Michael Weigend

Python

8. Auflage

GE-PACKT

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

mitp

Inhaltsverzeichnis

E Einleitung

E.1 Was ist Python?

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

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

E.2 Einige besondere Merkmale von Python

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

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

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

E.3 Python 2 und 3

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

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

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

Dieses Buch verwendet durchgehend die Syntax von Python 3.

E.4 Hinweise zum Lesen dieses Buches

Typographie

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

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

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

`range(`*zahl*`)`

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

Beispiele

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

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

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

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

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

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

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

Hinweise zum Aufbau der Kapitel

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

1 Basiskonzepte von Python

1.1 Python im interaktiven Modus

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

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

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

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

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

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

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

Wichtige Tastenkombinationen

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

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

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

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

1.2 Ausführung von Python-Skripten

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

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

Windows

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

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

   ```
   python meinSkript.py
   ```

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

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

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

Unix

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

```
python meinSkript.py
```

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

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

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

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

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

```
chmod 711 meinSkript.py
```

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

1.3 Die Zeilenstruktur

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

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

Erlaubt ist:

```
summe = 1 + 2
```

Nicht erlaubt ist:

```
summe = 1
+ 2
```

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

Explizites Verbinden von Zeilen

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

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

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

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

Implizites Verbinden von Zeilen

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

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

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

Einrückungen – Anweisungsblöcke

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

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

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

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

Das folgende Beispielskript zeigt einige typische Einrückungsfehler:

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

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

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

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

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

1.4 Deklaration der Codierung

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

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

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

```
# -*- coding: latin-1 -*-
import sys, os
…
#!/usr/bin/python
# -*- coding: iso-8859-15 -*-
import sys, os
...
```

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

1.5 Bezeichner (identifiers)

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

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

Schlüsselwörter (keywords)

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

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

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

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

Verwendung von Unterstrichen

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

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

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

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

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

Siehe auch: Kapitel 15.3

1.6 Objekte

Identität von Objekten

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

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

Änderbare und unveränderbare Objekte

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

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

Betrachten wir folgende Anweisungsfolge:

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

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

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

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

Typen und Klassen

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

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

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

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

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

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

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

Wahrheitswert

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

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

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

Namen

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

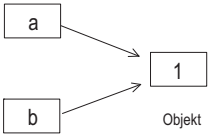

Abbildung 1.1: Variablen als Namen für Objekte

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

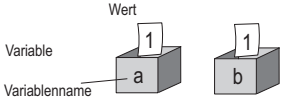

Abbildung 1.2: Variablen als Behälter für Werte

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

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

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

1.7 Die Standard-Typ-Hierarchie

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

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

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

1.8 Literale für einfache Datentypen

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

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

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

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

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

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

None

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

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

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

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

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

Zahlen

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

Ganze Zahlen

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

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

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

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

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

Dezimalzahlen

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

```
>>> 00000
0
```

Oktalzahlen

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

Hexadezimalzahlen

Das Hexadezimalsystem verwendet 16 Ziffern, die durch die üblichen Dezimalziffern `0` bis `9` und die sechs ersten Buchstaben des Alphabets dargestellt werden. Dabei repräsentiert `A` den numerischen Wert `10`, `B` den Wert `11` usw. und schließlich `F` den Wert `15`. Bei Python beginnen Hexadezimal-Literale mit dem Präfix `0x` oder `0X`, wobei das erste Zeichen wiederum die Ziffer null (und nicht der Buchstabe O) ist. Beispiele für Hexadezimalzahlen sind `0x10e3` oder `0XD423f2`, nicht aber `ox3F` (Buchstabe `o` am Anfang).

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

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

Binärzahlen

Analog zu Hexadezimal- und Oktalzahlen werden Literale für Binärzahlen gebildet. Sie beginnen mit einer Null, danach kommt der Buchstabe `b` bzw. `B` und eine Folge von Binärziffern, z.B. `0B101` oder `0b111`.

Gleitkommazahlen

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

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

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

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

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

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

$z = m * 10^e$

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

Gültige Literale sind:

`1.0e-10` entspricht der Zahl 0.0000000001

`2.1E+7` entspricht der Zahl 21000000

`.2e0` entspricht der Zahl 0.2

`001e2` entspricht der Zahl 100

Ungültig sind:

`0x10E1` (Hexadezimalzahl als Mantisse)

`0.1-E7` (Minuszeichen vor dem E)

`1.2e0.3` (keine ganze Zahl als Exponent)

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

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

```
>>> 1.2345678901234567890
1.2345678901234567
```

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

```
>>> 0.6 / 3
0.1999999999999999
```

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

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

Imaginäre Literale

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

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

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

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

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

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

Wahrheitswerte

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

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

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

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

1.9 Namensräume – lokale und globale Namen

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

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

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

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

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

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

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

Namensräume können auf verschiedene Weise erweitert werden:

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

Probieren Sie im interaktiven Modus Folgendes aus:

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

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

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

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

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

Die global-Anweisung

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

Betrachten Sie folgendes Beispiel:

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

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

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

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

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

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

S Stichwortverzeichnis

H

I

M

N

S