r'`) oder in seine Standardeingabe zu schreiben (`modus 'w'`). Zum Beispiel empfängt `dirlist = os.popen('ls -l *.py').read()` die Ausgabe des Unix-Befehls `ls`.

`befehl` ist ein beliebiger Befehlsstring, der in der Konsole oder Shell-Eingabeaufforderung des Systems eingegeben werden kann. `modus` kann `'r'` oder `'w'` sein, Standard ist `'r'`. `buffering` hat die gleiche Bedeutung wie bei der eingebauten Funktion `open`. `befehl` läuft unabhängig, sein Beendigungsstatus wird von der `close`-Methode des resultierenden Dateiobjekts zurückgeliefert. Ist dieser 0 (kein Fehler), wird `None` geliefert. Nutzen Sie `readline()` oder Iteration, um die Ausgabe zeilenweise zu lesen.

Python 2.x bietet zusätzlich die Varianten `popen2`, `popen3` und `popen4`, um eine Verbindung zu anderen Streams des gestarteten Befehls herzustellen (z.B. liefert `popen2` ein Tupel (`kind_stdin, kind_stdout`)). In Python 3.0 gibt es diese Aufrufe nicht mehr; nutzen Sie stattdessen `subprocess.Popen()`. Das `subprocess`-Modul von Version 2.4 und höher gestattet Skripten, neue Prozesse zu starten, ihre Eingabe-/Ausgabe-/Fehler-Pipes zu verbinden und Rückgabecodes zu erhalten. Siehe die Python Library Reference.

`spawn*(args...)`
: Eine Gruppe von Funktionen zum Starten von Programmen und Befehlen. Weitere Informationen liefern der Abschnitt »Prozesskontrolle« auf Seite 157 und die Python Library Reference. Das Modul `subprocess` ist die allgemein bevorzugte Alternative zu diesen Aufrufen.

Umgebungswerkzeuge

Diese Attribute exportieren die Ausführungsumgebung und den Kontext:

`environ`
: Das Dictionary-artige Objekt mit den Umgebungsvariablen. `os.environ['USER']` ist der Wert der Variablen `USER` in der Shell

(entspricht `$USER` unter Unix und `%USER%` unter DOS). Wird beim Programmstart initialisiert. Änderungen an `os.environ` durch Schlüsselzuweisungen werden über einen Aufruf von `putenv` (C) exportiert und von allen Prozessen geerbt, die von diesem Prozess gestartet werden, sowie von eventuell eingebundenem C-Code.

`putenv(varname, wert)`
: Setzt die Shell-Umgebungsvariable `varname` auf den String `wert`. Wirkt sich auf mit `system`, `popen`, `spawnv`, `fork` oder `execv` gestartete Subprozesse aus. Zuweisungen zu `os.environ_` Schlüsseln rufen automatisch `putenv` auf, aber `putenv`-Aufrufe aktualisieren `environ` nicht.

`getcwd()`
: Liefert das aktuelle Arbeitsverzeichnis als String.

`chdir(pfad)`
: Ändert das aktuelle Arbeitsverzeichnis für diesen Prozess in `pfad`, einem String mit einem Verzeichnisnamen. Nachfolgende Dateioperationen sind relativ zum neuen aktuellen Arbeitsverzeichnis.

`strerror(code)`
: Liefert die zu `code` korrespondierende Fehlermeldung.

`times()`
: Erzeugt ein Tupel mit fünf Elementen, die Informationen zur verbrauchte CPU-Zeit in Sekunden (Fließkomma) liefern: (*benutzer-zeit, system-zeit, kind-benutzer-zeit, kind-system-zeit, vergangene-echt-zeit*). Siehe auch den Abschnitt »Das Modul time« auf Seite 183.

`umask(mask)`
: Setzt die numerische `umask` auf `mask` und liefert den vorherigen Wert zurück.

`uname()`
: Liefert ein Tupel mit Strings, die das Betriebssystem beschreiben: (*systemname, knotenname, release, version, maschine*).

Dateideskriptorwerkzeuge

Die folgenden Funktionen bearbeiten Dateien über ihren Deskriptor (`dateideskriptor` ist ein ganzzahliger Wert, der eine Datei identifiziert). Auf Dateideskriptoren des `os`-Moduls basierende Dateien sind für elementarere Aufgaben gedacht und nicht das Gleiche wie die Dateiobjekte, die die eingebaute Funktion `open()` liefert (obgleich `os.fdopen` und die Dateibjektmethode `fileno` zwischen beiden konvertieren können). Bei der Arbeit mit Dateien sollten in der Regel Dateiobjekte und keine Deskriptoren genutzt werden.

`close(dateideskriptor)`
: Schließt den Dateideskriptor (kein Dateiobjekt).

`dup(dateideskriptor)`
: Erzeugt ein Duplikat des Dateideskriptors.

`dup2(dateideskriptor, dateideskriptor2)`
: Kopiert den Dateideskriptor `dateideskriptor` auf `dateideskriptor2` (der gegebenenfalls vorher geschlossen wird).

`fdopen(dateideskriptor [, modus [, puffergroesse]])`
: Liefert ein eingebautes Dateiobjekt, das mit dem Dateideskriptor (Integer) verbunden ist. `modus` und `puffergroesse` haben die gleiche Bedeutung wie bei der eingebauten Funktion `open()` (siehe den Abschnitt »Eingebaute Funktionen« auf Seite 105). Dient der Konvertierung von deskriptorbasierten Dateien in Dateiobjekte, die üblicherweise mit der eingebauten Funktion `open()` erzeugt werden. Tipp: Nutzen Sie `fileobj.fileno`, um ein Dateiobjekt in einen Deskriptor umzuwandeln.

`fstat(dateideskriptor)`
: Liefert den Dateistatus für den Dateideskriptor (wie `stat`).

`ftruncate(dateideskriptor, laenge)`
: Schneidet die dem Dateideskriptor zugeordnete Datei auf maximal `laenge` Bytes ab.

`isatty(dateideskriptor)`
: Liefert 1, wenn der Dateideskriptor geöffnet und mit einem tty-Gerät (oder einem tty-ähnlichen Gerät) verbunden ist.

`lseek(dateideskriptor, position, rel)`
: Setzt die aktuelle Position des Dateideskriptors (bei wahlfreiem Zugriff). `rel` gibt die relative Bedeutung von `position` an: 0 – ab Dateianfang, 1 – ab der aktuellen Position, 2 – ab Dateiende.

`open(dateiname, flags [, modus])`
: Öffnet eine deskriptorbasierte Datei und liefert den Dateideskriptor zurück (ein Integer und kein `stdio`-Dateiobjekt). Nur für elementarere Dateiaufgaben gedacht und nicht dasselbe wie die eingebaute Funktion `open()`. Der Standardwert für `modus` ist 0777 (oktal), und der aktuelle Wert von `umask` wird zuvor ausmaskiert. `flags` ist eine Bitmaske: Benutzen Sie `|`, um plattformneutrale und die im `os`-Modul definierten plattformspezifischen `flags`-Konstanten zu kombinieren (siehe Tabelle 18).

`pipe()`
: Siehe den Abschnitt »Prozesskontrolle« auf Seite 157.

`read(dateideskriptor, n)`
: Liest maximal `n` Bytes aus dem Dateideskriptor und liefert diese als String.

`write(dateideskriptor, string)`
: Schreibt alle Bytes im String in den Dateideskriptor.

Tabelle 18: Auszug der über ODER kombinierbaren Schalter für os.open

O_APPEND	O_EXCL	O_RDONLY	O_TRUNC
O_BINARY	O_NDELAY	O_RDWR	O_WRONLY
O_CREAT	O_NOCTTY	O_RSYNC	
O_DSYNC	O_NONBLOCK	O_SYNC	

Werkzeuge für Dateipfadnamen

Die folgenden Funktionen verarbeiten Dateien anhand ihres Pfadnamens (`pfad` ist der Pfadname einer Datei in String-Form). Lesen Sie auch den Abschnitt »Das Modul os.path« auf Seite 159. In Python 2.x enthält dieses Modul außerdem Werkzeuge für tempo-

räre Dateien, die in Python 3.0 durch das Modul `tempfile` ersetzt werden.

`chdir(path)`,

`getcwd()`
: Siehe den Abschnitt »Umgebungswerkzeuge« auf Seite 150.

`chmod(pfad, modus)`
: Ändert den Zugriffsmodus der Datei `pfad` in die Zahl `modus`.

`chown(pfad, uid, gid)`
: Ändert Eigentümer-/Gruppen-IDs von `pfad` in die Zahl `uid/gid`.

`link(quelle, ziel)`
: Erzeugt einen harten Link.

`listdir(pfad)`
: Liefert eine Liste mit den Namen aller Einträge im Verzeichnis `pfad`. Eine schnelle und portable Alternative zum Modul `glob` oder zum Auswerten von Verzeichnisausgaben von `os.popen()`. In Bezug auf Dateinamenserweiterung siehe auch das Modul `glob`.

`lstat(pfad)`
: Wie `stat`, folgt aber symbolischen Links nicht.

`mkfifo(pfad [, modus])`
: Erzeugt eine benannte *FIFO*-Pipe `pfad` mit den Zugriffsberechtigungen, die durch den numerischen Zugriffsmodus `modus` angegeben werden, ohne sie zu öffnen. Der Standardwert für `modus` ist oktal 0666. Der aktuelle Wert von `umask` wird zuvor aus `modus` ausmaskiert. FIFOs sind Pipes, die dateisystembasiert sind und wie normale Dateien geöffnet und verarbeitet werden können. Sie existieren, bis sie gelöscht werden.

`mkdir(pfad [, modus])`
: Erzeugt das Verzeichnis `pfad` mit dem Zugriffsmodus `modus`. Der Standardwert für Modus ist oktal 0777.

`makedirs(pfad [, modus])`
: Funktion zum rekursiven Erstellen von Verzeichnissen. Wie `mkdir`, erzeugt aber auch alle erforderlichen Zwischenverzeichnisse in `pfad`, die nicht existieren. Löst eine Ausnahme aus,

wenn das letzte Zielverzeichnis bereits existiert oder nicht erzeugt werden kann. Der Standardwert für `modus` ist oktal 0777.

`readlink(pfad)`
: Liefert den durch den symbolischen Link `pfad` referenzierten Pfad.

`remove(pfad)`
`unlink(pfad)`
: Löscht die Datei `pfad`. `remove` ist identisch mit `unlink`. Siehe `rmdir` und `removedirs` in dieser Liste zum Löschen von Verzeichnissen.

`removedirs(pfad)`
: Funktion zum rekursiven Löschen von Dateien. Ähnlich wie `rmdir`, nur dass nach erfolgreicher Löschung des untersten Verzeichnisses versucht wird, höher liegende Pfadsegmente zu löschen, bis der ganze Pfad verarbeitet ist oder ein Fehler auftritt. Löst eine Ausnahme aus, wenn das unterste Verzeichnis nicht entfernt werden konnte.

`rename(pfad, zielpfad)`
: Benennt die Datei `pfad` in `zielpfad` um bzw. verschiebt sie.

`renames(alterpfad, neuerpfad)`
: Funktion zum rekursiven Umbenennen von Verzeichnissen oder Dateien. Wie `rename`, verursacht aber zuerst, alle Verzeichnisse zu erzeugen, die der neue Pfadname verlangt. Nach der Umbenennung wird versucht, mithilfe von `removedirs` die Pfadsegmente des alten Pfadnamens zu bereinigen.

`rmdir(pfad)`
: Löscht das Verzeichnis `pfad`.

`stat(pfad)`
: Führt einen `stat`-Systemaufruf für `pfad` aus. Zurückgegeben wird ein Tupel aus Integern mit systemnahen Dateiinformationen. (Das Modul `stat` definiert die Elemente und bietet Tools für ihre Bearbeitung.)

`symlink(quelle, ziel)`
: Erzeugt einen symbolischen Link.

`utime(pfad, (zugriff, modifikation))`

Setzt das Datum von Dateizugriff und -modifikation.

`walk(top [, topdown=True [, onerror=None] [, follow-links=False]])`

Erzeugt die Dateinamen in einem Verzeichnisbaum, indem der Baum von oben nach unten bzw. von unten nach oben durchlaufen wird. Gibt für jedes Verzeichnis unterhalb des Verzeichnisses `top` (inklusive `top` selbst) ein Tripel (`dirpfad, dirnamen, dateinamen`) zurück. `dirpfad` ist ein String mit dem Pfad zum Verzeichnis, `dirnamen` eine Namensliste mit den Unterverzeichnissen in `dirpfad` (exklusive `.` und `..`) und `dateinamen` eine Namensliste der Dateien in `dirpfad`, die keine Verzeichnisse sind. Beachten Sie, dass die Namen in den Listen keine Pfadkomponenten enthalten. Den vollständigen Pfad (von `top`) zu einer Datei oder einem Verzeichnis in `dirpfad` erhalten Sie mit `os.path.join(dirpfad, name)`.

Falls das optionale Argument `topdown` wahr ist oder fehlt, wird das Tripel für ein Verzeichnis vor jenen für dessen Unterverzeichnisse erzeugt (Verzeichnisse werden von oben nach unten erzeugt). Ist `topdown` falsch ist, wird das Tripel zu einem Verzeichnis nach denen für die Unterverzeichnisse erzeugt (Verzeichnisse werden von unten nach oben erzeugt). Wenn das optionale `onerror` angegeben wird, sollte es eine Funktion sein, die mit einem Argument, einer Instanz von `os.error`, aufgerufen wird. Symlinks, die zu Verzeichnissen aufgelöst werden, wird standardmäßig nicht gefolgt; setzen Sie `followlinks` auf unterstützenden Systemen auf `True`, damit Verzeichnisse erreicht werden, auf die Symlinks zeigen.

Python 2.x bietet zusätzlich einen `os.path.walk()`-Aufruf mit einer ähnlichen Funktionalität zum Durchlaufen von Verzeichnisbäumen, die eine Event-Handler-Callback-Funktion statt eines Generators nutzt. In Python 3.0 wurde `os.path.walk()` aufgrund seiner Redundanz entfernt; nutzen Sie stattdessen `os.walk()`. Zu Dateinamenserweiterung siehe das Modul `glob`.

Prozesskontrolle

Die folgenden Funktionen erzeugen und verwalten Prozesse und Programme. Siehe auch den Abschnitt »Shell-Befehle« auf Seite 149 für alternative Wege, um Programme und Dateien zu starten.

`abort()`
: Sendet ein `SIGABRT`-Signal an den laufenden Prozess. Unter Unix wird standardmäßig ein Coredump erzeugt; unter Windows liefert der Prozess sofort Exitcode 3 zurück.

`execl(pfad, arg0, arg1,...)`
: Äquivalent zu `execv(pfad, (arg0, arg1,...))`.

`execle(pfad, arg0, arg1,..., env)`
: Äquivalent zu `execve(pfad, (arg0, arg1,...), env)`.

`execlp(pfad, arg0, arg1,...)`
: Äquivalent zu `execvp(pfad, (arg0, arg1,...))`.

`execve(pfad, args, env)`
: Wie `execv`, aber das Dictionary `env` ersetzt die Shell-Umgebung. `env` muss Strings auf Strings abbilden.

`execvp(pfad, args)`
: Wie `execv(pfad, args)`, bildet aber die Suchaktionen der Shell nach einer ausführbaren Datei in einer Verzeichnisliste nach. Die Verzeichnisliste wird aus `os.environ['PATH']` gebildet.

`execvpe(pfad, args, env)`
: Eine Kreuzung aus `execve` und `execvp`. Die Verzeichnisliste wird aus `os.environ['PATH']` gebildet.

`execv(pfad, args)`
: Startet die ausführbare Datei `pfad` mit dem Kommandozeilenparameter `args`, wobei das aktuelle Programm in diesem Prozess ersetzt wird (der Python-Interpreter). `args` kann ein Tupel oder eine Liste mit Strings sein. Das erste beginnt per Konvention mit dem Namen der ausführbaren Datei (`argv[0]`). Dieser Funktionsaufruf kehrt niemals zurück, es sei denn, während des Starts des neuen Programms tritt ein Fehler auf.

`_exit(n)`
: Beendet den Prozess unmittelbar mit Status `n`, ohne aufzuräumen. Wird normalerweise nur in Kindprozessen nach einem `fork` benutzt; der normale Weg zum Beenden ist ein Aufruf von `sys.exit(n)`.

`fork()`
: Spaltet einen Kindprozess ab (eine virtuelle Kopie des aufrufenden Prozesses, der parallel läuft). Der Kindprozess erhält 0 als Rückgabewert, der Elternprozess hingegen bekommt die Prozess-ID des Kindes zurück. In der Standard-Windows-Version von Python nicht verfügbar, aber in Cygwin Python.

`getpid()`
`getppid()`
: Liefert die Prozess-ID des laufenden Prozesses; `getppid()` liefert die Prozess-ID des Elternprozesses.

`getuid()`
`geteuid()`
: Liefert die Benutzer-ID des Prozesses; `geteuid` liefert die effektive Benutzer-ID.

`kill(pid, sig)`
: Bricht den Prozess mit der ID `pid` durch Senden von Signal `sig` ab. Zur Registrierung von Signal-Handlern siehe das Modul `signal`.

`mkfifo(path [, mode])`
: Siehe den Abschnitt »Werkzeuge für Dateipfadnamen« auf Seite 153 (Dateien, die zur Prozesssynchronisation genutzt werden).

`nice(increment)`
: Erhöht den `nice`-Wert des Prozesses (d.h., verringert seine CPU-Priorität).

`pipe()`
: Liefert ein Dateideskriptor-Tupel (`rfd`, `wfd`) zum Lesen und Schreiben einer neuen, anonymen (unbenannten) Pipe. Für Interprozesskommunikation.

`plock(op)`
: Sperrt Programmsegmente im Speicher. `op` (definiert in `<sys/lock.h>`) bestimmt, welche Segmente gesperrt werden.

`spawnv(modus, pfad, args)`
: Führt Programm `pfad` als neuen Prozess aus, wobei die in `args` angegebenen Parameter als Kommandozeile übergeben werden. `args` kann eine Liste oder ein Tupel sein. `modus` ist eine magische operationale Konstante mit den Werten `P_WAIT`, `P_NOWAIT`, `P_NOWAITO`, `P_OVERLAY` oder `P_DETACH`. Unter Windows in etwa äquivalent zu einer Kombination aus `fork` und `execv` (`fork` gibt es noch nicht in der Standard-Windows-Version, aber `popen` und `system`). Zu mächtigeren Alternativen siehe das Modul `subprocess`.

`spawnve(modus, pfad, args, env)`
: Wie `spawnv`, übergibt aber den Inhalt der Abbildung `env` als Shell-Umgebung für das gestartete Programm.

`wait()`
: Wartet auf die Beendigung eines Kindprozesses. Liefert ein Tupel mit der ID des Kindes und seinem Beendigungsstatus.

`waitpid(pid, optionen)`
: Wartet auf die Beendigung eines Kindprozesses mit der ID `pid`. `optionen` ist im Normalfall 0 oder `os.WNOHANG`, um ein Aufhängen zu vermeiden, wenn kein Kindstatus verfügbar ist. Wenn `pid` 0 ist, bezieht sich die Anfrage auf jedes Kind in der Prozessgruppe des aktuellen Prozesses. Vergleichen Sie dazu auch die in der Python Library Reference dokumentierten Funktionen zum Prüfen des Prozessbeendigungsstatus (zum Beispiel ruft `WEXITSTATUS(`*`status`*`)` den Beendigungsstatus ab).

Das Modul os.path

Dieses Modul bietet zusätzliche Dienste für Verzeichnispfadnamen und Portabilitätswerkzeuge. Es handelt sich um ein eingebettetes Modul: Seine Namen sind im Untermodul `os.path` in das Modul `os` eingebettet (auf die Funktion `exists` greift man beispielsweise zu, indem man `os` importiert und dann `os.path.exists` benutzt).

Die meisten Funktionen in diesem Modul akzeptieren ein Argument `pfad`, den Pfadnamenstring einer Datei (zum Beispiel `"C:\\dir1\\spam.txt"`). Verzeichnispfade werden generell gemäß den Konventionen der Plattform angegeben; ist kein Verzeichnispräfix angegeben, sind Pfade relativ zum aktuellen Arbeitsverzeichnis. Hinweis: Vorwärts-Slashes funktionieren normalerweise auf allen Plattformen als Verzeichnistrenner. In Python 2.x schließt dieses Modul ein `os.path.walk`-Werkzeug ein, das in Python 3.0 durch `os.walk` ersetzt wird.

`abspath(pfad)`
: Liefert eine normalisierte absolute Version von `pfad`. Auf den meisten Plattformen ist das äquivalent zu `normpath(join(os.getcwd(), pfad))`.

`basename(pfad)`
: das Gleiche wie die zweite Hälfte des von `split(pfad)` erzeugten Paars.

`commonprefix(liste)`
: Liefert das längste gemeinsame Pfadpräfix (Zeichen für Zeichen), das allen Pfaden in `liste` gemeinsam ist.

`dirname(pfad)`
: Das Gleiche wie die erste Hälfte des von `split(pfad)` erzeugten Paars.

`exists(pfad)`
: Wahr, wenn der String `pfad` als Dateipfadname existiert.

`expanduser(pfad)`
: Expandiert in `pfad` enthaltene ~-Benutzernamenelemente.

`expandvars(pfad)`
: Expandiert mit `$` in `pfad` eingebettete Umgebungsvariablen.

`getatime(pfad)`
: Liefert den Zeitpunkt des letzten Zugriffs auf `pfad` (Sekunden seit der Epoche).

`getmtime(pfad)`
: Liefert den Zeitpunkt der letzten Modifikation von `pfad` (Sekunden seit der Epoche).

`getsize(pfad)`
: Liefert die Größe der Datei `pfad` in Bytes.

`isabs(pfad)`
: Wahr, wenn der String `pfad` ein absoluter Pfad ist.

`isfile(pfad)`
: Wahr, wenn der String `pfad` eine reguläre Datei ist.

`isdir(pfad)`
: Wahr, wenn der String `pfad` ein Verzeichnis ist.

`islink(pfad)`
: Wahr, wenn der String `pfad` ein symbolischer Link ist.

`ismount(pfad)`
: Wahr, wenn der String `pfad` ein Einhängepunkt (Mount Point) ist.

`join(pfad1 [, pfad2 [, ...]])`
: Verbindet eine oder mehrere Pfadkomponenten auf intelligente Weise (unter Verwendung plattformabhängiger Konventionen für die Trennzeichen zwischen den Pfadkomponenten).

`normcase(pfad)`
: Normalisiert die Schreibweise eines Pfadnamens. Ohne Auswirkung unter Unix; bei von der Groß-/Kleinschreibung unabhängigen Dateisystemen wird in Kleinschreibung umgewandelt; unter Windows wird außerdem / in \ geändert.

`normpath(pfad)`
: Normalisiert einen Pfadnamen. Kollabiert überflüssige Trennzeichen und Aufwärtsverweise; unter Windows wird / in \ geändert.

`realpath(pfad)`
: Gibt den kanonischen Pfad der angegebenen Datei zurück, wobei alle eventuell vorhandenen symbolischen Links im Pfad aufgelöst werden.

`samefile(pfad1, pfad2)`
: Wahr, wenn beide Pfadnamenargumente auf die gleiche Datei bzw. das gleiche Verzeichnis verweisen.

`sameopenfile(fp1, fp2)`
: Wahr, wenn beide Dateiobjekte auf die gleiche Datei verweisen.

`samestat(stat1, stat2)`
: Wahr, wenn beide `stat`-Tupel auf die gleiche Datei verweisen.

`split(pfad)`
: Teilt `pfad` in (`head, tail`), wobei `tail` die letzte Pfadnamenkomponente ist und `head` alles bis hin zu `tail`. Entspricht dem Tupel (`dirname(pfad), basename(pfad)`).

`splitdrive(pfad)`
: Teilt `pfad` in ein Paar (`'drive:', rest`) (unter Windows).

`splitext(pfad)`
: Teilt `pfad` in (`root, ext`), wobei die letzte Komponente von `root` keinen `.` enthält und `ext` leer ist oder mit `.` beginnt.

`walk(pfad, visitor, daten)`
: In Python 2.x eine Alternative zu `os.walk`. In Python 3.0 entfernt: Nutzen Sie `os.walk`, nicht `os.path.walk`.

Das Mustervergleichsmodul re

Das Modul `re` ist die Standardschnittstelle für Mustervergleiche. Reguläre Ausdrücke (Regular Expressions, RE) werden als Strings angegeben. Dieses Modul muss importiert werden.

Modulfunktionen

`compile(pattern [, flags])`
: Übersetzt einen RE-String `pattern` in ein RE-Objekt für einen späteren Mustervergleich. Die auf der obersten Ebene des `re`-Moduls verfügbaren `flags` (kombinierbar durch den bitweisen Operator |) sind:

`A` *oder* `ASCII` *oder* `(?a)`
: Bewirkt, dass `\w`, `\W`, `\b`, `\B`, `\s` und `\S` nur ASCII-Vergleiche, keine vollständigen Unicode-Vergleiche durchführen. Ist nur bei Unicode-Mustern relevant und wird bei Bytemustern ignoriert. Beachten Sie, dass im Dienste der Rückwärtskompatibili-

tät `re.U` in Python 3.0 weiterhin vorhanden ist (ebenso das Synonym `re.UNICODE` sowie das eingebettete Gegenstück `?u`), aber redundant ist, da Vergleiche bei Strings standardmäßig in Unicode erfolgen (und Unicode-Vergleiche bei Bytes nicht zulässig sind).

`I` *oder* `IGNORECASE` *oder* `(?i)`
: Muster ignoriert Groß-/Kleinschreibung.

`L` *oder* `LOCALE` *oder* `(?L)`
: Macht `\w`, `\W`, `\b`, `\B`, `\s`, `\S`, `\d` und `\D` abhängig von der aktuellen Locale-Einstellung (Standard in Python 3.0 ist Unicode).

`M` *oder* `MULTILINE` *oder* `(?m)`
: Sucht in einzelnen Zeilen, nicht im gesamten String.

`S` *oder* `DOTALL` *oder* `(?s)`
: `.` erkennt *alle* Zeichen, einschließlich Newline.

`U` *oder* `UNICODE` *oder* `(?u)`
: Macht `\w`, `\W`, `\b`, `\B`, `\s`, `\S`, `\d` und `\D` abhängig von Unicode-Zeicheneigenschaften (neu in Version 2.0 und in Python 3.x überflüssig).

`X` *oder* `VERBOSE` *oder* `(?x)`
: Ignoriert Whitespace im Muster außerhalb von Zeichenklassen.

`match(pattern, string [, flags])`
: Wenn 0 oder mehr Zeichen am Anfang von `string` auf das String-Muster `pattern` passen, wird eine entsprechende `MatchObject`-Instanz zurückgeliefert. Gibt es keinen Treffer, wird `None` geliefert. `flags` wie in `compile`.

`search(pattern, string [, flags])`
: Sucht `string` nach einer Stelle ab, wo er auf das Muster `pattern` passt. Liefert die entsprechende `MatchObject`-Instanz oder `None`, wenn kein Vorkommen gefunden wird. `flags` wie in `compile`.

`split(pattern, string [, maxsplit=0])`
: Zerteilt `string` beim Vorkommen von `pattern`. Wenn in `pattern` einfangende `()` vorkommen, werden die aufgetretenen

Muster oder Teilmuster ebenfalls zurückgegeben. Ist `maxsplit` angegeben und > 0, wird nach `maxsplit` Treffern nicht weiter geteilt.

`sub(pattern, ersetzung, string [, count=0])`
: Liefert einen String, in dem die (`count` ersten) am weitesten links stehenden, nicht-überlappenden Vorkommen von `pattern` (ein String oder RE-Objekt) in `string` durch `ersetzung` ersetzt sind. `ersetzung` kann ein String sein oder eine Funktion, die ein einzelnes `MatchObject`-Argument akzeptiert und den Ersetzungsstring zurückgibt. `ersetzung` kann Ersetzungs-Escapes der Form `\1`, `\2` usw. enthalten, die sich auf Substrings aus erkannten Gruppen beziehen, oder `\0` für alles.

`subn(pattern, ersetzung, string [, count=0])`
: Wie `sub`, liefert aber ein Tupel: `(neuer-string, anzahl-der-ersetzungen)`.

`findall(pattern, string [, flags])`
: Liefert eine Liste von Strings mit allen nicht-überlappenden Übereinstimmungen von `pattern` in `string`. Wenn in `pattern` eine oder mehrere Gruppen vorkommen, wird eine Liste von Gruppen zurückgegeben.

`finditer(muster, string [, flags])`
: Gibt einen Iterator über alle nicht-überlappenden Vorkommen von Treffern des regulären Ausdrucks `muster` in `string` zurück.

`escape(string)`
: Maskiert in `string` alle nicht-alphanumerischen Zeichen mit einem Backslash, so dass er als String-Literal übersetzt werden kann.

Reguläre Ausdrucksobjekte

RE-Objekte werden von der Funktion `re.compile` zurückgegeben und haben folgende Attribute:

`flags`
: Das beim Übersetzen angegebene `flags`-Argument.

`groupindex`
: Dictionary aus {`group-name: group-number`} im Muster.

`pattern`
: Das beim Übersetzen angegebene Muster.

`match(string [, pos [, endpos]])`
`search(string [, pos [, endpos]])`
`split(string [, maxsplit=0])`
`sub(ersetzung, string [, count=0])`
`subn(ersetzung, string [, count=0])`
`findall(string [, pos[, endpos]])`
`finditer(string [, pos[, endpos]])`
: Wie die zuvor gezeigten Funktionen des Moduls `re`, aber `pattern` ist implizit, und `pos` und `endpos` kennzeichnen Anfangs- und Ende-Indizes für den untersuchten Substring.

Match-Objekte

Match-Objekte werden von erfolgreichen `match`- und `search`-Operationen zurückgeliefert und haben die folgenden Attribute (siehe die Python Library Reference für weitere Attribute, die hier nicht aufgeführt sind):

`pos, endpos`
: Bei `search()` oder `match()` angegebene Werte von `pos` und `endpos`.

`re`
: Das RE-Objekt, dessen `match()`- oder `search()`-Ergebnis das Objekt ist.

`string`
: An `match()` oder `search()` übergebener String.

`group([g1, g2,...])`
: Liefert Substrings, die durch einfangende Gruppen im Muster erkannt wurden. Akzeptiert 0 oder mehr Gruppennummern. Wird nur ein Argument angegeben, ist das Ergebnis der Substring, der von der durch die Nummer angegebenen Gruppe gefunden wurde. Bei mehreren Argumenten ist das Ergebnis ein Tupel mit einem erkannten Substring je Argument. Ganz ohne Argumente wird der komplette erkannte Substring zurückgeliefert. Ist die Gruppennummer 0, wird der gesamte

passende String geliefert, ansonsten der von der Gruppe mit dieser Nummer passende Substring (1...N, von links nach rechts). Die Argumente für die Gruppennummern können auch Gruppennamen sein.

`groups()`
: Liefert ein Tupel mit allen Gruppen des Vergleichs. Gruppen, die nicht erkannt wurden, haben den Wert `None`.

`groupdict()`
: Liefert ein Dictionary mit allen benannten Untergruppen des Vergleichs mit ihren Untergruppennamen als Schlüssel.

`start([gruppe]), end([gruppe])`
: Die Indizes des Anfangs bzw. Endes des durch `gruppe` erkannten Substrings (oder des gesamten erkannten Strings, wenn `gruppe` fehlt). Für ein Match-Objekt `M` gilt `M.string[M.start(g):M.end(g)]==M.group(g)`.

`span([gruppe])`
: Liefert das Tupel `(start(gruppe), end(gruppe))`.

`expand(template)`
: Erzeugt den String, der sich durch eine Backslash-Substitution des Vorlagenstrings `template` ergibt, ähnlich wie die Methode `sub`. Escape-Sequenzen wie `\n` werden in die entsprechenden Zeichen umgewandelt, und numerische (`\1`, `\2`) und benannte (`\g<1>`, `\g<name>`) Rückwärtsreferenzen werden durch die entsprechende Gruppe ersetzt.

Mustersyntax

Musterstrings werden durch Verkettung von Formen angegeben (siehe Tabelle 19) wie auch durch Escape-Sequenzen für Zeichenklassen (siehe Tabelle 20). Python-Escape-Sequenzen (z.B. `\t` für Tab) dürfen ebenso vorkommen. Musterstrings werden auf Textstrings angewandt und liefern ein Boolesches Ergebnis sowie gruppierte Substrings, die von Untermustern in Klammern gefunden wurden.

```
>>> import re
>>> patt = re.compile('hello[ \t]*(.*)')
```

```
>>> mobj = patt.match('hello  world!')
>>> mobj.group(1)
'world!'
```

In Tabelle 19 ist `C` ein beliebiges Zeichen, `R` ein beliebiger regulärer Ausdruck, und `m` und `n` sind Integer. Alle Formen erkennen normalerweise so viel Text wie möglich, es sei denn, es handelt sich um nicht-gierige Formen (die den kleinsten Text finden, der das gesamte Muster immer noch auf den Zielstring passen lässt).

Tabelle 19: Syntax für reguläre Ausdrücke

Form	Beschreibung
`.`	Erkennt jedes Zeichen (einschließlich Newline, wenn der Schalter `DOTALL` angegeben wurde).
`^`	Erkennt den Anfang des Strings (jeder Zeile, wenn im `MULTILINE`-Modus).
`$`	Erkennt das Ende des Strings (jeder Zeile, wenn im `MULTILINE`-Modus).
`C`	Jedes nicht spezielle Zeichen erkennt sich selbst.
`R*`	Kein oder mehrere Vorkommen des vorangehenden regulären Ausdrucks `R` (so viel wie möglich).
`R+`	Ein oder mehrere Vorkommen des vorangehenden regulären Ausdrucks `R` (so viel wie möglich).
`R?`	Kein oder ein Vorkommen des vorangehenden regulären Ausdrucks `R`.
`R{m}`	Erkennt genau `m` Wiederholungen des vorangehenden regulären Ausdrucks `R`.
`R{m,n}`	Erkennt `m` bis `n` Wiederholungen des vorangehenden regulären Ausdrucks `R`.
`R*?, R+?, R??, R{m,n}?`	Wie `*`, `+` und `?`, aber erkennt so wenige Zeichen/so selten wie möglich; *nicht-gierig*.
`[...]`	Definiert eine Zeichenklasse; z. B. erkennt `[a-zA-Z]` alle Buchstaben (siehe auch Tabelle 20).
`[^...]`	Definiert eine komplementäre Zeichenklasse; trifft zu, wenn das Zeichen nicht in der Menge vorhanden ist.

Tabelle 19: Syntax für reguläre Ausdrücke (Fortsetzung)

Form	Beschreibung
`\`	Maskiert Sonderzeichen (z. B. `*?+|()`) und leitet spezielle Sequenzen ein (siehe Tabelle 20). Aufgrund der Python-Regeln schreibt man es als `\\`.
`\\`	Erkennt ein Literal `\`; aufgrund von Pythons String-Regeln schreibt man im Muster `\\\\` oder nutzt die raw-Schreibweise `r'\\'`.
`\number`	Erkennt den Inhalt der Gruppe mit der entsprechenden Nummer: `(.+) \1` erkennt »42 42«
`R|R`	Alternative: erkennt linken oder rechten `R`.
`RR`	Verkettung: erkennt beide `R`.
`(R)`	Erkennt RE innerhalb von `()` und grenzt eine Gruppe ab (fängt erkannten Substring ein).
`(?: R)`	Wie `(R)`, aber ohne eine Gruppe abzugrenzen, die den erkannten Substring einfängt.
`(?= R)`	Lookahead-Zusicherung: trifft, wenn `R` als Nächstes trifft, aber verbraucht nichts vom String. (Zum Beispiel: `X (?=Y)` erkennt `X`, wenn `Y` darauf folgt.)
`(?! R)`	Negative Lookahead-Zusicherung: trifft, wenn `R` als Nächstes nicht trifft. Umkehrung von `(?=R)`.
`(?P<name> R)`	Erkennt RE innerhalb von `()` und grenzt eine benannte Gruppe ab. (Zum Beispiel: `r'(?P<id>[a-zA-Z_]\w*)'` definiert eine Gruppe namens `id`.)
`(?P=name)`	Erkennt den durch die frühere Gruppe `name` erkannten Text.
`(?#...)`	Ein Kommentar; wird ignoriert.
`(?letter)`	`letter` ist `a`, `i`, `L`, `m`, `s`, `x` oder `u`. Setzt Flag (`re.A`, `re.I`, `re.L`, usw.) für gesamten RE.
`(?<= R)`	Positive Lookbehind-Zusicherung: trifft nach einem Treffer des Musters `R` mit fester Breite.
`(?<! R)`	Negative Lookbehind-Zusicherung: trifft nach keinem Treffer des Musters `R` mit fester Breite.
`(?(id/name)yespattern|nopattern)`	Versucht, mit Muster `yespattern` zu vergleichen, wenn die Gruppe mit der angegebenen `id` oder dem angegebenen `name` existiert, ansonsten mit dem optionalen Muster `nopattern`.

Das Verhalten der Escape-Sequenzen \b, \B, \d, \D, \s, \S, \w und \W in Tabelle 20 wird von Schaltern gesteuert und ist in Python 3.0 standardmäßig Unicode-basiert, es sei denn, ASCII (?a) wird verwendet. Tipp: Nutzen Sie Raw-Strings (r'\n'), um die Eingabe der Backslashs in den Escape-Sequenzen aus Tabelle 20 zu vereinfachen.

Tabelle 20: Spezialsequenzen in regulären Ausdrücken

Sequenz	Beschreibung
\nummer	Erkennt Text der Gruppe *nummer* (von 1).
\A	Trifft nur am Anfang des Strings.
\b	Leerer String an Wortgrenze.
\B	Leerer String nicht an Wortgrenze.
\d	Jede Dezimalziffer (wie [0-9]).
\D	Jede Nicht-Dezimalziffer (wie [^0-9]).
\s	Jedes Whitespace-Zeichen (wie [\t\n\r\f\v]).
\S	Jedes Nicht-Whitespace-Zeichen (wie [^ \t\n\r\f\v]).
\w	Jedes alphanumerische Zeichen.
\W	Jedes nicht-alphanumerische Zeichen.
\Z	Trifft nur am Ende des Strings.

Module zur Objekt-Persistenz

Drei Module bilden die Schnittstelle zur Objekt-Persistenz:

dbm *(*anydbm *in Python 2.x)*
: Schlüsselbasierte Dateispeicher für Strings.

pickle *(und* cPickle *in Python 2.x)*
: Serialisiert/deserialisiert ein Objekt im Speicher in/aus Datei-Streams.

shelve
: Schlüsselbasierte persistente Objektspeicher: Serialisiert/deserialisiert Objekte in/aus dbm-Dateien.

Das Modul shelve implementiert persistente Objektspeicher. shelve benutzt zunächst das Modul pickle zur Konvertierung von Python-Objekten im Speicher in Bytes-Stream-Strings (Serialisierung) und dann das Modul dbm, um diese Strings in schlüsselbasierten Dateien abzulegen.

HINWEIS

In Python 2.x heißt dbm anydbm, das cPickle-Modul ist eine optimierte Version von pickle, die direkt importiert werden kann und automatisch von shelve genutzt wird, wenn sie vorhanden ist. In Python 3.0 wurde cPickle in _pickle umbenannt und wird automatisch von pickle genutzt, falls es vorhanden ist – es muss nicht direkt importiert werden und wird von shelve angefordert.

Beachten Sie auch, dass in Python 3.0 die Berkeley DB-Schnittstelle (bsddb) für dbm nicht mehr automatisch mit Python ausgeliefert wird, sondern als externe Open Source-Erweiterung verfügbar ist, die separat installiert werden muss (zu Ressourcen siehe die Python 3.0 Library Reference).

Die Module dbm und shelve

DBM ist ein Dateisystem mit Zugriff über Schlüssel: Strings werden über ihren Schlüssel gespeichert und abgerufen. Das Modul dbm wählt die entsprechende Implementierung von schlüsselbasierten Dateien in Ihrem Python-Interpreter aus und präsentiert eine Dictionary-ähnliche API für Skripten. Ein persistentes Objekt-Shelve wird wie eine einfache *dbm*-Datei verwendet, außer dass das Modul dbm durch shelve ersetzt wird und dass der gespeicherte wert nahezu jede Art von Python-Objekt sein kann (Schlüssel sind nach wie vor Strings). In vielerlei Hinsicht verhalten sich dbm-Dateien und -Shelves wie Dictionaries, die vor der Verwendung geöffnet und nach Änderungen geschlossen werden müssen; alle Operationen für Abbildungen und einige Dictionary-Methoden werden unterstützt.

`import shelve,`
`import dbm`

Ruft eine der verfügbaren `dbm`-Unterstützungsbibliotheken ab: `dbm.bsd`, `dbm.gnu`, `dbm.ndbm`, or `dbm.dumb`.

`open(…)`

```
datei = shelve.open(dateiname
   [, flag='c'
   [, protocol=None
   [, writeback=False]]])
datei = dbm.open(dateiname
   [, flag='r'
   [, mode]])
```

Erzeugt eine neue oder öffnet eine existierende `dbm`-Datei.

`flag` ist in `shelve` und `dbm` identisch (`shelve` leitet es an `dbm` weiter). `'r'` öffnet eine bestehende Datenbank zum Lesen (`dbm`-Standard), `'w'` öffnet eine bestehende Datenbank zum Lesen und Schreiben, `'c'` erstellt eine Datenbank, wenn sie nicht besteht (`shelve`-Standard), und `'n'` erstellt immer eine neue leere Datenbank. Das Modul `dbm.dumb` (wird in 3.0 als Standard verwendet, wenn keine andere Datei installiert ist) ignoriert `flag` – die Datenbank wird immer zum Aktualisieren geöffnet und wird erstellt, wenn sie noch nicht besteht.

Bei `dbm` gibt das optionale Argument `mode` den Unix-Modus der Datei an, wird aber nur genutzt, wenn die Datenbank erstellt werden muss. Standardwert ist der Oktalwert 0o666.

Bei `shelve` wird das Argument `protocol` an `pickle` weitergereicht. Es gibt die Pickle-Protokollnummer an (die weiter unten erläutert wird), die zur Speicherung von Shelve-Objekten genutzt wird; in Python 2.6 ist der Standardwert `0` und in Python 3.0 `2`. Standardmäßig werden Änderungen an aus Shelves abgerufenen Objekten nicht automatisch im Dateisystem gespeichert. Ist der optionale Parameter `writeback` auf `True` gesetzt, werden alle Einträge, auf die zugegriffen wird, im Speicher gecacht und beim Schließen geschrieben. Das vereinfacht die Veränderung veränderlicher Einträge im Shelve, kann aber Speicher für den Cache in Anspruch nehmen und das

Schließen verlangsamen, weil alle Einträge geschrieben werden müssen.

`datei['schlüssel'] = wert`
: Speichern: Erstellt oder ändert den Eintrag für `'schlüssel'`. Bei `dbm` ist `wert` ein String, bei `shelve` ein beliebiges Objekt.

`wert = datei['schlüssel']`
: Abrufen: Lädt den Wert für den Eintrag `'schlüssel'`. Bei `shelve` wird das Objekt im Speicher rekonstruiert.

`anzahl = len(dateiname)`
: Größe: Liefert die Anzahl der gespeicherten Einträge.

`index = datei.keys()`
: Index: Ruft die gespeicherten Schlüssel ab (kann in `for` oder einem anderen Iterationskontext verwendet werden).

`wert = datei.has_key('schlüssel')`
: Suche: Prüft, ob es einen Eintrag für `'schlüssel'` gibt.

`del datei['schlüssel']`
: Löschen: Entfernt den Eintrag für `'schlüssel'`.

`file.close()`
: Manuelles Schließen; bei manchen zugrunde liegenden `dbm`-Schnittstellen erforderlich zum Zurückschreiben von Änderungen auf die Platte.

pickle-Modul

Die `pickle`-Schnittstelle konvertiert nahezu beliebige Python-Objekte im Speicher in/aus serialisierten Byte-Streams. Diese Byte-Streams können zu jedem dateiähnlichen Objekt umgeleitet werden, das die erwarteten Lese-/Schreiboperationen bietet. Bei der Deserialisierung, dem »Entpickeln«, wird das ursprüngliche Objekt im Speicher wiederhergestellt (mit gleichem Wert, aber anderer Identität, d.h. neuer Adresse).

Siehe die vorangehende Notiz zu den optimierten Modulen in Python 2.x, `cPickle`, und Python 3.0, `_pickle`. Siehe auch die `makefile`-Methode von Socket-Objekten zum Versenden serialisierter Objekte über Netzwerke.

Pickle-Schnittstelle

`P = pickle.Pickler(fileobject [, protocol=None])`
: Erzeugt einen `Pickler` zum Speichern in ein Ausgabedateiobjekt.

`P.dump(object)`
: Schreibt ein Objekt in Datei/Stream des `Picklers`.

`pickle.dump(object, fileobject [, protocol=None])`
: Kombination der beiden vorherigen: Serialisiert `object` in Datei.

`string = pickle.dumps(object [, protocol=None])`
: Liefert eine serialisierte Darstellung des Objekts als String (in Python 3.0 ein `bytes`-String).

Ent-Pickle-Schnittstellen

`U = pickle.Unpickler(fileobject, encoding="ASCII", errors="strict")`
: Erzeugt einen `Unpickler` zum Laden aus einem Eingabedateiobjekt.

`object = U.load()`
: Liest ein Objekt aus Datei/Stream des `Unpicklers`.

`object = pickle.load(fileobject, encoding="ASCII", errors="strict")`
: Kombination der beiden vorherigen: Deserialisiert `object` aus Datei.

`object = pickle.loads(string, encoding="ASCII", errors="strict")`
: Liest Objekte aus einem Zeichenstring (in Python 3.0 ein `bytes`-String).

Verwendungshinweise

- In Python 3.0 sollten Dateien, die zum Speichern von Pickle-Objekten genutzt werden, bei allen Protokollen immer im binären Modus geöffnet werden, da der Pickler `bytes`-Strings erzeugt und Dateien im Textmodus das Schreiben von `bytes` nicht

unterstützen (Dateien im Textmodus werden in Python 3.0 als Unicode-Text kodiert und dekodiert).

- In Python 2.6 müssen Dateien, die zum Speichern von Pickle-Objekten verwendet werden, für alle Pickle-Protokolle >= 1 im Binärmodus geöffnet werden, damit die Übersetzung von Zeilenenden bei binären Pickle-Daten unterdrückt werden. Protokoll 0 ist ASCII-basiert, Dateien können also im Text- oder Binärmodus geöffnet werden, solange man dabei konsistent verfährt.
- `fileobject` ist ein offenes Dateiobjekt oder jedes Objekt, das die durch die Schnittstelle aufgerufenen Attribute besitzt. `Pickler` ruft die Dateimethode `write` mit einem String als Argument auf. `Unpickler` ruft die Dateimethode `read` mit einem Bytezähler und `readline` ohne Argumente auf.
- `protocol` ist ein optionales Argument, das das Format für die serialisierten Daten wählt und im `Pickler`-Konstruktor sowie in den `dump`- und `dumps`-Unterstützungsmethoden des Moduls verfügbar ist. Dieses Argument kann einen Wert zwischen `0...3` annehmen. Höhere Protokollnummern sind im Allgemeinen effizienter, können aber inkompatibel zu Unpicklern in früheren Python-Versionen sein. Die Standardprotokollnummer in Python 3.0 ist `3`, kann vom Python-2.x-Unpickler aber nicht verarbeitet werden. Das Standardprotokoll in Python 2.6 ist `0`. Dieses Protokoll ist weniger effizient, aber portabler. Bei `-1` wird automatisch das höchste unterstützte Protokoll genutzt. Bei der Deserialisierung wird das Protokoll durch die Dateninhalte vorgegeben.
- Die optionalen benannten Unpickler-Argumente `encoding` und `errors` sind nur in Python 3.0 verfügbar. Sie werden genutzt, um die 8-Bit-String-Instanzen zu deserialisieren, die mit Python 2.x serialisiert wurden. Die Standardwerte für diese Argumente sind `'ASCII'` und `'strict'`.
- `Pickler` und `Unpickler` sind exportierte Klassen, die durch Erweiterung angepasst werden können. In der Python Library Reference finden Sie die verfügbaren Methoden.

Das GUI-Modul tkinter und Werkzeuge

tkinter (Tkinter in Python 2.x und ein Modulpaket in Python 3.0) ist eine portable Bibliothek zum Aufbau grafischer Benutzeroberflächen (GUI) und gehört zu den Standardbibliotheksmodulen von Python. tkinter bietet eine objektbasierte Schnittstelle zur Open Source-Tk-Bibliothek und implementiert ein natives Look-and-Feel für in Python geschriebene GUIs unter Windows, X-Windows und Mac OS X. Es ist portabel, einfach zu benutzen, gut dokumentiert, ausgereift und wird vielfach eingesetzt und gut unterstützt. Andere portable GUI-Optionen für Python wie *wxPython* und *PyQT* sind externe Erweiterungen mit einem umfangreicheren Widget-Satz, in der Regel aber auch komplexeren Anforderungen an die Programmierung.

tkinter-Beispiel

In tkinter-Skripten sind *Widgets* konfigurierbare Klassen (z. B. Button, Frame), *Optionen* sind Schlüsselwortargumente (z. B. text="press"), und *Komposition* bezieht sich auf Objekteinbettung, nicht auf Pfadnamen (z. B. Label(Top,...)):

```
from tkinter import *               # Widgets, Konstanten
def msg():                          # Callback-Handler
    print('hello stdout...')
top = Frame()                       # Container erstellen
top.pack()
Label(top,  text="Hello world").pack(side=TOP)
widget = Button(top, text="press", command=msg)
widget.pack(side=BOTTOM)
top.mainloop()
```

tkinter-Kern-Widgets

Tabelle 21 führt die primären Widget-Klassen im tkinter-Modul auf. Es handelt sich dabei um wirkliche Python-Klassen, die erweitert und deren Instanzen in andere Objekte eingebettet werden können. Um ein Widget zu erzeugen, erstellen Sie eine Instanz der entsprechenden Klasse, konfigurieren sie und legen die Gestalt mit einer der Methoden der Geometriemanager-Schnittstelle fest (z. B.

`Button(text='hello').pack()`). Neben den Klassen in Tabelle 21 bietet das Modul `tkinter` eine große Menge an vordefinierten Namen (d.h. Konstanten), die zur Konfiguration von Widgets genutzt werden (z.B. `RIGHT`, `BOTH`, `YES`); diese werden automatisch aus `tkinter.constants` geladen (`Tkconstants` in Python 2.x).

Tabelle 21: Kern-Widget-Klassen des Moduls tkinter

Widget-Klasse	Beschreibung
`Label`	Einfache Meldungsfläche.
`Button`	Einfaches, beschriftetes Button-Widget.
`Frame`	Container, in dem andere Widget-Objekte gehalten und angeordnet werden.
`Toplevel, Tk`	Fenster auf oberster Ebene, die vom Fenstermanager verwaltet werden.
`Message`	Feld zur mehrzeiligen Textanzeige (Label).
`Entry`	Einfaches, einzeiliges Texteingabefeld.
`Checkbutton`	Button-Widget mit zwei Zuständen für Mehrfachauswahl.
`Radiobutton`	Button-Widget mit zwei Zuständen für Einfachauswahl.
`Scale`	Ein Schieber-Widget mit skalierbarer Position.
`PhotoImage`	Bildobjekt zur Platzierung von Full-Color-Bildern in andere Widgets.
`BitmapImage`	Bildobjekt zur Platzierung von Bitmap-Bildern in andere Widgets.
`Menu`	Mit einem `Menubutton` oder Fenster oberster Ebene assoziierte Optionen.
`Menubutton`	Button, der ein `Menu` auswählbarer Optionen/Untermenüs öffnet.
`Scrollbar`	Balken zum Scrollen anderer Widgets (z.B. `Listbox`, `Canvas`, `Text`).
`Listbox`	Liste von auswählbaren Namen.
`Text`	Mehrzeiliges Widget zum Lesen/Bearbeiten von Text mit Unterstützung für Fonts usw.
`Canvas`	Grafische Zeichenfläche: Linien, Kreise, Fotos, Text usw.
`OptionMenu`	*Zusammengesetzt*: Pull-down-Auswahlliste.
`PanedWindow`	Eine Multipane-Window-Schnittstelle.
`LabelFrame`	Ein Frame-Widget mit Label.
`Spinbox`	Ein Widget für Mehrfachauswahl.

Tabelle 21: Kern-Widget-Klassen des Moduls tkinter (Fortsetzung)

Widget-Klasse	Beschreibung
`ScrolledText`	Python 2.x-Name (in Python 3.0 im Modul `tkinter.scrolledtext` verfügbar); *Zusammengesetzt*: Text mit Scrollbalken.
`Dialog`	Python 2.x-Name (in Python 3.0 im Modul `tkinter.dialog` verfügbar); *Alt*: Erzeuger für allgemeine Dialoge (siehe die neuen Aufrufe im nächsten Abschnitt).

Häufige verwendete Dialogaufrufe

Modul tkinter.messagebox (tkMessageBox in Python 2.x)

```
showinfo(title=None, message=None, **optionen)
showwarning(title=None, message=None, **optionen)
showerror(title=None, message=None, **optionen)
askquestion(title=None, message=None, **optionen)
askokcancel(title=None, message=None, **optionen)
askyesno(title=None, message=None, **optionen)
askretrycancel(title=None, message=None, **optionen)
```

Modul tkinter.simpledialog (tkSimpleDialog in Python 2.x)

```
askinteger(title, prompt, **kw)
askfloat(title, prompt, **kw)
askstring(title, prompt, **kw)
```

Modul tkinter.colorchooser (tkColorChooser in Python 2.x)

```
askcolor(color = None, **optionen)
```

Modul tkinter.filedialog (tkFileDialog in Python 2.x)

```
class Open
class SaveAs
class Directory
askopenfilename(**optionen)
asksaveasfilename(**optionen)
askopenfile(mode="r", **optionen)
asksaveasfile(mode="w", **optionen)
askdirectory(**optionen)
```

Die Optionen beim Aufruf allgemeiner Dialoge sind `defaultextension` (zum Dateinamen hinzugefügt, wenn nicht explizit angegeben), `filetypes` (Sequenz aus (`marke`, `muster`)-Tupeln), `initialdir`

(ursprüngliches Verzeichnis, Klassen merken sich das), `initialfile` (ursprüngliche Datei), `parent` (Fenster, das die Dialogbox aufnimmt) und `title` (Titel der Dialogbox).

Zusätzliche tkinter-Klassen und Werkzeuge

Tabelle 22 führt einige `tkinter`-Schnittstellen und -Werkzeuge auf, die neben den Kern-Widget-Klassen und Standarddialogen ebenfalls häufiger genutzt werden.

Tabelle 22: Zusätzliche tkinter-Werkzeuge

Kategorie	Verfügbare Werkzeuge
Mit tkinter verknüpfte Variablenklassen	`StringVar`, `IntVar`, `DoubleVar`, `BooleanVar` (in `tkinter`-Modul).
Methoden zur Geometrieverwaltung	Die Widget-Objektmethoden `pack`, `grid`, `place` sowie Konfigurationsoptionen in Modulen.
Zeitgebundene Callbacks	Die Wiget-Methoden `after`, `wait` und `update`; Datei-I/O-Callbacks.
Andere Tkinter-Werkzeuge	Zugriff auf Zwischenablage; elementares `bind`/`Event`- Event-Verarbeitungs-Widget-Objekt-; Widget-`config`-Optionen; Unterstützung für modale Dialoge.
tkinter-Erweiterungen (suchen Sie im Web)	*PMW*: weitere Widgets, *PIL*: Bilder, Tree-Widgets, Font-Unterstützung, Drag-and-Drop, tix-Widgets, ttk-Widgets mit Themes usw.

Tcl/Tk-Python/tkinter-Zuordnungen

Tabelle 23 vergleicht Pythons `tkinter`-API mit der zugrunde liegenden Tk-Bibliothek, wie sie von der Sprache Tcl bereitgestellt wird. Im Allgemeinen sind Tcl-Befehlsstrings Python-Objekten zugeordnet. Im Einzelnen unterscheidet sich in Pythons `tkinter` die Tk-GUI-Schnittstelle auf folgende Weisen von Tcl:

Erzeugung
: Widgets werden als Klasseninstanzen erzeugt, indem man eine Widget-Klasse aufruft.

Eltern
: Eltern sind zuvor erzeugte Objekte, die dem Klassenkonstruktor eines Widgets übergeben werden.

Widget-Optionen
: Optionen des Konstruktors oder `config`-Aufrufs sind Schlüsselwortparameter oder indizierte Schlüssel.

Operationen
: Widget-Operationen (Aktionen) werden zu Methoden des `tkinter`-Widget-Klassenobjekts.

Callbacks
: Callback-Handler kann jedes aufrufbare Objekt sein: Funktion, Methode, `lambda`, Klasse mit `__call__`-Methode usw.

Erweiterung
: Widgets werden erweitert durch den Einsatz der Klassenvererbungsmechanismen von Python.

Komposition
: Schnittstellen werden durch Anfügen von Objekten erzeugt, nicht durch Verketten von Namen.

Verbundene Variablen
: Mit Widgets assoziierte Variablen sind `tkinter`-Klassenobjekte mit Methoden.

Tabelle 23: Tk–tkinter-Zuordnung

Operation	Tcl/Tk	Python/tkinter
Erzeugen	`frame .panel`	`panel = Frame()`
Eltern	`button .panel.quit`	`quit = Button(panel)`
Optionen	`button .panel.go -fg black`	`go = Button(panel, fg='black')`
Konfiguration	`.panel.go config -bg red`	`go.config(bg='red') go['bg'] = 'red'`
Aktionen	`.popup invoke`	`popup.invoke()`
Packen	`pack .panel -side left -fill x`	`panel.pack(side=LEFT, fill=X)`

Internetmodule und -werkzeuge

Dieser Abschnitt gibt einen Überblick über die Python-Unterstützung für die Internetprogrammierung.

Weithin benutzte Bibliotheksmodule

Dieser Abschnitt ist eine repräsentative Auswahl häufiger eingesetzter Module aus Pythons Internetkollektion; eine vollständigere Liste finden Sie in der Python Library Reference.

`socket`
: Unterstützung primitiver Netzwerkkommunikation (TCP/IP, UDP usw.). Schnittstellen zum Senden und Empfangen von Daten über Sockets im BSD-Stil: `socket.socket()` erzeugt ein Objekt mit Socket-Aufrufmethoden (z.B. `object.bind()`). Die meisten Protokoll- und Servermodule setzen auf diesem Modul auf.

`socketserver` *(`SocketServer` in Python 2.x)*
: Framework für allgemeine Thread- und Fork-basierte Netzwerkserver.

`xdrlib`
: Portable Kodierung binärer Daten (siehe auch das Modul `socket` weiter oben in dieser Liste).

`select`
: Schnittstellen zur `select`-Funktion unter Unix und Windows. Wartet auf Aktivität auf einer von `N` Dateien oder Sockets. Wird üblicherweise eingesetzt, um zwischen mehreren Streams zu multiplexen oder um Zeitbegrenzungen zu implementieren. Funktioniert unter Windows nur für Sockets, nicht für Dateien.

`cgi`
: Skriptunterstützung für serverseitiges CGI: `cgi.FieldStorage` parst den Eingabestreams; `cgi.escape` wendet HTML-Escape-Konventionen auf Ausgabe-Streams an. Parsen von und Zugriff auf Formularinformationen: Nachdem ein CGI-Skript `form=cgi.FieldStorage()` aufgerufen hat, ist `form` ein Dictionary-ähn-

liches Objekt mit einem Eintrag je Formularfeld (z.B. ist `form["name"].wert` der Text des Formularfelds `name`).

`urllib.request` *(*`urllib`*,* `urllib2` *in Python 2.X)*
: Ruft Webseiten und Ausgaben von Serverskripten von ihren Internetadressen (URLs) ab: `urllib.request.urlopen(`*`url`*`)` liefert ein Dateiobjekt mit `read`-Methoden, ebenso `urllib.request.urlretrieve(`*`remote`*, *`local`*`)`. Unterstützt HTTP, FTP, Gopher und lokale Datei-URLs.

`urllib.parse` *(*`urlparse` *in Python 2.x)*
: Zerlegt URL-Strings in Komponenten. Enthält auch Werkzeuge zum Maskieren von URL-Text: `urllib.parse.quote_plus(`*`str`*`)` führt eine URL-Kodierung für Text durch, der in HTML-Ausgabe-Streams eingefügt wird.

`ftplib`
: Module zum FTP-Protokoll. `ftplib` bietet Schnittstellen für den Internetdateitransfer in Python-Programmen. Nach `ftp=ftplib.FTP('sitename')` hat `ftp` Methoden für das Login, zum Wechsel des Arbeitsverzeichnisses, zum Herauf- und Herunterladen von Dateien und Verzeichnislisten usw. Unterstützt Binär- und Texttransfer und funktioniert auf jeder Maschine mit Python und einer Internetverbindung.

`poplib`, `imaplib`, `smtplib`
: Module für die Protokolle POP, IMAP (Mailabruf) und SMTP (Mailversand).

`email`-*Paket*
: Parst und konstruiert E-Mail-Nachrichten mit Headern und Anhängen. Enthält auch MIME-Unterstützung.

`http.client` *(*`httplib` *in Python 2),* `nntplib`, `telnetlib`
: Module für Clients für die Protokolle HTTP (Web), NNTP (News) und Telnet.

`http.server` *(*`CGIHTTPServer` *und* `SimpleHTTPServer` *in Python 2.x)*
: HTTP-Request-Serverimplementierungen.

`xml`-*Paket,* `html`-*Paket (*`htmllib` *in Python 2.x)*
: Parst MXML-Dokumente und HTML-Webseiteninhalte. Das `xml`-Paket unterstützt Parsen über DOM, SAX und ElementTree.

`xmlrpc`-*Paket (*`xmlrpclib` *in Python 2.x)*
: XML-RPC-Remote-Method-Call-Protokoll.

`uu`, `binhex`, `base64`, `binascii`, `quopri`
: Kodiert und dekodiert binäre (oder andere) Daten zur Übertragung als Text.

Tabelle 24 führt einige dieser Module nach Protokolltyp auf.

Tabelle 24: Ausgewählte Python-Internetmodule nach Protokoll

Protokoll	Übliche Funktion	Portnummer	Python-Modul
HTTP	Webseiten	80	`http.client`, `urllib.request`, `xmlrpc.*`
NNTP	Usenet-News	119	`nntplib`
FTP-Daten (Standard)	Dateitransfer	20	`ftplib`, `urllib.request`
FTP-Steuerung	Dateitransfer	21	`ftplib`, `urllib.request`
SMTP	E-Mail senden	25	`smtplib`
POP3	E-Mail empfangen	110	`poplib`
IMAP4	E-Mail empfangen	143	`imaplib`
Telnet	Kommandozeilen	23	`telnetlib`

Andere Module der Standardbibliothek

Dieser Abschnitt dokumentiert einige weitere Module der Standardbibliothek. Ergänzende Informationen zu allen eingebauten Werkzeugen finden Sie in der Python Library Reference. Nutzen Sie die PyPI-Websites (siehe »Sonstige Hinweise« auf Seite 194) oder Ihre Lieblingssuchmaschine, um mehr über externe Module und Werkzeuge in Erfahrung zu bringen.

Das Modul math

Das Modul `math` exportiert Werkzeuge der C-Standard-Mathematikbibliothek für die Verwendung in Python. Tabelle 25 zeigt die Exporte des Moduls; weitere Details finden Sie in der Python Library Reference. Siehe auch das Modul `cmath` in der Python-Bibli-

othek für komplexe Zahlen sowie das *NumPy*-System für fortgeschrittenes numerisches Arbeiten.

Tabelle 25: math-Modulexporte in Python 3.0 und 2.6

acos	acosh	asin	asinh	atan
atan2	atanh	ceil	copysign	cos
cosh	degrees	e	exp	fabs
factorial	floor	fmod	frexp	fsum
hypot	isinf	isnan	ldexp	log
log10	log1p	modf	pi	pow
radians	sin	sinh	sqrt	tan
tanh	trunc			

Das Modul time

Es folgt eine nicht erschöpfende Liste der Exporte des Moduls `time`. Weitere Informationen finden Sie in der Python Library Reference.

`clock()`
: Liefert die CPU-Zeit oder die seit Starten des Prozesses oder seit dem ersten Aufruf von `clock()` verstrichene Zeit. Genauigkeit und Semantik sind plattformabhängig (siehe die Python-Handbücher). Liefert Sekunden in Form einer Fließkommazahl. Nützlich für Benchmarks und zur Messung der Laufzeit alternativer Codeabschnitte.

`ctime([sekunden])`
: Konvertiert eine in Sekunden seit der Epoche ausgedrückte Zeitangabe in einen String, der die lokalisierte Zeit darstellt (z. B. `ctime(time())`). Der Standardwert für das optionale Argument ist die aktuelle Zeit.

`time()`
: Liefert eine Fließkommadarstellung der UTC-Zeit in Sekunden seit der Epoche. Unter Unix ist die Epoche 1970. Kann auf einigen Plattformen eine höhere Genauigkeit haben als `clock()` (siehe Python-Handbücher).

`sleep(sekunden)`
: Unterbricht die Ausführung des Prozesses (des aufrufenden Threads) für die angegebene Anzahl von Sekunden. `sekunden` kann ein `float` sein, um Sekundenbruchteile darzustellen.

Das Modul datetime

Dieses Modul ist ein Hilfsmittel für die Subtraktion von Datumsangaben, die Addition von Tagen zu Datumsangaben usw. Siehe Details in der Python Library Reference.

```
>>> from datetime import date, timedelta
>>> date(2009, 12, 17) - date(2009, 11, 29)
datetime.timedelta(18)
>>> date(2009, 11, 29) + timedelta(18)
datetime.date(2009, 12, 17)
```

Module zum Threading

Threads sind leichtgewichtige Prozesse, die einen gemeinsamen globalen Speicher besitzen (d.h., lexikalischen Geltungsbereich und die Interna des Interpreters) und alle parallel innerhalb des gleichen Prozesses laufen. Pythons Thread-Module funktionieren plattformübergreifend.

`_thread` *(*`thread` *in Python 2.x)*
: Das grundlegende und elementare Threading-Modul. Werkzeuge zum Starten, Stoppen und Synchronisieren von parallel laufenden Funktionen. Erzeugen eines Threads: `thread.start_new_thread(function, argstuple)`. Die Funktion `start_new` ist ein Synonym für `start_new_thread` (in Python 3.0 obsolet). Zur Synchronisation von Threads nutzen Sie Thread-Locks: `lock=thread.allocate_lock()`; `lock.acquire()`; *`ändere-Objekte`*; `lock.release()`.

`threading`
: Das Modul `threading` setzt auf `thread` auf und bietet anpassbare Threading-orientierte Klassen: `Thread`, `Condition`, `Semaphore`, `Lock` usw. Leiten Sie von `Thread` ab, um die Aktionsmethode `run`

zu überladen. Ist mächtiger als `_thread`, kann bei einfachen Anwendungsfällen aber mehr Code erfordern.

`queue` *(`Queue` in Python 2.x)*
: Eine Implementierung einer Mehrfachproduzenten-, Mehrfachverbraucher-FIFO-Warteschlange. Besonders nützlich für Anwendungen mit Threads (siehe die Python Library Reference). Sperrt die Operationen `get` und `put`, um den Zugriff auf die Daten in der Warteschlange zu synchronisieren.

Parsen von Binärdaten

Das Modul `struct` enthält eine Schnittstelle zum Parsen und zur Konstruktion von kompakten Binärdaten als Strings. Wird oft zusammen mit den binären Schreib-/Lesemodi `rb` und `wb` beim Öffnen von Dateien verwendet. Siehe das Python Library Manual für Details zu Formatdatentyp und Endian-Codes.

`string = struct.pack(format, v1, v2, ...)`
: Liefert einen im angegebenen Format verpackten String (`bytes` in 3.0, `str` in 2.6) mit den Werten `v1`, `v2`, usw. Die Argumente müssen den Werten genügen, die von den Typcodes des Formats vorgegeben werden. Der Formatstring kann über sein erstes Zeichen die Bit-Ordnung des Ergebnisses angeben sowie Wiederholungsangaben für die einzelnen Typcodes.

`tuple = struct.unpack(format, string)`
: Entpackt den String (`bytes` in 3.0, `str` in 2.6) gemäß dem angegebenen Formatstring.

`struct.calcsize(fmt)`
: Gibt die Größe von `struct` (und damit des Strings) entsprechend dem angegebenen Format zurück.

Das folgende Beispiel zeigt für Python 3.0 (Python 2.x nutzt gewöhnliche `str`-Strings statt `bytes`), wie Daten mit `struct` ein- und ausgepackt werden:

```
>>> import struct
>>> data = struct.pack('4si', 'spam', 123)
>>> data
b'spam{\x00\x00\x00'
```

```
>>> x, y = struct.unpack('4si', data)
>>> x, y
(b'spam', 123)
>>> open('data', 'wb').write(struct.pack('>if', 1, 2.0))
8
>>> open('data', 'rb').read()
b'\x00\x00\x00\x01@\x00\x00\x00'
>>> struct.unpack('>if', open('data', 'rb').read())
(1, 2.0)
```

Die portable SQL-Datenbank-API

Die portable SQL-Datenbank-API von Python bietet Skriptportabilität zwischen verschiedenen herstellerspezifischen SQL-Datenbankpaketen. Für jeden Hersteller installiert man das spezifische Erweiterungsmodul, schreibt seine Skripten aber gemäß der portablen Datenbank-API. Ihre Datenbankskripten werden größtenteils unverändert weiterfunktionieren, wenn Sie zu einem anderen Herstellerpaket migrieren.

Beachten Sie, dass die meisten Datenbankerweiterungsmodule nicht Teil der Python-Standardbibliothek sind (Sie müssen sie sich selbst beschaffen und separat installieren). Das Paket für die eingebettete prozessgebundene relationale Datenbank SQLite ist als Modul der Python-Standard-Bibliothek `sqlite3` verfügbar. Sie ist für die Speicherung von Programmdaten und die Entwicklung von Prototypen gedacht. Der Abschnitt »Module zur Objekt-Persistenz« auf Seite 169 liefert Informationen zu einfacheren Alternativen zur Speicherung von Daten.

Beispiel zur API-Verwendung

Dieses Beispiel nutzt das SQLite-Modul der Standard-Bibliothek. Die Verwendung bei Hochleistungsdatenbanken wie MySQL, PostgreSQL oder Oracle ist ähnlich, erfordert aber andere Verbindungsparameter und die Installation von Erweiterungsmodulen:

```
>>> from sqlite3 import connect
>>> conn = connect(r'C:\users\mark\misc\temp.db')
>>> curs = conn.cursor()
>>> curs.execute('create table jobs (name, title, pay)')
```

```
>>> prefix = 'insert into jobs values '
>>> curs.execute(prefix + "('Bob', 'dev', 100)")
>>> curs.execute(prefix + "('Sue', 'dev', 120)")
>>> curs.execute("select * from jobs where pay > 100")
>>> for (name, title, pay) in curs.fetchall():
...     print(name, title, pay)
...
Sue dev 120
>>> curs.execute("select name, pay from jobs").fetchall()
[('Bob', 100), ('Sue', 120)]
>>> query = "select * from jobs where title = ?"
>>> curs.execute(query, ('dev',)).fetchall()
[('Bob', 'dev', 100), ('Sue', 'dev', 120)]
```

Modulschnittstelle

Dieser und die folgenden Abschnitte enthalten eine unvollständige Liste der Exporte; sämtliche Details finden Sie in der API-Spezifikation unter *http://www.python.org/*.

`connect(parameter...)`
: Konstruktor für Verbindungsobjekte; stellt eine Verbindung zur Datenbank dar. Parameter sind herstellerspezifisch.

`paramstyle`
: Ein String, der die Art der Parameterformatierung angibt (z. B. im Stil `qmark = ?`).

`Warning`
: Ausnahme für wichtige Warnungen, wie zum Beispiel Datenverlust durch Abschneiden.

`Error`
: Diese Ausnahme ist die Basisklasse aller anderen Fehlerausnahmen.

Verbindungsobjekte

Verbindungsobjekte bieten die folgende Methoden:

`close()`
: Schließt die Verbindung sofort (und nicht erst, wenn `__del__` aufgerufen wird).

`commit()`
: Schreibt alle offenen Transaktionen in die Datenbank.

`rollback()`
: Rollt alle offenen Transaktionen zurück und stellt den Datenbankzustand vor dem Öffnen der Transaktion wieder her. Wenn eine Verbindung geschlossen wird, ohne zuvor die Änderungen zu bestätigen, werden alle schwebenden Transaktionen implizit rückgängig gemacht.

`cursor()`
: Erzeugt mithilfe der Verbindung ein neues Cursor-Objekt zum Absenden von SQL-Strings über die Verbindung.

Cursor-Objekte

Cursor-Objekte repräsentieren Datenbankzeiger, mit denen man den Kontext einer Abrufoperation steuern kann.

`description`
: Folge aus Sequenzen, deren sieben Elemente jeweils eine Spalte beschreiben: (*`name`*, *`type_code`*, *`display_size`*, *`internal_size`*, *`precision`*, *`scale`*, *`null_ok`*).

`rowcount`
: Liefert die Anzahl Zeilen, die das letzte `execute*` erzeugt hat (für DQL-Anweisungen wie `select`) oder die betroffen waren (für DML-Anweisungen wie `update` oder `insert`).

`callproc(prozedurname [,parameter])`
: Ruft eine gespeicherte Datenbankprozedur auf. Die Sequenz `parameter` muss einen Eintrag für jedes Argument besitzen, das die Prozedur erwartet. Das Ergebnis ist eine veränderte Kopie der Eingaben.

`close()`
: Schließt den Cursor sofort (und nicht erst beim Aufruf von `__del__`).

`execute(operation [,parameter])`
: Bereitet eine Datenbankoperation vor und führt sie aus (Abfrage oder Befehl). Parameter können als Liste von Tupeln

angegeben werden, um mehrere Datensätze in einer einzigen Operation einzufügen (aber executemany wird bevorzugt).

`executemany(operation, sequenz)`
: Bereitet eine Datenbankoperation (Abfrage oder Befehl) vor und führt sie mit allen Parametersequenzen oder -abbildungen in der Sequenz aus. Entspricht mehrfachen Aufrufen von execute.

`fetchone()`
: Holt die nächste Zeile einer Abfrageergebnismenge. Liefert eine einzelne Sequenz oder None, wenn keine weiteren Daten verfügbar sind.

`fetchmany([size=cursor.arraysize])`
: Holt die nächste Gruppe von Datensätzen eines Abfrageergebnisses als Sequenz von Sequenzen (z.B. eine Liste von Tupeln). Wenn keine weiteren Daten verfügbar sind, wird eine leere Sequenz zurückgegeben.

`fetchall()`
: Holt alle (verbleibenden) Zeilen eines Abfrageergebnisses als Sequenz von Sequenzen (z.B. eine Liste von Tupeln).

Typobjekte und Konstruktoren

`Date(year,month,day)`
: Erzeugt ein Objekt mit einem Datumswert.

`Time(hour,minute,second)`
: Erzeugt ein Objekt mit einem Zeitwert.

`None`
: SQL-NULL-Werte werden in der Ein- und Ausgabe durch das Python-None dargestellt.

Python-Eigenheiten und Tipps

Dieser Abschnitt enthält häufig verwendete Python-Programmiertricks und allgemeine Benutzungshinweise. In der Python Library Reference und der Sprachreferenz (*http://www.python.org/doc/*) finden Sie weitere Informationen zu den hier erwähnten Themen.

Sprachkern

- `S[:]` erzeugt eine flache Kopie einer Sequenz. `copy.deepcopy(X)` erzeugt vollständige (tiefe) Kopien. `list(L)` und `D.copy()` kopieren Listen bzw. Dictionaries.
- `L[:0]=[X,Y,Z]` fügt Elemente vor Ort am Anfang von Liste `L` ein.
- `L[len(L):]=[X,Y,Z]`, `L.extend([X,Y,Z])` und `L += [X,Y,Z]` hängen Elemente ans Ende an.
- `L.append(X)` und `X=L.pop()` können zur Implementierung von Stapeloperationen verwendet werden, wobei das Listenende dem obersten Stapelelement entspricht.
- Benutzen Sie `for key in D.keys():` zum Iterieren über Dictionaries oder einfach `for key in D:` in Version 2.2 und höher. In Python 3.0 sind diese beiden Formen äquivalent, da `keys` ein iterierbarer View ist.
- Nutzen Sie `for key in sorted(D):`, um in Version 2.4 und höher Dictionary-Schlüssel in sortierter Weise zu durchlaufen; die Form `K=D.keys(); K.sort(); for key in K:` funktioniert in Python 2.x ebenfalls, aber nicht in Python 3.0, da `keys`-Ergebnisse View-Objekte und keine Listen sind.
- `X=A or B or None` weist `X` das erste wahre Objekt unter `A` und `B` zu oder `None`, wenn beide falsch sind (d.h. 0 oder leer).
- `X,Y = Y,X` vertauscht die Werte von `X` und `Y`.
- `red, green, blue = range(3)` weist Integer-Folgen zu.
- Benutzen Sie `try/finally`-Anweisungen, um die Ausführung von beliebigem Beendigungscode sicherzustellen. Das ist besonders um Lock-Aufrufe herum hilfreich (vor dem `try` anfordern, in `finally` freigeben).
- Nutzen Sie `with/as`-Anweisungen, um zu sichern, dass objektspezifischer Beendigungscode ausgeführt wird. Für Objekte, die das Kontextmanager-Protokoll unterstützen (z.B. automatisches Schließen von Dateien, automatische Freigabe von Sperren).
- Packen Sie iterierbare Objekte in einen `list()`-Aufruf, um alle Ergebnisse in Python 3 interaktiv zu betrachten; dazu zählen `range()`, `map()`, `zip()`, `filter()`, `dict.keys()` und Weitere.

Umgebung

- Benutzen Sie `if __name__ == '__main__':` zum Anfügen von Selbsttestcode oder für den Aufruf einer Hauptfunktion am Fuß von Moduldateien. Die Abfrage gibt nur wahr zurück, wenn die Datei ausgeführt wird, aber nicht beim Importieren als Bibliothekskomponente.
- Verwenden Sie `data=open('filename').read()`, um den gesamten Inhalt einer Datei mit einem einzigen Ausdruck zu laden.
- Zum Iterieren über die Zeilen von Textdateien verwenden Sie ab Version 2.2 `for line in file:` (in älteren Versionen verwenden Sie `for line in file.readlines():`).
- Nutzen Sie `sys.argv`, um auf Kommandozeilenargumente zuzugreifen.
- Nutzen Sie `os.environ`, um auf die Shell-Einstellungen zuzugreifen.
- Die Standard-Streams sind: `sys.stdin`, `sys.stdout` und `sys.stderror`.
- Verwenden Sie `glob.glob("muster")`, um eine Liste von Dateien zu erhalten, die einem Muster entsprechen.
- Nutzen Sie `os.listdir('.')`, um eine Liste von Dateien und Unterverzeichnissen in einem Pfad zu erhalten.
- Nutzen Sie in Python 3.0 und 2.6 `os.walk`, um einen vollständigen Verzeichnisbaum zu durchlaufen (`os.path.walk` ist in Python 2.6 ebenfalls verfügbar).
- Zum Ausführen von Shell-Befehlen innerhalb von Python-Skripten können Sie `os.system('cmdline')`, `output=os.popen('cmdline', 'r').read()` nutzen. Die zweite Form liest die Standardausgabe des gestarteten Programms und kann auch genutzt werden, um zeilenweise zu lesen.
- Andere Streams gestarteter Programme sind in Python 3.0 über das Modul `subprocess` und in Python 2.x über die `os.popen2/3/4`-Aufrufe zugreifbar. Die Aufrufe `os.fork/os.exec*` haben auf Unix-artigen Plattformen eine ähnliche Wirkung.

- Auf Unix-artigen Plattformen können Sie eine Datei zu einem ausführbaren Skript machen, indem Sie am Anfang eine Zeile der Form `!/usr/bin/env python` oder `#!/usr/local/bin/python` einfügen und der Datei mit dem Befehl `chmod` Ausführungsberechtigung verleihen. Unter Windows können Dateien, dank der Registry, angeklickt und direkt ausgeführt werden.
- Die Funktion `dir([objekt])` ist nützlich zum Inspizieren von Attributnamensräumen; `print (objekt.__doc__)` liefert Ihnen häufig die Dokumentation zu einem Objekt.
- Die Funktion `help([objekt])` bietet Ihnen interaktive Hilfe zu Modulen, Funktionen, Typen und mehr. `help(str)` gibt Hilfe zum Typ `str` aus, `help("`*`modul`*`")` zu einem Modul, auch dann, wenn es noch nicht importiert wurde, und mit `help("`*`thema`*`")` erhalten Sie Hilfe zu Schlüsselwörtern und anderen Hilfethemen (nutzen Sie `"topics"`, um eine Liste der verfügbaren Hilfethemen aufzurufen).
- `print()` und `input()` (in Python 2.x `print` und `raw_input()`) nutzen die Streams `sys.stdout` und `sys.stdin`: Weisen Sie diesen dateiartige Objekte zu, um Eingabe/Ausgabe intern umzuleiten, oder nutzen Sie in Python 3.0 die Form `print(..., file=F)` (oder in Python 2.x die Form `print >> F, ...`).

Benutzung

- Benutzen Sie `from __future__ import` *`featurename`* zum Aktivieren von experimentellen Sprachfunktionen, die existierenden Code beeinträchtigen könnten.
- Intuitive Vermutungen über die Performance in Python-Programmen sind meistens falsch. Messen Sie immer, bevor Sie anfangen zu optimieren oder zu C wechseln. Verwenden Sie die Module `profile` und `time` (sowie *cProfile* und *timeit*).
- In den Modulen `unittest` (auch bekannt als PyUnit) und `doctest` finden sich Werkzeuge zur Durchführung von automatisierten Tests. `unittest` ist ein Klassen-Framework, `doctest` untersucht Dokumentationsstrings nach Tests und deren Ausgabetext.

- Das Bibliotheksmodul `pydoc` kann Dokumentationsstrings aus Modulen, Funktionen, Klassen und Methoden herausziehen und anzeigen.
- Zum Abschalten von Interpreter-Warnmeldungen bezüglich veralteten Modulen lesen Sie den Abschnitt »Warnungssystem« auf Seite 135 und zu `-W` den Abschnitt »Kommandozeilenoptionen« auf Seite 3.
- Optionen zum Verpacken und Verteilen von Python-Programmen finden Sie u.a. in *Distutils*, *PyInstaller*, *py2exe* und *eggs*.
- Mit *PyInstaller* oder *py2exe* lassen sich Python-Programme in *.exe*-Dateien für Windows umwandeln.
- Durch die Erweiterungen *NumPy*, *SciPy* und verwandte Pakete wird aus Python ein numerisch-wissenschaftliches Programmierwerkzeug mit Vektorobjekten und -operationen u.v.m.
- *ZODB* und andere bieten umfangreiche OODB-Unterstützung, die es ermöglicht, native Python-Objekte über einen Schlüssel zu speichern, mit *SQLObject*, *SQLAlchemy* und anderen können Objekt-Relationale-Zuordnungen definiert werden, die es ermöglichen, Klassen mit relationalen Tabellen zu nutzen.
- Zum automatischen Erzeugen von Verbindungscode zur Einbindung von C- und C++-Bibliotheken in Python dient *SWIG*.
- *IDLE* ist eine Python beiliegende Entwicklungsumgebung. Sie bietet Editoren mit Syntaxhervorhebung, Objektbrowser, Debugging usw. Weitere IDEs sind zum Beispiel *PythonWin*, *Komodo*, *PythonWorks*, *Boa Constructor*.
- Die *Emacs*-Hilfe enthält Tipps zum Schreiben und Ausführen von Python-Code im Emacs-Texteditor. Die meisten anderen Editoren unterstützen Python ebenfalls (z.B. Auto-Einrückung, Einfärbung), inklusive *VIM* und *IDLE*; siehe die Editoren-Seite auf *www.python.org*.
- Python 3.0-Portierung: Nutzen Sie unter Python 2.6 die Kommandozeilenoption `-3`, um Inkompatibilitätswarnungen zu aktivieren, und schauen Sie sich das Skript *2to3* an, das große Mengen von 2.x-Code automatisch so konvertiert, dass er unter 3.x läuft.

Sonstige Hinweise

- Wichtige Webadressen:

  http://www.python.org
  : Python-Homepage.

  http://www.python.org/pypi
  : Zusätzliche Python-Werkzeuge von Drittanbietern.

  http://starship.python.net
  : Viele Homepages aus der Python-Community.

  http://codespeak.net/pypy
  : Eine EU-finanzierte Python-Implementierung in Python.

  http://www.rmi.net/~lutz
  : Die Website des Autors.

- Python-Philosophie: `import this`.
- In Python-Beispielen sagt man `spam` und `eggs` statt `foo` und `bar`.
- Always look on the bright side of life.

Index

B

C

D

E

F

G

M

Q

R

S

T

U

V

W