# 3 Python-Programmierung

Die Unix-Shellprogrammierung ist geeignet, um einfache Aufgaben rund um das Dateisystem, die Benutzerverwaltung und Systemadministration schnell umzusetzen. Shellprogrammierung ist weniger geeignet, wenn die Aufgabe, die Sie programmieren wollen, entweder relativ komplex ist oder viel Interaktion mit anderen Programmen erfordert. Hierfür ist die Verwendung einer höheren getypten Programmiersprache sinnvoll.

Höhere Programmiersprachen, die der Shell-Programmierung noch im weitesten Sinne ähneln und die für diese Aufgaben in Betracht kämen, wären Skriptsprachen[1] wie Perl, Tcl, Ruby oder Python.Insbesondere Python eignet sich hervorragend für Programmieranfänger: Die Syntax ist einfach und klar und Python vereint die Paradigmen der prozeduralen, der funktionalen und der objektorientierten Programmierung. Besonders für Programmieranfänger sehr vorteilhaft ist die interaktive Interpreterumgebung, die Python anbietet, die Python-Shell. Ähnlich wie wir im vorherigen Kapitel das Programmieren in der Bash erlebt haben, stellt sich auch der Umgang mit Pythonkommandos in der Python-Shell dar.

Man kann dieses Kapitel deshalb als Anknüpfung an das vorige Kapitel betrachten: Python kann als flexiblerer Ersatz für Shellskripte dienen. Für die im weiteren Verlauf des Buches behandelten Konzepte, die weit über die Benutzerinteraktion mit einem Betriebssystem hinausgehen, wird die Programmiersprache Python als Grundlage dienen, nämlich für:

1. die wichtigsten Programmierparadigmen: Prozedurale Programmierung, Funktionale Programmierung und Objektorientierte Programmierung
2. die Programmierung mit regulären Ausdrücken
3. die Datenpersistenz und Datenbanken
4. das Internet und Internetprogrammierung
5. die Parallele Programmierung

Die in diesem Kapitel dargestellte Python-Einführung bietet einen Überblick über eine Teilmenge aller Sprachkonstrukte, die Python anbietet. Diese Einführung hat weniger das Ziel eine vollständige Präsentation aller Möglichkeiten in Python zu bieten, sondern mehr, die wichtigsten Konstrukte einer höheren Programmiersprache kennen zu lernen und für die folgenden Kapitel gewappnet zu sein. Es gibt zahlreiche frei ver-

**1** Skriptsprachen sind Programmiersprachen, die eigentlich für eher kleinere, überschaubare Aufgaben gedacht sind. Sie verzichten auf manche Sprachelemente, wie etwa die Deklaration von Variablennamen. Außerdem sind die meisten Skriptsprachen interpretiert, im Gegensatz zu C oder C++, die compiliert sind. Einige Skriptsprachen wie Python haben sich aber auch in großen Softwareprojekten bewährt.

fügbare Quellen im Internet, die komplettere Pythoneinführungen bieten [10, 12] und einen Überblick über die verfügbaren Module[2][9].

## 3.1 Arbeiten mit Python

Python ist eine besonders für Programmieranfänger geeignete Programmiersprache. Im Gegensatz zur Bash-Programmiersprache werden in Python jedoch große Softwareprojekte erstellt, und viele Spezialisten im Umfeld Data Science, Forensics, Web-Development und Ingenieurwissenschaften schreiben komplexe Algorithmen mit Python. Entsprechend vielfältig sind auch die Entwicklungswerkzeuge und der ganze „Zoo" an Bibliotheken und Versionen drumherum, die für die Python-Programmierung zur Verfügung stehen. Dieser Abschnitt gibt einen Überblick über die Entwicklungsumgebungen und die Versionsvielfalt sowie Empfehlungen, wo Sie als Programmieranfänger starten sollten und wie sie Python verwenden sollten, um die in diesem Kapitel vorgestellten Konzepte auszuprobieren und die Übungsaufgaben zu lösen.

### 3.1.1 Python 3 vs. Python 2

Interessiert man sich dafür, Python zu installieren, so wird man mit der Frage konfrontiert, ob man Python 3.*x* oder Python 2.*x* installieren und verwenden möchte. Hierbei steht das „*x*" für die Subversionsnummer. Die Python-2.*x*-Versionen sind alle rückwärtskompatibel. Was heißt das genau? Betrachten wir dafür zwei Python-Interpreter, einen der Python 2.*x* unterstützt und einen der Python 2.*y* unterstützt; nehmen wir weiter an, dass $x < y$, d.h., dass Python 2.*y* eine neuere Pythonversion ist. Rückwärtskompatibilität bedeutet, dass der Python-2.*y*-Interpreter immer auch fehlerfrei Programme ausführen kann, die für einen Python-2.*x*-Interpreter geschrieben wurden. Andersherum gilt dies natürlicherweise nicht: Ein älterer Python-2.*x*-Interpreter kann nicht alle Programme „verstehen", die für einen neueren Python-Interpreter geschrieben wurden.

Im Jahr 2008 erschien eine neue Python-Version mit der Versionsnummer 3. Guido van Rossum, der Erfinder der Programmiersprache Python, entschied sich dafür Python grundlegender zu überarbeiten. Obwohl Python 3 sich aus Sicht des Programmieranfängers nur unwesentlich von Python 2 unterscheidet, waren die Änderungen dann doch so groß, dass eine Rückwärtskompatibiltät nicht mehr sichergestellt werden konnte. In der Folge entwickelten sich die beiden Versionsstränge Python 2.*xx* und Python 3.*x* parallel weiter, und noch bis heute sind nicht alle (aber fast alle)

**2** Ein Pythonmodul ist ein Sammlung von Klassen-, Funktions- und Konstantendefinitionen, die durch ein Importieren des Moduls verwendet werden können.

Python-Bibliotheken auf die Python-3-Version übertragen. Die Python-2.*x*-Versionen werden allerdings nur noch wenige Jahre lang weiterentwickelt und gewartet, und so ist es durchaus sinnvoll, sich als Programmieranfänger die Python-3-Version zu besorgen. Dieses Buch verwendet ausschließlich Python 3.

### 3.1.2 Installation

Die einfachste Art, die Entwicklungsumgebung, die wir für das Erlernen von Python vorschlagen, zu installieren besteht darin, sich die Python-Distribution „Anaconda" zu installieren. Es handelt sich um eine frei verfügbare Sammlung von Packeten und die Notebook-Umgebung (siehe Abschnitt 3.1.5). Anaconda ist über

`https://www.continuum.io/downloads`

beziehbar. Für dieses Buch benötigen wir die Python 3.*x*-Installation (zu der Zeit, in der dieses Buch geschrieben wurde, war $x = 5$).

Die Installation dürfte einige Zeit in Anspruch nehmen. Nach der Installation von Anaconda stehen Ihnen alle Werkzeuge, die in diesem Buch verwendet werden, zur Verfügung.

### 3.1.3 Ein erstes Python-Programm

Wir schreiben nun unser erstes Python-Programm, öffnen hierzu die Bash und erzeugen eine Textdatei `HalloWelt.py`, die das Pythonkommando **`print`** enthält. Diese Datei führen wir anschließend aus. Das Zeichen `$` soll hierbei das Prompt der Bash sein und andeuten, dass die folgenden Zeilen interaktiv der Bash übergeben werden sollen:

```
$ echo 'print("Hallo Welt")' >>HalloWelt.py
$ python HalloWelt.py
Hallo Welt
```

Um komplexere Python-Programme zu erstellen, ist es ratsam einen Texteditor zu verwenden, der Syntaxhervorhebungen unterstützt. Hier bieten sich etwa `vi` oder `emacs` an, oder eine sogenannte integrierte Entwicklungsumgebung (auch: IDE vom englischen Integrated Development Environment) wie Spyder oder PyCharm.

Wollen wir den String `'Hallo Welt'` in der interaktiven Python-Shell ausgeben, so verfahren wir wir folgt (hierbei soll > das Prompt der Bash und `>>>` das Prompt der Python-Shell sein).

```
$ python
>>> print("Hallo Welt")
Hallo Welt
```

### 3.1.4 Die Python-Shell

Ähnlich wie mit Shell-Befehlen unter Unix, kann man Python auch in einer interaktiven Umgebung verwenden, die zunächst einem Taschenrechner vergleichbar zu funktionieren scheint. Sie erwartet die Eingabe eines Ausdrucks, berechnet das Ergebnis dieses Ausdrucks und gibt die String-Repräsentation dieses Ausdrucks aus. Eine Umgebung, die eine solche Funktionsweise anbietet, nennt man auch REPL-Umgebung (von Read-Evaluate-Print-Loop). In die REPL-Umgebung gelangt man, indem man den Befehl `python` in der Kommandozeile einer Shell ausführen lässt, so wie in folgendem Beispiel:

```
$ python
>>> print("Hallo")
Hallo
>>> 2+5
7
```

Man kann auch ein Python-Programm in die interaktive Python-Shell laden und etwa eine in dem Skript definierte Funktion `someFunction` testen. Dies geschieht mittels der Option `-i`:

```
$ python -i test.py
>>> someFunction
Hallo Welt
```

### 3.1.5 Python Notebooks

Die Python-Shell ist ein hervorragendes Werkzeug gerade für Programmieranfänger. Sie erlaubt es, Python-Programme direkt zu testen und mit den selbst geschriebenen Funktionen zu interagieren und zu experimentieren, Objekte auf ihre Eigenschaften hin zu untersuchen und ihre Dokumentation zu lesen und zu durchsuchen. Wenn man allerdings viele Zeilen Programmcode in der Shell schreibt und später einen Teil des Codes ändern möchte, so werden die Einschränkungen der Shell offensichtlich: Man muss mittels der ↑-Taste in der Kommandozeilen-Historie zurückblättern, und alle Programmzeilen bis auf die geänderte Programmzeile nochmals ausführen.

Schreibt man eine Skript-Datei, also eine Textdatei, die Python-Befehle enthält, hat man diesen Nachteil nicht und noch dazu den Vorteil, sich den Programmcode gemäß seiner Syntax farblich markiert anzeigen lassen zu können (Syntax-Highlighting), was die Lesbarkeit des Programm-Codes erleichtert. Syntax-Highlighting bieten alle gängigen integrierten Entwicklungsumgebungen ebenso wie die Editoren `vi` und `emacs`. Beim Arbeiten mit einer Skript-Datei verliert man jedoch auch die Möglichkeit der direkten Interaktion mit dem Programmcode, den die Python-Shell bietet.

Python Notebooks bringen die Vorteile beider Seiten zusammen: Notebooks bieten eine interaktive REPL-Umgebung, ermöglichen eine einfache Modifikation aller bisherigen Kommandos und erlauben darüberhinaus eine komfortable Möglichkeit der Dokumentation, die weit über die Möglichkeiten der üblichen Kommentar-Erstellung hinausgeht. Eine „Zelle“ in einem Notebook enthält standardmäßig Python-Code, der in der üblichen REPL-Manier ausgewertet und ausgedruckt wird; man kann jede Zelle jedoch leicht in eine Kommentarzelle umwandeln, die die Eingabe von Markdown-Text und sogar LaTeX-Formeln erlaubt. Zudem können Notebooks gespeichert und entsprechend mit anderen geteilt werden. Diese Eigenschaften machen Notebooks zum idealen Werkzeug für Python-Anfänger und für die Präsentation von Programmierkonzepten. Wir empfehlen die Verwendung von Notebooks zum Experimentieren mit Python-Code und zum Lösen der in diesem Buch enthaltenen Aufgaben.

Wenn Sie Anaconda wie in Abschnitt 3.1.2 empfohlen installiert haben, dann können sie ein Notebook über die folgende Bash- oder Windows-Kommandozeile starten.

```
$ jupyter notebook
[NotebookApp] 0 active kernels
[NotebookApp] The Jupyter Notebook is running at:
            http://localhost:8888/
```

Es wird ein lokaler Webserver gestartet. Dieser kommunizert die im Notebook eingegebenen Kommandos an einen Python-Kernel, einer Instanz des Python-Interpreters, und leitet die Ausgaben dieser Interpreter-Instanz wiederum an den Webserver zurück. Das `jupiter`-Kommando öffnet dann ein Browser-Fenter mit der Notebook-Oberfläche. Abbildung 3.1 zeigt ein Beispiel einer solchen Oberfläche nach Eingabe einiger Kommandos und einiger Markdown-Kommentare.

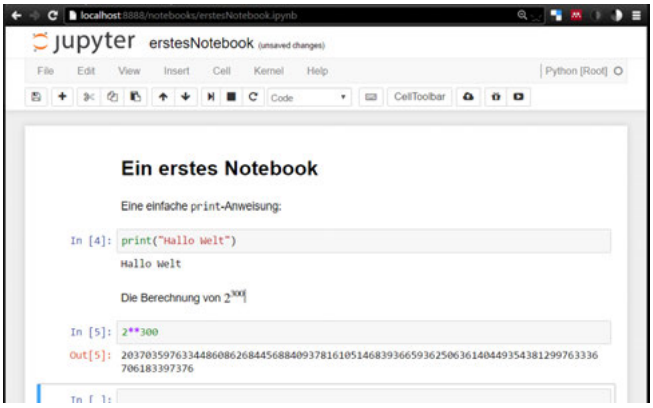



**Abb. 3.1.** Ein Python-Notebook im Fenster eines Web-Browsers mit einigen Kommando-Zellen und einigen Dokumentations-Zellen.

**Aufgabe 3.1**

Öffnen Sie ein Python-Notebook und machen Sie sich mit der Funktionsweise der Benutzeroberfläche vertraut, indem Sie unter dem Menüpunkt „Help“ die „User Interface Tour“ wählen und vollständig durchlaufen. Informieren Sie sich anschließend über die Formatierung der Markdown-Zellen. Informationen darüber finden Sie ebenfalls im Menüpunkt „Help“.

Geben Sie nun die Befehle und Markdown-Kommentare so ein, dass Ihr Notebook dem aus Abbildung 3.1 gleicht.

## 3.2 Einfache Datentypen

Die Bash kennt nur einen Datentyp, nämlich Strings. Python hingegen kennt eine ganze Reihe unterschiedlicher, teilweise auch aus einfacheren Datentypen zusammengesetzten Datentypen. In diesem Abschnitt wollen wir zunächst die einfachen Datentypen betrachten, die Python unterstützt, nämlich Zahlen, Boolesche Werte und Strings.

### 3.2.1 Zahlen

In Python gibt es vier Zahlen-Typen: Ganzzahlen (`int`), lange Ganzzahlen mit beliebig vielen Stellen (`long int`), Gleitpunktzahlen (`float`) und komplexe Zahlen (`complex`). Beispiele für Python-Zahlen: `12`, `3.141`, `4.23E-5` (Gleitpunkt-Darstellung), `0xFE` (hexadezimale Darstellung), `3/4` (Bruchzahlen), `2**100`, `1j+3` (komplexe Zahlen).

**Aufgabe 3.2**

Verwenden Sie ein Python-Notebook, um folgende Berechnungen durchzuführen:

(a) $2^{20}$
(b) $(12 - 5i) \cdot (4 + 3i)$
(c) $(2 + i)^{(2+i)}$

### 3.2.2 Boolesche Werte

Noch einfacher strukturiert sind Boolesche Werte vom Typ `bool`, nämlich `True` und `False`. Sie sind auch Ergebnis von Vergleichsoperationen, wie in folgendem Beispiel gezeigt.

```
>>> type(True)
bool
>>> 4 < 3
False
```

### 3.2.3 Strings

Strings sind Sequenzen einzelner Zeichen. Im Gegensatz zu Listen und Dictionaries (die wir später ausführlich behandeln) sind Strings *unveränderlich*, d. h. ist ein bestimmter String einmal definiert, so kann er nicht mehr verändert werden. Man hat die Wahl, Strings entweder in doppelte Anführungszeichen (also: `"..."`) oder in einfache Anführungszeichen (also: `'...'`) zu setzen. Die spezielle Bedeutung der Anführungszeichen kann, ganz ähnlich wie in der Bash, mit dem Backspace (also: \) entfernt werden. Syntaktisch korrekte Python-Strings wären demnach beispielsweise: `"Hallo"`,`'Hallo'`, `'"Hallo"'`, `'\'\''`, `"Python's"`, `'Hallo Welt'`, usw.

Verwendet man dreifache Anführungszeichen (also: `"""..."""` oder `'''...'''`), so kann man auch mehrzeilige Strings definieren.

**Aufgabe 3.3**

Geben Sie mit dem Python `print`-Kommando den folgende Text aus:

```
Strings in Python koennen entweder mit "double ticks" oder
mit 'einfachen ticks' umschlossen werden.
```

### 3.2.4 Variablen

Variablen sind, genau wie in anderen Programmiersprachen auch, (veränderliche) Platzhalter für bestimmte Werte. Variablennamen müssen mit einem Buchstaben oder mit dem Zeichen „_“ beginnen und dürfen keine Leerzeichen oder Sonderzeichen (außer eben dem Zeichen „_“) enthalten. Korrekte Variablennamen sind beispielsweise `i`, `_i`, `Kaese`, `kaese`; die Zeichenketten `2dinge` oder `leer zeichen` wären beispielsweise keine korrekten Variablennamen.

Python ist, im Gegensatz zu vielen gängigen Programmiersprachen, nicht statisch getypt; d. h. der Typ einer Variablen muss nicht schon vor der Ausführung eines Programms festgelegt sein, sondern er wird dynamisch, also während der Programmausführung, bestimmt.

Das hat den Vorteil, dass Variablen nicht deklariert werden müssen. Variablendeklarationen in anderen Programmiersprachen dienen ja hauptsächlich dazu, den Typ

einer Variablen vor Programmausführung festzulegen. In Python kann man Variablen einfach ohne vorige Deklaration einen Wert zuweisen, wie in folgendem Beispiel:

```
x = 2j +4
```

Der Python-Interpreter leitet den Typ der Variablen aus der ersten Zuweisung ab. Außerdem kann die Verwendung von Variablen grundsätzlich flexibler erfolgen als bei statisch getypten Programmiersprachen. Folgendes Beispiel zeigt, dass eine Variable sogar abhängig von einer Programm-abhängigen Bedingung einen Wert eines anderen Typs erhalten kann. Das Beispiel verwendet die `if`-Anweisung, die später in Abschnitt 3.3.2 eingeführt wird.

```
if gespraechig:
  x = "Guten Morgen"
else:
  x = 12**12
print(x)
```

### 3.2.5 Operatoren

Die folgende Tabelle zeigt eine Auswahl an Operatoren, die Python anbietet, um Ausdrücke zu verknüpfen. Einige Operatoren verlangen, dass Ihre Parameter einen bestimmten Typ haben, etwa einen Sequenztyp, d.h. eine Sammlung von Werten in einer bestimmten Reihenfolge (z.B. Listen), oder eine Kollektion von Werten, d.h. eine Sammlung von Werten, wobei die Werte nicht notwendigerweise eine bestimmte Reihenfolge haben müssen (z.B. Mengen). Einige der Operatoren sind polymorph, d. h. auf unterschiedliche Typen anwendbar. So kann beispielsweise der Operator + sowohl auf Strings, als auch auf Integer, als auch auf Komplexe Zahlen angewendet werden. Jeder dieser Anwendungsmöglichkeiten liegt jedoch ein anderer Algorithmus zugrunde.

| `X + Y`, `X - Y` | Plus/Konkatenation, Minus | | |
|---|---|---|---|
| Beispiele: | `>>> 2+3`<br>`5` | `>>> '2'+'3'`<br>`'23'` | `>>> [1,2,3]+[10]`<br>`[1,2,3,10]` |
| `X * Y`, `X ** Y` | Multiplikation / Vervielfachung, Potenzierung | | |
| Beispiele: | `>>> 2*6`<br>`12` | `>>> '2'*6`<br>`'222222'` | `>>> [0,1]*3`<br>`[0,1,0,1,0,1]` |
| `X / Y`, `X // Y`<br>`X % Y` | Division, restlose Division<br>Rest (bei der Division) | | |

| | |
|---|---|
| Beispiele: | `>>> 2.0/3` `0.66666666` `>>> 2/3` `0` `>>> 17%7` `3` |
| `X<Y, X <= Y`<br>`X>Y, X >= Y` | Kleiner, kleinergleich (lexikografisch bei Sequenzen).<br>Größer, größergleich (lexikografisch bei Sequenzen)./ |
| Beispiele: | `>>> 4<2` `False` `>>> 'big'<'small'` `True` `>>> [1,100]<[2,1]` `True` |
| `X == Y, X != Y`<br>`X is Y, X is not Y`<br>`X & Y, X \| Y, X ^ Y`<br>`~X`<br>`X << Y, X >> Y` | Gleichheit, Ungleichheit (Werte)<br>Objektgleichheit, Objektungleichheit<br>bitweises „Und“, bitweises „Oder“, bitweises exkl. „Oder“;<br>bitweise Negation.<br>Schiebe `X` nach links, rechts um `Y` Bits. |
| Beispiele: | `>>> 9 & 10` `8` `>>> 10 \| 6` `14` `>>> 3 << 4` `48` |
| `X & Y`<br><br>`X \| Y`<br><br>`X ^ Y` | Bitweise Und-Verknüpfung: Die einzelnen Bits der Binärdarstellungen der Zahlen `X` und `Y` werden Und-verknüpft.<br>Bitweise Oder-Verknüpfung: Die einzelnen Bits der Binärdarstellungen der Zahlen `X` und `Y` werden Oder-verknüpft.<br>Bitweise Exklusiv-Oder-Verknüpfung: Die einzelnen Bits der Binärdarstellungen der Zahlen `X` und `Y` werden Exklusiv-Oder-verknüpft, d. h. es kommt eine `1` als Ergebnis, wenn genau eines der entsprechenden Bits aus `X` oder `Y` eine `1` enthält. |
| Beispiele | `12 ≙ 0b1100`<br>`6 ≙ 0b110`<br><br>`>>> 12 ^ 6`<br>`10`<br><br>`>>> 12 & 6`<br>`4`<br><br>`>>> 12 \| 6`<br>`14` |
| `X and Y`<br>`X or Y`<br>`not X`<br>`X in S`<br><br>`X not in S` | Wenn `X` falsch, dann `X`, andernfalls `Y`;<br>wenn `X` falsch, dann `Y`, andernfalls `X`;<br>wenn `X` falsch, dann `True`, andernfalls `False`;<br>Test auf Enthaltensein eines Elements `X` in einer Sequenz oder Kollektion `S` von Werten;<br>Test auf Nicht-Enthaltensein eines Elements `X` in einer Sequenz oder Kollektion `S` von Werten. |

| Beispiele: | >>> True and False<br>False<br><br>>>> 4 in [1,2]<br>False | >>> 'he' in 'hei'<br>True |
|---|---|---|

Der Vergleich bei Sequenzen erfolgt *lexikografisch*, d.h. nach den selben Prinzipien, wie wir es gewohnt sind, Wörter in einem Wörterbuch zu ordnen: Zunächst wird nach dem ersten Buchstaben (bzw. dem ersten Element der Sequenz) sortiert; sind ersten Buchstaben zweier Wörter (bzw. die ersten beiden Elemente zweier Sequenzen) identisch, so erfolgt die Sortierung nach den zweiten Buchstaben (bzw. nach den zweiten Elementen der beiden Sequenzen), usw.

**Aufgabe 3.4**

(a) Was hat `+` für eine Bedeutung für Strings?
(b) Was hat `*` für eine Bedeutung für Strings?
(c) Was ist der Unterschied zwischen dem Ausdruck `4 & 5` und dem Ausdruck `4 and 5`?
(d) Finden Sie einen Fall, bei dem der Vergleichsoperator == und der Operator `is` unterschiedliche Ergebnisse liefern.
(e) Geben Sie ein Beispiel an für die Verwendung des Operators `in`.

## 3.3 Grundlegende Konzepte

### 3.3.1 Einrücktiefe

Eine grundlegende und in vielen anderen Programmiersprachen nicht gängige Eigenschaft von Python ist die Bedeutung der *Einrücktiefe*, also der am Zeilenanfang befindlichen Leerzeichen. Diese Leerzeichen gehören zur Syntax, und der Python-Interpreter liefert eine Fehlermeldung, wenn diese nicht richtig gesetzt sind. Anweisungen, die zusammengehören, müssen die gleiche Einrücktiefe haben.

Der Anweisungsblock, der einer `if`-Anweisung, `while`-Anweisung oder `for`-Anweisung folgt, wird also nicht explizit eingeklammert, wie es in vielen anderen Programmiersprachen wie C, C++ oder Java üblich ist, sondern die Anweisungen werden durch den Python-Interpreter durch ihre Einrücktiefe als zum selben Anweisungsblock gehörig erkannt.

### 3.3.2 Kontrollfluss

Üblicherweise werden die Kommandos in einem Python-Skript einfach sequentiell, also nacheinander, beginnend von der ersten Programmzeile bis zur letzten, abgearbeitet. Es gibt jedoch einige Befehle, die diesen Kontrollfluss umleiten können, etwa wenn bestimmte Bedingungen erfüllt sind, oder während die Elemente einer Sequenz durchlaufen werden. Python bietet drei Kommandos zur Steuerung des Kontrollflusses an: die `if`-Anweisung, die `for`-Anweisung und die `while`-Anweisung. Diesen Anweisungen ist gemein, dass ihnen ein Anweisungsblock folgt; im Falle der `if`-Anweisung können es auch mehrere Anweisungsblöcke sein, falls der `else`-Zweig oder der `elif`-Zweig verwendet wird.

Betrachten wir nun die Syntax dieser Anweisungen:

**Die `if`-Anweisung**

```
if ⟨test⟩:
    ⟨Anweisungsfolge⟩
[elif ⟨test⟩:
    ⟨Anweisungsfolge⟩ ]*
[else:
    ⟨Anweisungsfolge⟩ ]
```

Die `if`-Anweisung wählt eine aus mehreren Anweisungsfolgen aus. Ausgewählt wird diejenige Anweisungsfolge, die zum *ersten* ⟨*test*⟩ mit wahrem Ergebnis gehört; die `else`-Klausel entspricht einem `elif True`.

**Die `while`-Anweisung**

```
while ⟨test⟩:
    ⟨Anweisungsfolge⟩
[else:
    ⟨Anweisungsfolge⟩ ]
```

Die `while`-Anweisung stellt die allgemeinste Schleife dar. Die erste ⟨*Anweisungsfolge*⟩ wird solange ausgeführt, wie ⟨*test*⟩ wahr ergibt. Die zweite ⟨*Anweisungsfolge*⟩ wird ausgeführt, sobald die Schleife normal (d. h. ohne Verwendung der `break`-Anweisung) verlassen wird.

**Die `for`-Anweisung**

```
for ⟨variable⟩ in ⟨sequenz⟩:
    ⟨Anweisungsfolge⟩
[else:
    ⟨Anweisungsfolge⟩ ]
```

Die `for`-Anweisung ist eine Schleife über Sequenzen, wie Listen, Tupel oder Strings. Pythons `for`-Anweisung funktioniert ähnlich der `for`-Anweisung der Bash: Die Anzahl der Durchläufe entspricht immer der Länge der Sequenz ⟨*sequenz*⟩, und in jedem Durchlauf nimmt die Variable ⟨*variable*⟩ den Wert eines Elements in der Sequenz an.

Ist ein Teil der Syntax in obigen Beschreibungen in eckigen Klammern eingeschlossen (also: [...]), so bedeutet dies, dass der entsprechende Teil optional ist. Ist der in eckigen Klammern eingeschlossene Teil von einem Stern gefolgt (also: [...]*), so bedeutet

dies, dass der entsprechende Teil beliebig oft (auch 0-mal) wiederholt werden kann. Beispielsweise kann der `elif`-Teil der `if`-Anweisung beliebig oft (und eben auch 0-mal) hintereinander verwendet werden.

### Beispiele

Das in Listing 3.22 gezeigte Programm verwendet die `while`-Schleife und ein `if`-Kommando, um ein einfaches Ratespiel zu implementieren. Abbildung 3.2 zeigt ein Ablaufdiagramm, das den Kontrollfluss des Skripts aus Listing 3.22 veranschaulicht.

```
zahl = 23
weiter = True

while weiter:
    geraten = int(input('Zahl eingeben:'))
    if geraten == zahl:
        print('Super, richtig geraten')
        weiter = False
    else:
        print('Falsch geraten')
else:
    print('While-Schleife wurde beendet')
```

**Listing 3.22.** Implementierung eines einfachen Ratespiels als Beispiel für die Verwendung einer while-Schleife.

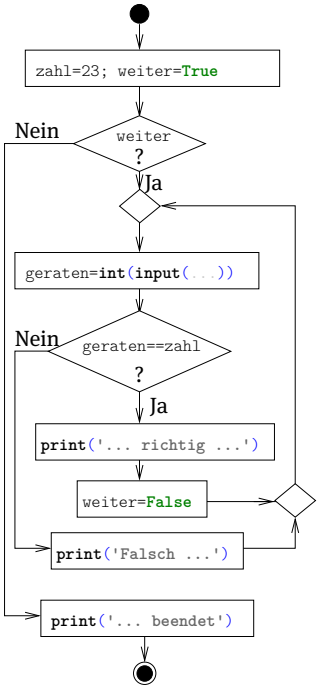



**Abb. 3.2.** Ablaufdiagramm, das den Kontrollfluss des Skripts aus Listing 3.22 veranschaulicht.

Die `while`-Schleife führt die Kommandos des folgenden Kommando-Blocks (die Kommandos bis einschließlich der drittletzten Zeile) solange aus, bis der Boolesche Ausdruck im Argument von `while` den Wert `False` annimmt. Das ist dann der Fall, wenn der erste Block des `if`-Kommandos ausgeführt wird, dessen letztes Kommando die Zuweisung des Booleschen Wertes `False` an die Variable `weiter` darstellt.

Beachten Sie in diesem Zusammenhang den Unterschied zwischen der Zuweisung „=“ und dem Operator „==“. Der Operator „==“ ist Teil eines Booleschen Ausdrucks,

der für einen der Werte `True` oder `False` steht. Dagegen ist die Zuweisung „=“ kein Ausdruck (steht also auch nicht für einen bestimmten Wert), sondern ein Kommando, dass einer Variablen einen Wert zuweist.

Folgende im Programmbeispiel verwendete Funktionen sind noch nicht bekannt:

- Die Python-Funktion `input(`⟨*string*⟩`)` liest einen String von der Tastatur ein (ganz ähnlich wie das `read`-Kommando der Bash) und verwendet ⟨*string*⟩ als Eingabeaufforderung.
- Die Python-Funktion `int(`⟨*ausdruck*⟩`)` konvertiert ⟨*ausdruck*⟩ in eine `int`-Zahl.

**Aufgabe 3.5**

(a) Tippen Sie das vorige Beispiel in die Zelle eines Python-Notebooks, und führen Sie die Zelle aus.

(b) Erweitern Sie das Programm so, dass statt der Ausgabe „`Die while-Schleife wurde beendet`“ eine Ausgabe erfolgt, die informiert, wie oft sie geraten haben (etwa „`Sie haben 6 Rate-Versuche gebraucht.`“).

(c) Erweitern Sie das Programm so, dass das Ratespiel vier mal mit vier unterschiedlichen Zahlen abläuft; am Ende sollen Sie über den besten Rate-Lauf und den schlechtesten Rate-Lauf informiert werden, etwa so:
`Ihr schlechtester Lauf: 8 Versuche; ihr bester Lauf: 3 Versuche.`

Listing 3.23 zeigt als weiteres Beispiel ein Programm, das die `for`-Schleife verwendet, um die Zahlen von 1 bis 4 auszugeben:

```
for i in range(1,5):
    print(i)
else:
    print('Die for-Schleife ist zu Ende.')
```

**Listing 3.23.** Ein Beispiel für die Verwendung der for-Schleife: Ein Programm, das die Zahlen von 1 bis 4 ausgibt.

Im Programmbeispiel wird die Python-Funktion `range` verwendet. Diese erzeugt eine Sequenz ganzer Zahlen im angegebenen Bereich. Eine Besonderheit in Python ist die Tatsache, dass Bereichsangaben für ganzzahlige Intervalle immer *exklusive* der rechten Grenze zu interpretieren sind. So erzeugt der Aufruf von `range(`$a$, $b$`)` alle ganzen Zahlen zwischen *einschließlich* $a$ und *ausschließlich* $b$. Zu beachten ist außerdem, dass ein Aufruf von `range` die Sequenz nicht sofort erzeugt, sondern die Elemente nur dann erzeugt werden, wenn sie wirklich gebraucht werden. Man nennt eine solche Auswertungs-Strategie auch *lazy* oder nicht-strikt.

Wendet man die Funktion `list(...)` oder `tuple(...)` auf ein `range`-Objekt an, so „zwingt“ man Python, die spezifizierte Sequenz sofort in Form einer Liste zu erzeugen. Folgendes Beispiel zeigt dieses „Verhalten“ eines `range`-Objekts:

```
>> range(1,5)
range(1, 5)
>> list(range(1,5))
[1, 2, 3, 4]
>> tuple(range(1,5))
(1, 2, 3, 4)
```

Was die Werte betrifft, die ein `range`-Objekt erzeugt, gilt also:

```
list(range(a,b))== [a,a+1,...,b-2, b-1]
```

Optional kann man der `range`-Funktion auch als drittes Argument eine Schrittweite übergeben. Beispielsweise erzeugt `range(1,9,2)` eine Sequenz, die die Werte 1, 3, 5 und 7 enthält. Es gilt also in diesem Fall

```
list(range(a,b,c))== [a,a+c,a+2c, ..., b-2c, b-c]
```

Übergibt man `range` nur ein einziges Argument, so beginnt die Ergebnisliste bei 0. Es gilt also

```
list(range(a))== [0,1,...,a-2, a-1]
```

**Aufgabe 3.6**

Schreiben Sie ein Pythonskript, das die Quadrate aller durch 3 teilbarer Zahlen zwischen 1 und 999 ausgibt.

**Aufgabe 3.7**

(a) Schreiben Sie ein Pythonskript, das die Summe aller Quadratzahlen zwischen 1 und 100 ausgibt.
(b) Schreiben Sie ein Pythonskript, das eine Zahl $n$ von der Tastatur einliest und den Wert $\sum_{i=0}^{n} i^3$ zurückliefert.
(c) Schreiben Sie ein Pythonskript, das zwei Zahlen $n$ und $m$ von der Tastatur einliest und den Wert $\sum_{i=n}^{m} i^3$ zurückliefert.

**Aufgabe 3.8**

Schreiben Sie ein Pythonskript, das alle Primzahlen zwischen 1 und 1000 auf dem Bildschirm ausgibt.

**Aufgabe 3.9**

Schreiben Sie ein Pythonskript, das Ihnen die vier kleinsten perfekten Zahlen ausgibt.
Eine natürliche Zahl heißt perfekt, wenn sie genauso groß ist, wie die Summe Ihrer positiven echten Teiler (d. h. Teiler außer sich selbst). Beispielsweise ist 6 eine perfekte Zahl, da sie Summe ihrer Teiler ist, also 6 = 1 + 2 + 3.

### 3.3.3 Schleifenabbruch

Die beiden im Folgenden vorgestellten Kommandos, `break` und `continue` geben dem Programmierer mehr Flexibilität im Umgang mit Schleifen; man sollte diese aber möglichst sparsam verwenden, denn sie können Programme schwerer verständlich und damit auch schwerer wartbar[3] werden lassen.

Mit der `break`-Anweisung kann man vorzeitig aus einer Schleife aussteigen; auch ein vorhandener `else`-Zweig wird dann nicht mehr gegangen. Ein Beispiel:

```
while True:
  i = int(input('Bitte eine Zahl eingeben: '))
  if i == 0: break
print('Fertig')
```

**Aufgabe 3.10**

Implementieren Sie das vorige Beispiel: wie verhält sich dieses Skript?

Mit der `continue`-Anweisung kann man die restlichen Anweisungen im aktuellen Schleifendurchlauf überspringen und sofort zum Schleifen„kopf“ springen.

### 3.3.4 Anweisungen vs. Ausdrücke

Gerade für den Programmieranfänger ist es wichtig, sich des Unterschieds bewusst zu sein zwischen einer *Anweisung* und eines Ausdrucks. Eine Anweisung, tut etwas, wie etwa den Zustand des Programms bzw. Systems verändern: Dies geschieht etwa durch eine Variablenzuweisung, die Veränderung des Speicherinhalts oder die Ausführung

**3** Spricht man in der Softwaretechnik von Wartbarkeit, so an meint man damit im Allgemeinen die Einfachheit ein Programm im nachhinein anzupassen oder zu erweitern. Je übersichtlicher und besser strukturiert ein Programm bzw. Softwaresystem ist, desto besser wartbar ist es.

einer Bildschirmausgabe. Ein Ausdruck dagegen repräsentiert einen bestimmten Wert durch dessen Berechnung der Zustand des Programms nicht verändert wird.

Einige Beispiele: Der Python-Code `x=5+3` stellt eine Anweisung dar, nämlich die, der Variablen `x` einen Wert zuzuweisen. Die rechte Seite dieser Zuweisung, nämlich `5+3`, ist dagegen ein Ausdruck, der für den Wert `8` steht. Man beachte in diesem Zusammenhang den Unterschied zwischen „=“, das immer Teil einer Zuweisung (also: eines Kommandos) ist und „==“, das einen Vergleich darstellt (also einen Wahrheitswert zurückliefert) und folglich immer Teil eines Ausdrucks ist: Der Python-Code `5==3` ist also ein Ausdruck, der für den Wert `False` steht.

**Aufgabe 3.11**

Viele Anweisungen enthalten Ausdrücke als Komponenten. Gibt es auch Ausdrücke, die Anweisungen als Komponenten enthalten?

In der interaktiven Python-Shell und in Python-Notebooks kann der Programmierer sowohl Anweisungen als auch Ausdrücke eingeben. Die Python-Shell geht aber jeweils unterschiedlich mit diesen um: Wird ein Kommando eingegeben, so führt die Python-Shell das Kommando aus. Wird dagegen ein Ausdruck eingegeben, so wird der Ausdruck zunächst ausgewertet und anschließend die String-Repräsentation des Ausdrucks ausgegeben.

Neben der in Abschnitt 3.3.2 vorgestellten `if`-Anweisung bietet Python auch die Möglichkeit, Ausdrücke mit `if` zu strukturieren.

$$\langle expr_1 \rangle \texttt{ if } \langle condition \rangle \texttt{ else } \langle expr_2 \rangle$$

Dieser Ausdruck steht für den Wert des Ausdrucks $\langle expr_1 \rangle$ falls der Ausdruck $\langle condition \rangle$ dem Wahrheitswert `True` entspricht, andernfalls steht dieser `if`-Ausdruck für den Wert des Ausdrucks $\langle expr_2 \rangle$. Betrachten wir dazu folgende Beispiele:

```
>>> x=3 ; y=4
>>> 'a' if x+1==y else 'b'
a
>>> 'Hallo Welt'[7 if x==y else 9]
e
```

Man sieht: Der `if`-Ausdruck ist ein vollwertiger Ausdruck, der an jeder beliebigen Stelle verwendet werden kann, an der Python einen Ausdruck erwartet.

**Aufgabe 3.12**

Welchen Wert haben die folgenden Python-Ausdrücke. Überlegen Sie erst und überprüfen sie dann mittels Eingabe der Ausdrücke in ein Python-Notebook.

(a) `'Hallo'[4 if (4 if 4==2 else 3)==3 else 5]`
(b) `'Hallo'+'welt' if str(2-1)==str(1)else 'Welt'`
(c) `('hallo'if len('bla')==3 else 'welt')[2 if len('b')==len('w') else 3]`

Was passiert, wenn man die Klammern um den ersten `if`-Audsdruck weglässt?

### 3.3.5 Funktionen

Schreiben Sie komplexere Skripte, so entspricht es einem guten Programmierstil, das Skript in kleinere einfachere Aufgabenstellungen aufzuteilen, und die einfacheren Teile dann so zusammenzusetzen, dass das gewünschte Ergebnis entsteht. Der Grund für diese Empfehlung ist, dass der Mensch kleine Aufgaben viel einfacher durchdenken, überblicken, entwerfen und programmieren kann. Die gängigste Möglichkeit, ein Programm in einfachere Teile aufzuteilen, sind Funktionen. Eine Funktion ist nichts anders als ein Block Python-Code, der durch Aufruf der Funktion ausgeführt wird. Dieser Code-Block hat im Allgemeinen einen Namen und erhält eine Liste von Parametern, denn viele Funktionen sind parametriert. Jede Funktion löst eine einfache Teilaufgabe und am Ende werden die Funktionen dann entsprechend kombiniert, meist durch Hintereinanderausführung.

**Aufgabe 3.13**

Ist eine Funktion eine Anweisung oder ein Ausdruck?

#### 3.3.5.1 Definition mit dem Schlüsselwort `def`

Man kann Funktionen mit Hilfe des Schlüsselwortes `def` definieren. Viele Funktionen liefern nach Beendigung einen bestimmten Rückgabewert zurück. Diesen kann man mit dem Schlüsselwort `return` bestimmen. Nach Ausführung des `return`-Kommandos wird die Funktion verlassen.

Beispielsweise liefert die im folgenden Listing definierte Funktion `getMax(x,y)` die größere der beiden übergebenen Zahlen zurück:

```
def getMax(a,b):
  if a > b:
    return a
  else:
    return b
```

Der Aufruf `getMax(4,10)` würde also den Wert `10` zurückliefern.

Alle in einer Funktion verwendeten Variablen sind *lokal*, d. h. außerhalb der Funktion weder sichtbar noch verwendbar und nur innerhalb der Funktion gültig. Weist man einer bestimmten Variablen, die es im Hauptprogramm bzw. aufrufenden Programm schon gibt, einen Wert zu, so wird die Hauptprogramm-Variable dadurch weder gelöscht noch verändert. In der Funktion arbeitet man auf einer Kopie der Variablen, die von der Variablen des Hauptprogramms entkoppelt ist. Warum ist ein solches Verhalten sinnvoll? In der Informatik hat es sich bewährt, Funktionen inklusive ihrer Wirkungen und Informationen, die sie erzeugen, möglichst gut gegen andere Funktionen und gegen das Hauptprogramm „abzuschotten“. Man bezeichnet dieses Prinzip auch als „Datenkapselung“ oder „Information Hiding“. Lässt man beispielsweise den Code auf der linken Seite durch Python ausführen, so ergibt sich die auf der rechten Seite gezeigte Ausgabe:

```
def func(x):
  print('x ist' +str(x))
  x=2
  print('Lokales x ist jetzt' +
str(x))

x=50
func(x)
print('x ist immer noch' +str(x))
```

erzeugt ⟹

```
x ist 50
Lokales x ist jetzt 2
x ist immer noch 50
```

Solange `x` kein neuer Wert zugewiesen wurde, wird das `x` aus dem Hauptprogramm verwendet; erst nach der Zuweisung wird ein „neues“ lokales `x` in der Funktion verwendet, die vom `x` des Hauptprogramms abgekoppelt ist. So wird sichergestellt, dass der Wert der Variablen `x` des Hauptprogramms nicht überschrieben wird und nach dem Funktionsaufruf wieder verfügbar ist.

Möchte man diesen Mechanismus umgehen und nicht auf einer lokalen Kopie arbeiten, so muss man in der Funktion die Variable `x` mit dem Schlüsselwort **`global`** deklarieren.

Eine Besonderheit von Python sind die sogenannten *DocStrings*: Die Zeile, die direkt auf die mit dem Schlüsselwort **`def`** eingeleitete erste Zeile der Funktionsdefinition folgt, kann einen ggf. mehrzeiligen String enthalten. Auf diesen String kann man über das Attribut `__doc__` direkt zugreifen oder mittels der **`help`**-Funktion. Beispiel:

```
>>> def getMax(a,b):
      """Liefert das Maximum zweier Zahlen"""
      # Body der Funktion
>>> help(getMax)
Help on function getMax in module __main__:
```

```
getMax(a, b)
    Liefert das Maximum zweier Zahlen,
    das mittels eines Vergleichs bestimmt wurde.
```

**Aufgabe 3.14**

Schreiben Sie ein Pythonskript, das Ihnen alle Zwillingsprimzahlen zwischen 1 und 1000 ausgibt. Verwenden Sie dazu eine (selbstgeschriebene) Funktion `isPrim(n)`, die `True` zurückliefert, wenn `n` eine Primzahl ist, und andernfalls `False` zurückliefert.

#### 3.3.5.2 Definiton mit einem `lambda`-Ausdruck

Es gibt eine weitere Art, Funktionen zu definieren und zwar mit sogenannten `lambda`-Ausdrücken. Besonders dann, wenn man die Möglichkeiten der Funktionalen Programmierung in Python nutzt, weiß man die Definition einer Funktion durch einen `lambda`-Ausdruck zu schätzen. Das hängt damit zusammen, dass man häufig eine Funktion einmalig verwenden möchte, etwa als Übergabeparameter einer `map`, `filter` oder `reduce`-Funktion; siehe hierzu auch die Abschnitte 3.5.2, 3.5.3 und 3.5.4. Genau das ist mit einem `lambda`-Ausdruck mit der folgenden Syntax einfach möglich.

$$\texttt{lambda}\ \langle parameter_1\rangle, \langle parameter_2\rangle, \ldots : \langle ausdr\rangle$$

Dem Schlüsselwort `lambda` folgt eine durch Kommata getrennte Liste von Parametern, die mit einem Doppelpunkte abgeschlossen ist. Der darauffolgende Ausdruck ist der Rückgabewert der Funktion. Der `lambda`-Ausdruck selbst ist lediglich ein Funktionsobjekt; um den Rückgabewert zu berechnen, muss die Funktion zunächst mit Parametern aufgerufen werden.

Betrachten wir nun im Folgenden einige Beispiele:

```
>>> f = lambda x,y: x+y
>>> f(2,3)
5
```

Die erste Zeile definiert `f` mittels eines `lambda`-Ausdrucks als Funktion, die zwei Parameter `x` und `y` erwartet und deren Summe zurückliefert. Diese Definition von `f` ist äquivalent zur folgenden Funktionsdefinition von `f2` mittels des Schlüsselworts `def`:

```
>>> def f2(x,y): return x+y
>>> f2(2,3)
5
```

Ein mittels `lambda`-Ausdruck definiertes Funktionsobjekt kann auch ohne den Umweg einer Variablenzuweisung direkt als Funktion verwendet werden:

```
>>> (lambda x,y,z: x in z or y in z)('a','Hallo','Welt')
True
```

Funktionen können selbst wiederum Funktionen als Argument erwarten. Solche Funktionen nennt man auch *Funktionen höherer Ordnung*:

```
>>> (lambda x,y,z: x(y) +x(z))(lambda a: a+1, 10,11)
23
```

Durch die Erzeugung einer unbenannten Funktion mittels eines `lambda`-Ausdrucks ist es bequem möglich, einen Funktionsparameter für eine Funktion höherer Ordnung an Ort und Stelle zu erzeugen, ohne vorher mittels `def` eine Funktion zu definieren.

## 3.4 Zusammengesetzte Datentypen

Python besitzt mehrere zusammengesetzte Datentypen, darunter Strings (`str`), Listen (`list`), Tupel (`tuple`), Mengen (`set`) und sog. *Dictionaries* (`dict`), das sind Mengen von Schlüssel-Wert-Paaren, die einen schnellen Zugriff auf die Werte über die entsprechenden Schlüssel erlauben. Strings, Listen und Tupel sind sog. *Sequenzen* (auch Folgen oder Arrays genannt). Sequenzen sind iterierbar, d. h. man kann sie beispielsweise mittels einer `for`-Schleife durchlaufen.

### 3.4.1 Listen und Sequenzen

Python-Listen sind Sequenzen von durch Kommata getrennten Werten, eingeschlossen in eckigen Klammern. Listen können beliebig geschachtelt werden, d. h. die Elemente einer Liste können wiederum Listen sein, deren Elemente wiederum Listen sein könnten, usw. Folgende Python-Werte sind beispielsweise Listen:

`[]` (eine leere Liste), `[5,3,10,23]`, `['spam', [1,2,3], 3.14]`

**Indizierung**
Sei `S` eine Variable, die ein Objekt enthält, das einen Sequenz-Typ besitzt – also beispielsweise einen String, eine Liste oder ein Tupel, dann sind die folgenden Indizierungs-Operationen auf `S` anwendbar:

| | |
|---|---|
| `S[i]` | Selektiert Einträge an einer bestimmten Position. Negative Indizes zählen dabei vom Ende her. |

Beispiele:

| | |
|---|---|
| `S[0]` | Liefert das erste Element der Sequenz `S`; |
| `S[-2]` | liefert das zweitletzte Element der Sequenz `S`; |
| `['ab','xy'][-1][0]` | liefert `'x'` zurück. |

**Slicing (Teilbereichsbildung)**

Über Slicing kann man sich eine Teilsequenz erzeugen:

| | |
|---|---|
| `S[i:j]` | Selektiert einen zusammenhängenden Bereich einer Sequenz. Die Selektion erfolgt von einschließlich Index `i` bis ausschließlich Index `j`. |
| `S[:j]` | Die Selektion erfolgt vom ersten Element der Sequenz bis ausschließlich Index `j`. |
| `S[i:]` | Die Selektion erfolgt vom einschließlich Index `i` bis zum letzten Element der Sequenz. |

Beispiele:

| | |
|---|---|
| `S[1:5]` | Selektiert den zusammenhängenden Bereich aller Elemente ab einschließlich Index 1 bis ausschließlich Index 5; |
| `S[3:]` | selektiert alle Elemente von `S` ab Index 3 |
| `S[:-1]` | selektiert alle Elemente von `S`, bis auf das letzte |
| `S[:]` | selektiert alles, vom ersten bis zum letzten Element. |

**Extended Slicing**

Über einen zusätzlichen Parameter kann man sich die Schrittweite einstellen, mit der Elemente aus der Sequenz gezogen werden.

| | |
|---|---|
| `S[i:j:k]` | Durch `k` kann eine Schrittweite vorgegeben werden. |

Beispiele:

| | |
|---|---|
| `S[::2]` | Selektiert jedes zweite Element; |
| `S[::-1]` | selektiert alle Elemente von `S` in umgekehrter Reihenfolge; |
| `S[4:1:-1]` | selektiert die Elemente von rechts nach links ab Position 4 bis ausschließlich 1; |
| `'Welt'[::-1]` | ergibt `'tleW'`; |
| `'hallo welt'[-2::-2]` | ergibt `'lwolh'`; |
| `list(range(51)[::-10])` | ergibt `[50, 40, 30, 20, 10, 0]`; |

**Zuweisungen und Indizierung**

Handelt es sich bei der Sequenz um eine Liste, so kann – da Listen ja veränderliche Objekte sind – auch eine Zuweisung über Slicing erfolgen. Es folgen zwei Beispiele, wie Teile von Listen mittels Zuweisungen verändert werden können.

```
>>> l = list(range(6))
>>> l[2:5] = ['x']*3
>>> l
[0,1,'x','x','x',5]
```

```
>>> l = ['x']*6
>>> l[::2]=[0]*3
>>> l
[0,'x',0,'x',0,'x']
```

```
>>> l = list(range(7))
>>> l[-3::-1]=range(5)
>>> l
[4, 3, 2, 1, 0, 5, 6]
```

### 3.4.2 Allgemeine Sequenzoperationen

Folgende Funktionen sind auf alle Sequenzen anwendbar. Die meisten der hier aufgeführten Funktionen liefern Rückgabewerte zurück.

| | |
|---|---|
| `len(S)` | Liefert die Länge der Sequenz `S` zurück. |

Beispiele:

| | |
|---|---|
| `len('hallo')` | Liefert die Länge des String zurück, nämlich 5. |
| `len([1,[2,3]])` | Liefert die Länge der Liste zurück, nämlich 2. |

| | |
|---|---|
| `min(S)` | Liefert das minimale Element der Sequenz `S` zurück. |
| `max(S)` | Liefert das maximale Element der Sequenz `S` zurück. |

Beispiele:

| | |
|---|---|
| `max('hallo')` | Liefert die maximale Element des Strings, nämlich `'o'` zurück. |
| `max([101,123,99])` | Liefert die Zahl 123 zurück. |

| | |
|---|---|
| `sum(S)` | Liefert die Summe der Elemente der Sequenz `S` zurück. |

Beispiele:

| | |
|---|---|
| `sum(range((100))` | Berechnet $\sum_{i=0}^{99}$ und liefert entsprechend 4950 zurück. |

| | |
|---|---|
| `del(S[i])` | Löscht einen Eintrag einer Sequenz. |

| | |
|---|---|
| `del(S[i:j:k])` | `del` kann auch mit Slicing und Extended Slicing verwendet werden.<br>`del` kann man nur auf veränderliche Sequenzen anwenden. |

Beispiele:

| | |
|---|---|
| `l = list(range(10))`<br>`del(l[::2])` | Löscht jedes zweite Element der Liste; `l` hat also nach Ausführung der beiden Kommandos den Wert `[1, 3, 5, 7, 9]`. |

**Aufgabe 3.15**

Bestimmen Sie den Wert der folgenden Ausdrücke:

(a) `[range(1,100)[1],range(1,100)[2]]`
(b) `[range(1,10), range(10,20)][1][2]`
(c) `['Hello',2,'World'][0][2]+['Hello',2,'World'][0]`
(d) `len(range(1,100))`
(e) `len(range(100,200)[0:50:2])`

Hinweis: Versuchen Sie es zunächst ohne die Hilfe von Python und überprüfen danach erst Ihre Überlegungen mit Hilfe des Python-Interpreters.

**Aufgabe 3.16**

Wie können Sie in folgendem Python-Ausdruck

`[[x],[[[y]]]]`

auf den Wert von `y` zugreifen?

**Aufgabe 3.17**

Lösen sie die folgenden Aufgaben durch einen Python-Einzeiler:

(a) Erzeugen Sie die Liste aller geraden Zahlen zwischen 1 und 20.
(b) Erzeugen Sie die Liste aller durch 5 teilbaren Zahlen zwischen 0 und 100.
(c) Erzeugen Sie die Liste aller durch 7 teilbaren Zahlen zwischen 0 und 100; die Liste soll dabei umgekehrt sortiert sein, d. h. die größten Elemente sollen am Listenanfang und die kleinsten Elemente am Listenende stehen.

### 3.4.3 Wichtige Operationen auf Listen

Folgende Liste zeigt eine Auswahl der wichtigsten Listenoperationen angewendet auf einer Liste `L`:

| | |
|---|---|
| `L.append(x)` | Die `append`-Operation liefert keinen Wert zurück, sondern verändert die Liste `l` und fügt `x` am Ende von `L` ein. |

Beispiel:

| | |
|---|---|
| `l = list(range(5))`<br>`l.append('y')` | In diesem Fall wird der String `'y'` hinten an die Liste `l` angefügt. Danach hat `l` den Wert `[0,1,2,3,4,'y']`. |

| | |
|---|---|
| `L.sort()` | Auch `sort` liefert keinen Wert zurück, sondern verändert die Liste `L`. Nach Ausführung des Kommandos ist die Liste `L` aufsteigend sortiert. |

Beispiel:

| | |
|---|---|
| `l = [5,2,9,1]`<br>`l.sort()` | Danach hat `l` den Wert `[1,2,5,9]`. |

| | |
|---|---|
| `L.reverse()` | Dreht die Reihenfolge der Listenelemente um (und verändert so `L`). Liefert keinen Wert zurück, sondern verändert `L`. |

Beispiel:

| | |
|---|---|
| `l = list(range(5))`<br>`l.reverse()` | Danach hat `l` den Wert `[4,3,2,1,0]`. |

| | |
|---|---|
| `L.insert(i,x)` | Fügt `x` zwischen Indexposition `i` und Indexposition `i+1` in die Liste `L` ein.<br>`L[i:i] = [x]` hätte übrigens den gleichen Effekt. |

Beispiel:

| | |
|---|---|
| `l = list(range(5))`<br>`l.insert(3,"drei")` | Danach hat `l` den Wert `[0,1,2,"drei",3,4]`. |

| | |
|---|---|
| `L.count(x)` | Liefert die Anzahl der Vorkommen von `x` in `L` zurück. |

Beispiel:

| | |
|---|---|
| `[1,2,3,1].count(1)` | Liefert die Zahl `2` zurück. |

| | |
|---|---|
| `L.remove(x)` | Liefert keinen Wert zurück, sondern verändert `L`. In diesem Fall wird das erste Auftreten von `x` in `L` gelöscht. |

Beispiel:

| | |
|---|---|
| `l = [1,2,3,1]`<br>`l.remove(1)}` | Löscht das erste Vorkommen von `1` in der Liste. |

Diese Aufzählung der Operationen auf Listen ist nicht vollständig. Man kann sich alle vorhandenen Operationen eines Datentyps mit Hilfe der Python-Funktion **`dir`** ausgeben lassen. Der Aufruf

```
>>> dir(list)
... 'title', 'translate', 'upper', 'zfill']
```

liefert eine sortierte Stringliste aller Operations-Namen zurück, die für den Datentyp **`list`** definiert sind. Detailliertere Informationen liefert die Python-Funktion **`help`**, die vergleichbar mit dem `man`-Kommando der Bash ist. Wünscht man detaillierte Informationen zur Syntax einer Operation so kann man das Kommando **`help`** verwenden. Mit einem Aufruf von **`help`**`(list.append)` erhält man etwa die Hilfeinformation für die Operation `append`.

**Aufgabe 3.18**

Geben Sie in ein Python-Notebook das Kommando

```
[1,2,3].remove(1)
```

ein. Was liefert das Kommando zurück? Wie erklären Sie sich das Ergebnis?

**Aufgabe 3.19**

Geben Sie ein möglichst kurzes Pythonkommando / Pythonskript an, das …

(a) …die Anzahl der für den Datentyp **`dict`** definierten Operationen ausgibt.
(b) …die Anzahl der für den Datentyp **`list`** definierten Operationen ausgibt, die mit `'c'` beginnen.
(c) …die Länge des längsten Operationsnamens der auf dem Datentyp **`list`** definierten Operationen ausgibt.

### 3.4.4 Referenzen

Eine Zuweisung wie etwa

```
x = y
```

bewirkt im Allgemeinen nicht, dass eine neue Kopie eines Objektes `y` angelegt wird, sondern nur, dass `x` auf den Teil des Hauptspeichers zeigt, an dem sich `y` befindet. Man sagt auch, dass `x` und `y` Referenzen sind, die auf das selbe Objekt verweisen. Dies gilt immer für Werte zusammengesetzter Datentypen wie beispielsweise Listen, nicht jedoch für einfache Datentypen wie Zahlen oder Boolesche Werte. Normalerweise braucht sich der Programmierer darüber keine Gedanken zu machen. Es gibt jedoch immer wieder knifflige Situationen, in denen es von Vorteil ist, sich dieser Tatsache bewusst zu sein. Ein einfaches Beispiel:

```
>>> a = [1,2,3]
>>> b = a
>>> a.append(5)
>>> print(b)
[1,2,3,5]
```

Will man, dass `b` auf eine Kopie der Liste `a` verweist und `b` nicht nur, wie oben, ein neuer Zeiger auf die gleiche Liste darstellt, dann kann man dies folgendermaßen bewerkstelligen:

```
>>> b = a[:]
```

Dabei ist in obigem Fall `a[:]` dasselbe wie `a[0:2]` (siehe Seite 70 zur Erklärung der Indizierungsoperatoren) und bewirkt, dass auf der rechen Seite der Zuweisung eine Kopie der Liste erzeugt wird.

**Aufgabe 3.20**

Was ist der Wert der Variablen `a`, `b` und `c` nach der Eingabe der folgenden Kommandos in den Python-Interpreter:

```
>>> a = ['a','ab','abc']
>>> b = a
>>> b.append('abcd')
>>> c = b[:]
>>> c[0] = '0'
```

### 3.4.5 Tupel

Tupel sind Listen sehr ähnlich, mit dem Unterschied, dass sie unveränderlich sind, genauso wie auch Strings. Man kann Tupel also immer dann verwenden, wenn man sicher davon ausgehen kann, dass eine bestimmte Sammlung von Werten sich nicht verändern wird. Tupel werden in normalen runden Klammern notiert. Tupel können genauso wie andere Sequenzen auch indiziert werden. Beispiele:

```
>>> x = ('Das', 'ist', 'ein', 'Tupel')
>>> x[1]
'ist'
>>> x[2][0]
'e'
```

Versucht man einen Teil des Tupels zu verändern, so wird eine Fehlermeldung ausgegeben:

```
>>> x[0] = 'Hier'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: 'tuple' object does not support item assignment
```

Tupel können zusammen mit Strings verwendet werden, um Objekte formatiert in Strings einzufügen, wie in folgendem Beispiel gezeigt:

```
>>> alter = 50
>>> name = 'Tobias'
>>> print('%s ist noch keine %d Jahre alt.' % (name,alter))
Tobias ist noch keine 50 Jahre alt.
```

Python wandelt hier jeden Teil des nach dem %-Zeichen übergebenen Tupels in einen gemäß den Formatangaben entsprechenden String um, und setzt diesen dann anstelle der Formatangaben ein.

### 3.4.6 Dictionaries

Ein Dictionary-Objekt stellt eine Repräsentation einer Zuordnung von Schlüsseln auf Werte dar. Diese Datenstruktur ermöglichst einen sehr schnellen Zugriff auf Werte, die über einen Schlüssel gesucht werden; der Zugriff ist auch dann noch schnell, wenn die Datenstruktur viele Millionen Einträge enthält. Ein intuitives Anwendungsbeispiel ist ein Adressbuch, das bestimmte Namen – in diesem Falle die Schlüssel – auf Adressen – in diesem Falle die Werte – abbildet. Ein Dictionary-Objekt sollte die folgenden drei Operationen effizient unterstützen: **1.** Das Einfügen eines neuen Wertes `w` mit dem Schlüssel `s`. **2.** Das Finden eines bestimmten Wertes `w` anhand seines Schlüssels `s`. **3.** Das Löschen eines Schlüssels `s` zusammen mit dem zugehörigen Wert `w`.

Aufgrund der Tatsache, dass der Informatiker eine effiziente Unterstützung der Dictionary-Operationen häufig benötigt, bietet Python hierfür einen eigenen internen Typ `dict` an. Während Listen in eckigen Klammern und Tupel in runden Klammern notiert werden, werden Dictionary-Objekte in geschweiften Klammern geschrieben:

{ ⟨*schlüssel1*⟩ : ⟨*wert1*⟩ , ⟨*schlüssel2*⟩ : ⟨*wert2*⟩ , ...}

Ein einfaches Dictionary-Objekt, das eine Zuordnung von Namen auf Emailadressen realisiert, könnte folgendermaßen definiert werden:

```
>>> ab = { 'Carlo' :'carlo@web.de',
           'Hannes' :'hannes.pichelshuber@gmail.de',
           'Madita' :'madita.lindgren@gmx.de' }
```

In diesem Fall sind die Strings `'Carlo'`, `'Hannes'` und `'Madita'` die Schlüssel und die Strings, die die Emailadressen enthalten, sind die Werte.

Die Operationen „Einfügen“ und „Suchen“ werden über den Indizierungsoperator `[...]` angesprochen, so dass sich die Verwendung eines Dictionary-Objektes teilweise anfühlt wie die Verwendung eines Listen- oder Tupelobjekts. In folgendem Beispiel wird der dem Schlüssel `'Hannes'` zugeordnete Wert gesucht:

```
>>> ab['Hannes']
'hannes.pichelshuber@gmail.de'
```

Ein neuer Eintrag kann einfach mittels einer Zuweisung erzeugt werden. Der neue Schlüssel wird mittels Indizierungsoperator angegeben.

```
>>> ab['Matilda']='matilda@gmx.de'
>>> ab['Matilda']
'matilda@gmx.de'
```

Auch das Überschreiben eines bestehenden Eintrags ist in dieser Weise möglich:

```
>>> ab['Hannes']='hannes@gmx.de'
>>> ab['Hannes']
'hannes@gmx.de'
```

Die Funktion `del` implementiert die Löschfunktion. So kann man Einträge mit einem bestimmten Schlüsselwert aus dem Dictionary löschen.

```
>>> del(ab['Madita'])
>>> print('Es gibt %d Eintraege im Objekt ab' % len(ab))
'Es gibt 2 Eintraege im Objekt ab'
```

Man sieht in vorigem Beispiel, dass man viele für Sequenzen definierte Funktionen anwenden kann.

Die Typen von Schlüsseln und Werten können beliebig gewählt werden und sich auch innerhalb eines Dictionary-Objekts unterscheiden. So könnte man beispielsweise folgende Einträge dem Objekt `ab` hinzufügen:

```
>>> ab[1] = 2
>>> ab[(1,2)] = [1,2]
>>> ab[1]
2
```

Wichtig zu wissen ist, dass man nur unveränderliche Werte als Schlüssel verwenden kann, also insbesondere *keine* Listen:

```
>>> ab[[1,2]] = 0
TypeError: unhashable type: 'list'
```

Intern ist die Dictionary-Struktur als Hashtabelle organisiert, und nur Python-Objekte, die „hashable“ sind, eignen sich für Schlüssel eines Dictionary-Objekts.

**Aufgabe 3.21**

Erklären Sie, was das Problem wäre, wenn man auch veränderliche Werte (wie beispielsweise Listen) als Schlüssel in einem Dictionary-Objekt zulassen würde.

Die Reihenfolge, in der Elemente in einem Dictionary-Objekt angeordnet sind, ist nicht spezifiziert. Dennoch sind Dictionary-Objekte iterierbar:

```
>>> for x in ab:
      print(x)
Carlo
1
(1, 2)
Matilda
Hannes
```

Im Folgenden geben wir noch einige häufig verwendete Operationen über Dictionary-Objekte an. Die Variable `D` bezeichnet im Folgenden immer ein Dictionary-Objekt.

| | |
|---|---|
| `D.keys()` | Die `keys`-Operation liefert alle im Dictionary-Objekt `D` enthaltenen Schlüssel als eine Art Menge zurück. Diese Menge vom Typ `dict_keys` ist iterierbar, und einige Sequenz-Funktionen, wie beispielsweise **`len`**, können darauf angewendet werden. |

Beispiel:

| | |
|---|---|
| `ab.keys()` | Liefert folgendes `dict_keys`-Objekt zurück: `dict_keys(['Carlo', 1, (1,2), 'Matilda', 'Hannes'])`. |

| | |
|---|---|
| `D.values()` | Die `values`-Operation liefert alle in `D` enthaltenen Werte in einem Objekt vom Typ `dict_values` zurück. |

Beispiel:

| | |
|---|---|
| `ab.values()` | Liefert folgendes `dict_values`-Objekt zurück: `dict_values(['carlo@web.de', 2, [1, 2], ...])` |

| | |
|---|---|
| `D.items()` | Die `items`-Operation liefert alle Schlüssel-Wert-Paare in Form jeweils eines Tupels zurück. Diese Tupel werden in einem Mengen-artigen Objekt vom Typ `dict_items` zusammengefasst zurückgeliefert. |

Beispiel:

| | |
|---|---|
| `ab.items()` | Liefert folgendes `dict_items`-Objekt zurück: `dict_items([('Carlo', 'carlo@web.de'), (1, 2), ...])` |

**Aufgabe 3.22**

Gegeben sei ein Dictionary-Objekt `d`. Schreiben Sie eine Python-Funktion `flip_dict`, die ein neues Dictionary-Objekt zurückliefert, bei dem die Werte aus `d` den entsprechenden Schlüsseln aus `d` zugeordnet sind. Beispiel:

```
>>> flip_dict({1:"eins", 2:"zwei"})
{"eins":1, "zwei":2}
```

### 3.4.7 Strings (Fortsetzung)

Häufig gebraucht, sowohl für große Programmierprojekte als auch für viele kleine nützliche Skripts, sind Funktionen, die Strings verändern, erzeugen oder durchsuchen. Häufig sind diese auch im Zusammenhang mit Dateimanipulation relevant (siehe Abschnitt 3.6).

Wie schon in Abschnitt 3.2.3 erwähnt, sind Strings, genau wie Listen und Tupel, Sequenzen. Man kann daher auch alle Sequenzoperationen, wie Indizierung, Slicing usw. (siehe Abschnitt 3.4.1) auf sie anwenden. Im Gegensatz zu Listen sind Strings aber unveränderlich, d. h. ein einmal definierter String kann nicht verändert werden. Man kann also weder einzelne Zeichen aus ihm herauslöschen (wie dies etwa die Listenfunktion `del` macht), noch kann man an einen bestehenden String Zeichen anfügen (wie dies die Listenmethode `append` macht).

Im folgenden stellen wir die wichtigsten String-Kommandos vor, gruppiert in die Abschnitte „Durchsuchen“, „Aufteilen, Zusammenfügen“ und „Formatieren“. Die Variable `S` steht in den Beispielen stets für ein String-Objekt.

**Suchen**

| | |
|---|---|
| `S.find(S2)` | Liefert den Offset des ersten Vorkommens von `S2` in `S` zurück. |

Beispiel:

| | |
|---|---|
| `>>> s = "Wanna Banana"`<br>`>>> s.find("an")` | Liefert den Index 1 zurück, denn ab Indexposition 1 beginnt der Teilstring `'an'`. |

| | |
|---|---|
| `s.replace(S1,S2)` | Liefert einen String zurück, in dem alle Vorkommen von `S1` durch `S2` ersetzt sind. |

Beispiel:

| | |
|---|---|
| `>>> s = "Wanna Banana"`<br>`>>> s.replace("an","bn")` | Jedes Vorkommen von `'an'` wird durch `'bn'` ersetzt. Das `replace`-Kommano liefert in diesem Fall also den String `'Wbnna Bbnbna'` zurück. |

| | |
|---|---|
| `S.startswith(S1)` | Liefert `True` zurück, falls `S` mit `S1` beginnt. Andernfalls wird `False` zurückgeliefert. |

Beispiel:

| | |
|---|---|
| `>>> "Hallo".startswith("Ha")` | Dieser Ausdruck liefert den Wahrheitswert `True` zurück. |

| | |
|---|---|
| `S.endswith(S1)` | Liefert `True` zurück, falls der String `S` mit `S1` endet; andernfalls wird der Wahrheitswert `False` zurückgeliefert. |

Beispiel:

| | |
|---|---|
| `>>> "Hallo".endswith("loo")` | Dieser Ausdruck liefert den Wahrheitswert `False` zurück. |

**Aufteilen und Zusammenfügen**

| | |
|---|---|
| `S.split(S1)` | Gibt eine Liste von Wörtern von `S` zurück, mit `S1` als Trenner. Wird kein Trenner angegeben, so wird automatisch das Leerzeichen `' '` als Trenner verwendet. |

Beispiel:

| | |
|---|---|
| `>>> s = "1.;2.;3."`<br>`>>> s.split(";")` | Liefert die Stringliste `['1.', '2.', '3.']` zurück. |

| | |
|---|---|
| `S.partition(sep)` | Sucht nach dem Trenner `sep` in `S` und liefert ein 3-Tupel `(head,sep,tail)` zurück, wobei `head` der Teil vor `sep`, und `tail` der Teil nach `sep` ist. |

Beispiel:

| | |
|---|---|
| `>>> s = "1.;2.;3."`<br>`>>> s.partition(";")` | Liefert das Tupel `('1.', ';', '2.;3.')` zurück. |

| | |
|---|---|
| `S.join(strs)` | Verkettet die Strings in der Liste `strs` zu einem einzigen String mit S als Trenner. |

Beispiel:

| | |
|---|---|
| `>>> strs = ["Hi","out","there"]`<br>`>>> "! ".join(strs)` | Liefert den String `'Hi! out! there'` zurück. |

**Formatieren**

| | |
|---|---|
| `S.capitalize()` | Macht das erste Zeichen von S zu einem Großbuchstaben. |
| `S.upper()` | Wandelt alle Buchstaben in Großbuchstaben um. |
| `S.lower()` | Wandelt alle Buchstaben in Kleinbuchstaben um. |

**Aufgabe 3.23**

Schreiben Sie eine Python-Funktion `zipString`, die zwei Strings als Argumente übergeben bekommt und einen String zurückliefert, der eine „verschränkte“ Kombination der beiden übergebenen Strings ist. Beispielanwendungen der Funktion:

```
>>> zipString('Hello','World')
'HWeolrllod'
>>> zipString('Bla','123')
'B1l2a3'
```

**Aufgabe 3.24**

Schreiben Sie eine Python-Funktion `wordsLower`, die einen String übergeben bekommt und einen String zurückliefert, in dem alle Wörter kleingeschrieben sind. Beispielanwendung der Funktion:

```
>>> wordsLower("Hallo Welt, ich heisse Tobias")
"hallo welt, ich heisse tobias"
```

## 3.5 Funktionale Programmierung

Viele der gängigen Programmiersprachen, wie C, C++, Java, C#, sind *Imperative Programmiersprachen*. Das Paradigma der *Funktionalen Programmierung* unterscheidet sich vom für die meisten Programmierer gewohnten Paradigma der Imperativen Programmierung. In der Imperativen Programmierung verwendet man überwiegend *Anweisungen*[4], die etwas „tun", d. h. den Zustand des Programms bzw. des Speichers bzw. den Zustand von Peripheriegeräten (wie etwa des Bildschirms) verändern – etwa indem sie etwas auf dem Bildschirm ausgeben, den Wert einer Variablen verändern oder einen Teil des Hauptspeichers oder eine Datei manipulieren. Auch `for`- oder `while`-Schleifen sind typische Anweisungen: In jedem Schleifendurchlauf verändert sich der Zustand der Schleifenvariablen.

In der funktionalen Programmierung verwendet man überwiegend *Ausdrücke*, die strenggenommen nichts „tun", sondern lediglich für einen bestimmten Wert stehen, etwa einen Integerwert, einen String oder eine Liste. Im Gegensatz zu Anweisungen verändern Ausdrücke den Zustand von Objekten, des Speichers oder der Peripheriegeräte nicht. Viele Programmierfehler entstehen dadurch, dass der Programmierer den Überblick über die durch das Programm erzeugten Zustände verloren hat. Programmiert man dagegen hauptsächlich mit Ausdrücken, so schließt man diese Fehlerquelle aus. Entsprechend lohnt es sich immer in Erwägung zu ziehen, eine imperative Schleife durch eine Listenkomprehension, eine `map`-Anweisung oder eine `filter`-Anweisung zu ersetzen.

Python ist keine rein funktionale Sprache, und ein Pythonprogrammierer wird immer auch Anweisungen verwenden müssen. In diesem Abschnitt stellen wir mit den Listenkomprehensionen und den Funktionen `map`, `filter`, `reduce` und `enumerate` die wichtigsten Werkzeuge vor, um mit weniger Anweisungen und weniger Zustandsveränderungen auszukommen.

**4** Siehe Abschnitt 3.3.4 für die Unterscheidung zwischen Anweisungen und Ausdrücken; um diesen Abschnitt zu verstehen ist es notwendig, zu verstehen, was der Unterschied zwischen einem Ausdruck und einer Anweisung ist.

### 3.5.1 Listenkomprehensionen

Listenkomprehensionen sind Ausdrücke, keine Kommandos, und stehen also für einen bestimmten Wert. Man kann Listenkomprehensionen als das funktionale Pendant zur imperativen Schleife betrachten. Sie sind insbesondere für Mathematiker leicht verständlich, denn sie besitzen eine mit der Mengennotation, den „set comprehensions“ der Mathematik, vergleichbare Notation. Die Menge

$$\{2 \cdot x \mid x \in \{1, \ldots 20\},\ x \text{ durch 3 teilbar }\}$$

entspricht[5] hierbei der Python-Listenkomprehension

```
[ 2*x for x in range(1,21)if x%3==0 ]
```

Jede Listenkomprehension besteht mindestens aus einem in eckigen Klammern `[...]` eingeschlossenen Ausdruck, gefolgt von einer oder mehreren sogenannten `for`-Klauseln. Jede `for`-Klausel kann optional durch eine `if`-Klausel eingeschränkt werden.

```
[⟨ausdr⟩ for ⟨variable1⟩ in ⟨sequenz1⟩ [if ⟨bedingung1⟩]
        for ⟨variable2⟩ in ⟨sequenz2⟩ [if ⟨bedingung2⟩] ...]
```

Der Bedingungsausdruck einer `if`-Klausel hängt im Allgemeinen ab von einer (oder mehreren) durch vorangegangene `for`-Klauseln gebundenen Variablen. Dieser Bedingungsausdruck filtert all diejenigen Variablen der jeweiligen Sequenz aus, für die er den Wahrheitswert `False` liefert, oder anders ausgedrückt: Die jeweilige Variable nimmt nur diejenigen Werte der jeweiligen Sequenz an, für die der Bedingungsausdruck den Wahrheitswert `True` ergibt.

Der Wert der Listenkomprehension ist die Liste aller Werte des Ausdrucks ⟨*ausdr*⟩ für alle Kombinationen von Werten von ⟨*variable1*⟩, ⟨*variable2*⟩, usw.

Abbildung 3.3 veranschaulicht, wie sich die Werte einer Listenkomprehension mit einer `for`-Klausel und einer `if`-Klausel zusammensetzen.

**5** Man sollte sich jedoch bewusst sein, dass mathematische Mengen und Python-Listen sich in folgenden beiden Punkten unterscheiden: **1.** In Mengen spielt die Reihenfolge, in der die Elemente der Menge notiert werden, keine Rolle. **2.** Ein bestimmtes Element kann höchstens einmal in der Menge auftauchen. Bei Python-Listen dagegen spielt die Reihenfolge in der die Elemente in der Liste auftauchen sehr wohl eine Rolle – es gilt beispielsweise `[1,2]!=[2,1]`; außerdem können Elemente mehrmals in einer Python-Liste auftauchen. Insofern entsprechen Python-Listen genau genommen eher der mathematischen Tupel-Notation.

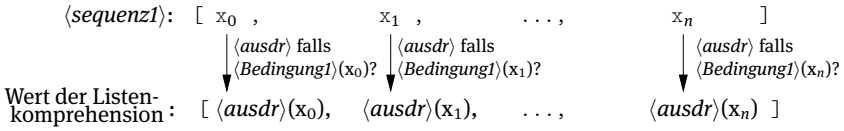



**Abb. 3.3.** Darstellung, wie sich die Werte einer Listenkomprehension mit einer `for`-Klausel und einer `if`-Klausel zusammensetzen. Die Ausdrücke ⟨*sequenz1*⟩,⟨*bedingung1*⟩ und ⟨*ausdr*⟩ beziehen sich hier auf die entsprechenden Platzhalter, die in obiger Syntaxbeschreibung verwendet wurden. Wie man sieht, ist der Wert der Listenkomprehension *immer* eine Liste, deren Elemente durch Auswertung von ⟨*ausdr*⟩ in Abhängigkeit der einzelnen Elemente der Liste ⟨*sequenz1*⟩ entstehen.

### Beispiele

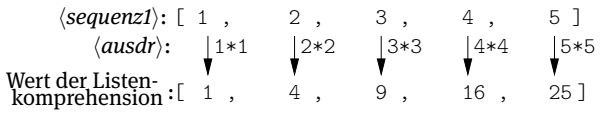
Um mit Listenkomprehensionen vertrauter zu werden, präsentieren wir im folgenden einige Beispiele. Diese zeigen jeweils eine Listenkomprehension, die mit Python ausgewertet wird.

Wir gehen in vielen der präsentierten Beispiele darauf ein, welchen Wert die einzelnen Platzhalter der obigen Syntaxbeschreibung haben, d. h. wir geben häufig der Klarheit halber an, was der jeweilige Wert der Platzhalter ⟨*ausdr*⟩, ⟨*ausdr1*⟩, ⟨*sequenz1*⟩, ⟨*bedingung1*⟩, usw. ist.

**i)** Erzeugung einer Liste aller Quadratzahlen von $1^2$ bis $5^2$:

```
>>> [x*x for x in range(1,6) ]
[1, 4, 9, 16, 25]
```

Der Wert des Platzhalters ⟨*ausdr*⟩ in obiger Syntaxbeschreibung entspricht hier dem Ausdruck `x*x`. Die Sequenz ⟨*sequenz1*⟩ entspricht `range(1,6)`. Für jeden Wert in `range(1,6)`, also für jeden Wert in `[1,2,3,4,5]`, wird ein Listeneintrag der Ergebnisliste durch Auswertung des Ausdrucks `x*x` erzeugt. Ergebnis ist also `[1*1, 2*2, ...]`. Die folgende Abbildung veranschaulicht dies nochmals:




**ii)** Erzeugung einer Liste aller durch 3 oder durch 7 teilbaren Zahlen zwischen 1 und 20:

```
>>> [x for x in range(1,20)
...         if x%3==0 or x%7==0 ]
[3, 6, 7, 9, 12, 14, 15, 18]
```

Der Wert des Platzhalters ⟨*ausdr*⟩ entspricht hier dem nur aus einer Variablen bestehenden Ausdruck `x`. Die Sequenz ⟨*sequenz1*⟩ entspricht `range(1,20)`. Der Bedingungsausdruck ⟨*bedingung1*⟩ entspricht `x%3==0 or x%7==0`. Hier wird also eine Liste erzeugt, die aus Einträgen `x` in der Sequenz `range(1,20)` besteht, für die der Ausdruck `x%3==0 or x%7==0` den Wert `True` ergibt.

**Aufgabe 3.25**

(a) Schreiben Sie eine Python-Funktion `teiler(n)`, die die Liste aller Teiler einer als Parameter übergebenen Zahl `n` zurückliefert. Verwenden Sie zur Erzeugung der Liste der Teiler eine Listenkomprehension. Beispielanwendung:

```
>>> teiler(45)
>>> [1, 3, 5, 9, 15]
```

(b) Geben Sie unter Verwendung der Funktion `teiler` einen Python-Ausdruck an, der eine Liste aller Zahlen zwischen 1 und 1000 ermittelt, die genau 5 Teiler besitzen.

(c) Geben Sie unter Verwendung der Funktion `teiler` einen Python-Ausdruck an, der diejenige Zahl zwischen 1 und 1000 ermittelt, die die meisten Teiler besitzt.

**iii)** Erzeugung einer Liste aller möglichen Tupel von Zahlen aus 1 bis 10.

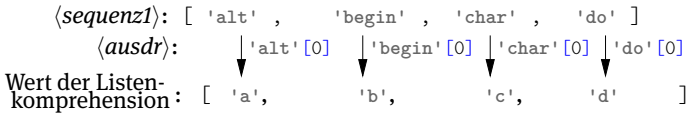
```
>>> [(x,y) for x in range(1,10)
...         for y in range(1,10)]
[(1, 1), (1, 2), ...,(1,9), (2,1), (2,2), ...(9, 9)]
```

Der Platzhalter ⟨*ausdr*⟩ entspricht in diesem Fall dem Tupel `(x,y)`, der Platzhalter ⟨*sequenz1*⟩ entspricht `range(1,10)` und der Platzhalter ⟨*sequenz2*⟩ entspricht `range(1,10)`. Man sieht: Es können beliebig viele `for`-Klauseln hintereinander stehen, was einer Schachtelung von `for`-Schleifen entspricht. Im ersten Durchlauf hat `x` den Wert `1` und `y` durchläuft für diesen einen festen Wert von `x` die Zahlen von `1` bis (ausschließlich) `10`. Im zweiten Durchlauf hat `x` den Wert `2` und `y` durchläuft wiederum die Zahlen von `1` bis ausschließlich `10`, usw. Jede dieser beiden `for`-Klauseln könnte (auch wenn dies in obigem Beispiel nicht der Fall ist) ein `if`-Statement verwenden, das die Werte für `x` bzw. `y` einschränkt.

**iv)** Erzeugung der jeweils ersten Zeichen von in einer Liste befindlichen Strings:

```
>>> [x[0] for x in ['alt','begin','char','do']]
['a','b','c','d']
```

Der Platzhalter ⟨*ausdr*⟩ entspricht hier dem Ausdruck `x[0]`, und der Platzhalter ⟨*sequenz1*⟩ entspricht der Stringliste `['alt','begin',...]`. Die Schleifenvariable `x` durchläuft nacheinander die Strings `'alt'`, `'begin'`, usw. In jedem Durchlauf wird das erste Zeichen des jeweiligen Strings in die Ergebnisliste eingefügt. Die folgende Abbildung veranschaulicht dies nochmals:




**Aufgabe 3.26**

Verwenden Sie eine Listenkomprehension, um eine Liste zu erzeugen, die alle Methoden- und Attributnamen der Klasse `str` enthält, die ...

(a) mit `'a'` beginnen.
(b) mit `'er'` enden.
(c) mehr als 10 Zeichen enthalten.
(d) den String `'cod'` als Teilstring enthalten.

**Tipp**: Mit `dir(str)` erhalten Sie die Liste aller Methoden- und Attributnamen der Klasse `str`. Für obige Aufgaben benötigen Sie evtl. die String-Methoden `startswith`, `endswith`, die Funktion `len`, und die Operation `in`.

**Aufgabe 3.27**

Verwenden Sie eine Listenkomprehension, die ...

(a) eine Liste der Längen aller Methoden- und Attributnamen der Klasse `str` erzeugt.
(b) die Länge des längsten Methoden- oder Attributnamens der Klasse `str` berechnet.
(c) den Namen des längsten Methoden- oder Attributnamens der Klasse `str` zurückliefert.

**v)** Erzeugung der Liste aller Wörter aus einer Liste von Strings, die weniger als 3 Zeichen enthalten:

```
>>> strlst = ['Ich bin klein', 'und so fein', 'ja das ist eben so']
>>> [w for s in strlst for w in s.split() if len(w)<3]
['so', 'ja', 'so']
```

Diese Listenkomprehension besteht aus zwei geschachtelten `for`-Schleifen. Die äußere `for`-Schleife läuft über die in `strlst` enthaltenen Strings; `s` hat also im ersten Durchlauf den Wert `'Ich bin klein'`, im zweiten Durchlauf den Wert `'und so fein'` und im dritten Durchlauf den Wert `'ja das ist eben so'`. Schauen wir uns den ersten Durchlauf der äußeren Schleife näher an: In diesem ersten Durchlauf hat `s` den Wert `'Ich bin klein'` und `w` läuft über die Liste

```
'Ich bin klein'.split()
```

Die Funktion `split` teilt einen String nach einem bestimmten Vorkommen eines Zeichens auf, in diesem Fall standardmäßig nach Leerzeichen; der Ausdruck hat also den Wert `['Ich','bin','klein']` und die innere Schleife läuft über die drei Strings `'Ich'`, `'bin'` und `'klein'`. Von diesen Wörtern `w` werden nur diejenigen „durchgelassen“, die die Bedingung `len(w)<3` erfüllen. Im ersten Durchlauf der äußeren Schleife sind das gar keine, im zweiten Durchlauf ist das das Wort `'so'` und im letzten Durchlauf sind es die Wörter `'ja'` und `'so'`.

**Aufgabe 3.28**

Gegeben Sei ein langer String, der `'\n'`-Zeichen (also Newline-Zeichen, oder Zeilentrenner-Zeichen) enthält. Geben Sie – evtl. unter Verwendung einer Listenkomprehension – einen Ausdruck an, der …

(a) die Anzahl der Zeilen zurückliefert, die dieser String enthält.
(b) alle Zeilen zurückliefert, die weniger als 5 Zeichen enthalten.
(c) alle Zeilen zurückliefert, die das Wort `'Gruffelo'` enthalten.

**Aufgabe 3.29**

Programmieren Sie mit Hilfe einer Listenkomprehension eine Funktion `prims(n)`, die die Liste aller Primzahlen bis `n` zurückliefert.

*Anmerkung*: Häufig sehe ich, dass Programmieranfänger die Ergebnisse mit der `print`-Funktion „zurückgeben“. Eine Funktion, die Ergebnisse auf dem Bildschirm ausgibt, ist jedoch im Allgemeinen als Teil eines größeren Prorgammes wenig nützlich. Eine Funktion, die Ergebnisse mit Hilfe von `return` zurückliefert, kann dagegen ihre Ergebnisse an andere Funktionen weitergeben.

**Aufgabe 3.30**

Geben Sie eine Listenkomprehension an, mit der Sie die Liste aller Quersummen der Zahlen von 1 bis 1000 ausgeben können.

**Aufgabe 3.31**

Erzeugen Sie eine Liste aller Perfekten Zahlen zwischen 1 und 10000 mit Hilfe einer Python-Listenkomprehension.

Ein Zahl heißt perfekt, wenn Sie gleich der Summe ihrer positiven echten (d. h.

die Zahl selbst gehört nicht dazu) Teiler ist. Beispielsweise ist 6 eine Perfekte Zahl, denn 6 = 1 + 2 + 3.

**Aufgabe 3.32**

Wir wollen Zufallsexperimente mit einem 6-seitigen Würfel durchführen. Sie können hierzu die Funktion `randint` aus der Bibliothek `random` verwenden und diese folgendermaßen in ein Python-Notebook oder ein Python-Skript einbinden:

```
>>> from random import randint
```

(a) Erzeugen Sie mittels einer Listenkomprehension eine Liste mit 10 zufälligen Werten zwischen 1 und 6 – um einen 10-maligen Wurf mit einem 6-seitigen Würfel zu simulieren.
(b) Berechnen Sie eine Schätzung der Wahrscheinlichkeit, dass bei einem 10-maligen Wurf keine 1 fällt.

    Zählen Sie zur Beantwortung dieser Frage die 10er-Würfe von (sagen wir) 100000 10er-Würfen, die keine 1 enthalten. Realisieren Sie dies durch eine Listenkomprehension.
(c) Berechnen Sie eine Schätzung der Wahrscheinlichkeit, dass bei einem 10-maligen Wurf genau drei mal eine 1 fällt.

### 3.5.2 Die `map`-Funktion

Die `map`-Funktion verknüpft mehrere Listen elementweise mit einer als Parameter übergebenen Funktion:

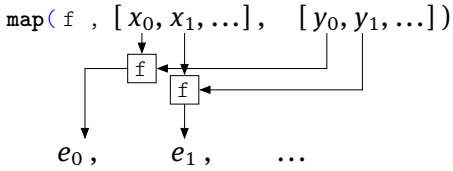
$$\texttt{map}(\langle fkt\rangle, \langle sequenz_1\rangle, \langle sequenz_2\rangle, \ldots)$$

Die `map`-Funktion ist eine sog. *Funktion höherer Ordnung* (engl.: *higher-order function*); das ist eine Funktion, die als Parameter eine Funktion erwartet oder eine Funktion als Ergebnis liefert.

Funktionsweise: Die `map`-Funktion ruft die Funktion $\langle fkt\rangle$ für alle ersten Elemente der Listen $\langle sequenz_1\rangle$, $\langle sequenz_2\rangle$, ... auf, anschließend für alle zweiten Elemente, usw. Die jeweils erhaltenen Werte werden als spezielle Sequenz vom Typ `map` zurückgeliefert. Diese Sequenz wird nicht sofort ausgewertet, sondern erst dann, wenn die Elemente explizit angefordert werden, etwa durch eine `for`-Schleife, oder wenn man Python durch die Funktion `list(...)` zwingt, das Ergebnis in eine Liste zu überführen. Dieses Verhalten ist besonders dann sinnvoll, wenn es sich um sehr lange Se-

quenzen handelt, die bei einer sofortigen Auswertung viel Speicherplatz benötigen würden.

Folgendes Beispiel veranschaulicht die Funktionsweise der `map`-Funktion, die mit einer zweistelligen Funktion `f` zwei Listen $[x_0, x_1, \ldots]$ und $[y_0, y_1, \ldots]$ elementweise verknüpft und daraus eine neue Sequenz $e_0, e_1, \ldots$ erzeugt:




```
>>> m = map(lambda x,y: x+y, [1,3,5], [10,100,1000])
>>> m
<map at 0xde31f5dba8>
>>> list(m)
[11, 103, 1005]
>>> list(m)
[]
```

Der Aufruf von `map` in der ersten Zeile erzeugt ein Sequenz-Objekt, das die Ergebnisse zunächst aber noch nicht auswertet und dessen String-Repräsentation lediglich die Information enthält, dass sich das Objekt an Stelle `0xde31f5dba8` befindet. Erst nachdem wir das Objekt durch den Aufruf `list(m)` dazu „zwingen“, berechnet es seine Ergebnisse und liefert diese in einer Liste zurück. Etwas eigenartig mag das Verhalten des letzten Aufrufs erscheinen: Hat das `map`-Objekt einmal seine Inhalte ausgewertet, ist es „erschöpft“ und kann keine Werte mehr produzieren.

Die `map`-Funktion kann natürlich auch mit einer Liste und einer einstelligen Funktion verwendet werden, wie in folgendem Beispiel gezeigt:

```
>>> list(map(lambda x: x[:2],
         ['piraten', 'raten', 'teils', 'zu', 'schlimmen', 'taten']))
['pi','ra','te','zu','sc','ta']
```

Der an die `map`-Funktion übergebene `lambda`-Ausdruck ist einstellig, d. h. er erwartet genau einen Parameter, der eine (indizierbare) Sequenz sein muss, und liefert die ersten beiden Elemente davon zurück.

Berechnungen, die eine `map`-Funktion verwenden, die mit einer einstelligen Funktion und einer einzigen Liste arbeiten, können auch mittels einer Listenkomprehensionen ausgedrückt werden. Obiges Beispiel, unter Verwendung von Listenkomprehensionen, hat die folgende Form:

```
>>> [x[:2] for x in ['piraten', 'raten', 'teils', 'zu',
                     'schlimmen', 'taten']]
```

```
['pi','ra','te','zu','sc','ta']
```

Ein Unterschied ist, dass man bei Verwendung von `map` die Kontrolle hat, wann die erzeugte Sequenz ausgewertet wird; eine Listenkomprehension berechnet immer sofort die gesamte Liste.

**Aufgabe 3.33**

Verwenden Sie die `map`-Funktion, um einer (String-)Liste von Zeilen Zeilennummern hinzuzufügen. Der Ausdruck

```
['Erste Zeile', 'Zweite Zeile', 'Und die dritte Zeile']
```

sollte also eine Sequenz erzeugen, die die folgenden Einträge hat:

```
'1. Erste Zeile', '2. Zweite Zeile', '3. Und die dritte Zeile'
```

### 3.5.3 Die `filter`-Funktion

Mit Hilfe der `filter`-Funktion können bestimmte Elemente einer Liste oder Sequenz ausgefiltert werden, wenn sie eine bestimmte Eigenschaft nicht erfüllen.

`filter` (⟨*function*⟩, ⟨*sequenz*⟩)

Die `filter`-Funktion liefert nur diejenigen Elemente von ⟨*sequenz*⟩ als Liste zurück, für die das Funktionsargument ⟨*function*⟩, angewandt auf das jeweilige Sequenzelement, den Wert `True` zurückliefert. Die übergebene Funktion muss einstellig sein und als Rückgabewert einen Booleschen Wert erzeugen. Ein Aufruf von `filter` liefert ein Objekt vom Typ `filter` zurück, das eine Sequenz darstellt, die ihren eigentlichen Inhalt erst dann auswertet, wenn es der Programmierer explizit verlangt. Das `filter`-Objekt hat – abgesehen vom eigentlichen Namen des Typs – die gleichen Eigenschaften, wie das im vorherigen Abschnitt besprochene `map`-Objekt.

Beispielsweise kann man folgendermaßen alle Zahlen zwischen 2 und 25 erhalten, die weder durch 2 noch durch 3 teilbar sind:

```
>>> f = filter(lambda x: x%2 != 0 and x%3 != 0, range(2,25))
>>> f
<filter at 0x68773b7748>
>>> list(f)
[5, 7, 11, 13, 17, 19, 23]
```

Berechnungen, die eine `filter`-Funktion verwenden, können immer auch mit einer Listenkomprehension ausgedrückt werden, mit dem Unterschied jedoch, dass Listen-

komprehensionen eine fertig ausgewertete Liste produzieren. Obiges Beispiel unter Verwendung einer Listenkomprehension hat die folgende Form:

```
>>> [x for x in range(2,25) if x%2 !=0 and x%3 != 0]
[5, 7, 11, 13, 17, 19, 23]
```

**Aufgabe 3.34**

Verwenden Sie `filter` und/oder `map`, um ...

(a) sich alle Methodennamen des Typs `string` ausgeben zu lassen, die kürzer als 6 Zeichen sind.
(b) sich die Länge des längsten Methodennamens des Typs `string` ausgeben zu lassen.
(c) zu zählen, wie viele Methodennamen des Typs `list` mit `'a'` beginnen.
(d) zu zählen, wie viele Methodennamen des Typs `string` ein `'a'` enthalten und mehr als 5 Zeichen lang sind.
(e) sich die Länge des längsten Methodennamens des Typs `string` ausgeben zu lassen, der ein `'a'` enthält.

Hinweis: Die Liste aller Methodennamen von beispielsweise `string` erhält man mit dem Pythonkommando `dir(string)`.

**Aufgabe 3.35**

Wir wollen alle möglichen Wörter mit bestimmten Eigenschaften erzeugen. Gegeben seien die folgenden Definitionen:

```
C = 'abcdefghijklmnopqrstuvwxyz'
D = map(str, range(0,10)) ; A = C+D
```

(a) Schreiben Sie eine Pythonfunktion `woerter(n)`, die *alle* Wörter der Länge `n` mit Buchstaben aus `A` erzeugt.
*Tipp*: Zur Lösung kommt man elegant rekursiv, indem man überlegt, was welchen Wert `woerter(0)` und `woerter(1)` haben muss und anschließend überlegt, wie man `woerter(n)` aus `woerter(n-1)` berechnen kann.
(b) Schreiben Sie eine Pythonfunktion `w1d(n)`, die *alle* Wörter der Länge `n` mit Buchstaben aus `A` erzeugt, die *mindestens* eine Ziffer enthalten.
(c) Schreiben Sie eine Pythonfunktion `wdE(n)`, die *alle* Wörter der Länge `n` mit Buchstaben aus `A` erzeugt, die mit einer Ziffer enden.

*Tipp*: `c.isdigit()` testet, ob das Zeichen `c` eine Ziffer ist.

### 3.5.4 Die `reduce`-Funktion

Neben der `map`- und der `filter`-Funktion, ist die `reduce`-Funktion eine weitere im Rahmen der Funktionalen Programmierung häufig verwendete Funktion höherer Ordnung. Während in der Python 2.*x*-Version die `reduce`-Funktion noch initial verfügbar war, befindet sich diese in den Python 3.*x*-Versionen in dem Modul `functools` und muss mittels

```
from functools import reduce
```

in den Namensraum geladen werden.

Die `reduce`-Funktion verknüpft die Elemente einer Liste bzw. einer Sequenz nacheinander mit einer zwei-stelligen Funktion.

`reduce(`⟨*function*⟩`,`⟨*sequenz*⟩`)`

Die Verknüpfung erfolgt von links nach rechts. Um die Funktionsweise von `reduce` zu veranschaulichen gehen wir von folgender Definition einer zweistelligen Funktion aus – das Symbol $\oplus$ ist hierbei Platzhalter für eine beliebige Verknüpfung der beiden Parameter `x` und `y`:

```
>>> f = lambda x,y: x ⊕ y
```

Dann wertet sich der folgende Ausdruck

```
>>> reduce( f, [x0, x1, x2, ..., xn] )
```

aus zu: $(\cdots(((x_0 \oplus x_1) \oplus x_2) \oplus \ldots) \oplus x_n)$

**Beispiele**

Wir geben im Folgenden einige Beispiele für die Verwendung der `reduce`-Funktion:

**i) Aufsummieren aller ungeraden Zahlen von 1 bis 1000**

```
>>> reduce(lambda x,y: x+y, range(1,1000,2))
250000
```

Berechnet die Summe $(\cdots((1+3)+5)+\ldots+999)$. Die gleiche Berechnung kann man auch mit `sum(range(1,1000,2))` durchführen.

**ii) Verknüpfen einer Menge von Strings zu einem String der aus einer Menge von Zeilen besteht**

```
>>> f = lambda x,y: x+'\n'+y
>>> reduce(f,['Erste Zeile', 'Zweite Zeile', 'Dritte Zeile'])
>>> 'Erste Zeile\nZweite Zeile\nDritte Zeile'
>>> print(reduce(f,['Erste Zeile','Zweite Zeile','Dritte Zeile']))
Erste Zeile
Zweite Zeile
Dritte Zeile
```

Wie schon in Abschnitt 3.2.5 angedeutet, ist der +-Operator überladen, d. h. seine Funktionsweise ist abhängig vom Typ der Operanden. Sind die Operanden Zahlen, so entspricht der +-Operator einer Addition; sind die Operanden dagegen Sequenzen, dann entspricht der +-Operator einer Konkatenation, d. h. einem Zusammenketten von Sequenzen. Die Funktion `f` ist so definiert, dass sie zwei Strings mit dem Newline-Zeichen `'\n'` als Trenner zusammenkettet. Die `reduce`-Funktion verkettet entsprechend alle Strings in der Liste und fügt ein `'\n'`-Zeichen zwischen jeweils zwei Strings ein.

**iii) Umwandeln einer als String repräsentierten Hexadezimal-Zahl in einen Python Integerwert.**

```
>>> hexNum = '12fb3a' ; l = len(hexNum)
>>> def f(x,y): return x+y
>>> reduce(f,[ c2h(hexNum[i])*16**(l-i) for i in range(l)])
1243962
```

Wir gehen davon aus, dass die in der folgenden Aufgabe durch den Leser zu definierenden Funktion `c2h` eine als Zeichen repräsentierte hexadezimale Ziffer in den entsprechenden `int`-Wert umwandelt. Die Listenkomprehension innerhalb der `reduce`-Funktion erzeugt die Liste

$$[1 * 16^5, 2 * 16^4, 15 * 16^3, 11 * 16^2, 3 * 16^1, 10 * 16^0]$$

Diese Werte müssen nur noch aufsummiert werden – dies erledigt die `reduce`-Funktion – und man erhält den Wert

$$\begin{aligned} & \sum_{i=0}^{5} \texttt{hexNum}_i \cdot 16^{(5-i)} \\ = \quad & 1 * 16^5 + 2 * 16^4 + 15 * 16^3 + 11 * 16^2 + 3 * 16^1 + 10 * 16^0 \end{aligned}$$

und dies entspricht genau der Dezimalrepräsentation der als String repräsentierten hexadezimalen Zahl `'12fb3a'`.

**Aufgabe 3.36**

Schreiben Sie eine Funktion `c2h(c)`, die das Zeichen `c` in einen `int`-Wert umwandelt, falls `c in '0123456789abcdefABCDEF'`, d. h., falls `c` eine hexadezimale Ziffer repräsentiert. Es soll beispielsweise gelten:

```
>>> c2h('5')
5
>>> c2h('e')
14
```

**iv) Umwandeln einer als String repräsentierten Hexadezimal-Zahl in einen Integerwert unter Verwendung des *Horner-Schemas*:**

Nehmen wir an es sei eine hexadezimale Zahl $h_0h_1h_2h_3h_4$ gegeben. Will man daraus die entsprechende Dezimalzahl über

$$h_0 * 16^4 + h_1 * 16^3 + h_2 * 16^2 + h_3 * 16^1 + h_4 * 16^0$$

berechnen, so ist dies wenig effizient. Der Grund dafür ist, dass zur Berechnung der Potenzen sehr viele (nämlich 4+3+2) Multiplikationen durchgeführt werden müssen, und Multiplikationen sind recht rechenintensiv. Die gleiche Berechnung kann folgendermaßen mit nur vier Multiplikationen durchgeführt werden:

$$(((h_0 * 16 + h_1) * 16 + h_2) * 16 + h_3) * 16 + h4$$

Dieses Berechnungs-Schema ist das sog. *Horner-Schema*. Eine Implementierung kann elegant mit Hilfe der `reduce`-Funktion folgendermaßen erfolgen:

```
>>> hexNum = '12fb3a' ; l = len(hexNum)
>>> def f(x,y): return 16*x +y
>>> reduce(f, [c2h(h) for h in hexNum])
1243962
```

Die Listenkomprehension `[c2h(h)for h in hexNum]` erzeugt zunächst eine Liste der Integerwerte, die den einzelnen Ziffern in `hexNum` entsprechen – hier wäre das die Liste `[1,2,15,11,3,10]`. Die `reduce`-Funktion verknüpft dann die Elemente der Liste mit der Funktion `f` und verwendet so das Horner-Schema, um die Dezimalrepräsentation der Hexadezimalzahl `'12fb3a'` zu berechnen.

### Aufgaben

**Aufgabe 3.37**

Verwenden Sie das mittels `reduce`-Funktion implementierte Horner-Schema, um eine als String repräsentierte Binärzahl in die entsprechende Dezimalzahl umzuwandeln.

**Aufgabe 3.38**

Verwenden Sie die `reduce`-Funktion, um eine Funktion `prod(lst)` zu definieren, die alle in `lst` befindlichen Zahlen aufmultipliziert.

**Aufgabe 3.39**

Verwenden Sie die `reduce`-Funktion, um eine Funktion `max(lst)` zu definieren, die das maximale in `lst` befindliche Element zurückliefert.

**Aufgabe 3.40**

Verwenden Sie die `reduce`-Funktion, um eine Liste von Tupeln „flachzuklopfen“ und in eine einfache Liste umzuwandeln. Beispiel: Die Liste

```
[(1,10), ('a','b'), ([1], [2])]
```

sollte dadurch beispielsweise in die Liste

```
[1,10,'a','b',[1],[2]]
```

umgewandelt werden.

## 3.6 Dateien und Verzeichnisse

Python wird mit einer großen Zahl von *Standard-Bibliotheken* geliefert. Das sind vorprogrammierte Codeteile, die wichtige, häufig benötigte Funktionen bereitstellen. Einige wichtige Bibliotheken, die wir in diesem Abschnitt kennenlernen werden, sind

| | |
|---|---|
| `sys` | Bietet Zugriff auf Umgebungsvariablen, Standardströme (stdin, stdout, stderr), die Kommandozeile |
| `os` | Ist die plattformunabhängige[6] Schnittstelle zum Betriebssystem. |
| `os.path` | Ist ein „Untermodul“ von *os* und bietet Möglichkeiten zur Bearbeitung von Pfaden. |

Ein Modul kann mit

`import` ⟨*modulname*⟩

geladen werden. Man kann sich alle in einem Modul definierten Funktionsnamen als Stringliste ausgeben lassen mit dem Kommando

6 Mit „plattformunabhängig“ ist gemeint: unabhängig vom konkreten Betriebssystem; das bedeutet hier, dass die entsprechenden Kommandos immer das gleiche Ergebnis liefern, egal unter welchem Betriebssystem sie ausgeführt werden.

`dir(⟨modulname⟩)`

Will man etwa auf Parameter zugreifen, die über die Kommandozeile übergeben wurden, die sogenannten *Kommandozeilenparameter*, so kann man die von Modul `sys` bereitgestellte Variable `argv` verwenden. Die Variable `argv` enthält eine Liste von Strings. Der *i*-te Eintrag entspricht dem *i*-ten über die Kommandozeile übergebenen Parameter. Im 0-ten Eintrag, also im String `sys.argv[0]`, ist immer der Name des ausführenden Kommandos gespeichert. Ein Beispiel (das `$`-Zeichen stellt das Prompt der Unix-Shell Bash dar):

```
$ ( echo 'import sys' ; echo 'print(sys.argv)' ) >komm.py
$ python komm.py erstesArg zweitesArg
$  ['komm.py', 'erstesArg', 'zweitesArg']
```

### 3.6.1 Datei-Objekte

Die eingebaute Funktion `open`

`open(⟨string⟩,⟨modus⟩)`

erzeugt ein Dateiobjekt, über das man lesend und/oder schreibend auf Dateien zugreifen kann; diese erwartet zwei String-Argumente: einen Dateinamen und einen Modus (`'w'` für schreibenden Zugriff, `'r'` für lesenden Zugriff).

#### Schreiben in Dateien

Ein Dateiobjekt besitzt die Methode

`⟨dateiobj⟩.write(⟨string⟩)`

die ⟨*string*⟩ in ⟨*dateiobj*⟩ schreibt. Beispielsweise erzeugen die folgenden Pythonkommandos eine neue Datei `datei.txt`, die die beiden mit der `write`-Methode übergebenen Strings enthält:

```
>>> datei = open('datei.txt', 'w')
>>> datei.write('Hallo Welt\n')
>>> datei.write('Hier eine 2. Zeile.\n')
>>> datei.close()
```

Die `close`-Methode schließt ein Dateiobjekt. Normalerweise ist es jedoch nicht notwendig eine Datei explizit zu schließen, sondern der Python-Interpreter entscheidet im Allgemeinen selbst darüber, wann ein „guter“ Zeitpunkt dafür ist, ein Dateiobjekt aus dem Speicher zu entfernen.

**Aufgabe 3.41**

Schreiben Sie ein kleines Unix-Shellskript, das genau dasselbe Ergebnis liefert, wie das obige Python-Skript.

**Aufgabe 3.42**

Schreiben Sie eine Python-Funktion `writeLinesToFile`, die zwei Argumente übergeben bekommen soll: Einen Dateinamen und eine Liste von Strings; sie soll eine Datei mit dem übergebenen Namen erzeugen und jeden String der Liste in eine extra Zeile dieser Datei schreiben. So soll der Aufruf

```
writeLinesToFile ('datei.txt',['Hallo Welt', 'Hier eine 2. Zeile'])
```

die gleiche Wirkung haben wie das Skript in vorigem Beispiel.

Die Methode

⟨*dateiobj*⟩`.writelines(`⟨*liste*⟩`)`

kann übrigens eine Stringliste in eine Datei schreiben. Beispielsweise kann man mit folgendem Kommando

```
>>> open('test.txt','w').writelines(['Hallo\n','Welt\n'])
```

zwei Zeilen in die Datei `test.txt` schreiben.

#### Lesen von Dateien

Die Methode

⟨*dateiobj*⟩`.read()`

gibt den kompletten Inhalt einer Datei in einem Python-String zurück. Folgendes Kommando gibt beispielsweise die ersten 11 Zeichen der Datei `text.txt` zurück:

```
>>> open('test.txt').read()[:10]
'Hallo Welt\n'
```

**Aufgabe 3.43**

Geben Sie ein Pythonkommando an, mit dem der Inhalt der Datei `test.txt` ...

(a) rückwärts ausgegeben wird.
(b) zur Hälfte ausgegeben wird.
(c) komplett bis auf das letzte und das vorletzte Zeichen ausgegeben wird.

Die Methode

⟨*dateiobj*⟩`.readlines()`

gibt eine Liste von Strings zurück; jeder Eintrag in dieser Liste entspricht einer Zeile von ⟨*dateiobj*⟩. Folgendes Kommando liest beispielsweise die Zeilen der Datei `test.txt` aus:

```
>>> open('test.txt').readlines()
['Hallo Welt\n', 'Hier eine 2. Zeile\n']
```

**Aufgabe 3.44**

Was bewirkt das Kommando `open('datei.txt').readlines()[:-2]`

**Aufgabe 3.45**

Schreiben Sie eine Python-Funktion `myHead`, die einen Dateinamen und eine Zahl *n* als Argument erhält und die ersten *n* Zeilen der Datei auf dem Bildschirm ausgibt. Der Aufruf `myHead('test.txt',4)` sollte beispielsweise die ersten 4 Zeilen der Datei `test.txt` auf dem Bildschirm ausgeben.

**Aufgabe 3.46**

Programmieren Sie das Shell-Kommando `head` in Python nach und nennen Sie Ihr eigenes Kommando `myhead`. Das selbst geschriebene `myhead` soll zwei Optionen kennen:

- `-c` [-]*N*: Die ersten *N* Bytes jeder Datei werden ausgegeben. Falls ein „-“ vorgestellt ist, dann sollen alle Bytes außer den *N* letzten ausgegeben werden.
- `-n` [-]*N*: Die ersten *N* Zeilen jeder Datei werden ausgegeben. Falls ein „-“ vorgestellt ist, dann sollten alle Zeilen außer den *N* letzten ausgegeben werden.

Die Verwendung von `readlines` hat allerdings insbesondere bei großen Dateien einen Nachteil: es liest den *gesamten* Inhalt einer Datei sofort aus, und dadurch wird viel Speicherplatz verbraucht. Will man das vermeiden, dann kann man den *Iterator* verwenden, der in jedem Datei-Objekt vorhanden ist. Einen Iterator enthalten auch Listen, Strings, Tupel, Dictionaries usw.; mit ihm kann man nach und nach die Einträge eines Sequenz-Objekts durchlaufen. Der Datei-Iterator liest immer so viele Zeilen aus der Datei aus, wie auch tatsächlich gebraucht werden. Man kann etwa eine `for`-Schleife verwenden und damit die Zeilen einer Datei einzeln auslesen:

```
>>> datei = open('datei.txt','r')
```

```
>>> for zeile in datei
...     print zeile,
Hallo Welt
Hier eine 2. Zeile
```

Würde man sich etwa nur für die ersten 1000 Zeilen einer sehr langen Datei interessieren – das ist häufig bei Log-Dateien der Fall – täte man gut daran, die Datei nicht mittels `open(...).readlines()` zu öffnen, sondern mittels `for` die ersten 1000 Zeilen auszulesen.

**Aufgabe 3.47**

Verwenden Sie die `map`-Funktion, um genau dasselbe wie mit der gerade eben vorgestellten Listenkomprehension zu implementieren, nämlich die Ausgabe der Liste der großgeschriebenen Zeilen der Datei `datei.txt`.

**Aufgabe 3.48**

Geben Sie ein Python-Kommando an, das die Liste der Länge aller Zeilen der Datei `test.txt` ausgibt.

**Aufgabe 3.49**

(a) Schreiben Sie eine Python-Funktion, die in der Datei `test.txt` alle Vorkommen des Wortes `eines` in das Wort `keines` umwandelt und dies in die (neu anzulegende) Datei `test.txt.2` speichert.

(b) Wir verallgemeinern die Lösung zu Teilaufgabe (a): Schreiben Sie eine Python-Funktion `dateiRepl`, die drei Argumente übergeben bekommt: einen Dateinamen und zwei Strings. Es sollen dann alle Vorkommen des einen Strings durch den anderen String ersetzt werden, und das Ergebnis in einer neuen Datei mit Endung `.2` gespeichert werden. Der Aufruf `dateiRepl('test.txt','eines','keines')` soll also genau den Effekt haben, der in der vorigen Teilaufgabe verlangt wurde.

### 3.6.2 Dateimanipulation mit Listenkomprehensionen

Den Datei-Iterator kann man auch in Listenkomprehensionen verwenden, wie etwa in folgendem Beispiel; hier sollen alle Klein- in Großbuchstaben umwandelt und als Zeilenliste zurückliefert werden.

```
>>> [line.upper() for line in open('datei.txt')]
['HALLO WELT', 'HIER EINE 2. ZEILE']
```

Mit Listenkomprehensionen und entsprechenden String- und Sequenzfunktionen lassen sich elegant einfach Dateimanipulationen oder Suchoperationen in Dateien ausdrücken. Wir geben im Folgenden einige Beispiele hierfür an.

**i) Gib alle Zeilen aus `test.txt` aus, die weniger als 5 Zeichen enthalten.**

```
[zeile for zeile in open('test.txt') if len(zeile)<5]
```

Die Schleifenvariable `zeile` durchläuft nacheinander alle Zeilen der Datei `test.txt`. Aufgrund der `if`-Klausel besteht die resultierende Liste aber nur aus den Zeilen, für die `len(zeile)<5` gilt.

**Aufgabe 3.50**

Verwenden Sie eine Listenkomprehension, um alle Zeilen der Datei `test.txt` auszugeben, die …

(a) … mit der Zeichenkette `'Py'` beginnen.
(b) … die die Zeichenkette `'Python'` enthalten.
(c) … deren letztes Zeichen ein Kleinbuchstabe ist.

**ii) Was ist die Länge der längsten Zeile der Datei `test.txt`?**

```
max([len(zeile) for zeile in open('text.txt')])
```

Die Listenkomprehension erzeugt eine Liste aller Zeilenlängen der Datei `test.txt`, d. h. eine Integerliste. Die Listenfunktion `max` liefert den größten Integerwert, also die Länge der längsten Zeile, zurück.

**Aufgabe 3.51**

Berechnen Sie unter Verwendung einer Listenkomprehension …

(a) die Summe aller Längen von Zeilen, die mit `'a'` beginnen.
(b) die Länge der kürzesten Zeile, die keine Leerzeichen enthält.

**iii) Welches ist die längste Zeile der Datei `test.txt`?**

```
max([(len(zeile),zeile) for zeile in open('test.txt')])[1]
```

Diese Fragestellung ist der vorigen sehr ähnlich, nur wollen wir hier nicht die Länge, sondern die Zeile selbst zurückgeliefert haben, d. h. wir müssen in der Listenkomprehension beide Informationen strukturiert, etwa in einem Tupel, mitführen: sowohl die Längen der Zeilen (die für die Maximumsbildung benötigt werden) als auch die Zeilen selbst. Dass die Zeilenlänge die erste Komponente der Tupel bildet, die durch

die Listenkomprehension erzeugt wird, ist kein Zufall: Sortierungen, Minimums- und Maximumsbildung erfolgen *lexikografisch*. Ausschlaggebend für diese Sortierungen ist zunächst die erste Komponente einer Struktur. Sollten die ersten Komponenten zweier Objekte identisch sein, so ist die zweite Komponente für deren Sortierung ausschlaggebend, usw. Die „alphabetische" Sortierung beispielsweise in einem Telefonbuch stellt ein Beispiel einer lexikografischen Sortierung dar: Zuerst wird nach dem ersten Buchstaben gruppiert, dann, sollte der erste Buchstabe zweier Wörter identisch sein, wird nach dem zweiten Buchstaben gruppiert, usw.

Die Anwendung der `max`-Funktion auf obige Listenkomprehension liefert also dasjenige Tupel aus Zeile und zugehöriger Zeilenlänge zurück, das die maximale Länge hat. Um die Zeile selbst zu erhalten, müssen wir mittels des Indexoperators ⟨*seq*⟩`[1]` nur noch die zweite Komponente dieses Tupels extrahieren.

Wir haben gerade eben ein häufig verwendetes Muster kennengelernt, das in der Literatur auch als „Decor-Undecor-Idiom" bezeichnet wird: Listenwerte werden mit Informationen angereichert, d. h. „dekoriert", um sie nach bestimmten Kriterien zu sortieren. Anschließend wird diese Dekoration wieder entfernt („undecor"), was in obigem Beispiel mittels des Indexoperators ⟨*seq*⟩`[1]` erfolgt.

**Aufgabe 3.52**

Verwenden Sie eine Listenkomprehension, um diejenige Zeile der Datei `'test.txt'` auszugeben, die …

(a) …am häufigsten das Zeichen `'a'` enthält.
(b) …die meisten Wörter enthält.
(c) …das längste Wort enthält.

**iv) Wie viele Leerzeilen enthält die Datei `test.txt`?**

```
len([lz for lz in open('test.txt') if len(lz)==1])
```

Die erzeugte Liste enthält nur diejenigen Zeilen aus der Datei `test.txt`, die genau ein Zeichen enthalten, d. h. für die `len(zeile)==1` gilt. Dies sind genau die Leerzeilen, denn diese bestehen lediglich aus dem Newline-Zeichen `'\n'`. Um zu wissen, wie viele Leerzeilen gefunden wurden, müssen die Einträge in der Listenkomprehension gezählt werden. Dies erledigt die Anwendung der `len`-Funktion.

**v) Welche in der Datei `test.txt` enthaltenen Wörter beginnen mit `'a'`.**

```
[w for zeile in open('test.txt')
  for w in zeile.split() if w.startswith('a')]
```

Dies lässt sich am besten über eine Listenkomprehension lösen, die zwei geschachtelte Schleifen verwendet. Die äußere Schleife durchläuft die Zeilen der Datei `test.txt`. Für jede einzelne Zeile durchläuft die Variable `w` der inneren Schleife die Liste der Wör-

ter dieser Zeile, erzeugt durch den Ausdruck `zeile.split()`. Durch die `if`-Anweisung werden nur diejenigen Wörter „durchgelassen“, die mit `'a'` beginnen.

**Aufgabe 3.53**

Was ist der „Sinn“ des folgenden Statements?

```
[x[:-1] for x in open('test.txt')]
```

### 3.6.3 Verzeichnisse

Eine häufige Aufgabe besteht darin, eine bestimmte Operation auf eine große Menge von Dateien in einem bestimmten Verzeichnis oder in einem ganzen Verzeichnisbaum anzuwenden. Möglicherweise wollen Sie bestimmte Ersetzungen in allen `*.tex`-Dateien in einem bestimmten Verzeichnis vornehmen oder nach bestimmten Java-Schlüsselwörtern in allen Java-Dateien auf Ihrem Home-Verzeichnisbaum suchen.

Manche dieser Aufgaben sind vielleicht in der Bash einfacher zu realisieren, aber in Python können Sie diese Aufgaben plattformunabhängig programmieren[7]. Außerdem stehen Ihnen unter Python viel mächtigere Möglichkeiten, wie strukturierte Datentypen, Iteratoren oder Listenkomprehensionen zur Verfügung als unter der Bash. Je komplexer Ihre Aufgabe ist, die Sie realisieren wollen, desto empfehlenswerter ist die einer höheren getypten Programmiersprache wie Python anstelle der Verwendung eines Bash-Skripts.

#### Durch die Einträge eines Verzeichnisses iterieren

Der Aufruf

`os.listdir(⟨dir⟩)`

liefert eine String-Liste der Namen aller Einträge im Verzeichnis ⟨*dir*⟩ zurück. Ein Beispiel:

```
>>> import os
>>> os.listdir('.')
['graphen.jpg', 'hs-albsig.jpg', 'Analysis Klausur.tex']
```

7 Ein Bash-Skript ist immer plattformabhängig, d. h. in diesem Fall: nur auf Unix-Systemen oder Unix-artigen-Systemen lauffähig – nicht jedoch unter Windows.

**Aufgabe 3.54**

Geben Sie einen Python-Einzeiler an, mit dem Sie ...

(a) sich die Anzahl der Einträge im aktuellen Verzeichnis ausgeben lassen.
(b) sich die Längen aller Dateinamen im aktuellen Verzeichnis ausgeben lassen.
(c) sich die Anzahl der Dateien im aktuellen Verzeichnis mit Endung `.c` und mit Endung `.py` ausgeben lassen.
(d) sich alle Dateien mit Endung `.txt` aus dem aktuellen Verzeichnis ausgeben lassen.
(e) sich die Anzahl der Zeilen aller Dateien mit Endung `.txt` ausgeben lassen.

#### Durch die Einträge eines Verzeichnisbaums iterieren

Der Aufruf

`os.walk(`⟨*dir*⟩`)`

liefert einen Iterator über den Verzeichnisbaum unterhalb des Verzeichnisses `dir` zurück. Jeder Aufruf dieses Iterators liefert ein 3-Tupel der folgenden Form zurück:

(⟨*dirpath*⟩, ⟨*dirnames*⟩, ⟨*filenames*⟩)

Dabei ist ⟨*dirpath*⟩ ein String, der den Pfad zum momentan besuchten Verzeichnis enthält, ⟨*dirnames*⟩ ist eine Liste von Strings, die die Namen der Unterverzeichnisse des momentan besuchten Verzeichnisses enthält und ⟨*filenames*⟩ ist eine Liste von Strings, die die Namen der Dateien, die keine Verzeichnisse sind, im momentan besuchten Verzeichnis enthält.

Folgendes Programmbeispiel findet alle Dateien mit Endung `.tex`, in denen sich der String `python` befindet, im Verzeichnisbaum unterhalb `'.'`:

```
>>> import os.path
>>> treffer = []
>>> for (verzName, verzeichnisse, dateien) in os.walk('.'):
      for dateiname in dateien:
        if dateiname.endswith('.tex'):
          pfadname = os.path.join(verzName, dateiname)
          if 'python' in open(pfadname).read():
            treffer.append(dateiname)
>>> treffer
['blatt3.tex', 'a4.tex', 'Python_Uebungsblatt.tex', 'einfinf2.tex',
'python.tex', 'einfinf.tex', 'KlausurEinfInf0910.tex']
```

Einige Erläuterungen hierzu: `verzName` ist immer der Name des Verzeichnisses, das gerade angelaufen wurde; `verzeichnisse` ist die Liste der Unterverzeichnisse (für die wir uns in diesem Fall nicht interessieren) und `dateien` ist die Liste aller Datei-

en im Verzeichnis, das gerade angelaufen wurde. Mit der String-Methode `endswith` wird geprüft, ob der Name der entsprechenden Datei mit `.tex` endet. Mit der Funktion `os.path.join` werden der Pfadname und der Dateiname zusammengehängt[8]. Diese vollständigen Pfadnamen benötigen wir für die Funktion `open`, die den gesamten Inhalt der Datei in einen String einliest; mit der Operation `in` (siehe Abschnitt 3.2.5) können wir testen, ob dieser den String `'python'` enthält.

Mit noch weniger Codezeilen kommt man aus, wenn man für diese Aufgabe eine Listenkomprehension verwendet:

```
>>> [for (verzName, verzeichnisse, dateien) in os.walk('.')
      for dateiname in dateien if dateiname.endswith('.tex')
       if 'python' in open(join(verzName, dateiname)).read()]
```

**Aufgabe 3.55**

(a) Schreiben Sie ein Python-Skript, das Ihnen die Datei mit dem längsten Namen (unter allen Dateien in Ihrem kompletten Homeverzeichnis) zurückliefert.
(b) Schreiben Sie ein Python-Skript, das Ihnen diejenige `.py`-Datei (unter alle `.py`-Dateien in Ihrem kompletten Home-Verzeichnisbaum) zurückliefert, die die längste Zeile besitzt.

**Aufgabe 3.56**

Schreiben Sie eine Funktion, die zählt, wie oft das Schlüsselwort `for` in allen Python-Dateien in Ihrem kompletten Homeverzeichnisbaum vorkommt.

## 3.7 Objektorientierte Programmierung

Zentral für die objektorientierte Programmierung ist die Möglichkeit neue *Klassen* erzeugen zu können. Eine Klasse ist eigentlich nichts anderes ist als ein Python-Typ, genau wie `int`, `str`, `list` oder `dict`. Ein bestimmter Wert einer Klasse wird im Sprachjargon der Objektorientierten Programmierung auch als *Objekt* bezeichnet. Die Aussage

„`[2,4,6]` ist ein Wert vom Typ `list`“

würde man im Sprachjargon der Objektorientieren Programmieren dagegen eher als

**8** Wie das genau geschieht ist abhängig vom Betriebssystem. Unter Unix werden Pfade mit \ zusammengehängt; unter Windows dagegen mit /.

„`[2,4,6]` ist ein Objekt der Klasse **`list`“**

formulieren.

Die Kommandos, die eine Klasse anbietet, um ihre Objekte zu manipulieren oder Berechnungen auf ihnen auszuführen, werden im Sprachjargon der Objektorientierten Programmierung als *Methoden* bezeichnet. Der Aufruf eines Kommandos `f`, das Berechnungen über ein Objekt ⟨*obj*⟩ durchführt und zusätzliche Argument ⟨$arg_1$⟩, ⟨$arg_2$⟩, ... bekommt, wird im Zusammenhang mit der Objektorientierten Programmieren nicht als `f(`⟨*obj*⟩`,`⟨$arg_1$⟩`,` ⟨$arg_2$⟩`, ...)` geschrieben, sondern in der Syntax

⟨*obj*⟩`.f(`⟨$arg_1$⟩`,` ⟨$arg_2$⟩`, ...)`

geschrieben.

Kommandos wie beispielsweise **`list`**`.append`, **`dict`**`.keys` oder **`str`**`.endswidth` werden wir in diesem Kapitel also grundsätzlich als *Methoden* bezeichnen.

### 3.7.1 Definition und Verwendung einer Klasse

Zunächst ist es notwendig, dass die Leser ein grobe Vorstellung davon bekommen, was eine Klasse ist und wie sie verwendet wird. Grob gesprochen, ist eine Klasse eine Gruppierung von Daten und dazugehörigen Funktionen, die in diesem Zusammenhang Methoden genannt werden. Die in einer Klasse enthaltenen „Dinge“ sollten konzeptuell zusammengehören. Häufig beziehen sich Klassen auf Dinge der realen Welt, wie etwa Kunden, Personen, Autos, graphische Objekte oder andere real vorhandenen Dinge, die in der Software repräsentiert sein sollen.

Die Syntax zur Erzeugung einer neuen Klasse lautet:

**`class`** ⟨*name*⟩`:`
    ⟨$kommando_1$⟩
    ...
    ⟨$kommando_n$⟩

Die Kommandos werden bei Erzeugung der Klasse alle ausgeführt. Typischerweie sind diese Kommandos Methoden-Definitionen oder Variablen-Zuweisungen. Die Variablen, denen innerhalb einer Klasse ein Wert zugewiesen wird, werden üblicherweise als *Attribute* bezeichnet. Listing 3.41 zeigt ein Beispiel einer einfachen Klassendefinition:

```
class Auto:
  typ = 'VW Golf'
  def sagHallo(self):
    print('Hallo, ich bin ein Auto vom Typ' +Auto.typ)
```

**Listing 3.41.** Definition einer einfachen Klasse

In Zeile 2 wird eine Variable `typ` definiert und in Zeile 3 eine Methode `sagHallo`. Eine solche Variable heißt auch Attribut oder Klassenvariable. Alle Methoden haben Zugriff auf diese Klassenvariable über die Referenz `Auto.typ`. Jede Methode *muss* als erstes Argument die Variable „**`self`“ übergeben bekommen. Die Variable **`self`** enthält immer die Referenz auf das Objekt selbst. Dieses wird bei jedem Methodenaufruf implizit mit übergeben. Betrachten wir beispielsweise den Methodenaufruf `lst.append(3)`: In diesem Fall ist **`self`** nichts anderes als das Listenobjekt `lst`.

Durch die Zuweisung

```
>>> einAuto = Auto()
```

kann man eine *Instanz* der Klasse erzeugen, d. h. eine Variable vom Typ `Auto`, im OO-Sprachjargon als ein *Objekt* bezeichnet, in diesem Fall ein Objekt der Klasse `Auto`. Auf die Methode `sagHallo` kann man mittels `einAuto.sagHallo` zugreifen:

```
>>> Auto.typ
'VW Golf'
>>> einAuto.sagHallo()
'Hallo, ich bin ein Auto'
```

Enthält eine Klassendefinition die Methode `__init__`, so wird diese Methode bei jedem Erzeugen eines Objektes automatisch ausgeführt. Neben dem obligaten Argument **`self`** kann die `__init__`-Methode noch weitere Argumente enthalten, und die Erzeugung von Objekten kann so abhängig von bestimmten Parametern erfolgen. Listing 3.42 zeigt eine modifizierte Definition der Klasse `Auto`:

```
class Auto:
  anzAutos = 0

  def __init__(self, t, f):
    self.typ = t
    self.farbe = f
    Auto.anzAutos += 1

  def __del__(self):
    Auto.anzAutos -= 1

  def ueberDich(self):
    print("Ich bin ein %ser %s. Die Klasse enthaelt %d Autos." % \
          (self.farbe, self.typ, Auto.anzAutos))
```

**Listing 3.42.** Definition einer komplexeren `Auto`-Klasse

Bei der Erzeugung einer neuen Instanz von `Auto` wird nun immer automatisch die `__init__`-Methode ausgeführt, die neben **`self`** zwei weitere Argumente erwartet, die dann in Zeile 6 und 7 den (Objekt-)Attributen `typ` und `farbe` zugewiesen werden. Man

sieht, dass man mittels `self.typ` bzw. `self.farbe` auf die Attribute `typ` bzw. `farbe` des aktuellen Objektes zugreifen kann.

Die Attribute `self.typ` und `self.farbe` gehören also zu *einem* bestimmten Objekt der Klasse `Auto` und können für unterschiedliche Objekte unterschiedliche Werte annehmen. Dagegen ist das in Zeile 2 definierte Attribut `anzAutos` ein Klassenattribut, d. h. es gehört nicht zu einer bestimmten Instanz von `Auto`, sondern ist global für alle Objekte der Klasse sichtbar. Gleiches gilt für alle Methodendeklarationen – auch sie sind global für alle Objekte der Klasse sichtbar.

Bei jeder Erzeugung einer Klasseninstanz erhöhen wir die Variable `anzAutos` um Eins. Die in Zeile 10 definierte spezielle Methode `__del__` wird immer dann automatisch aufgerufen, wenn mittels des `del`-Kommandos ein Objekt der Klasse gelöscht wird; in Zeile 11 erniedrigen wir die Variable `anzAutos` um Eins, wenn ein Objekt gelöscht wird.

Wir erzeugen in den Variablen `a1`, `a2` und `a3` drei verschiedene Variablen vom Typ `Auto`:

```
>>> a1 = Auto("Mercedes-Benz", "gruen")
>>> a2 = Auto("BMW", "rot")
>>> a3 = Auto("VW Golf", Schwarz")
```

und können uns nun mittels der Methode `ueberDich` Informationen über das jeweilige Objekt ausgeben lassen:

```
>>> a1.ueberDich()
Ich bin ein gruener Mercedes-Benz. Die Klasse enthaelt 3 Autos.
>>> del(a1)
>>> a2.ueberDich()
Ich bin ein roter BMW. Die Klasse enthaelt momentan 2 Autos.
```

Man kann auch eine neue Klasse erzeugen, die auf den Attributen und Methoden einer anderen Klasse basiert – im Jargon der Objektorientierten Programmierung nennt man das auch *Vererbung*. Es ist grundsätzlich so, dass eine Klasse, die ein spezielleres Konzept modelliert, immer von der Klasse erbt, die ein allgemeineres Konzept repräsentiert. Stellen wir uns vor, wir wollten die drei Klassen `Hochschulangehoeriger`, `Professor` und `InformatikProfessor` definieren. Dann wäre es sinnvoll, die Klasse `Professor` von der Klasse `Hochschulangehoeriger` erben zu lassen und die Klasse `InformatikProfessor` von der Klasse `Professor` erben zu lassen. Oder anknüpfend an unser obiges Beispiel mit der Klasse `Auto`, könnte man sich vorstellen, eine neue Klasse `Oldtimer` zu definieren, die von der Klasse `Auto` erbt. Das Alter eines Autos beispielsweise könnte uns nur dann interessieren, wenn es sich um einen Oldtimer handelt. Diese Tatsache kann man wie folgt in Python realisieren:

```
class Oldtimer(Auto):
  def __init__(self, t, f, a):
    Auto.__init__(self,t,f)
```

```
    self.alter = a
  def ueberDich(self):
    Auto.ueberDich(self)
    print "Ausserdem bin ich %d Jahr alt" % self.alter
```

Wie man sieht, muss gleich bei der Deklaration der Klasse die Klasse, von der geerbt wird (hier: `Auto`) in Klammern mit angegeben werden. Ebenfalls zu sehen ist, dass man die `__init__`-Methode der Basisklasse explizit aufrufen muss. Gleiches gilt auch für andere gleichlautende Methoden: Die Methode `ueberDich` muss die gleichlautende Methode der Basisklasse explizit aufrufen. Wir können nun ein Objekt vom Typ `Oldtimer` folgendermaßen erzeugen und verwenden:

```
>>> o1 = Oldtimer("BMW", "grau", 50)
>>> o1.ueberDich()
Ich bin ein grauer BMW; du hast momentan 3 Autos
Ausserdem bin ich 50 Jahr alt
```

Neben der `__init__`-Methode und der `__del__`-Methode gibt es in Python noch eine Reihe weiterer Methoden mit spezieller Bedeutung, unter anderem:

- `__str__(self)`: Diese Methode berechnet die String-Repräsentation eines bestimmten Objektes; sie wird durch Pythons interne Funktion `str(...)` und durch die `print`-Funktion aufgerufen.
- `__cmp__(self,x)`: Diese Methode wird bei Verwendung von Vergleichsoperationen aufgerufen; sie sollte eine negative ganze Zahl zurückliefern, falls `self<x`; sie sollte `0` zurückliefern, falls `self==x`; sie sollte eine positive ganze Zahl zurückliefern, falls `self>x`.
- `__getitem__(self,i)`: Wird bei der Auswertung des Ausdrucks `self[i]` ausgeführt.
- `__setitem__(self,i,v)`: Wird bei einer Zuweisung `self[i]=v` ausgeführt.
- `__len__(self)`: Wird bei der Ausführung der Python internen Funktion `len(...)` aufgerufen.

**Aufgabe 3.57**

Erstellen Sie eine Python-Klasse `Auto`. Die Klasse soll die Methoden `fahre`, `kaputt` und `repariere` haben.
Nach dreimaligem Fahren (d. h. nach dreimaligem Aufruf der Methode `fahre`) geht ein Auto kaputt. Danach kann man es nicht mehr fahren. Nach Aufruf von `repariere` ist das Auto repariert und es kann wieder gefahren werden, nach dreimal ist es wieder kaputt, usw. Die Methode `kaputt` liefert `True` oder `False` zurück, je nachdem, ob das Auto kaputt ist oder nicht. Beispiel:

```
>>> einAuto = Auto()
>>> einAuto.fahre()
```

```
"Ich fahre"
>>> einAuto.fahre()
"Ich fahre"
>>> einAuto.fahre()
"Ich fahre"
>>> einAuto.fahre()
"Ich bin kaputt"
>>> einAuto.repariere()
"Jetzt kann ich wieder fahren"
>>> einAuto.fahre()
"Ich fahre"
```

Programmieren Sie in Python die Klasse `Auto`, die sich wie oben beschrieben verhält – inklusive der drei Methoden `fahre`, `repariere` und `kaputt`.

**Aufgabe 3.58**

Erstellen Sie eine Pythonklasse `Stack`. Die Stack-Datenstruktur, auch Stapel-Datenstruktur genannt, bietet eine Sammlung von Elementen an, die logisch in Form eines Stapels (von Elementen, Aufgaben, usw.) angeordnet sind. Eingefügt werden kann ein neues Element immer nur oben auf dem Stapel. Das Einfügen bezeichnet man in diesem Zusammenhang auch als Push-Operation. Ausgelesen werden kann ein Element ebenfalls nur von oben. Das Auslesen eines Elements und anschließendes Löschen bezeichnet man im Zusammenhang mit der Stapel-Datenstruktur als Pop-Operation.

Nachfolgendes Beispiel soll die Funktionsweise der Klasse `Stack` veranschaulichen:

```
>>> s = Stack()
>>> s.push('f') ; s.push('g') ; s.push('x') ; s.push('a')
>>> s.pop()
'a'
>>> s.push('b')
>>> s.pop()
'b'
>>> s.pop()
'x'
```