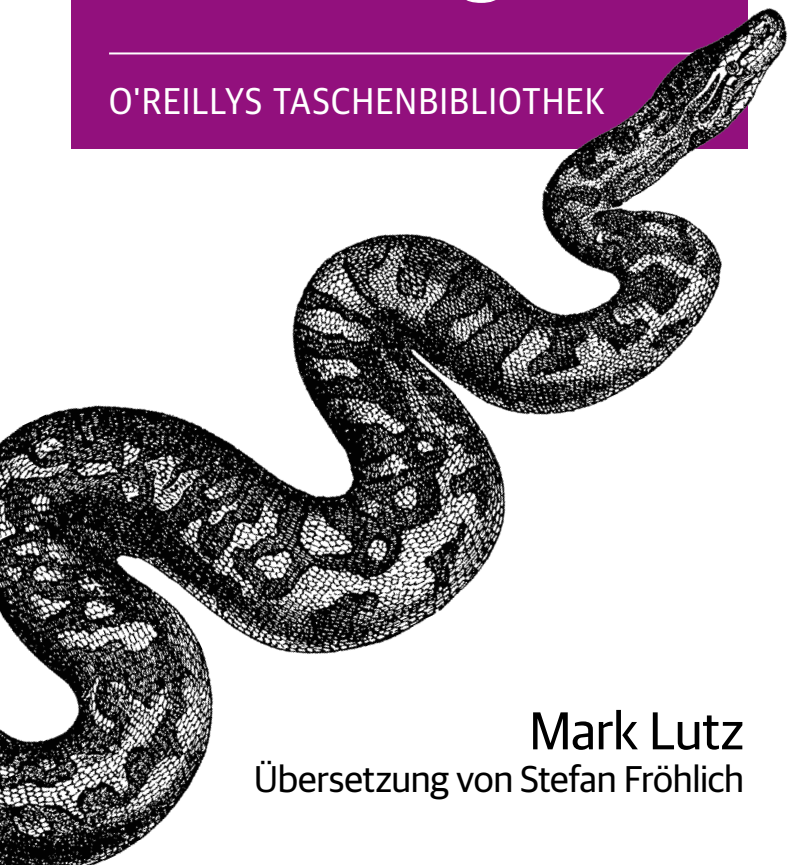
O'REILLY®

5. Auflage
Für Python 3.x und 2.7

Python
kurz & gut

O'REILLYS TASCHENBIBLIOTHEK



Mark Lutz
Übersetzung von Stefan Fröhlich

5. AUFLAGE

Python

kurz & gut

Mark Lutz

Deutsche Übersetzung von
Stefan Fröhlich

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.
Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

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

Copyright der deutschen Ausgabe:
© 2014 O'Reilly Verlag GmbH & Co. KG
1. Auflage 1999
2. Auflage 2002
3. Auflage 2005
4. Auflage 2010
5. Auflage 2014

Die Originalausgabe erschien 2014 unter dem Titel
Python Pocket Reference, Fifth Edition bei O'Reilly Media, Inc.

Die Darstellung einer Felsenpython im Zusammenhang
mit dem Thema Python ist ein Warenzeichen von O'Reilly Media, Inc.

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

Übersetzung: Stefan Fröhlich, Berlin
Lektorat: Susanne Gerbert, Köln
Korrektorat: Kathrin Jurgenowski, Köln
Produktion: Karin Driesen, Köln
Umschlaggestaltung: Michael Oreal, Köln
Satz: Reemers Publishing Services GmbH, Krefeld, www.reemers.de,
Druck: fgb freiburger graphische betriebe; www.fgb.de

ISBN: 978-3-95561-770-7

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Einführung **1**

Typografische Konventionen **2**

Verwendung der Python-Befehlszeile **3**
Python-Optionen 4
Angabe des Programms 6
Befehlsoptionen in Python 2.X 7

Umgebungsvariablen in Python **8**
Operationale Variablen 8
Befehlszeilen-Optionsvariablen 10

Verwendung des Python Launchers unter Windows **10**
Dateidirektiven des Launchers 11
Launcher-Befehlszeilen 11
Launcher-Umgebungsvariablen 12

Integrierte Typen und Operatoren **12**
Operatoren und Vorrang 12
Hinweise zum Gebrauch von Operatoren 14
Operationen nach Kategorien 17
Hinweise zu Operationen für Sequenzen 21

Besondere integrierte Typen **22**
Zahlen 22
Strings 25
Unicode-Strings 44
Listen 48

Dictionaries 55
Tupel 60
Dateien 61
Sets 66
Andere Typen und Konvertierungen 69

Anweisungen und Syntax **70**
Syntaxregeln 71
Namensregeln 73

Spezifische Anweisungen **75**
Zuweisungsanweisungen 76
Ausdrucksanweisungen 80
print-Anweisungen 82
Die if-Anweisung 84
Die while-Anweisung 85
Die for-Anweisung 85
Die pass-Anweisung 86
Die break-Anweisung 86
Die continue-Anweisung 86
Die del-Anweisung 86
Die def-Anweisung 87
Die return-Anweisung 92
Die yield-Anweisung 92
Die global-Anweisung 94
Die nonlocal-Anweisung 95
Die import-Anweisung 95
Die from-Anweisung 99
Die class-Anweisung 101
Die try-Anweisung 103
Die raise-Anweisung 106
Die assert-Anweisung 108
Die with-Anweisung 108
Python 2.X-Anweisungen 110

Namensraum und Gültigkeitsregeln **111**
Qualifizierte Namen: Namensräume von Objekten 111
Unqualifizierte Namen: lexikalische Gültigkeitsbereiche 112
Verschachtelte Gültigkeitsbereiche und Funktionsabschlüsse 114

Objektorientierte Programmierung ... **115**
Klassen und Instanzen ... 116
Pseudoprivate Attribute ... 117
Klassen neuen Stils ... 118
Formale Vererbungsregeln ... 119

Methoden zur Operatorüberladung ... **124**
Methoden für alle Typen ... 125
Methoden für Sammlungen (Sequenzen, Mappings) ... 132
Methoden für Zahlen (binäre Operatoren) ... 134
Methoden für Zahlen (andere Operationen) ... 137
Methoden für Deskriptoren ... 138
Methoden für Kontextmanager ... 139
Methoden zur Operatorüberladung in Python 2.X ... 139

Integrierte Funktionen ... **143**
Integrierte Funktionen in Python 2.X ... 166

Integrierte Ausnahmen ... **173**
Superklassen: Kategorien ... 173
Spezifische Ausnahmen ... 175
Spezifische OSError-Ausnahmen ... 179
Ausnahmen aus der Kategorie Warnungen ... 181
Warning Framework ... 182
Integrierte Ausnahmen in Python 3.2 ... 183
Integrierte Ausnahmen in Python 2.X ... 184

Integrierte Attribute ... **184**

Module der Standardbibliothek ... **185**

Modul sys ... **186**

Modul string ... **195**
Funktionen und Klassen ... 196
Konstanten ... 197

Systemmodul os ... **197**
Administrationswerkzeuge ... 199
Portierbarkeitskonstanten ... 200
Shell-Befehle ... 201

Umgebungswerkzeuge 203
Dateideskriptorwerkzeuge 204
Dateipfadwerkzeuge 207
Prozesssteuerung 211
Modul os.path 215

Mustervergleichsmodul re 218
Modulfunktionen 218
Pattern-Objekte 220
Match-Objekte 221
Muster-Syntax 222

Module für die Persistenz von Objekten 225
Module shelve und dbm 226
Modul pickle 229

tkinter - GUI-Modul und Tools 232
tkinter-Beispiel 232
Wichtige tkinter-Widgets 232
Allgemeine Dialogaufrufe 234
Zusätzliche tkinter-Klassen und Tools 235
Zuordnung Tcl/Tk zu Python/tkinter 235

Internetmodule und Tools 237

Andere Module der Standardbibliothek 239
Modul math 240
Modul time 240
Modul timeit 242
Modul datetime 243
Modul random 243
Modul json 244
Modul subprocess 244
Modul enum 245
Modul struct 246
Thread-Module 247

Pythons SQL-Datenbank-API 248
Anwendungsbeispiele zur API 249
Modulschnittstelle 250

Verbindungsobjekte 250
Cursor-Objekte 251
Typobjekte und Konstruktoren 252

Weitere Tipps und Idiome **252**
Tipps zum Sprachkern 252
Tipps zur Umgebung 254
Tipps zur Benutzung 256
Sonstige Hinweise 258

Index **259**

Python – kurz & gut

Einführung

Python ist eine universelle Programmiersprache, die mehrere Programmierparadigmen unterstützt: sowohl objektorientierte, funktionale als auch prozedurale Codestrukturen. Python wird sowohl für eigenständige Programme als auch für Skripten in den verschiedensten Anwendungsbereichen eingesetzt und gilt generell als eine der am weitesten verbreiteten Programmiersprachen der Welt.

Zu den wichtigsten Merkmalen von Python gehören der Fokus auf lesbaren Code und die Verwendung von Bibliotheken sowie ein Design, das die Produktivität der Entwickler, die Qualität der Software, die Portierbarkeit von Programmen und die Integration von Komponenten optimiert. Python-Programme laufen auf den meisten üblichen Plattformen, darunter Unix und Linux, Windows und Macintosh, Java und .NET, Android und iOS und andere.

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

Die vorliegende *fünfte Auflage* behandelt gleichermaßen Python 3.X und 2.X. Sie befasst sich in erster Linie mit Python 3.X, dokumentiert aber auch die Unterschiede zu Version 2.X. Diese Auflage wurde auf den Stand der aktuellen Python-Versionen *3.4* und *2.7*

aktualisiert. Der Inhalt dieses Buchs gilt aber größtenteils auch für ältere Versionen der Python-Zweige 3.X und 2.X.

Diese Ausgabe bezieht sich außerdem auf alle wichtigen Implementierungen von Python – einschließlich CPython, PyPy, Jython, IronPython und Stackless – und wurde den neuesten Änderungen entsprechend hinsichtlich Sprache, Bibliotheken und bewährten Arbeitsweisen aktualisiert und erweitert. Dazu gehören die neuen Abschnitte zu MRO und `super()`, formalen Vererbungsalgorithmen, Importen, Kontextmanagern, der Einrückung von Codeblöcken sowie häufig verwendeten Bibliotheksmodulen und Tools einschließlich `json`, `timeit`, `random`, `subprocess`, `enum` sowie zum Python Launcher für Windows.

Typografische Konventionen

In diesem Buch werden folgende Auszeichnungskonventionen verwendet:

`[]`
: Eckige Klammern kennzeichnen in Syntaxformaten optionale Elemente, sind aber soweit angegeben auch Teil der Syntax (z.B. in Listen).

`*`
: In Syntaxformaten werden Elemente, hinter denen ein Sternchen steht, null oder mehrmals wiederholt. Das Sternchen wird teilweise aber auch in der Syntax von Python verwendet (z.B. für die Multiplikation).

a | *b*
: Alternativen werden in Syntaxformaten mit vertikale Balken gekennzeichnet. Der vertikalen Balken wird teilweise auch in der Syntax von Python verwendet (z.B. für die Vereinigung).

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

`Nichtproportionalschrift`
: Kennzeichnet Code, Befehle, Befehlszeilenoptionen, Namen von Modulen, Funktionen, Attributen, Variablen und Klassen.

`Nichtproportionalschrift kursiv`
: Wird für zu ersetzende Parameternamen in der Syntax für Kommandozeilen, Ausdrücke, Funktionen und Klassen verwendet.

`Funktion()`
: Soweit nicht anders angegeben, werden aufrufbare Funktionen und Klassen mit zwei darauf folgenden runden Klammern gekennzeichnet, um sie von anderen Attributtypen zu unterscheiden.

Siehe »Abschnittsüberschrift«
: Verweise auf andere Abschnitte dieses Buchs werden mit der jeweiligen Abschnittsüberschrift in Anführungszeichen angegeben.

In diesem Buch bedeuten »3.X« und »2.X«, dass sich ein Thema auf alle gebräuchlichen Versionen des entsprechenden Python-Zweigs bezieht. Genauere Versionsnummern zeigen, dass ein Thema nur für bestimmte Versionen gilt (z.B. bezieht sich »2.7« nur auf Version 2.7). Da sich die Gültigkeit durch Änderungen in künftigen Python-Versionen ändern kann, sollten Sie auch immer einen Blick auf die »What's New«-Dokumente werfen, die momentan für die nach diesem Buch veröffentlichten Pythons unter *http://docs.python.org/3/whatsnew/index.html* gepflegt werden.

Verwendung der Python-Befehlszeile

Mit Befehlszeilen können Sie Python-Programme von einer System-Shell aus starten. Befehlszeilen haben das folgende Format:

```
python [optionen*]
  [ skriptdatei | -c befehl | -m modul | - ] [arg*]
```

In diesem Format steht *python* für die ausführbare Datei des Python-Interpreters – entweder mit dem vollständigen Verzeichnispfad oder nur das Wort `python`, das von der System-Shell ausgelöst wird (z.B. über die `PATH`-Einstellung). Für Python selbst gedachte Befehlszeilenoptionen (*optionen*) stehen vor dem Namen des auszuführenden Programmcodes. Argumente für den auszuführenden Code kommen danach (*arg*).

Python-Optionen

Die Elemente von *optionen* in Befehlszeilen werden von Python selbst verwendet. In Python 3.X gibt es folgende Optionen (siehe Abschnitt »Befehlsoptionen in Python 2.X«, Seite 7 für die Unterschiede zu 2.X):

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

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

`-d`
: Schaltet die Debugging-Ausgabe für den Parser an (für Entwickler des Python-Core).

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

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

`-i`
: Wechselt nach der Ausführung eines Skripts in den interaktiven Modus. Tipp: nützlich für die Postmortem-Fehlersuche; siehe

auch `pdb.pm()`, wie in den Python Library-Handbüchern beschrieben.

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

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

`-q`
: Unterdrückt beim interaktiven Start die Ausgabe der Versions- und Urheberrechtsinformationen (ab Python 3.2).

`-s`
: Das User-Siteverzeichnis nicht zum Modulsuchpfad `sys.path` hinzufügen.

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

`-u`
: Erzwingt, dass *stdout* und *stderr* ungepuffert und binär arbeiten.

`-v`
: Gibt bei jeder Initialisierung eines Moduls eine Meldung aus, von wo das Modul geladen wurde. Wiederholen Sie diese Option für ausführlichere Meldungen.

`-V`
: Gibt die Python-Versionsnummer aus und beendet die Ausführung (alternativ können Sie auch `--version` eingeben).

`-W` *`arg`*
: Warnungssteuerung: *`arg`* hat die Form *`Aktion`*`:`*`Meldung`*`:`*`Kategorie`*`:`*`Modul`*`:`*`Zeilennummer`*. Siehe auch Abschnitt »Warning Framework«, Seite 182 und Abschnitt »Ausnahmen aus der Kategorie Warnungen«, Seite 181 weiter unten sowie die Dokumentation zum Modul `warnings` in der Python Library Reference (unter *http://www.python.org/doc/*).

`-x`
: Überspringt die erste Zeile des Quellcodes, wodurch Sie auch Unix-fremde Schreibweisen von `#!`*`cmd`* verwenden können.

`-X` *`option`*
: Legt eine implementationsspezifische Option fest (ab Python 3.2). Zulässige Werte für *`option`* finden Sie in der Implementierungsdokumentation.

Angabe des Programms

Der auszuführende Code und dafür zu übergebende Befehlszeilenargumente werden in Python-Befehlszeilen folgendermaßen angegeben:

`skriptdatei`
: Name der Python-Skriptdatei, die als Hauptdatei eines Programms ausgeführt werden soll (z.B. führt *`python`* `main.py` den Code in `main.py` aus). Der Name des Skripts kann ein absoluter oder relativer Dateipfad sein (relativ zu ».«) und wird in `sys.argv[0]` zur Verfügung gestellt. Auf manchen Plattformen können Sie die Komponente *`python`* auch weglassen, wenn Sie die Befehlszeile mit dem Namen der Skriptdatei beginnen und keine Optionen an Python selbst übergeben möchten.

`-c` *`befehl`*
: Gibt den auszuführenden Python-Code (als String) an (z.B. führt *`python`* `-c "print('spam' * 8)"` eine print-Anweisung in Python aus). `sys.argv[0]` enthält den Wert `'-c'`.

`-m` *`modul`*
: Führt ein Modul als Skript aus: Sucht nach *`modul`* in `sys.path` und führt dieses als Hauptdatei aus (zum Beispiel führt *`python`* `-m pdb s.py` das Python-Debugger-Modul `pdb` aus dem Standardbibliotheksverzeichnis mit dem Argument `s.py` aus). *`modul`* darf auch der Name eines Pakets sein (z.B. `idlelib.idle`). `sys.argv[0]` enthält den vollständigen Pfad des Moduls.

`-`
: Liest Python-Befehle aus dem Standardeingabestream *stdin* (Standard) und wechselt in den interaktiven Modus, falls es

sich bei *stdin* um ein »tty« (interaktives Gerät) handelt. `sys.argv[0]` erhält den Wert `'-'`.

arg *
: Gibt an, dass der Rest der Befehlszeile an die Skriptdatei oder den Befehl übergeben und in der internen String-Liste `sys.argv[1:]` erscheint.

Wenn *`skriptdatei`*, *`befehl`* oder *`modul`* nicht angegeben werden, wechselt Python in den interaktiven Modus, nimmt Befehle vom *stdin* entgegen (unter Verwendung von GNU *readline* für die Eingabe falls installiert) und legt `sys.argv[0]` auf den Wert `''` fest (Leerstring) – außer wenn die in dieser Liste erklärte Option »-« angegeben wird.

Außer über herkömmliche Befehlszeilen an der Eingabeaufforderung einer System-Shell können Sie Python-Programme üblicherweise auch folgendermaßen ausführen: Indem Sie in einem grafischen Dateibrowser auf den jeweiligen Dateinamen klicken; indem Sie Funktionen der Python-Standardbibliothek aufrufen (z.B. `os.popen()`); über Programmstart-Menüoptionen in IDEs wie IDLE, Komodo, Eclipse, NetBeans usw.

Befehlsoptionen in Python 2.X

Python 2.X unterstützt dasselbe Befehlszeilenformat, aber nicht die Option `-b`, die sich auf Änderungen des String-Typs in Python 3.X bezieht, ebenso wenig wie die in 3.X neu hinzugekommenen Optionen `-q` und `-X`. In den Versionen 2.6 und 2.7 werden dagegen zusätzlich folgende Optionen unterstützt (teilweise auch in früheren Versionen):

`-t` und `-tt`
: `-t` gibt Warnmeldungen für inkonsistente Mischungen von Tabs und Leerzeichen in Einrückungen aus. Die Option `-tt` gibt statt Warnungen Fehlermeldungen aus. Python 3.X behandelt solche Mischungen immer als Syntaxfehler (siehe auch Abschnitt »Syntaxregeln«, Seite 71).

`-Q`
: Divisionsbezogene Optionen: `-Qold` (Standard), `-Qwarn`, `-Qwarnall` und `-Qnew`. Diese Optionen wurden in Python 3.X durch die echte Division subsumiert (siehe auch Abschnitt »Hinweise zum Gebrauch von Operatoren«, Seite 14).

`-3`
: Gibt Warnmeldungen für jegliche Python 3.X-Inkompatibilitäten im Code aus, die das Tool *2to3* der Python-Standardinstallation nicht auf einfache Weise beheben kann.

`-R`
: Aktiviert ein Pseudozufalls-Salt, um vorhersehbare Hash-Werte verschiedener Typen zwischen separaten Aufrufen des Interpreters zum Schutz vor Denial-of-Service-Angriffen zu verhindern. Neu ab Python 2.6.8. Diese Option gibt es im 3.X-Zweig aus Kompatibilitätsgründen ab Version 3.2.3. Die Zufallserzeugung von Hash-Werten ist ab Version 3.3 Standard.

Umgebungsvariablen in Python

Umgebungsvariablen (auch als *Shell*-Variablen bekannt), sind systemweite, programmübergreifende Einstellungen, die für die globale Konfiguration verwendet werden.

Operationale Variablen

Die folgenden wichtigen, vom Benutzer konfigurierbaren Umgebungsvariablen beeinflussen das Verhalten von Skripten:

`PYTHONPATH`
: Erweitert den Standardsuchpfad für importierte Moduldateien. Das Format dieses Variablenwerts ist dasselbe wie für die Shell-Einstellung `PATH`: Verzeichnispfade, die durch Doppelpunkte voneinander getrennt sind (Semikola unter Windows). Wenn diese Umgebungsvariable definiert ist, werden bei Modulimporten die importierten Dateien oder Verzeichnisse von links nach rechts in jedem in `PYTHONPATH` aufgeführten Verzeichnis gesucht. Steht in `sys.path` (dem vollständigen Modulsuchpfad für die am weitesten links stehenden Komponenten in absoluten Impor-

ten) nach dem Verzeichnis des Skripts und vor den Verzeichnissen der Standardbibliotheken. Siehe auch `sys.path` im Abschnitt »Modul sys«, Seite 186, und im Abschnitt »Die import-Anweisung«, Seite 95.

`PYTHONSTARTUP`
: Wenn diese Variable den Namen einer lesbaren Datei enthält, werden die Python-Befehle in dieser Datei ausgeführt, bevor die erste Eingabeaufforderung im interaktiven Modus angezeigt wird (nützlich für die Definition häufig verwendeter Tools).

`PYTHONHOME`
: Wenn dieser Wert festgelegt ist, wird er als alternatives Präfixverzeichnis für die Bibliotheksmodule (oder `sys.prefix`, `sys.exec_prefix`) verwendet. Der Standard-Modulsuchpfad nutzt `sys.prefix/lib`.

`PYTHONCASEOK`
: Wenn dieser Wert festgelegt ist, wird die Groß-/Kleinschreibung in import-Anweisungen ignoriert (derzeit nur unter Windows und OS X).

`PYTHONIOENCODING`
: Weisen Sie den String *`Kodierung`*`[:`*`Fehler-Handler`*`]` zu, um die standardmäßige Unicode-Kodierung (und einen optionalen Fehler-Handler) für Textübertragungen zu den Streams *stdin*, *stdout* und *stderr* zu überschreiben. Diese Einstellung kann in manchen Shells für Nicht-ASCII-Texte erforderlich sein (versuchen Sie beispielsweise die Einstellung `utf8` oder eine andere, falls die Ausgabe nicht korrekt erfolgt).

`PYTHONHASHSEED`
: Bei Angabe von »random« wird beim Seeden der Hash-Werte für `str`-, `bytes`- und `datetime`-Objekte ein Zufallswert verwendet. Als Wert ist auch ein Integer im Bereich `0`...`4.294.967.295` zulässig, um Hash-Werte mit einem vorhersehbaren Seed zu erhalten (ab Python 3.2.3 bzw. 2.6.8).

PYTHONFAULTHANDLER
: Falls diese Variable gesetzt ist, registriert Python beim Start Handler, um bei unbehebbaren Fehlern einen Traceback auszugeben (ab Python 3.3, entspricht `-X faulthandler`).

Befehlszeilen-Optionsvariablen

Die folgenden Umgebungsvariablen sind gleichbedeutend mit einigen Befehlszeilenoptionen von Python (siehe Abschnitt »Python-Optionen«, Seite 4):

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`.

PYTHONWARNINGS
: Wenn nicht leer, gleichbedeutend mit Option `-W`, selber Wert. Akzeptiert auch einen durch Kommata getrennten String als Äquivalent für mehrere `-W`-Optionen. (Ab Python 3.2 und 2.7.)

Verwendung des Python Launchers unter Windows

Unter Windows (und nur unter Windows), installieren Python 3.3 und spätere Versionen einen Skript-Launcher, der für frühere Ver-

sionen auch separat verfügbar ist. Dieser Launcher besteht aus den ausführbaren Dateien `py.exe` (Konsole) und `pyw.exe` (Fenster), die ohne `PATH`-Einstellungen aufgerufen werden können, durch Dateinamenszuordnung für die Ausführung von Python-Dateien registriert sind und drei Möglichkeiten zur Auswahl der Python-Version bieten: mit Unix-artigen »#!«-Direktiven oben im Skript, Befehlszeilenargumenten und konfigurierbaren Standardwerten.

Dateidirektiven des Launchers

Der Launcher erkennt »#!«-Zeilen im oberen Teil von Skriptdateien, die Python-Versionen in einer der folgenden Formen zitieren. Dabei ist * entweder: *leer*, um die Standardversion zu verwenden (aktuell `2`, falls installiert, ähnlich wie ohne »#!«-Zeile); eine *Major*-Versionsnummer (z.B. `3`), um die neueste installierte Version dieses Zweigs zu starten; oder eine vollständige Versionsangabe von *`Major.Minor`*, optional gefolgt von `-32`, falls Sie eine 32-Bit-Installation bevorzugen (z.B. `3.1-32`):

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

Am Zeilenende können Sie beliebige Argumente für Python (`python.exe`) angeben. Python 3.4 und später suchen unter Umständen in `PATH` nach »#!«-Zeilen, die `python` ohne explizite Versionsnummer angeben.

Launcher-Befehlszeilen

Der Launcher kann auch von einer System-Shell aus über Befehlszeilen der folgenden Form aufgerufen werden:

```
py [pyarg] [pythonarg*] skript.py [skriptarg*]
```

Allgemeiner ausgedrückt kann alles, was in einem `python`-Befehl nach der Komponente *`python`* stehen kann, auch nach dem optionalen *`pyarg`* in einem `py`-Befehl stehen und wird Wort für Wort an das gestartete Python übergeben. Das gilt auch für die Angaben `-m`,

-c und -, siehe Abschnitt »Verwendung der Python-Befehlszeile«, Seite 3.

Der Launcher akzeptiert für das optionale *pyarg* folgende Argumentformen, die dem Teil mit dem * am Ende der »#!«-Zeile entsprechen:

```
-2          Neueste installierte 2.X-Version starten
-3          Neueste installierte 3.X-Version starten
-X.Y        Angegebene Version starten (X ist 2 oder 3)
-X.Y-32     Angegebene 32-Bit-Version starten
```

Falls beide Angaben vorhanden sind, haben die Befehlszeilenargumente vor anderen, in den »#!«-Zeilen definierten Werten Vorrang. Bei einer Standardinstallation werden »#!«-Zeilen unter Umständen auf mehr Kontexte angewandt (z.B. Klicks auf Symbole).

Launcher-Umgebungsvariablen

Auch der Launcher kennt optionale Umgebungsvariablen, mit deren Hilfe die Versionswahl generell oder in bestimmten Fällen angepasst werden kann (z.B. fehlendes »#!«, nur Major-Angabe oder py-Befehlsargument):

```
PY_PYTHON   Version für Standardfälle (sonst 2)
PY_PYTHON3  Version für 3-Partials (z.B. 3.2)
PY_PYTHON2  Version für 2-Partials (z.B. 2.6)
```

Diese Einstellungen werden nur bei der Ausführung von Dateien über den Launcher verwendet, nicht wenn *python* direkt aufgerufen wird.

Integrierte Typen und Operatoren

Operatoren und Vorrang

Tabelle 1 zeigt die Ausdrucksoperatoren in Python. Die Operatoren in den unteren Zellen dieser Tabelle haben einen höheren Vorrang (also eine stärkere Bindung), wenn sie ohne Klammern mit anderen Operatoren gemischt werden.

Atomare Terme und Dynamische Typisierung

In Tabelle 1 können die ersetzbaren Ausdruckselemente *X*, *Y*, *Z*, *i*, *j* und *k* Folgendes sein:

- *Variablennamen*, die durch den zuletzt zugewiesenen Wert ersetzt werden
- *Literale*, definiert im Abschnitt »Besondere integrierte Typen«, Seite 22
- *Verschachtelte Ausdrücke* aus einer beliebigen Zeile dieser Tabelle, möglicherweise in Klammern

Python-*Variablen* folgen einem *dynamischen Typisierungsmodell* – sie werden nicht deklariert und erst bei Zuweisung eines Wertes angelegt. Variablen haben Objektverweise als Wert und können auf einem beliebigen Objekttyp verweisen. Außerdem müssen Variablen zugewiesen werden, bevor sie in einem Ausdruck verwendet werden können, und haben keinen Standardwert. Bei Variablennamen kommt es immer auf die Groß-/Kleinschreibung an (siehe Abschnitt »Namensregeln«, Seite 73). Die *Objekte*, auf die Variablen verweisen, werden automatisch erzeugt und automatisch vom *Garbage Collector* (verwendet Referenzzähler in CPython) von Python zurückgefordert, wenn sie nicht mehr verwendet werden.

Außerdem muss in Tabelle 1 *attr* der literale Name eines Attributs (ohne Anführungszeichen) sein. *args1* ist eine formale Argumentenliste wie im Abschnitt »Die def-Anweisung«, Seite 87, definiert. *args2* ist eine Eingabeargumentliste, wie im Abschnitt »Ausdrucksanweisungen«, Seite 80, definiert. Und das Literal `...` qualifiziert in 3.X (und nur da) als atomarer Ausdruck.

Die Syntax für Komprehensionen und Datenstrukturliterale (Tupel, Liste, Dictionary und Set), die in den letzten drei Zeilen von Tabelle 1 abstrakt dargestellt wird, wird im Abschnitt »Besondere integrierte Typen«, Seite 22, definiert.

Tabelle 1: Python 3.X: Ausdrucksoperatoren und deren Vorrang

Operator	Beschreibung
`yield` *X*	Generatorfunktionsergebnis (liefert `send()`-Wert)
`lambda` *args1*`:` *X*	Erzeugt eine anonyme Funktion (liefert beim Aufruf *X* zurück)

Tabelle 1: Python 3.X: Ausdrucksoperatoren und deren Vorrang (Fortsetzung)

Operator	Beschreibung
X `if` *Y* `else` *Z*	Ternäre Auswahl (*X* wird nur ausgewertet, wenn *Y* wahr ist)
X `or` *Y*	Logisches `OR`: *Y* wird nur ausgewertet, wenn *X* nicht wahr ist
X `and` *Y*	Logisches `AND`: *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, Teilmengen und Obermengen von Sets
X == *Y*, *X* != *Y*	Gleichheitsoperatoren
X | *Y*	Bitweises `OR`, Vereinigungsmenge von Sets
X ^ *Y*	Bitweises exklusives `OR`, Symmetrische Differenz von Sets
X & *Y*	Bitweises `AND`, Schnittmenge von Sets
X << *Y*, *X* >> *Y*	Schiebt *X* um *Y* Bits nach links bzw. rechts
X + *Y*, *X* – *Y*	Addition/Verkettung, Subtraktion/Set-Differenz
X * *Y*, *X* % *Y*,*X* / *Y*, *X* // *Y*	Multiplikation/Wiederholung, Rest/Formatieren, Division, restlose Division
-*X*, +*X*	Unäre Negation, Identität
~*X*	Bitweises NOT-Komplement (Umkehrung)
X ** *Y*	Potenzierung
X[*i*]	Indizierung (Sequenzen, Mappings, andere)
X[*i*:*j*:*k*]	Slicing (alle drei Grenzen optional)
X(*args2*)	Aufruf (Funktion, Methode, Klasse und andere aufrufbare Objekte)
X.*attr*	Attributreferenz
(....)	Tupel, Ausdruck, Generatorausdruck
[....]	Liste, Listenkomprehension
{....}	Dictionary, Set, Dictionary- und Set-Komprehension

Hinweise zum Gebrauch von Operatoren

- Nur in Python 2.X kann Wertungleichheit entweder als *X* != *Y* oder *X* <> *Y* geschrieben werden. In Python 3.X wurde die zweite Option wegen Redundanz entfernt.

- Nur in Python 2.X ist ein Ausdruck mit Backticks `` `X` `` gleichbedeutend mit `repr(X)` und wandelt Objekte für die Anzeige in Strings um. In Python 3.X verwenden Sie stattdessen die besser lesbaren integrierten Funktionen `str()` und `repr()`.
- Sowohl in Python 3.X als auch in 2.X schneidet die *`restlose Division`* `X // Y` die Restbruchteile immer ab und liefert als Ergebnis eine ganze Zahl.
- Der Ausdruck `X / Y` führt in 3.X eine *echte Division* durch (die ein Fließkommaergebnis zurückliefert, wobei der Rest erhalten bleibt). In Python 2.X führt dieser Ausdruck dagegen eine *klassische Division* durch (bei der für ganze Zahlen der Rest abgeschnitten wird), außer die echte Division aus 3.X ist über `from __future__ import division` oder die Python-Option `-Qnew` aktiviert.
- Die Syntax `[....]` wird sowohl für Listenliterale als auch für Listenkomprehensionsausdrücke verwendet. Letztere durchlaufen eine implizite Schleife und sammeln die Ergebnisse der Ausdrücke in einer neuen Liste.
- Die Syntax `(....)` wird für Tupel, Ausdrücke und Generatorausdrücke verwendet – eine Form der Listenkomprehension, die Ergebnisse auf Anfrage liefert, anstatt eine Ergebnisliste aufzubauen. In allen drei Konstruktionen können Klammern manchmal ausgelassen werden.
- Die Syntax `{....}` wird für Dictionary-Literale benutzt. In Python 3.X und 2.7 dient sie auch für Set-Literale und Dictionary- sowie Set-Komprehensionen. In 2.6 und früheren Versionen verwenden Sie `set()` und Schleifenanweisungen..
- `yield` und der ternäre Auswahlausdruck `if/else` stehen ab Python 2.5 zur Verfügung.`yield` liefert `send()`-Argumente in Generatoren. `if/else` ist die Kurzschreibweise für eine mehrzeilige `if`-Anweisung. Für `yield` sind Klammern erforderlich, wenn der Ausdruck auf der rechten Seite einer Zuweisungsanweisungen nicht alleine steht.

- Vergleichsoperatoren können verkettet werden: `X < Y < Z` liefert dieselbe Ergebnis wie `X < Y and Y < Z`, jedoch wird `Y` in der verketteten Form 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.X sind Größenvergleiche unterschiedlicher Typen zulässig – Zahlen werden in einen gemeinsamen Typ konvertiert, andere gemischte Typen werden nach dem Typnamen geordnet. In Python 3.X sind Größenvergleiche nicht-numerischer gemischter Typen nicht zulässig und lösen Ausnahmen aus, einschließlich der Sortierung über Proxy-Objekte.
- Größenvergleiche für Dictionaries werden in Python 3.X (im Gegensatz zu Gleichheitsprüfungen) nicht mehr unterstützt. Ein möglicher Ersatz in 3.X ist der Vergleich von `sorted(adict.items())`.
- Aufrufausdrücke unterstützen Positions- und Schlüsselwortargumente sowie eine beliebig große Anzahl beider Arten, siehe Abschnitt »Ausdrucksanweisungen«, Seite 80, und Abschnitt »Die def-Anweisung«, Seite 87, für die Aufrufsyntax.
- Python 3.X gestattet die Verwendung von Ellipsen (`...`, intern bekannt unter dem Namen `Ellipsis`) als atomare Ausdrücke an beliebigen Stellen im Quellcode. In manchen Kontexten (z.B. Funktion-Stubs, typenunabhängige Variableninitialisierung) können sie als Alternative zu `pass` oder `None` verwendet werden.
- Es war zwar noch ungewiss, als dieses Buch geschrieben wurde: Es kann aber sein, dass in Python 3.5 oder einer späteren Version *unter Umständen* die Syntax `*X` und `**X` so verallgemeinert wird, so dass sie auch in Datenstrukturliteralen und Komprehensionen verwendet werden kann, um Sammlungen in einzelne Elemente zu entpacken – ähnlich wie derzeit in Funktionsaufrufen. Weitere Informationen dazu finden Sie im Abschnitt »Zuweisungsanweisungen«, Seite 76.

Operationen nach Kategorien

In diesem Abschnitt werden abschließende Klammern der Einfachheit halber bei Methodennamen der Form `__X__` weggelassen. Im Wesentlichen unterstützen alle integrierten Typen die in Tabelle 2 genannten *Vergleichs-* und *Booleschen* Operationen (jedoch unterstützt Python 3.X keine Größenvergleiche für Dictionaries oder gemischte nicht-numerische Typen).

Der Boolesche Wert *true* bedeutet eine beliebige Zahl ungleich null oder ein beliebiges nicht leeres Sammlungsobjekt (Liste, Dictionary etc.). Alle Objekte haben einen Booleschen Wert. Die integrierten Namen `True` und `False` sind den Wahrheitswerten »wahr« und »falsch« zugeordnet und verhalten sich wie die Integer `1` und `0` mit einem eigenen Anzeigeformat. Das besondere Objekt `None` hat den Wert »falsch« und kommt in verschiedenen Python-Kontexten vor.

Vergleiche liefern `True` oder `False` und werden in zusammengesetzten Objekten automatisch rekursiv angewandt, um ein Ergebnis zu ermitteln.

Die Auswertung der Booleschen Operatoren `and` und `or` wird abgebrochen (Kurzschluss), sobald ein Ergebnis bekannt ist, und liefert das Operandenobjekt zurück – der Wert links oder rechts vom Operator – dessen Boolescher Wert zum Ergebnis führt.

Tabelle 2: Vergleiche und Boolesche Operationen

Operator	Beschreibung
`X < Y`	Streng kleiner als[1]
`X <= Y`	Kleiner oder gleich
`X > Y`	Streng größer als
`X >= Y`	Größer oder gleich
`X == Y`	Gleich (gleicher Wert)

1 Für die Implementierung von Vergleichsausdrücken siehe die Klassen für spezielle Vergleiche (z.B. `__lt__` für `<`) in 3.X und 2.X sowie die Methode `__cmp__` in 2.X für allgemeine Vergleiche im Abschnitt »Methoden zur Operatorüberladung«, Seite 124.

Tabelle 2: Vergleiche und Boolesche Operationen (Fortsetzung)

Operator	Beschreibung
X `!=` *Y*	Ungleich (nur in Python 2.X gleichbedeutend mit *X*`<>`*Y*)[2]
X `is` *Y*	Objektgleichheit
X `is not` *Y*	Negierte Objektgleichheit
X < *Y* < *Z*	Verkettete Vergleiche
`not` *X*	Wenn *X* falsch ist, dann `True`; ansonsten `False`
X `or` *Y*	Wenn *X* falsch ist *Y*; ansonsten *X*
X `and` *Y*	Wenn *X* falsch ist *X*; ansonsten *Y*

Die Tabelle 3 mit Tabelle 6 definieren gemeinsame Operationen für die drei wichtigsten Typkategorien – *Sequenz* (positionell geordnet), *Mapping* (Zugriff über Schlüssel) und *Zahl* (alle numerischen Typen) – sowie die für *veränderbare* Typen in Python verfügbaren Operationen. Die meisten Typen exportieren außerdem zusätzliche typenspezifische Operationen (z.B. Klassen), wie im Abschnitt »Besondere integrierte Typen«, Seite 22, beschrieben.

Tabelle 3: Operationen für Sequenzen (Strings, Listen, Tupel, Bytes, Bytearray)

Operation	Beschreibung	Klassenmethode
X `in` *SX* `not in` *S*	Enthalten in	`__contains__`, `__iter__`, `__getitem__`[3]
S1 + *S2*	Verkettung	`__add__`
S * *N*, *N* * *S*	Wiederholung	`__mul__`
S[*i*]	Indizierung nach Position	`__getitem__`

2 `!=` und `<>` bedeuten in 2.X Wertungleichheit. Aber `!=` ist in 2.X die bevorzugte und in 3.X die einzige zulässige Syntax. `is` führt eine Identitätsprüfung durch, `==` testet auf Wertgleicheit und ist deshalb in der Regel wesentlich nützlicher.

3 Siehe auch Abschnitt »Das Iterationsprotokoll«, Seite 52, für weitere Informationen zu diesen Methoden und ihrem Zusammenspiel. Wenn `__contains__` definiert ist, hat es Vorrang vor `__iter__`, und `__iter__` hat Vorrang vor `__getitem__`.