

# Kapitel 2

## Die Programmiersprache Python

Im vorangegangenen Kapitel haben wir Ihnen einen Überblick über dieses Buch gegeben und besprochen, in welcher Weise Sie es lesen sollten. Jetzt wenden wir uns der Programmiersprache Python zu und beginnen mit einer Einführung in die Geschichte und die grundlegenden Konzepte. Die beiden letzten Abschnitte dieses Kapitels behandeln die Einsatzmöglichkeiten und -gebiete von Python. Betrachten Sie dieses Kapitel also als erzählerische Einführung in die Thematik, die den darauffolgenden fachlichen Einstieg vorbereitet.

### 2.1 Historie, Konzepte, Einsatzgebiete

#### 2.1.1 Geschichte und Entstehung

Die Programmiersprache Python wurde Anfang der 90er-Jahre von dem Niederländer *Guido van Rossum* am *Centrum voor Wiskunde en Informatica* (CWI) in Amsterdam entwickelt. Ursprünglich war sie als Skriptsprache für das verteilte Betriebssystem Amoeba gedacht. Der Name Python lehnt sich nicht etwa an die Schlangenart an, sondern ist eine Hommage an die britische Komikertruppe Monty Python.

Vor Python hatte van Rossum an der Entwicklung der Programmiersprache ABC mitgewirkt, die mit dem Ziel entworfen wurde, so einfach zu sein, dass sie problemlos einem interessierten Laien ohne Programmiererfahrung beigebracht werden kann. Die Erfahrung aus positiver und negativer Kritik an ABC nutzte van Rossum für die Entwicklung von Python. Er schuf damit eine Programmiersprache, die mächtig und zugleich leicht zu erlernen ist.

Mit der Version 3.0, die im Dezember 2008 erschien, wurde die Sprache von Grund auf überarbeitet. Dabei sind viele kleine Unschönheiten und Designfehler beseitigt worden, die man in bisherigen Versionen aufgrund der Abwärtskompatibilität stets in der Sprache behalten musste.

Mittlerweile hat sich Python zu einer der beliebtesten Programmiersprachen ihres Typs entwickelt und nimmt bei Popularitätsindizes von Programmiersprachen<sup>1</sup> regelmäßig Spitzenpositionen ein.

---

<sup>1</sup> zum Beispiel *TIOBE*, *RedMonk* oder *PYPL*

Seit 2001 existiert die gemeinnützige *Python Software Foundation*, die die Rechte am Python-Code besitzt und Lobbyarbeit für Python betreibt. So organisiert die Python Software Foundation beispielsweise die PyCon-Konferenz, die jährlich in den USA stattfindet. Auch in Europa finden regelmäßig größere und kleinere Python-Konferenzen statt.

### 2.1.2 Grundlegende Konzepte

Grundsätzlich handelt es sich bei Python um eine imperative Programmiersprache, die jedoch noch weitere *Programmierparadigmen* in sich vereint. So ist es beispielsweise möglich, mit Python objektorientiert und funktional zu programmieren. Sollten Sie mit diesen Begriffen im Moment noch nichts anfangen können, seien Sie unbesorgt, schließlich sollen Sie ja die Programmierung mit Python und damit die Anwendung der verschiedenen Paradigmen in diesem Buch lernen.

Obwohl Python viele Sprachelemente gängiger Skriptsprachen implementiert, handelt es sich um eine interpretierte Programmiersprache. Der Unterschied zwischen einer Programmier- und einer Skriptsprache liegt im *Compiler*. Ähnlich wie Java oder C# verfügt Python über einen Compiler, der aus dem Quelltext ein Kompilat erzeugt, den sogenannten *Byte-Code*. Dieser Byte-Code wird dann in einer virtuellen Maschine, dem *Python-Interpreter*, ausgeführt.

Ein weiteres Konzept, das Python zum Beispiel mit Java gemeinsam hat, ist die *Plattformunabhängigkeit*. Ein Python-Programm ist auf allen Betriebssystemen unmodifiziert lauffähig, die vom Python-Interpreter unterstützt werden. Darunter fallen insbesondere die drei großen Desktop-Betriebssysteme Windows, Linux und OS X.

Im Lieferumfang von Python ist neben dem Interpreter und dem Compiler eine umfangreiche *Standardbibliothek* enthalten. Diese Standardbibliothek ermöglicht es dem Programmierer, in kurzer Zeit übersichtliche Programme zu schreiben, die sehr komplexe Aufgaben erledigen können. So bietet Python beispielsweise umfassende Möglichkeiten zur Netzwerkkommunikation oder zur Datenspeicherung. Da die Standardbibliothek die Programmiermöglichkeiten in Python wesentlich bereichert, widmen wir ihr im dritten und teilweise auch vierten Teil dieses Buches besondere Aufmerksamkeit.

Ein Nachteil der Programmiersprache ABC, den van Rossum bei der Entwicklung von Python beheben wollte, war ihre fehlende Flexibilität. Ein grundlegendes Konzept von Python ist es daher, es dem Programmierer so einfach wie möglich zu machen, die Standardbibliothek beliebig zu erweitern. Da Python selbst, als interpretierte Programmiersprache, nur eingeschränkte Möglichkeiten zur maschinennahen Programmierung bietet, können maschinennahe oder zeitkritische Erweiterungen problemlos in C geschrieben werden. Das ermöglicht die *Python API*.

In Abbildung 2.1 ist das Zusammenwirken der bisher angesprochenen Konzepte Pythons zusammengefasst: Ein Python-Programm wird vom Python-Interpreter ausgeführt, der dabei eine umfangreiche Standardbibliothek bereitstellt, die vom Programm verwendet werden kann. Außerdem erlaubt es die Python API einem externen C-Programm, den Interpreter zu verwenden oder zu erweitern.

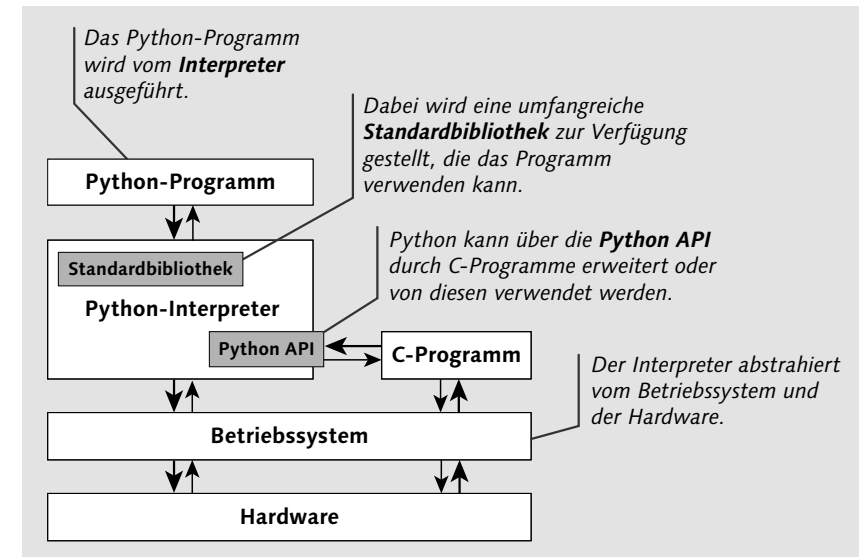


Abbildung 2.1 Veranschaulichung der grundlegenden Konzepte Pythons

Als letztes grundlegendes Konzept von Python soll erwähnt werden, dass Python unter der *PSF-Lizenz* steht. Das ist eine von der Python Software Foundation entworfene Lizenz für Open-Source-Software, die wesentlich weniger restriktiv ist als beispielsweise die GNU General Public License. So erlaubt es die PSF-Lizenz, den Python-Interpreter lizenzkostenfrei in Anwendungen einzubetten und mit diesen auszuliefern, ohne dass der Code offengelegt werden muss oder Lizenzkosten anfallen. Diese Politik macht Python auch für kommerzielle Anwendungen attraktiv.

### 2.1.3 Einsatzmöglichkeiten und Stärken

Die größte Stärke von Python ist *Flexibilität*. So kann Python beispielsweise als Programmiersprache für kleine und große Applikationen, als serverseitige Programmiersprache im Internet oder als Skriptsprache für eine größere C- oder C++-Anwendung verwendet werden. Auch abseits des klassischen Marktes breitet sich Python beispielsweise im Embedded-Bereich aus. So existieren Python-Interpreter für diverse Smartphone- bzw. Tablet-Systeme oder beispielsweise den Raspberry Pi.

Python ist aufgrund seiner *einfachen Syntax* leicht zu erlernen und gut zu lesen. Außerdem erlauben es die automatische Speicherverwaltung und die umfangreiche

Standardbibliothek, mit kleinen Programmen bereits sehr komplexe Probleme anzugehen. Aus diesem Grund eignet sich Python auch zum *Rapid Prototyping*. Bei dieser Art der Entwicklung geht es darum, in möglichst kurzer Zeit einen lauffähigen Prototyp als eine Art Machbarkeitsstudie einer größeren Software zu erstellen, die dann später in einer anderen Programmiersprache implementiert werden soll. Mithilfe eines solchen Prototyps lassen sich Probleme und Designfehler bereits entdecken, bevor die tatsächliche Entwicklung der Software begonnen wird.

Eine weitere Stärke Pythons ist die bereits im vorangegangenen Abschnitt angesprochene *Erweiterbarkeit*. Aufgrund dieser Erweiterbarkeit können Python-Entwickler aus einem reichen Fundus von Drittanbieterbibliotheken schöpfen. So gibt es etwa Anbindungen an die gängigsten GUI-Toolkits, die das Erstellen von Python-Programmen mit grafischer Benutzeroberfläche ermöglichen.

### 2.1.4 Einsatzbeispiele

Python erfreut sich großer Bekanntheit und Verbreitung sowohl bei Softwarefirmen als auch in der Open-Source-Gemeinschaft. Die Palette der Produkte, die zumindest zum Teil in Python geschrieben wurden, reicht von Webanwendungen (Google Mail, Google Maps, YouTube, Dropbox, reddit) über Filesharing-Plattformen (BitTorrent, Morpheus) und Entwicklungswerkzeuge (Mercurial, SCons) bis hin zu Computerspielen (Civilization IV, Battlefield 2, Eve Online).

Viele Anwendungen unterstützen Python als Skriptsprache für Erweiterungen. Beispiele dafür sind die Grafikanwendungen Maya, Blender, Cinema 4D, Paint Shop Pro und GIMP.

Neben den genannten gibt es unzählige weitere bekannte Anwendungen, die in Python geschrieben wurden oder in deren Umfeld Python eingesetzt wird. Lassen Sie sich anhand der oben dargestellten Beispiele sagen, dass Python eine beliebte, verbreitete und moderne Programmiersprache ist, die es sich lohnt zu erlernen.

## 2.2 Die Verwendung von Python

Die jeweils aktuelle Version von Python können Sie von der offiziellen Python-Website unter <http://www.python.org> als Installationsdatei für Ihr Betriebssystem herunterladen und installieren. Alternativ finden Sie Python 3.4 auf der CD, die diesem Buch beiliegt.

Grundsätzlich werden, wenn man einmal von Python selbst absieht, zwei wichtige Komponenten installiert: der interaktive Modus und IDLE.

Im sogenannten *interaktiven Modus*, auch *Python-Shell* genannt, können einzelne Programmzeilen eingegeben und die Ergebnisse direkt betrachtet werden. Der inter-

aktive Modus ist damit unter anderem zum Lernen der Sprache Python interessant und wird deshalb in diesem Buch häufig verwendet.

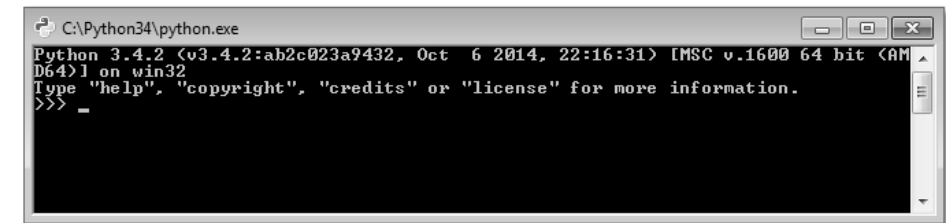


Abbildung 2.2 Python im interaktiven Modus (Python-Shell)

Bei *IDLE* (Integrated DeveLopment Environment) handelt es sich um eine rudimentäre Python-Entwicklungsumgebung mit grafischer Benutzeroberfläche. Beim Starten von IDLE wird zunächst nur ein Fenster geöffnet, das eine Python-Shell beinhaltet. Zudem kann über den Menüpunkt **FILE • NEW WINDOW** eine neue Python-Programmdatei erstellt und editiert werden. Nachdem die Programmdatei gespeichert wurde, kann sie über den Menüpunkt **RUN • RUN MODULE** in der Python-Shell von IDLE ausgeführt werden. Abgesehen davon, bietet IDLE dem Programmierer einige Komfortfunktionen wie beispielsweise das farbige Hervorheben von Code-Elementen (»Syntax Highlighting«) oder eine automatische Code-Vervollständigung.

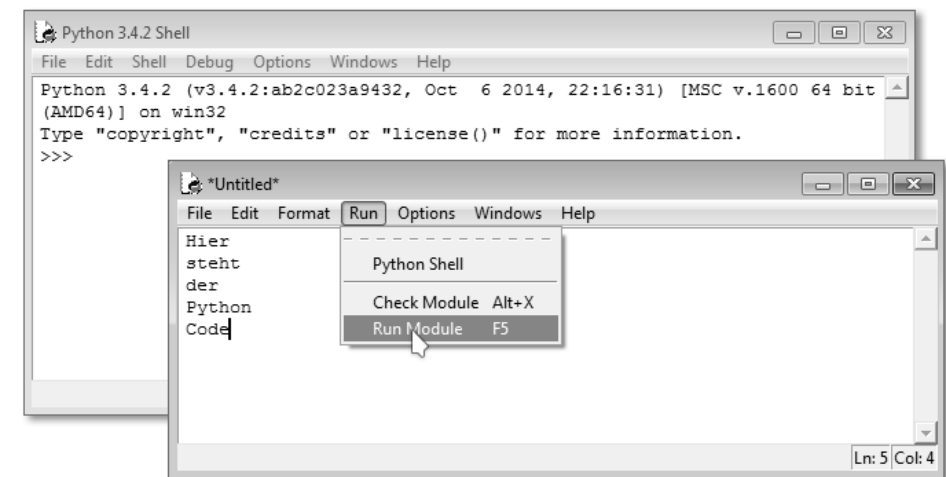


Abbildung 2.3 Die Entwicklungsumgebung IDLE

Wenn Sie mit IDLE nicht zufrieden sind, finden Sie eine Übersicht über die verbreitetsten Python-Entwicklungsumgebungen im Anhang dieses Buches. Zudem erhalten Sie auf der offiziellen Python-Website unter <http://wiki.python.org/moin/PythonEditors> eine umfassende Auflistung von Entwicklungsumgebungen und Editoren für Python.

Die folgenden Abschnitte geben Ihnen eine kurze Einführung dazu, wie Sie den interaktiven Modus und IDLE auf Ihrem System starten und verwenden.

### 2.2.1 Windows

Sie finden die Windows-Installationsdatei von Python 3.4 auf der dem Buch beigelegten CD-ROM.

Nach der Installation von Python unter Windows sehen Sie im Wesentlichen zwei neue Einträge im Startmenü: PYTHON (COMMAND LINE) und IDLE (PYTHON GUI). Ersterer startet den interaktiven Modus von Python in der Kommandozeile (»schwarzes Fenster«) und Letzterer die grafische Entwicklungsumgebung IDLE.

### 2.2.2 Linux

Beachten Sie, dass Python bei vielen Linux-Distributionen bereits im Lieferumfang enthalten ist. Die meisten Distributionen werden dabei standardmäßig Python 2.x mitbringen. Python 3.4 muss eventuell über den Paketmanager Ihrer Distribution nachinstalliert werden. Die beiden Versionen können aber problemlos gleichzeitig installiert sein.

Sollten Sie eine Distribution ohne Paketmanager einsetzen oder sollte Python 3.4 nicht verfügbar sein, müssen Sie den Quellcode von Python selbst kompilieren und installieren. Dazu können Sie den Anweisungen der im Quelltext enthaltenen *Readme*-Datei folgen.

Sie finden den Quellcode von Python 3.4 auf der dem Buch beigelegten CD-ROM.

Nach der Installation starten Sie den interaktiven Modus bzw. IDLE aus einer Shell heraus mit den Befehlen `python` bzw. `idle`.



#### Hinweis

Bei vielen Distributionen werden Sie Python 3.x mit einem anderen Befehl, beispielsweise `python3`, starten müssen, da diese Python 2.x und 3.x parallel installieren.

### 2.2.3 OS X

Sie finden die Installationsdatei für OS X von Python 3.4 auf der dem Buch beigelegten CD-ROM.

Nach der Installation von Python starten Sie den interaktiven Modus und IDLE, ähnlich wie bei Linux, aus einer Terminal-Sitzung heraus mit den Befehlen `python3` bzw. `idle`.

## TEIL I

# Einstieg in Python

*Herzlich willkommen im ersten Teil dieses Buches. Hier finden Sie eine einsteigerfreundliche Einführung in die grundlegenden Elemente von Python. Wir beginnen damit, einfache Kommandos im interaktiven Modus auszuprobieren, und wenden das erworbene Wissen danach in einem ersten einfachen Beispielprogramm an. Es folgen Kapitel zu den grundlegenden Kontrollstrukturen und eine Einführung in das Python zugrunde liegende Laufzeitmodell. Abschließend finden Sie als Übergang zum zweiten Teil ein Kapitel zur Verwendung von Funktionen, Methoden und Attributen.*

## Kapitel 3

### Erste Schritte im interaktiven Modus

3

Startet man den Python-Interpreter ohne Argumente, gelangt man in den sogenannten *interaktiven Modus*. Dieser Modus bietet dem Programmierer die Möglichkeit, Kommandos direkt an den Interpreter zu senden, ohne zuvor ein Programm erstellen zu müssen. Der interaktive Modus wird häufig genutzt, um schnell etwas auszuprobieren oder zu testen. Zum Schreiben wirklicher Programme ist er allerdings nicht geeignet. Dennoch möchten wir hier mit dem interaktiven Modus beginnen, da er Ihnen einen schnellen und unkomplizierten Einstieg in die Sprache Python ermöglicht.

Dieser Abschnitt soll Sie mit einigen Grundlagen vertraut machen, die zum Verständnis der folgenden Kapitel wichtig sind. Am besten setzen Sie die Beispiele dieses Kapitels am Rechner parallel zu Ihrer Lektüre um.

Zur Begrüßung gibt der Interpreter einige Zeilen aus, die Sie in ähnlicher Form jetzt auch vor sich haben müssten:

```
Python 3.4.1 (default, May 19 2014, 17:23:49)
[GCC 4.9.0 20140507 (prerelease)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Nach der Eingabeaufforderung (`>>>`) kann beliebiger Python-Code eingegeben werden. Bei Zeilen, die nicht mit `>>>` beginnen, handelt es sich um Ausgaben des Interpreters.

Zur Bedienung des interaktiven Modus sei noch gesagt, dass er über eine *History-Funktion* verfügt. Das heißt, dass Sie über die `↑`- und `↓`-Tasten alte Eingaben wieder hervorholen können und nicht erneut eingeben müssen. Auch das Verändern der hervorgeholten Eingaben ist möglich.

Wir beginnen mit der Einführung einiger grundlegender *Datentypen*. Dabei beschränken wir uns zunächst auf ganze Zahlen, Gleitkommazahlen, Zeichenketten, Listen und Dictionarys. Es gibt dabei bestimmte Regeln, nach denen man Instanzen dieser Datentypen, beispielsweise einen Zahlenwert oder eine Zeichenkette, zu schreiben hat, damit diese vom Interpreter erkannt werden. Eine solche Schreibweise nennt man *Literal*.

### 3.1 Ganze Zahlen

Als erstes und einfachstes Beispiel erzeugen wir im interaktiven Modus eine ganze Zahl. Der Interpreter antwortet darauf, indem er ihren Wert ausgibt:

```
>>> -9
-9
>>> 1139
1139
>>> +12
12
```

Das Literal für eine ganze Zahl besteht dabei aus den Ziffern 0 bis 9. Zudem kann ein positives oder negatives Vorzeichen vorangestellt werden. Eine Zahl ohne Vorzeichen wird stets als positiv angenommen.

Es ist möglich, mehrere ganze Zahlen durch *Operatoren* wie +, -, \* oder / zu einem *Term* zu verbinden. In diesem Fall antwortet der Interpreter mit dem Wert des Terms:

```
>>> 5 + 9
14
>>> 5 * -9
-45
```

Wie Sie sehen, lässt sich Python ganz intuitiv als eine Art Taschenrechner verwenden. Das nächste Beispiel ist etwas komplexer und umfasst gleich mehrere miteinander verknüpfte Rechenoperationen:

```
>>> (21 - 3) * 9 + 6
168
>>> 21 - 3 * 9 + 6
0
```

Hier zeigt sich, dass der Interpreter die gewohnten mathematischen Rechengesetze anwendet und das erwartete Ergebnis ausgibt. Das Divisionsergebnis zweier ganzer Zahlen ist nicht zwingend wieder eine ganze Zahl, weswegen der Divisionsoperator das Ergebnis stets als *Gleitkommazahl* zurückgibt:

```
>>> 3/2
1.5
>>> 2/3
0.6666666666666666
>>> 4/4
1.0
```

#### Hinweis

Diese Eigenschaft unterscheidet Python von vielen anderen Programmiersprachen. Dort wird bei der Division zweier ganzer Zahlen eine ganzzahlige Division durchgeführt und das Ergebnis ebenfalls als ganze Zahl zurückgegeben. Diese sogenannte *Integer-Division*<sup>1</sup> wurde in Python-Versionen vor 3.0 auch angewendet.



### 3.2 Gleitkommazahlen

Das Literal für eine Gleitkommazahl besteht aus einem Vorkommaanteil, einem Dezimalpunkt und einem Nachkommaanteil. Wie schon bei den ganzen Zahlen ist es möglich, ein Vorzeichen anzugeben:

```
>>> 0.5
0.5
>>> -123.456
-123.456
>>> +1.337
1.337
```

Beachten Sie, dass es sich bei dem Dezimaltrennzeichen um einen Punkt handeln muss. Die in Deutschland übliche Schreibweise mit einem Komma ist nicht zulässig. Gleitkommazahlen lassen sich ebenso intuitiv in Termen verwenden wie die ganzen Zahlen:

```
>>> 1.5 / 2.1
0.7142857142857143
```

So viel zunächst zu ganzen numerischen Datentypen. Im zweiten Teil des Buches werden wir auf diese grundlegenden Datentypen zurückkommen und sie in aller Ausführlichkeit behandeln. Doch nun zu einem weiteren wichtigen Datentyp, den Zeichenketten.

<sup>1</sup> Die Integer-Division kann durchaus ein gewünschtes Verhalten sein, führt aber gerade bei Programmieranfängern häufig zu Verwirrung und wurde deshalb in Python 3.0 abgeschafft. Wenn Sie eine Integer-Division in Python 3 durchführen möchten, müssen Sie den Operator // verwenden:

```
>>> 3//2
1
>>> 2//3
0
```



### 3.3 Zeichenketten

Neben den Zahlen sind *Zeichenketten*, auch *Strings* genannt, von entscheidender Bedeutung. Strings ermöglichen es, Text vom Benutzer einzulesen, zu speichern, zu bearbeiten oder auszugeben.

Um einen String zu erzeugen, wird der zugehörige Text in doppelte Hochkommata geschrieben:

```
>>> "Hallo Welt"
'Hallo Welt'
>>> "abc123"
'abc123'
```

Die einfachen Hochkommata, die der Interpreter verwendet, um den Wert eines Strings auszugeben, sind eine äquivalente Schreibweise zu den von uns verwendeten doppelten Hochkommata, die Sie auch benutzen dürfen:

```
>>> 'Hallo Welt'
'Hallo Welt'
```

Ähnlich wie bei Ganz- und Gleitkommazahlen gibt es auch Operatoren für Strings. So fügt der Operator + beispielsweise zwei Strings zusammen:

```
>>> "Hallo" + " " + "Welt"
'Hallo Welt'
```

### 3.4 Listen

Wenden wir uns nun den Listen zu. Eine Liste ist eine geordnete Ansammlung von Elementen beliebigen Datentyps. Um eine Liste zu erzeugen, werden die Literale der Werte, die sie enthalten soll, durch Kommata getrennt in eckige Klammern geschrieben:

```
>>> [1,2,3]
[1, 2, 3]
>>> ["Dies", "ist", "eine", "Liste"]
['Dies', 'ist', 'eine', 'Liste']
```

Die Elemente einer Liste müssen nicht alle den gleichen Typ haben und können insbesondere selbst wieder Listen sein, wie folgendes Beispiel zeigt:

```
>>> ["Python", 1, 2, -7 / 4, [1,2,3]]
['Python', 1, 2, -1.75, [1, 2, 3]]
```

Ähnlich wie Strings lassen sich Listen mit dem Operator + um Elemente erweitern, + bildet die Aneinanderreihung zweier Listen.

```
>>> [1,2,3] + ["Python", "ist", "super"]
[1, 2, 3, 'Python', 'ist', 'super']
```

### 3.5 Dictionarys

Der fünfte und letzte Datentyp, den Sie an dieser Stelle kennenlernen, ist das Dictionary (dt. Wörterbuch). Ein Dictionary speichert Zuordnungen von Schlüsseln zu Werten. Um ein Dictionary zu erzeugen, werden die Schlüssel/Wert-Paare durch Kommata getrennt in geschweifte Klammern geschrieben. Zwischen einem Schlüssel und dem zugehörigen Wert steht dabei ein Doppelpunkt:

```
>>> d = {"schlüssel1" : "wert1", "schlüssel2" : "wert2"}
```

Über einen Schlüssel können Sie auf den dahinterliegenden Wert zugreifen. Dazu werden ähnlich wie bei den Listen eckige Klammern verwendet:

```
>>> d["schlüssel1"]
'wert1'
>>> d["schlüssel2"]
'wert2'
```

Über diese Zugriffsoperation können Sie auch Werte modifizieren bzw. neue Schlüssel/Wert-Paare in das Dictionary eintragen:

```
>>> d["schlüssel2"] = "wert2.1"
>>> d["schlüssel2"]
'wert2.1'
>>> d["schlüssel3"] = "wert3"
>>> d["schlüssel3"]
'wert3'
```

Sowohl Schlüssel, als auch Werte können andere Datentypen haben, als die an dieser Stelle verwendeten Strings. Darauf werden wir zu gegebener Zeit zurückkommen.

Auf der in den vorangegangenen Abschnitten vorgestellten grundlegenden Darstellung der fünf Datentypen Ganzzahl, Gleitkommazahl, String, Liste und Dictionary werden wir in den folgenden Abschnitten aufbauen, bis wir im zweiten Teil des Buches ausführlich auf alle in Python eingebauten Datentypen eingehen werden.

3.6 Variablen

Es ist in Python möglich, einer Zahl oder Zeichenkette einen Namen zu geben. Dazu werden der Name auf der linken und das entsprechende Literal auf der rechten Seite eines Gleichheitszeichens geschrieben. Eine solche Operation wird *Zuweisung* genannt.

```
>>> name = 0.5
>>> var123 = 12
>>> string = "Hallo Welt!"
>>> liste = [1,2,3]
```

Die mit den Namen verknüpften Werte können später ausgegeben oder in Berechnungen verwendet werden, indem der Name anstelle des jeweiligen Wertes eingegeben wird:

```
>>> name
0.5
>>> 2 * name
1.0
>>> (var123 + var123) / 3
8
>>> var123 + name
12.5
```

Es ist genauso möglich, dem Ergebnis einer Berechnung einen Namen zu geben:

```
>>> a = 1 + 2
>>> b = var123 / 4
```

Dabei wird immer zuerst die Seite rechts vom Gleichheitszeichen ausgewertet. So wird beispielsweise bei der Anweisung `a = 1 + 2` stets zuerst das Ergebnis von `1 + 2` bestimmt, bevor dem entstandenen Wert ein Name zugewiesen wird.



Hinweis

Beachten Sie bei der Verwendung von Variablen, dass Python *case sensitive* ist. Dies bedeutet, dass bei Bezeichnern zwischen Groß- und Kleinschreibung unterschieden wird. In der Praxis heißt das, dass die Bezeichner `otto` und `Otto` nicht identisch sind, sondern zwei verschiedene Werte haben können.

Ein Variablenname, auch *Bezeichner* genannt, darf seit Python-Version 3.0 aus nahezu beliebigen Buchstaben und dem Unterstrich bestehen. Nach mindestens einem führenden Buchstaben oder Unterstrich dürfen auch Ziffern verwendet wer-

den.<sup>2</sup> Beachten Sie, dass auch Umlaute und spezielle Buchstaben anderer Sprachen erlaubt sind, wie folgendes Beispiel zeigt:

```
>>> äöüßèè = 123
>>> äöüßèè
123
```

Solche Freiheiten, was Bezeichner angeht, finden sich in anderen Programmiersprachen selten. Nicht zuletzt deshalb empfehlen wir Ihnen, sich auf das englische Alphabet zu beschränken. Die fehlenden Umlaute und das `ß` fallen auch bei deutschen Bezeichnern kaum ins Gewicht und wirken im Quellcode eher verwirrend als natürlich.

Bestimmte *Schlüsselwörter*<sup>3</sup> sind in Python für die Sprache selbst reserviert und dürfen nicht als Bezeichner verwendet werden. Die folgende Tabelle gibt Ihnen eine Übersicht über alle in Python reservierten Wörter.

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

Tabelle 3.1 Schlüsselwörter in Python

Zum Schluss möchten wir noch einen weiteren Begriff einführen: Alles, was mit numerischen Literalen – also Ganz- oder Gleitkommazahlen, Variablen und Operatoren – formuliert werden kann, wird als *arithmetischer Ausdruck* bezeichnet. Ein solcher Ausdruck kann also so aussehen:

```
(a * a + b) / 12
```

Alle bisher eingeführten Operatoren `+`, `-`, `*` und `/` werden daher als *arithmetische Operatoren* bezeichnet.

2 Häufig werden Variablen, die nur eine lokale und kurzfristige Bedeutung haben, mit einem kurzen, oft einbuchstabigen Namen versehen. Dabei sollten Sie beachten, dass die Buchstaben »o«, »O«, »l« und »I« in manchen Schriftarten wie Zahlen aussehen und damit für einen Variablennamen ungeeignet sind.

3 Unter einem Schlüsselwort versteht man in der Programmierung ein Wort, das eine bestimmte Bedeutung trägt, beispielsweise den Programmablauf steuert. In Python existieren zum Beispiel die Schlüsselwörter `if` und `for`, die eine Fallunterscheidung bzw. eine Schleife einleiten.



3.7 Logische Ausdrücke

Neben den arithmetischen Operatoren gibt es einen zweiten Satz von Operatoren, die das Vergleichen von Zahlen ermöglichen:

```
>>> 3 < 4
True
```

Hier wird getestet, ob 3 kleiner ist als 4. Auf solche Vergleiche antwortet der Interpreter mit einem *Wahrheitswert*, also mit `True` (dt. »wahr«) oder `False` (dt. »falsch«). Ein Vergleich wird mithilfe eines sogenannten *Vergleichsoperators*, in diesem Fall `<`, durchgeführt.

Die folgende Tabelle führt die Vergleichsoperatoren auf:

Vergleich	Bedeutung
3 == 4	Ist 3 gleich 4? Beachten Sie das doppelte Gleichheitszeichen, das den Vergleich von einer Zuweisung unterscheidet.
3 != 4	Ist 3 ungleich 4?
3 < 4	Ist 3 kleiner als 4?
3 > 4	Ist 3 größer als 4?
3 <= 4	Ist 3 kleiner oder gleich 4?
3 >= 4	Ist 3 größer oder gleich 4?

Tabelle 3.2 Vergleiche in Python

Allgemein kann für 3 und 4 ein beliebiger arithmetischer Ausdruck eingesetzt werden. Wenn zwei arithmetische Ausdrücke durch einen der oben genannten Operatoren miteinander verglichen werden, erzeugt man einen *logischen Ausdruck*:

```
(a - 7) < (b * b + 6.5)
```

Neben den bereits eingeführten arithmetischen Operatoren gibt es drei logische Operatoren, mit denen Sie das Ergebnis eines logischen Ausdrucks verändern oder zwei logische Ausdrücke miteinander verknüpfen können.

Der Operator `not` kehrt das Ergebnis eines Vergleichs um, macht also aus `True` `False` und aus `False` `True`. Der Ausdruck `not (3 < 4)` ist also das Gleiche wie `3 >= 4`:

```
>>> not (3 < 4)
False
>>> not (4 < 3)
True
```

Der Operator `and` bekommt zwei logische Ausdrücke als Operanden und ergibt nur dann `True`, wenn sowohl der erste Ausdruck als auch der zweite `True` ergeben haben. Er entspricht damit der umgangssprachlichen »Und«-Verknüpfung zweier Satzteile. Im Beispiel kann dies so aussehen:

```
>>> (3 < 4) and (5 < 6)
True
>>> (3 < 4) and (4 < 3)
False
```

Der Operator `or` entspricht dem umgangssprachlichen »oder«. Er bekommt zwei logische Ausdrücke als Operanden und ergibt nur dann `False`, wenn sowohl der erste Ausdruck als auch der zweite `False` ergeben haben. Der Operator ergibt also `True`, wenn mindestens einer seiner Operanden `True` ergeben hat:

```
>>> (3 < 4) or (5 < 6)
True
>>> (3 < 4) or (4 < 3)
True
>>> (5 > 6) or (4 < 3)
False
```

Wir haben der Einfachheit halber hier nur Zahlen miteinander verglichen. Selbstverständlich ergibt ein solcher Vergleich nur dann einen Sinn, wenn komplexere arithmetische Ausdrücke miteinander verglichen werden. Durch die vergleichenden Operatoren und die drei *booleschen Operatoren* `not`, `and` und `or` können schon sehr komplexe Vergleiche erstellt werden.

Hinweis

Beachten Sie, dass bei allen Beispielen aus Gründen der Übersicht Klammern gesetzt wurden. Durch Prioritätsregelungen der Operatoren untereinander sind diese überflüssig. Das bedeutet, dass jedes hier vorgestellte Beispiel auch ohne Klammern wie erwartet funktionieren würde. Trotzdem ist es gerade am Anfang sinnvoll, durch Klammerung die Zugehörigkeiten visuell eindeutig zu gestalten. Eine Tabelle mit den Prioritätsregeln für Operatoren, der sogenannten *Operatorrangfolge*, finden Sie in Kapitel 11, »Operatoren«.



3.8 Funktionen und Methoden

In diesem Abschnitt vermitteln wir Ihnen ein grundlegendes Wissen über Funktionen und einige Konzepte der objektorientierten Programmierung. Dabei beschrän-

ken wir uns auf die Aspekte, die in den folgenden Kapiteln benötigt werden. Beide Themen werden in Kapitel 19 bzw. Kapitel 21 noch einmal ausführlich behandelt.

### 3.8.1 Funktionen

In Python können Teile eines Programms in *Funktionen* gekapselt und danach über einen *Funktionsaufruf* ausgeführt werden. Das Ziel dieses Vorgehens ist es, Redundanz im Quellcode zu vermeiden. Funktionalität, die häufig benötigt wird, sollte stets nur einmal als Funktion implementiert und dann als solche im weiteren Programm verwendet werden. Außerdem kann der Einsatz von Funktionen die Les- und Wartbarkeit des Quellcodes deutlich erhöhen.

Python bietet einen Satz eingebauter Funktionen (*Built-in Functions*), die der Programmierer zu jeder Zeit verwenden kann. Als Beispiel dient in diesem Abschnitt die eingebaute Funktion `max`, die das größte Element einer Liste bestimmt:

```
>>> max([1,5,2,7,9,3])
9
```

Eine Funktion wird aufgerufen, indem man den Funktionsnamen, gefolgt von den *Funktionsparametern* in Klammern, schreibt. Im Beispiel erwartet die Funktion `max` genau einen Parameter, nämlich eine Liste der zu betrachtenden Werte. Das Ergebnis der Berechnung wird als *Rückgabewert* der Funktion zurückgegeben. Sie können sich vorstellen, dass der Funktionsaufruf im Quelltext durch den Rückgabewert ersetzt wird.

Es gibt eine Variante der Funktion `max`, die anstelle des größten Elements einer Liste den größten ihr übergebenen Parameter bestimmt. Um einer Funktion mehrere Parameter zu übergeben, werden diese beim Funktionsaufruf durch Kommata getrennt in die Klammern geschrieben:

```
>>> max(1,5,3)
5
```

Selbstverständlich können Sie in Python eigene Funktionen definieren, an dieser Stelle genügt es jedoch, zu wissen, wie bereits vorhandene Funktionen verwendet werden können. In Kapitel 19 kommen wir noch einmal ausführlich auf Funktionen zu sprechen.

### 3.8.2 Methoden

Das Erzeugen eines Wertes eines bestimmten *Datentyps*, etwa das Erzeugen einer ganzen Zahl über ihr Literal, wird *Instanzieren* genannt und der entstandene Wert

*Instanz*. So ist beispielsweise 1 eine Instanz des Datentyps »ganze Zahl« oder `[4,5,6]` eine Instanz des Datentyps »Liste«. Der Datentyp einer Instanz legt einerseits fest, welche Daten gespeichert werden, und definiert andererseits einen Satz von Operationen, die auf diesen Daten durchgeführt werden können. Ein Teil dieser Operationen wird durch Operatoren abgebildet, so bietet beispielsweise der Datentyp »Gleitkommazahl« den Operator `+` zum Addieren zweier Gleitkommazahlen an. Für die einfachen numerischen Datentypen sind einige wenige Operatoren ausreichend, um mit ihnen arbeiten zu können. Bei komplexeren Datentypen, beispielsweise den Listen, ist eine ganze Reihe von Operationen denkbar, die allein über Operatoren nicht abgebildet werden können. Für solche Fälle können Datentypen *Methoden* definieren. Dabei handelt es sich um Funktionen, die im Kontext einer bestimmten Instanz ausgeführt werden.

Der Datentyp »Liste« bietet zum Beispiel eine Methode `sort` an, mit deren Hilfe eine Liste sortiert werden kann. Um eine Methode aufzurufen, wird eine Instanz (oder eine Referenz darauf), gefolgt von einem Punkt und dem Methodenaufruf, geschrieben. Dieser ist wie ein Funktionsaufruf aufgebaut:

```
>>> liste = [2,7,3,2,7,8,4,2,5]
>>> liste.sort()
>>> liste
[2, 2, 2, 3, 4, 5, 7, 7, 8]
```

Ein weiteres Beispiel bietet die Methode `count` des Datentyps »String«, die zählt, wie oft ein Zeichen in einem String vorkommt.

```
>>> "Hallo Welt".count("l")
3
```

Das hier erworbene Wissen über Funktionen und Methoden wird zu gegebener Zeit vertieft. Im Folgenden wird die für den Anfang wohl wichtigste in Python eingebaute Funktion besprochen: `print`.

## 3.9 Bildschirmausgaben

Auch wenn wir hin und wieder auf den interaktiven Modus zurückgreifen werden, ist es unser Ziel, möglichst schnell echte Python-Programme zu schreiben. Es ist eine Besonderheit des interaktiven Modus, dass der Wert eines eingegebenen Ausdrucks automatisch ausgegeben wird. In einem normalen Programm müssen Bildschirmausgaben dagegen vom Programmierer erzeugt werden. Um den Wert einer Variablen auszugeben, wird in Python die Funktion `print` verwendet:

```
>>> print(1.2)
1.2
```

Beachten Sie, dass mittels `print`, im Gegensatz zur automatischen Ausgabe des interaktiven Modus, nur der Wert an sich ausgegeben wird. So wird bei der automatischen Ausgabe der Wert eines Strings in Hochkommata geschrieben, während dies bei `print` nicht der Fall ist:

```
>>> "Hallo Welt"
'Hallo Welt'
>>> print("Hallo Welt")
Hallo Welt
```

Auch hier ist es problemlos möglich, anstelle eines konstanten Wertes einen Variablennamen zu verwenden:

```
>>> var = 9
>>> print(var)
9
```

Oder Sie geben das Ergebnis eines Ausdrucks direkt aus:

```
>>> print(-3 * 4)
-12
```

Außerdem ermöglicht `print` es, mehrere Variablen oder Konstanten in einer Zeile auszugeben. Dazu werden die Werte durch Kommata getrennt angegeben. Jedes Komma wird bei der Ausgabe durch ein Leerzeichen ersetzt:

```
>>> print(-3, 12, "Python rockt")
-3 12 Python rockt
```

Das ist insbesondere dann hilfreich, wenn Sie nicht nur einzelne Werte, sondern auch einen kurzen erklärenden Text dazu ausgeben möchten. So etwas können Sie auf die folgende Weise erreichen:

```
>>> var = 9
>>> print("Die magische Zahl ist:", var)
Die magische Zahl ist: 9
```

Abschließend ist noch zu sagen, dass `print` nach jeder Ausgabe einen Zeilenvorschub ausgibt. Es wird also stets in eine neue Zeile geschrieben.

### Hinweis

Die Funktion `print` wurde in Python 3.0 eingeführt. In früheren Python-Versionen konnten Bildschirmausgaben über das Schlüsselwort `print` erzeugt werden:

```
>>> print "Dies", "ist", "Python", 2
Dies ist Python 2
```

In den meisten Fällen sind die fehlenden Klammern der einzige Unterschied. Näheres zu den Unterschieden zwischen Python 2 und 3 erfahren Sie in Kapitel 43.



# Kapitel 4

## Der Weg zum ersten Programm

Nachdem wir im interaktiven Modus spielerisch einige Grundelemente der Sprache Python behandelt haben, möchten wir dieses Wissen jetzt auf ein tatsächliches Programm übertragen. Im Gegensatz zum interaktiven Modus, der eine wechselseitige Interaktion zwischen Programmierer und Interpreter ermöglicht, wird der Quellcode eines Programms in eine Datei geschrieben. Diese Datei wird als Ganzes vom Interpreter eingelesen und ausgeführt.

In den folgenden Abschnitten lernen Sie die Grundstrukturen eines Python-Programms kennen und werden Ihr erstes einfaches Beispielprogramm schreiben.

### 4.1 Tippen, kompilieren, testen

In diesem Abschnitt werden die Arbeitsabläufe besprochen, die nötig sind, um ein Python-Programm zu erstellen und auszuführen. Ganz allgemein sollten Sie sich darauf einstellen, dass wir über einen Großteil des Buches ausschließlich *Konsolenanwendungen* schreiben werden. Eine Konsolenanwendung hat eine rein textbasierte Schnittstelle zum Benutzer und läuft in der *Konsole* (auch *Shell*) des jeweiligen Betriebssystems ab. Für die meisten Beispiele und auch für viele reale Anwendungsfälle reicht eine solche textbasierte Schnittstelle aus.<sup>1</sup>


Grundsätzlich besteht ein Python-Programm aus einer oder mehreren Programmdateien. Diese Programmdateien haben die Dateiendung *.py* und enthalten den Python-Quelltext. Dabei handelt es sich um nichts anderes als um Textdateien. Programmdateien können also mit einem normalen Texteditor bearbeitet werden.

Nachdem eine Programmdatei geschrieben wurde, besteht der nächste Schritt darin, sie auszuführen. Wenn Sie IDLE verwenden, kann die Programmdatei bequem über den Menüpunkt **RUN • RUN MODULE** ausgeführt werden. Sollten Sie einen Editor verwenden, der keine vergleichbare Funktion unterstützt, müssen Sie in einer Kommandozeile in das Verzeichnis der Programmdatei wechseln und, abhängig von Ihrem Betriebssystem, verschiedene Kommandos ausführen.

---

<sup>1</sup> Selbstverständlich ermöglicht Python auch die Programmierung grafischer Benutzeroberflächen. Dies wird in Kapitel 39 behandelt.

## Windows

Unter Windows reicht es, den Namen der Programmdatei einzugeben und mit  zu bestätigen. Im folgenden Beispiel, zu sehen in Abbildung 4.1, wird die Programmdatei *programm.py* im Ordner *C:\Ordner* ausgeführt. Dazu müssen Sie ein Konsolenfenster unter **START • PROGRAMME • ZUBEHÖR • EINGABEAUFFORDERUNG** starten.

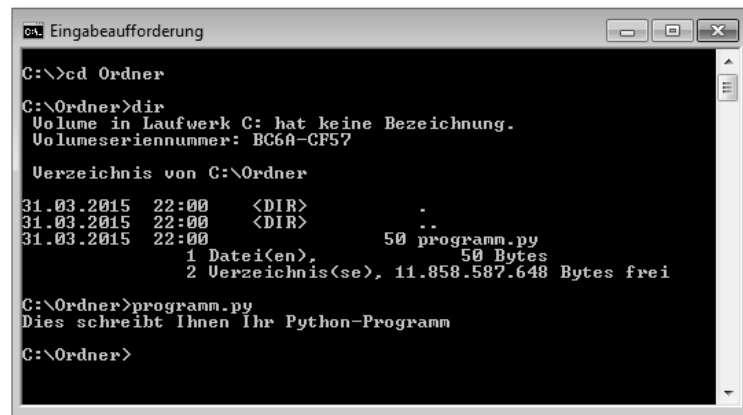


Abbildung 4.1 Ausführen eines Python-Programms unter Windows

Bei »Dies schreibt Ihnen Ihr Python-Programm« handelt es sich um eine Ausgabe des Python-Programms in der Datei *programm.py*, die beweist, dass das Python-Programm tatsächlich ausgeführt wurde.



### Hinweis

Unter Windows ist es auch möglich, ein Python-Programm durch einen Doppelklick auf die jeweilige Programmdatei auszuführen. Das hat aber den Nachteil, dass sich das Konsolenfenster sofort nach Beenden des Programms schließt und die Ausgaben des Programms somit nicht erkennbar sind.

## Linux und OS X

Unter Unix-ähnlichen Betriebssystemen wie Linux oder OS X wechseln Sie ebenfalls in das Verzeichnis, in dem die Programmdatei liegt, und starten dann den Python-Interpreter mit dem Kommando `python`, gefolgt von dem Namen der auszuführenden Programmdatei. Im folgenden Beispiel wird die Programmdatei *programm.py* unter Linux ausgeführt, die sich im Verzeichnis */home/user/ordner* befindet:

```
user@HOST ~ $ cd ordner
user@HOST ~/ordner $ python programm.py
Dies schreibt Ihnen Ihr Python-Programm
```

Bitte beachten Sie den Hinweis aus Abschnitt 4.1.2, der besagt, dass das Kommando, mit dem Sie Python 3.4 starten, je nach Distribution von dem hier demonstrierten `python` abweichen kann.

### 4.1.1 Shebang

Unter einem Unix-ähnlichen Betriebssystem wie beispielsweise Linux können Python-Programmdateien mithilfe eines *Shebangs*, auch *Magic Line* genannt, direkt ausführbar gemacht werden. Dazu muss die erste Zeile der Programmdatei in der Regel folgendermaßen lauten:

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

In diesem Fall wird das Betriebssystem dazu angehalten, diese Programmdatei immer mit dem Python-Interpreter auszuführen. Unter anderen Betriebssystemen, beispielsweise Windows, wird die Shebang-Zeile ignoriert.

Beachten Sie, dass der Python-Interpreter auf Ihrem System in einem anderen Verzeichnis als dem hier angegebenen installiert sein könnte. Allgemein ist daher folgende Shebang-Zeile besser, da sie vom tatsächlichen Installationsort Pythons unabhängig ist:

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

Beachten Sie außerdem, dass das Executable-Flag der Programmdatei gesetzt werden muss, bevor die Datei tatsächlich ausführbar ist. Das geschieht mit dem Befehl

```
chmod +x dateiname
```

Die in diesem Buch gezeigten Beispiele enthalten aus Gründen der Übersichtlichkeit keine Shebang-Zeile. Das bedeutet aber ausdrücklich nicht, dass vom Einsatz einer Shebang-Zeile abzuraten wäre.

### 4.1.2 Interne Abläufe

Bislang haben Sie eine ungefähre Vorstellung davon, was Python ausmacht und wo die Stärken dieser Programmiersprache liegen. Außerdem haben wir Ihnen das Grundwissen zum Erstellen und Ausführen einer Python-Programmdatei vermittelt. Doch in den vorangegangenen Abschnitten sind Begriffe wie »Compiler« oder »Interpreter« gefallen, ohne erklärt worden zu sein. In diesem Abschnitt möchten wir uns daher den internen Vorgängen widmen, die beim Ausführen einer Python-Programmdatei ablaufen. Abbildung 4.2 veranschaulicht, was beim Ausführen einer Programmdatei namens *programm.py* geschieht.

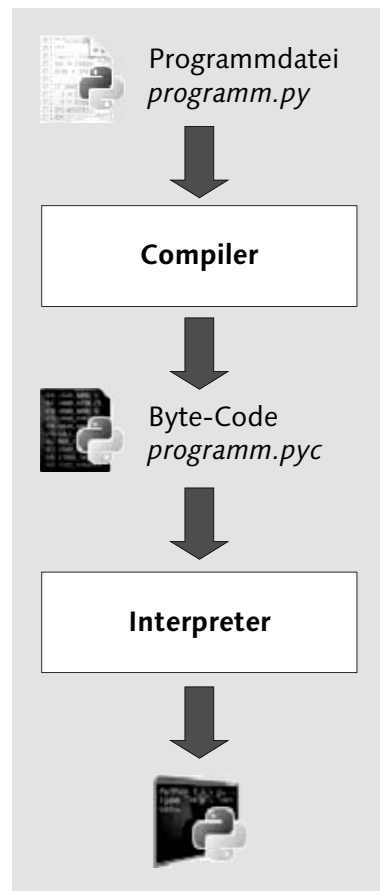


Abbildung 4.2 Kompilieren und Interpretieren einer Programmdatei

Wenn die Programmdatei *programm.py*, wie zu Beginn des Kapitels beschrieben, ausgeführt wird, passiert sie zunächst den *Compiler*. Als Compiler wird ein Programm bezeichnet, das von einer formalen Sprache in eine andere übersetzt. Im Falle von Python übersetzt der Compiler von der Sprache Python in den *Byte-Code*. Dabei steht es dem Compiler frei, den generierten Byte-Code im Arbeitsspeicher zu behalten oder als *programm.pyc* auf der Festplatte zu speichern.

Beachten Sie, dass der vom Compiler generierte Byte-Code, im Gegensatz etwa zu C- oder C++-Kompilaten, nicht direkt auf dem Prozessor ausgeführt werden kann. Zur Ausführung des Byte-Codes wird eine weitere Abstraktionsschicht, der *Interpreter*, benötigt. Der Interpreter, häufig auch *virtuelle Maschine* (engl. *virtual machine*) genannt, liest den vom Compiler erzeugten Byte-Code ein und führt ihn aus.

Dieses Prinzip einer interpretierten Programmiersprache hat verschiedene Vorteile. So kann derselbe Python-Code beispielsweise unmodifiziert auf allen Plattformen

ausgeführt werden, für die ein Python-Interpreter existiert. Allerdings laufen Programme interpretierter Programmiersprachen aufgrund des zwischengeschalteten Interpreters in der Regel auch langsamer als ein vergleichbares C-Programm, das direkt auf dem Prozessor ausgeführt wird.<sup>2</sup>

## 4.2 Grundstruktur eines Python-Programms

Um Ihnen ein Gefühl für die Sprache Python zu vermitteln, möchten wir Ihnen zunächst einen Überblick über ihre Syntax geben. Das Wort *Syntax* kommt aus dem Griechischen und bedeutet »Satzbau«. Unter der Syntax einer Programmiersprache ist die vollständige Beschreibung erlaubter und verbotener Konstruktionen zu verstehen. Die Syntax wird durch eine Grammatik festgelegt, an die sich der Programmierer zu halten hat. Tut er es nicht, so verursacht er den allseits bekannten *Syntax Error*.

Python macht dem Programmierer sehr genaue Vorgaben, wie er seinen Quellcode zu strukturieren hat. Obwohl erfahrene Programmierer darin eine Einschränkung sehen mögen, kommt diese Eigenschaft gerade Programmierneulingen zugute, denn unstrukturierter und unübersichtlicher Code ist eine der größten Fehlerquellen in der Programmierung.

Grundsätzlich besteht ein Python-Programm aus einzelnen *Anweisungen*, die im einfachsten Fall genau eine Zeile im Quelltext einnehmen. Folgende Anweisung gibt beispielsweise einen Text auf dem Bildschirm aus:

```
print("Hallo Welt")
```

Einige Anweisungen lassen sich in einen *Anweisungskopf* und einen *Anweisungskörper* unterteilen, wobei der Körper weitere Anweisungen enthalten kann:

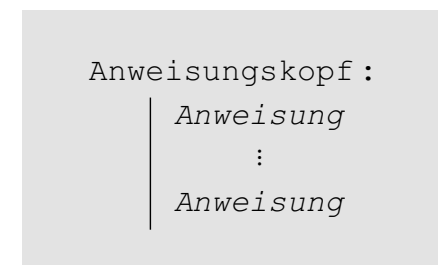


Abbildung 4.3 Struktur einer mehrzeiligen Anweisung

<sup>2</sup> Zumindest dann, wenn der Interpreter keine Laufzeitoptimierung über eine Just-in-Time-Kompilierung durchführt. Mit PyPy lernen Sie einen solchen Python Interpreter in Abschnitt 35.6.9, »Alternative Interpreter: PyPy«, kennen.



Das kann in einem konkreten Python-Programm etwa so aussehen:

```
if x > 10:
    print("x ist größer als 10")
    print("Zweite Zeile!")
```

Die Zugehörigkeit des Körpers zum Kopf wird in Python durch einen Doppelpunkt am Ende des Anweisungskopfes und durch eine tiefere Einrückung des Anweisungskörpers festgelegt. Die Einrückung kann sowohl über Tabulatoren als auch über Leerzeichen erfolgen, wobei Sie gut beraten sind, beides nicht zu vermischen. Wir empfehlen eine Einrückungstiefe von jeweils vier Leerzeichen.

Python unterscheidet sich hier von vielen gängigen Programmiersprachen, in denen die Zuordnung von Anweisungskopf und Anweisungskörper durch geschweifte Klammern oder Schlüsselwörter wie »Begin« und »End« erreicht wird.



#### Hinweis

Ein Programm, in dem sowohl Leerzeichen als auch Tabulatoren verwendet wurden, kann vom Python-Compiler anstandslos übersetzt werden, da jeder Tabulator intern durch acht Leerzeichen ersetzt wird. Dies kann aber zu schwer auffindbaren Fehlern führen, denn viele Editoren verwenden standardmäßig eine Tabulatorweite von vier Leerzeichen. Dadurch scheinen bestimmte Quellcodeabschnitte gleich weit eingerückt, obwohl sie es de facto nicht sind. Bitte stellen Sie Ihren Editor so ein, dass jeder Tabulator automatisch durch Leerzeichen ersetzt wird, oder verwenden Sie ausschließlich Leerzeichen zur Einrückung Ihres Codes.

Möglicherweise fragen Sie sich jetzt, wie Anweisungen, die über mehrere Zeilen gehen, mit dem interaktiven Modus vereinbar sind, in dem ja immer nur eine Zeile bearbeitet werden kann. Nun, generell werden wir, wenn ein Code-Beispiel mehrere Zeilen lang ist, versuchen, den interaktiven Modus zu vermeiden. Dennoch ist die Frage berechtigt. Die Antwort: Es wird ganz intuitiv zeilenweise eingegeben. Wenn der Interpreter erkennt, dass eine Anweisung noch nicht vollendet ist, ändert er den Eingabeprompt von `>>>` in `...`. Geben wir einmal unser oben dargestelltes Beispiel in den interaktiven Modus ein:

```
>>> x = 123
>>> if x > 10:
...     print("Der Interpreter leistet gute Arbeit")
...     print("Zweite Zeile!")
...
Der Interpreter leistet gute Arbeit
Zweite Zeile!
>>>
```

Beachten Sie, dass Sie, auch wenn eine Zeile mit `...` beginnt, die aktuelle Einrückungstiefe berücksichtigen müssen. Darüber hinaus kann der Interpreter das Ende des Anweisungskörpers nicht automatisch erkennen, da dieser beliebig viele Anweisungen enthalten kann. Deswegen muss ein Anweisungskörper im interaktiven Modus durch Drücken der -Taste beendet werden.

#### 4.2.1 Umbrechen langer Zeilen

Prinzipiell können Quellcodezeilen beliebig lang werden. Viele Programmierer beschränken die Länge ihrer Quellcodezeilen jedoch, damit beispielsweise mehrere Quellcodezeilen nebeneinander auf den Bildschirm passen oder der Code auch auf Geräten mit einer festen Zeilenbreite angenehm zu lesen ist. Eine geläufige maximale Zeilenlänge ist 80 Zeichen. Innerhalb von Klammern dürfen Sie Quellcode zwar beliebig umbrechen, doch an vielen anderen Stellen sind Sie an die strengen syntaktischen Regeln von Python gebunden. Durch Einsatz der Backslash-Notation ist es möglich, Quellcode an nahezu beliebigen Stellen in eine neue Zeile umzubrechen:

```
>>> var \
... = \
... 10
>>> var
10
```

Grundsätzlich kann ein Backslash überall da stehen, wo auch ein Leerzeichen hätte stehen können. Daher ist auch ein Backslash innerhalb eines Strings möglich:

```
>>> "Hallo \
... Welt"
'Hallo Welt'
```

Beachten Sie dabei aber, dass eine Einrückung des umbrochenen Teils des Strings Leerzeichen in den String schreibt. Aus diesem Grund sollten Sie folgende Variante, einen String in mehrere Zeilen zu schreiben, vorziehen:

```
>>> "Hallo " \
... "Welt"
'Hallo Welt'
```

#### 4.2.2 Zusammenfügen mehrerer Zeilen

Genauso, wie Sie eine einzeilige Anweisung mithilfe des Backslashes auf mehrere Zeilen umbrechen, können Sie mehrere einzeilige Anweisungen in einer Zeile zusammenfassen. Dazu werden die Anweisungen durch ein Semikolon voneinander getrennt:

```
>>> print("Hallo"); print("Welt")
Hallo
Welt
```

Anweisungen, die aus einem Anweisungskopf und einem Anweisungskörper bestehen, können auch ohne Einsatz eines Semikolons in eine Zeile gefasst werden, sofern der Anweisungskörper selbst aus nicht mehr als einer Zeile besteht:

```
>>> x = True
>>> if x: print("Hallo Welt")
...
Hallo Welt
```

Sollte der Anweisungskörper mehrere Zeilen lang sein, können diese durch ein Semikolon zusammengefasst werden:

```
>>> x = True
>>> if x: print("Hallo"); print("Welt")
...
Hallo
Welt
```

Alle durch ein Semikolon zusammengeführten Anweisungen werden so behandelt, als wären sie gleich weit eingerückt. Allein ein Doppelpunkt vergrößert die Einrückungstiefe. Aus diesem Grund gibt es im oben genannten Beispiel keine Möglichkeit, in derselben Zeile eine Anweisung zu schreiben, die nicht mehr im Körper der `if`-Anweisung steht.



#### Hinweis

Beim Einsatz des Backslashes und vor allem des Semikolons entsteht schnell unleserlicher Code. Verwenden Sie beide Notationen daher nur, wenn Sie meinen, dass es der Lesbarkeit und Übersichtlichkeit dienlich ist.

### 4.3 Das erste Programm

Als Einstieg in die Programmierung mit Python erstellen wir hier ein kleines Beispielprogramm, das Spiel Zahlenraten. Die Spielidee ist folgende: Der Spieler soll eine im Programm festgelegte Zahl erraten. Dazu stehen ihm beliebig viele Versuche zur Verfügung. Nach jedem Versuch informiert ihn das Programm darüber, ob die geratene Zahl zu groß, zu klein oder genau richtig gewesen ist. Sobald der Spieler die Zahl erraten hat, gibt das Programm die Anzahl der Versuche aus und wird beendet.

Aus Sicht des Spielers soll das Ganze folgendermaßen aussehen:

```
Raten Sie: 42
Zu klein
Raten Sie: 10000
Zu groß
Raten Sie: 999
Zu klein
Raten Sie: 1337
Super, Sie haben es in 4 Versuchen geschafft!
```

Kommen wir vom Ablaufprotokoll zur konkreten Implementierung in Python:

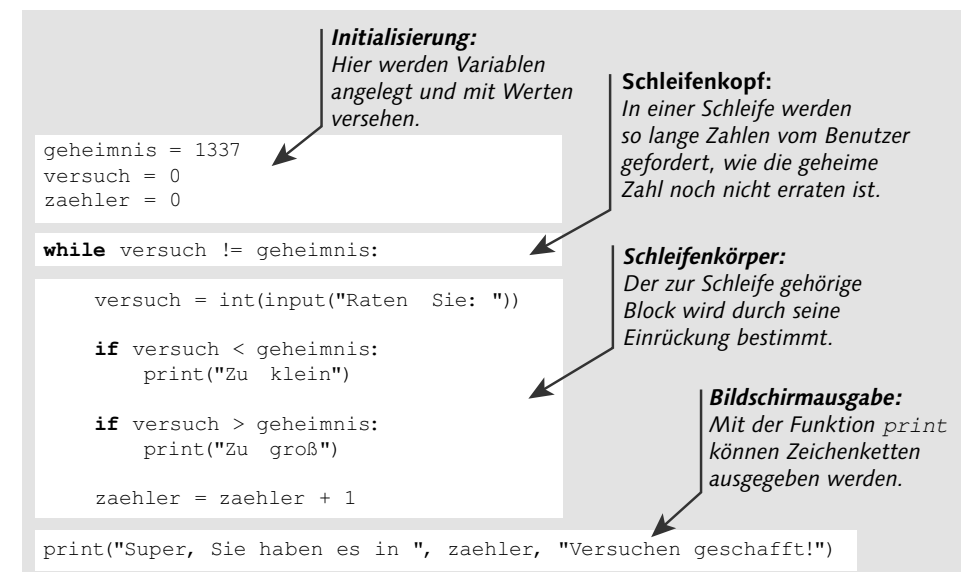


Abbildung 4.4 Zahlenraten, ein einfaches Beispiel

Die hervorgehobenen Bereiche des Programms werden im Folgenden noch einmal ausführlich diskutiert.

#### Initialisierung

Bei der Initialisierung werden die für das Spiel benötigten Variablen angelegt. Python unterscheidet zwischen verschiedenen Datentypen, wie etwa Zeichenketten, Ganz- oder Fließkommazahlen. Der Typ einer Variablen wird zur Laufzeit des Programms anhand des ihr zugewiesenen Wertes bestimmt. Es ist also nicht nötig, einen Datentyp explizit anzugeben. Eine Variable kann im Laufe des Programms ihren Typ ändern.

In unserem Spiel werden Variablen für die gesuchte Zahl (`geheimnis`), die Benutzereingabe (`versuch`) und den Versuchszähler (`zaehler`) angelegt und mit Anfangswerten versehen. Dadurch, dass `versuch` und `geheimnis` zu Beginn des Programms verschiedene Werte haben, ist sichergestellt, dass die Schleife anläuft.

### Schleifenkopf

Eine `while`-Schleife wird eingeleitet. Eine `while`-Schleife läuft so lange, wie die im Schleifenkopf genannte Bedingung (`versuch != geheimnis`) erfüllt ist, also in diesem Fall, bis die Variablen `versuch` und `geheimnis` den gleichen Wert haben. Aus Benutzersicht bedeutet dies: Die Schleife läuft so lange, bis die Benutzereingabe mit der zu erratenden Zahl übereinstimmt.

Den zum Schleifenkopf gehörigen Schleifenkörper erkennt man daran, dass die nachfolgenden Zeilen um eine Stufe weiter eingerückt wurden. Sobald die Eindrückung wieder um einen Schritt nach links geht, endet der Schleifenkörper.

### Schleifenkörper

In der ersten Zeile des Schleifenkörpers wird eine vom Spieler eingegebene Zahl eingelesen und in der Variablen `versuch` gespeichert. Dabei wird mithilfe von `input("Raten Sie: ")` die Eingabe des Benutzers eingelesen und mit `int` in eine ganze Zahl konvertiert (von engl. *integer*, »ganze Zahl«). Diese Konvertierung ist wichtig, da Benutzereingaben generell als String eingelesen werden. In unserem Fall möchten wir die Eingabe jedoch als Zahl weiterverwenden. Der String `"Raten Sie: "` wird vor der Eingabe ausgegeben und dient dazu, den Benutzer zur Eingabe der Zahl aufzufordern.

Nach dem Einlesen wird einzeln geprüft, ob die eingegebene Zahl `versuch` größer oder kleiner als die gesuchte Zahl `geheimnis` ist, und mittels `print` eine entsprechende Meldung ausgegeben. Schließlich wird der Versuchszähler `zaehler` um eins erhöht.

Nach dem Hochzählen des Versuchszählers endet der Schleifenkörper, da die nächste Zeile nicht mehr unter dem Schleifenkopf eingerückt ist.

### BildschirmAusgabe

Die letzte Programmzeile gehört nicht mehr zum Schleifenkörper. Das bedeutet, dass sie erst ausgeführt wird, wenn die Schleife vollständig durchlaufen, das Spiel also gewonnen ist. In diesem Fall werden eine Erfolgsmeldung sowie die Anzahl der benötigten Versuche ausgegeben. Das Spiel ist beendet.

Erstellen Sie jetzt Ihr erstes Python-Programm, indem Sie den Programm-Code in eine Datei namens `spiel.py` schreiben und ausführen. Ändern Sie den Startwert von `geheimnis`, und spielen Sie das Spiel.

## 4.4 Kommentare

Sie können sich sicherlich vorstellen, dass es nicht das Ziel ist, Programme zu schreiben, die auf eine Postkarte passen würden. Mit der Zeit wird der Quelltext Ihrer Programme umfangreicher und komplexer werden. Irgendwann ist der Zeitpunkt erreicht, da bloßes Gedächtnistraining nicht mehr ausreicht, um die Übersicht zu bewahren. Spätestens dann kommen Kommentare ins Spiel.

Ein *Kommentar* ist ein kleiner Text, der eine bestimmte Stelle des Quellcodes erläutert und auf Probleme, offene Aufgaben oder Ähnliches hinweist. Ein Kommentar wird vom Interpreter einfach ignoriert, ändert also am Ablauf des Programms nichts.

Die einfachste Möglichkeit, einen Kommentar zu verfassen, ist der *Zeilenkommentar*. Diese Art des Kommentars wird mit dem `#`-Zeichen begonnen und endet mit dem Ende der Zeile:

```
# Ein Beispiel mit Kommentaren
print("Hallo Welt!") # Simple Hallo-Welt-Ausgabe
```

Für längere Kommentare bietet sich ein *Blockkommentar* an. Ein Blockkommentar beginnt und endet mit drei aufeinanderfolgenden Anführungszeichen:<sup>3</sup>

```
""" Dies ist ein Blockkommentar,
er kann sich über mehrere Zeilen erstrecken. """
```

Kommentare sollten nur gesetzt werden, wenn sie zum Verständnis des Quelltextes beitragen oder wertvolle Informationen enthalten. Jede noch so unwichtige Zeile zu kommentieren führt dazu, dass man den Wald vor lauter Bäumen nicht mehr sieht.

## 4.5 Der Fehlerfall

Vielleicht haben Sie bereits mit dem Beispielprogramm aus Abschnitt 4.3, »Das erste Programm«, gespielt und sind dabei auf eine solche oder ähnliche Ausgabe des Interpreters gestoßen:

```
File "spiel.py", line 8
    if guess < secret
        ^
SyntaxError: invalid syntax
```

<sup>3</sup> Eigentlich wird mit dieser Notation kein Blockkommentar erzeugt, sondern ein mehrzeiliger String, der sich aber auch dazu eignet, größere Quellcodebereiche »auszukommentieren«.

Es handelt sich dabei um eine Fehlermeldung, die in diesem Fall auf einen Syntaxfehler im Programm hinweist. Können Sie erkennen, welcher Fehler hier vorliegt? Richtig, es fehlt der Doppelpunkt am Ende der Zeile.

Python stellt bei der Ausgabe einer Fehlermeldung wichtige Informationen bereit, die bei der Fehlersuche hilfreich sind:

- Die erste Zeile der Fehlermeldung gibt Aufschluss darüber, in welcher Zeile innerhalb welcher Datei der Fehler aufgetreten ist. In diesem Fall handelt es sich um die Zeile 8 in der Datei *spiel.py*.
- Der mittlere Teil zeigt den betroffenen Ausschnitt des Quellcodes, wobei die genaue Stelle, auf die sich die Meldung bezieht, mit einem kleinen Pfeil markiert ist. Wichtig ist, dass dies die Stelle ist, an der der Interpreter den Fehler erstmalig feststellen konnte. Das ist nicht unbedingt gleichbedeutend mit der Stelle, an der der Fehler gemacht wurde.
- Die letzte Zeile spezifiziert den Typ der Fehlermeldung, in diesem Fall einen Syntax Error. Dies sind die am häufigsten auftretenden Fehlermeldungen. Sie zeigen an, dass der Compiler das Programm aufgrund eines formalen Fehlers nicht weiter übersetzen konnte.

Neben dem Syntaxfehler gibt es eine Reihe weiterer Fehlertypen, die an dieser Stelle nicht alle im Detail besprochen werden sollen. Wir möchten jedoch noch auf den `IndentationError` (dt. »Einrückungsfehler«) hinweisen, da er gerade bei Python-Anfängern häufig auftritt. Versuchen Sie dazu einmal, folgendes Programm auszuführen:

```
i = 10
if i == 10:
print("Falsch eingerückt")
```

Sie sehen, dass die letzte Zeile eigentlich einen Schritt weiter eingerückt sein müsste. So, wie das Programm jetzt geschrieben wurde, hat die `if`-Anweisung keinen Anweisungskörper. Das ist nicht zulässig, und es tritt ein `IndentationError` auf:

```
File "indent.py", line 3
    print("Falsch eingerückt")
    ^
```

```
IndentationError: expected an indented block
```

Nachdem wir uns mit diesen Grundlagen vertraut gemacht haben, kommen wir zu den Kontrollstrukturen, die es dem Programmierer erlauben, den Programmfluss zu steuern.

## Kapitel 19

# Funktionen

Aus der Mathematik kennen Sie den Begriff der *Funktion*, mit dem eine Zuordnungsvorschrift bezeichnet wird. Die Funktion  $f(x) = x^2$  ordnet beispielsweise dem Parameter  $x$  sein Quadriertes zu. Eine Funktion im mathematischen Sinne besteht aus einem Namen, einer Liste von Parametern und einer Berechnungsvorschrift für den Funktionswert.

In der Programmierung findet sich das mathematische Konzept der Funktion wieder. Wir haben beispielsweise bereits die eingebaute Funktion `len` besprochen, die die Länge eines iterierbaren Objekts berechnet. Dazu bekommt sie das entsprechende Objekt als Parameter übergeben und gibt das Ergebnis in Form eines Rückgabewertes zurück.

```
>>> len("Dieser String ist ein Parameter")
31
```

Offensichtlich besteht hier eine gewisse Analogie zum mathematischen Begriff der Funktion. Eine Funktion in der Programmierung besteht aus einem *Funktionsnamen*, einer Liste von *Funktionsparametern* und einem Code-Block, dem *Funktionskörper*. Bei einem Funktionsaufruf wird dann der Funktionskörper unter Berücksichtigung der übergebenen Parameter ausgeführt. Eine Funktion in Python kann, wie `len`, einen *Rückgabewert* haben oder nicht.<sup>1</sup>

Funktionen werden in der Programmierung dazu eingesetzt, um Redundanzen im Quellcode zu vermeiden. Das bedeutet, dass Code-Stücke, die in der gleichen oder einer ähnlichen Form öfter im Programm benötigt werden, nicht jedes Mal neu geschrieben, sondern in einer Funktion gekapselt werden. Diese Funktion kann dann an den Stellen, an denen sie benötigt wird, aufgerufen werden. Darüber hinaus bilden Funktionen ein elegantes Hilfsmittel, um einen langen Quellcode sinnvoll in Unterprogramme aufzuteilen. Das erhöht die Les- und Wartbarkeit des Codes.

Im Folgenden erläutern wir die Handhabung einer bestehenden Funktion am Beispiel von `range`. Vieles des hier Gesagten kennen Sie bereits aus Kapitel 8, »Funktio-

---

<sup>1</sup> In Python wird nicht, wie beispielsweise in PASCAL, zwischen den Begriffen *Funktion* und *Prozedur* unterschieden. Unter einer Prozedur versteht man eine Funktion, die keinen Rückgabewert hat.

nen, Methoden und Attribute«, wir möchten es an dieser Stelle trotzdem noch einmal wiederholen.

Die eingebaute Funktion `range` wurde in Abschnitt 5.2.6, »Die for-Schleife als Zählschleife«, eingeführt und erzeugt ein iterierbares Objekt über eine begrenzte Anzahl fortlaufender ganzer Zahlen:

```
ergebnis = range(0, 10, 2)
```

Im Beispiel oben wurde `range` aufgerufen; man nennt dies den *Funktionsaufruf*. Dazu wird hinter den Namen der Funktion ein (möglicherweise leeres) Klammernpaar geschrieben. Innerhalb dieser Klammern stehen, durch Kommata getrennt, die *Parameter* der Funktion. Wie viele es sind und welche Art von Parametern eine Funktion erwartet, hängt von der Definition der Funktion ab und ist sehr verschieden. In diesem Fall benötigt `range` drei Parameter, um ausreichend Informationen zu erlangen. Die Gesamtheit der Parameter wird *Funktionsschnittstelle* genannt. Konkrete, über eine Schnittstelle übergebene Instanzen heißen *Argumente*. Ein *Parameter* hingegen bezeichnet einen Platzhalter für Argumente.

Nachdem die Funktion abgearbeitet wurde, wird ihr Ergebnis zurückgegeben. Sie können sich bildlich vorstellen, dass der Funktionsaufruf, wie er im Quelltext steht, durch den Rückgabewert ersetzt wird. Im Beispiel oben haben wir dem Rückgabewert von `range` direkt einen Namen zugewiesen und können auf ihn fortan über `ergebnis` zugreifen. So können wir beispielsweise in einer `for`-Schleife über das Ergebnis des `range`-Aufrufs iterieren:

```
>>> for i in ergebnis:
...     print(i)
...
0
2
4
6
8
```

Es ist auch möglich, das Ergebnis des `range`-Aufrufs mit `list` in eine Liste zu überführen:

```
>>> liste = list(ergebnis)
>>> liste
[0, 2, 4, 6, 8]
>>> liste[3]
6
```

So viel vorerst zur Verwendung vordefinierter Funktionen. Python erlaubt es Ihnen, eigene Funktionen zu schreiben, die Sie nach demselben Schema verwenden kön-

nen, wie es hier beschrieben wurde. Im nächsten Abschnitt werden wir uns damit befassen, wie Sie eine eigene Funktion erstellen.

## 19.1 Schreiben einer Funktion

Bevor wir uns an konkreten Quelltext wagen, möchten wir rekapitulieren, was eine Funktion ausmacht, was also bei der Definition einer Funktion anzugeben ist:

- Eine Funktion muss einen *Namen* haben, über den sie in anderen Teilen des Programms aufgerufen werden kann. Die Zusammensetzung des Funktionsnamens erfolgt nach denselben Regeln wie die Namensgebung einer Referenz.<sup>2</sup>
- Eine Funktion muss eine *Schnittstelle* haben, über die Informationen vom aufrufenden Programmteil in den Kontext der Funktion übertragen werden. Eine Schnittstelle kann aus beliebig vielen (unter Umständen auch keinen) Parametern bestehen. Funktionsintern wird jedem dieser Parameter ein Name gegeben. Sie lassen sich dann wie Referenzen im Funktionskörper verwenden.
- Eine Funktion muss einen *Wert* zurückgeben. Jede Funktion gibt automatisch `None` zurück, wenn der Rückgabewert nicht ausdrücklich angegeben wurde.

Zur Definition einer Funktion wird in Python das Schlüsselwort `def` verwendet. Syntaktisch sieht die Definition folgendermaßen aus:

```
def Funktionsname(parameter_1, ..., parameter_n):
    Anweisung
    :
    Anweisung
```

Abbildung 19.1 Definition einer Funktion

Nach dem Schlüsselwort `def` steht der gewählte Funktionsname. Dahinter werden in einem Klammernpaar die Namen aller Parameter aufgelistet. Nach der Definition der Schnittstelle folgen ein Doppelpunkt und, eine Stufe weiter eingerückt, der Funktionskörper. Bei dem Funktionskörper handelt es sich um einen beliebigen Code-Block, in dem die Parameternamen als Referenzen verwendet werden dürfen. Im Funktionskörper dürfen auch wieder Funktionen aufgerufen werden.

Betrachten wir einmal die konkrete Implementierung einer Funktion, die die Fakultät einer ganzen Zahl berechnet und das Ergebnis auf dem Bildschirm ausgibt:

<sup>2</sup> Das bedeutet, dass sich der Funktionsname aus großen und kleinen Buchstaben, Zahlen sowie dem Unterstrich (»\_«) zusammensetzen, allerdings nicht mit einer Zahl beginnen darf. Seit Python 3.0 dürfen auch Buchstaben verwendet werden, die nicht im englischen Alphabet enthalten sind.



```
def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    print(ergebnis)
```

Anhand dieses Beispiels können Sie gut nachvollziehen, wie der Parameter `zahl` im Funktionskörper verarbeitet wird. Nachdem die Berechnung erfolgt ist, wird `ergebnis` mittels `print` ausgegeben. Die Referenz `zahl` ist nur innerhalb des Funktionskörpers definiert und hat nichts mit anderen Referenzen außerhalb der Funktion zu tun.

Wenn Sie das obige Beispiel jetzt speichern und ausführen, werden Sie feststellen, dass zwar keine Fehlermeldung angezeigt wird, aber auch sonst nichts passiert. Nun, das liegt daran, dass wir bisher nur eine Funktion definiert haben. Um sie konkret im Einsatz zu sehen, müssen wir sie mindestens einmal aufrufen. Folgendes Programm liest in einer Schleife Zahlen vom Benutzer ein und berechnet deren Fakultät mithilfe der soeben definierten Funktion:

```
def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    print(ergebnis)

while True:
    eingabe = int(input("Geben Sie eine Zahl ein: "))
    fak(eingabe)
```

Sie sehen, dass der Quellcode in zwei Komponenten aufgeteilt wurde: zum einen in die Funktionsdefinition oben und zum anderen in das auszuführende Hauptprogramm unten. Das Hauptprogramm besteht aus einer Endlosschleife, in der die Funktion `fak` mit der eingegebenen Zahl als Parameter aufgerufen wird.

Betrachten Sie noch einmal die beiden Komponenten des Programms. Es wäre im Sinne der Kapselung der Funktionalität erstrebenswert, das Programm so zu ändern, dass sich das Hauptprogramm allein um die Interaktion mit dem Benutzer und das Anstoßen der Berechnung kümmert, während das Unterprogramm `fak` die Berechnung tatsächlich durchführt. Das Ziel dieses Ansatzes ist es vor allem, dass die Funktion `fak` auch in anderen Programmteilen zur Berechnung einer weiteren Fakultät aufgerufen werden kann. Dazu ist es unerlässlich, dass `fak` sich ausschließlich um die Berechnung kümmert. Es passt nicht in dieses Konzept, dass `fak` das Ergebnis der Berechnung selbst ausgibt.

Idealerweise sollte unsere Funktion `fak` die Berechnung abschließen und das Ergebnis an das Hauptprogramm zurückgeben, sodass die Ausgabe dort erfolgen kann.

Dies erreichen Sie durch das Schlüsselwort `return`, das die Ausführung der Funktion sofort beendet und einen eventuell angegebenen Rückgabewert zurückgibt.

```
def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    return ergebnis
```

```
while True:
    eingabe = int(input("Geben Sie eine Zahl ein: "))
    print(fak(eingabe))
```

Eine Funktion kann zu jeder Zeit im Funktionsablauf mit `return` beendet werden. Die folgende Version der Funktion prüft vor der Berechnung, ob es sich bei dem übergebenen Parameter um eine negative Zahl handelt. Ist das der Fall, wird die Abhandlung der Funktion abgebrochen:

```
def fak(zahl):
    if zahl < 0:
        return None
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    return ergebnis
```

```
while True:
    eingabe = int(input("Geben Sie eine Zahl ein: "))
    ergebnis = fak(eingabe)
    if ergebnis is None:
        print("Fehler bei der Berechnung")
    else:
        print(ergebnis)
```

In der zweiten Zeile des Funktionskörpers wurde mit `return None` explizit der Wert `None` zurückgegeben, was nicht unbedingt nötig ist. Der folgende Code ist äquivalent:

```
if zahl < 0:
    return
```

Vom Programmablauf her ist es egal, ob Sie `None` explizit oder implizit zurückgeben. Aus Gründen der Lesbarkeit ist `return None` in diesem Fall trotzdem sinnvoll, denn es handelt sich um einen ausdrücklich gewünschten Rückgabewert. Er ist Teil der Funktionslogik und nicht bloß ein Nebenprodukt des Funktionsabbruchs.

Die Funktion `fak`, wie sie in diesem Beispiel zu sehen ist, kann zu jeder Zeit zur Berechnung einer Fakultät aufgerufen werden, unabhängig davon, in welchem Kontext diese Fakultät benötigt wird.

Selbstverständlich können Sie in Ihrem Quelltext mehrere eigene Funktionen definieren und aufrufen. Das folgende Beispiel soll bei Eingabe einer negativen Zahl keine Fehlermeldung, sondern die Fakultät des Betrags dieser Zahl ausgeben:

```
def betrag(zahl):
    if zahl < 0:
        return -zahl
    else:
        return zahl

def fak(zahl):
    ergebnis = 1
    for i in range(2, zahl+1):
        ergebnis *= i
    return ergebnis

while True:
    eingabe = int(input("Geben Sie eine Zahl ein: "))
    print(fak(betrag(eingabe)))
```

Für die Berechnung des Betrags einer Zahl gibt es in Python auch die Built-in Funktion `abs`. Diese werden Sie noch in diesem Kapitel kennenlernen.

Ein Begriff soll noch eingeführt werden, bevor wir uns den Funktionsparametern widmen. Eine Funktion kann über ihren Namen nicht nur aufgerufen, sondern auch wie eine Instanz behandelt werden. So ist es beispielsweise möglich, den Typ einer Funktion abzufragen. Die folgenden Beispiele nehmen an, dass die Funktion `fak` im interaktiven Modus verfügbar ist:

```
>>> type(fak)
<class 'function'>
>>> p = fak
>>> p(5)
120
>>> fak(5)
120
```

Durch die Definition einer Funktion wird ein sogenanntes *Funktionsobjekt* erzeugt, das über den Funktionsnamen referenziert wird.

## 19.2 Funktionsparameter

Wir haben bereits oberflächlich besprochen, was Funktionsparameter sind und wie sie verwendet werden können, doch das ist bei Weitem noch nicht die ganze Wahrheit. In diesem Abschnitt werden Sie drei alternative Techniken zur Übergabe von Funktionsparametern kennenlernen.

### 19.2.1 Optionale Parameter

Zu Beginn dieses Kapitels wurde die Verwendung einer Funktion anhand der Built-in Funktion `range` erklärt. Sicherlich wissen Sie aus Abschnitt 5.2.6 über die `for`-Schleife noch, dass der letzte der drei Parameter der `range`-Funktion optional ist. Das bedeutet zunächst einmal, dass dieser Parameter beim Funktionsaufruf weggelassen werden kann. Ein optionaler Parameter muss funktionsintern mit einem Wert vorbelegt sein, üblicherweise einem Standardwert, der in einem Großteil der Funktionsaufrufe ausreichend ist. Bei der Funktion `range` regelt der dritte Parameter die Schrittweite und ist mit 1 vorbelegt. Folgende Aufrufe von `range` sind also äquivalent:

- ▶ `range(2, 10, 1)`
- ▶ `range(2, 10)`

Dies ist interessant, denn oftmals hat eine Funktion ein Standardverhalten, das sich durch zusätzliche Parameter an spezielle Gegebenheiten anpassen lassen soll. In den überwiegenden Fällen, in denen das Standardverhalten jedoch genügt, wäre es umständlich, trotzdem die für diesen Aufruf überflüssigen Parameter anzugeben. Deswegen sind vordefinierte Parameterwerte oft eine sinnvolle Ergänzung einer Funktionsschnittstelle.

Um einen Funktionsparameter mit einem Default-Wert vorzubelegen, wird dieser Wert bei der Funktionsdefinition zusammen mit einem Gleichheitszeichen hinter den Parameternamen geschrieben. Die folgende Funktion soll, je nach Anwendung, die Summe von zwei, drei oder vier ganzen Zahlen berechnen und das Ergebnis zurückgeben. Dabei soll der Programmierer beim Aufruf der Funktion nur so viele Zahlen angeben müssen, wie er benötigt:

```
def summe(a, b, c=0, d=0):
    return a + b + c + d
```

Um eine Addition durchzuführen, müssen mindestens zwei Parameter übergeben worden sein. Die anderen beiden werden mit 0 vorbelegt. Sollten sie beim Funktionsaufruf nicht explizit angegeben werden, fließen sie nicht in die Addition ein. Die Funktion kann folgendermaßen aufgerufen werden:

```
>>> summe(1, 2)
3
>>> summe(1, 2, 3)
6
>>> summe(1, 2, 3, 4)
10
```

Beachten Sie, dass optionale Parameter nur am Ende einer Funktionsschnittstelle stehen dürfen. Das heißt, dass auf einen optionalen kein nicht-optionaler Parameter mehr folgen darf. Diese Einschränkung ist wichtig, damit alle angegebenen Parameter eindeutig zugeordnet werden können.

### 19.2.2 Schlüsselwortparameter

Neben den bislang verwendeten sogenannten *Positional Arguments* (Positionsparameter) gibt es in Python eine weitere Möglichkeit, Parameter zu übergeben. Solche Parameter werden *Keyword Arguments* (Schlüsselwortparameter) genannt. Es handelt sich dabei um eine alternative Technik, Parameter beim Funktionsaufruf zu übergeben. An der Funktionsdefinition ändert sich nichts. Betrachten wir dazu unsere Summenfunktion, die wir im vorangegangenen Abschnitt geschrieben haben:

```
def summe(a, b, c=0, d=0):
    return a + b + c + d
```

Diese Funktion kann auch folgendermaßen aufgerufen werden:

```
summe(d=1, b=3, c=2, a=1)
```

Dazu werden im Funktionsaufruf die Parameter, wie bei einer Zuweisung, auf den gewünschten Wert gesetzt. Da bei der Übergabe der jeweilige Parametername angegeben werden muss, ist die Zuordnung unter allen Umständen eindeutig. Das erlaubt es dem Programmierer, Schlüsselwortparameter in beliebiger Reihenfolge anzugeben.

Es ist möglich, beide Formen der Parameterübergabe zu kombinieren. Dabei ist zu beachten, dass keine Positional Arguments auf Keyword Arguments folgen dürfen, Letztere also immer am Ende des Funktionsaufrufs stehen müssen.

```
summe(1, 2, c=10, d=11)
```

Beachten Sie außerdem, dass nur solche Parameter als Keyword Arguments übergeben werden dürfen, die im selben Funktionsaufruf nicht bereits als Positional Arguments übergeben wurden.

### 19.2.3 Beliebige Anzahl von Parametern

Rufen Sie sich noch einmal die Verwendung der eingebauten Funktion `print` in Erinnerung:

```
>>> print("P")
P
>>> print("P", "y", "t", "h", "o", "n")
P y t h o n
>>> print("P", "y", "t", "h", "o", "n", " ", "i", "s", "t", " ",
... "s", "u", "p", "e", "r")
P y t h o n   i s t   s u p e r
```

Offensichtlich ist es möglich, der Funktion `print` eine beliebige Anzahl von Parametern zu übergeben. Diese Eigenschaft ist nicht exklusiv für die `print`-Funktion, sondern es können auch eigene Funktionen definiert werden, denen beliebig viele Parameter übergeben werden können.

Für beide Formen der Parameterübergabe (Positional und Keyword) gibt es eine Notation, die es einer Funktion ermöglicht, beliebig viele Parameter entgegenzunehmen. Bleiben wir zunächst einmal bei den Positional Arguments. Betrachten Sie dazu folgende Funktionsdefinition:

```
def funktion(a, b, *weitere):
    print("Feste Parameter:", a, b)
    print("Weitere Parameter:", weitere)
```

Zunächst einmal werden ganz klassisch zwei Parameter `a` und `b` festgelegt und zusätzlich ein dritter namens `weitere`. Wichtig ist der Stern vor seinem Namen. Bei einem Aufruf dieser Funktion würden `a` und `b`, wie Sie das bereits kennen, die ersten beiden übergebenen Instanzen referenzieren. Interessant ist, dass `weitere` fortan ein Tupel referenziert, das alle zusätzlich übergebenen Instanzen enthält. Anschaulich wird dies, wenn wir folgende Funktionsaufrufe betrachten:

```
funktion(1, 2)
funktion(1, 2, "Hallo Welt", 42, [1,2,3,4])
```

Die Ausgabe der Funktion im Falle des ersten Aufrufs ist:

```
Feste Parameter: 1 2
Weitere Parameter: ()
```

Der Parameter `weitere` referenziert also ein leeres Tupel. Im Falle des zweiten Aufrufs sieht die Ausgabe folgendermaßen aus:

Feste Parameter: 1 2

Weitere Parameter: ('Hallo Welt', 42, [1, 2, 3, 4])

Der Parameter `weitere` referenziert nun ein Tupel, in dem alle über `a` und `b` hinausgehenden Instanzen in der Reihenfolge enthalten sind, in der sie übergeben wurden.

An dieser Stelle möchten wir die im vorangegangenen Beispiel definierte Funktion `summe` dahingehend erweitern, dass sie die Summe einer vom Benutzer festgelegten Zahl von Parametern berechnen kann:

```
def summe(*parameter):
    s = 0
    for p in parameter:
        s += p
    return s
```

Das folgende Beispiel demonstriert die Anwendung der weiterentwickelten Funktion `summe` im interaktiven Modus:

```
>>> summe(1, 2, 3, 4, 5)
15
>>> summe(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
78
```

Diese Art, einer Funktion das Entgegennehmen beliebig vieler Parameter zu ermöglichen, funktioniert ebenso für Keyword Arguments. Der Unterschied besteht darin, dass der Parameter, der alle weiteren Instanzen enthalten soll, in der Funktionsdefinition mit zwei Sternen geschrieben werden muss. Außerdem referenziert er später kein Tupel, sondern ein Dictionary. Dieses Dictionary enthält den jeweiligen Parameternamen als Schlüssel und die übergebene Instanz als Wert. Betrachten Sie dazu folgende Funktionsdefinition:

```
def funktion(a, b, **weitere):
    print("Feste Parameter:", a, b)
    print("Weitere Parameter:", weitere)
```

und diese beiden dazu passenden Funktionsaufrufe:

```
funktion(1, 2)
funktion(1, 2, johannes="ernesti", peter="kaiser")
```

Die Ausgabe nach dem ersten Funktionsaufruf sieht folgendermaßen aus:

Feste Parameter: 1 2  
Weitere Parameter: {}

Der Parameter `weitere` referenziert also ein leeres Dictionary. Nach dem zweiten Aufruf sieht die Ausgabe so aus:

Feste Parameter: 1 2  
Weitere Parameter: {'johannes': 'ernesti', 'peter': 'kaiser'}

Beide Techniken können zusammen verwendet werden, wie folgende Funktionsdefinition zeigt:

```
def funktion(*positional, **keyword):
    print("Positional:", positional)
    print("Keyword:", keyword)
```

Der Funktionsaufruf

```
funktion(1, 2, 3, 4, hallo="welt", key="word")
```

gibt diese Werte aus:

Positional: (1, 2, 3, 4)  
Keyword: {'hallo': 'welt', 'key': 'word'}

Sie sehen, dass `positional` ein Tupel mit allen Positions- und `keyword` ein Dictionary mit allen Schlüsselwortparametern referenziert.

### 19.2.4 Reine Schlüsselwortparameter

Es ist möglich, Parameter zu definieren, die ausschließlich in Form von Schlüsselwortparametern übergeben werden dürfen. Solche reinen Schlüsselwortparameter<sup>3</sup> werden bei der Funktionsdefinition nach dem Parameter geschrieben, der beliebig viele Positionsargumente aufnimmt:

```
def f(a, b, *c, d, e):
    print(a, b, c, d, e)
```

In diesem Fall besteht die Funktionsschnittstelle aus den beiden Positionsparametern `a` und `b`, der Möglichkeit für weitere Positionsparameter `*c` und den beiden reinen Schlüsselwortparametern `d` und `e`. Es gibt keine Möglichkeit, die Parameter `d` und `e` zu übergeben, außer in Form von Schlüsselwortparametern.

```
>>> f(1, 2, 3, 4, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() needs keyword-only argument d
```

<sup>3</sup> engl. *keyword-only parameters*

```
>>> f(1, 2, 3, 4, 5, d=4, e=5)
1 2 (3, 4, 5) 4 5
```

Wie bei Positionsparametern müssen reine Schlüsselwortparameter angegeben werden, sofern sie nicht mit einem Default-Wert belegt sind:

```
>>> def f(a, b, *c, d=4, e=5):
...     print(a, b, c, d, e)
...
>>> f(1, 2, 3)
1 2 (3, 4, 5) 4 5
```

Wenn zusätzlich die Übergabe beliebig vieler Schlüsselwortparameter ermöglicht werden soll, folgt die dazu notwendige **\*\***-Notation nach den reinen Schlüsselwortparametern am Ende der Funktionsdefinition:

```
def f(a, b, *args, d, e, **kwargs):
    print(a, b, args, d, e, kwargs)
```

Es ist auch möglich, reine Schlüsselwortparameter zu definieren, ohne gleichzeitig beliebig viele Positionsparameter zuzulassen. Dazu werden die reinen Schlüsselwortparameter in der Funktionsschnittstelle durch einen **\*** von den Positionsparametern getrennt.

```
>>> def f(a, b, *, c, d):
...     print(a, b, c, d)
...
>>> f(1, 2, 3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 2 positional arguments (4 given)
>>> f(1, 2, c=3, d=4)
1 2 3 4
```

### 19.2.5 Entpacken einer Parameterliste

In diesem Abschnitt lernen Sie eine weitere Möglichkeit kennen, Parameter an eine Funktion zu übergeben. Dazu stellen wir uns vor, wir wollten mithilfe der in Abschnitt 19.2.3, »Beliebige Anzahl von Parametern«, definierten erweiterten Version der `summe`-Funktion die Summe aller Einträge eines Tupels bestimmen. Dazu ist momentan die folgende Notation nötig:

```
>>> t = (1, 4, 3, 7, 9, 2)
>>> summe(t[0], t[1], t[2], t[3], t[4], t[5])
26
```

Das ist sehr umständlich. Zudem laufen wir der Allgemeinheit der Funktion `summe` zuwider, denn die Anzahl der Elemente des Tupels `t` muss stets bekannt sein. Wünschenswert ist ein Weg, eine in einem iterierbaren Objekt gespeicherte Liste von Argumenten direkt einer Funktion übergeben zu können. Dieser Vorgang wird *Entpacken* genannt.

Das Entpacken eines iterierbaren Objekts geschieht dadurch, dass der Funktion das Objekt mit einem vorangestellten Sternchen (**\***) übergeben wird. Im folgenden Beispiel wird das von der eingebauten Funktion `range` erzeugte iterierbare Objekt verwendet, um mithilfe der Funktion `summe` die Summe der ersten 100 natürlichen Zahlen zu berechnen:<sup>4</sup>

```
>>> summe(*range(101))
5050
```

Beim Funktionsaufruf wird der Funktion jedes Element des iterierbaren Objekts, in diesem Fall also die Zahlen von 0 bis 100, als gesonderter Parameter übergeben. Das Entpacken einer Parameterliste funktioniert nicht nur im Zusammenhang mit einer Funktion, die beliebig viele Parameter erwartet, sondern kann auch mit der ursprünglichen Funktion `summe`, die die Summe von maximal vier Parametern bestimmt, verwendet werden:

```
def summe(a, b, c=0, d=0):
    return a + b + c + d
```

Beachten Sie dabei, dass das zu entpackende iterierbare Objekt auch maximal vier (und mindestens zwei) Elemente bereitstellt:

```
>>> t = (6, 3, 9, 12)
>>> summe(*t)
30
>>> summe(*[4, 6, 12, 7, 9])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: summe() takes at most 4 positional arguments (5 given)
```

Analog zum Entpacken eines Tupels zu einer Liste von Positionsparametern kann ein Dictionary zu einer Liste von Schlüsselwortparametern entpackt werden. Der Unterschied in der Notation besteht darin, dass zum Entpacken eines Dictionarys zwei Sternchen vorangestellt werden müssen:

<sup>4</sup> Das beim Funktionsaufruf von `range(n)` zurückgegebene iterierbare Objekt durchläuft alle ganzen Zahlen von 0 bis einschließlich `n-1`. Daher muss im Beispiel 101 anstelle von 100 übergeben werden.

```
>>> d = {"a" : 7, "b" : 3, "c" : 4}
>>> summe(**d)
14
```

Abschließend ist noch zu erwähnen, dass die Techniken zum Entpacken von Parametern miteinander kombiniert werden können, wie folgendes Beispiel zeigt:

```
>>> summe(1, *(2,3), **{"d" : 4})
10
```

Beachten Sie allgemein, dass die hier vorgestellten Notationen nur innerhalb eines Funktionsaufrufs verwendet werden dürfen und außerhalb dessen zu einem Fehler führen.

### 19.2.6 Seiteneffekte

Bisher haben wir diese Thematik geschickt umschifft, doch Sie sollten immer im Hinterkopf behalten, dass sogenannte *Seiteneffekte* (engl. *side effects*) immer dann auftreten können, wenn eine Instanz eines mutablen Datentyps, also zum Beispiel einer Liste oder eines Dictionarys, als Funktionsparameter übergeben wird.

In Python werden bei einem Funktionsaufruf keine Kopien der als Parameter übergebenen Instanzen erzeugt, sondern es wird funktionsintern mit Referenzen auf die Argumente gearbeitet.<sup>5</sup> Betrachten Sie dazu folgendes Beispiel:

```
>>> def f(a, b):
...     print(id(a))
...     print(id(b))
...
>>> p = 1
>>> q = [1,2,3]
>>> id(p)
134537016
>>> id(q)
134537004
>>> f(p, q)
134537016
134537004
```

Im interaktiven Modus definieren wir zuerst eine Funktion *f*, die zwei Parameter *a* und *b* erwartet und deren jeweilige Identität ausgibt. Anschließend werden zwei

<sup>5</sup> Diese Methode der Parameterübergabe wird *Call by Reference* genannt. Demgegenüber steht das Prinzip *Call by Value*, bei dem funktionsintern auf Kopien der Argumente gearbeitet wird. Letztere Variante ist frei von Seiteneffekten, aber aufgrund des Kopierens langsamer.

Referenzen *p* und *q* angelegt, die eine ganze Zahl bzw. eine Liste referenzieren. Dann lassen wir uns die Identitäten der beiden Referenzen ausgeben und rufen die angelegte Funktion *f* auf. Sie sehen, dass die ausgegebenen Identitäten gleich sind. Es handelt sich also sowohl bei *p* und *q* als auch bei *a* und *b* im Funktionskörper um Referenzen auf dieselben Instanzen. Dabei macht es zunächst einmal keinen Unterschied, ob die referenzierten Objekte Instanzen eines veränderlichen oder unveränderlichen Datentyps sind.

Trotzdem ist die Verwendung eines unveränderlichen Datentyps grundsätzlich frei von Seiteneffekten, da dieser bei Veränderung automatisch kopiert wird und alte Referenzen davon nicht berührt werden. Sollten wir also beispielsweise *a* im Funktionskörper um eins erhöhen, werden nachher *a* und *p* verschiedene Instanzen referenzieren. Dies führt dazu, dass bei der Verwendung unveränderlicher Datentypen in Funktionsschnittstellen keine Seiteneffekte auftreten können.<sup>6</sup>

Diese Sicherheit können uns veränderliche Datentypen, etwa Listen oder Dictionarys, nicht geben. Dazu folgendes Beispiel:

```
def f(liste):
    liste[0] = 42
    liste += [5,6,7,8,9]
```

```
zahlen = [1,2,3,4]
```

```
print(zahlen)
f(zahlen)
print(zahlen)
```

Zunächst wird eine Funktion definiert, die eine Liste als Parameter erwartet und diese im Funktionskörper verändert. Im Hauptprogramm wird eine Liste angelegt und ausgegeben. Danach wird die Funktion aufgerufen und die Liste erneut ausgegeben. Die Ausgabe des Beispiels sieht folgendermaßen aus:

```
[1, 2, 3, 4]
[42, 2, 3, 4, 5, 6, 7, 8, 9]
```

Es ist zu erkennen, dass sich die Änderungen nicht allein auf den Kontext der Funktion beschränken, sondern sich auch im Hauptprogramm auswirken. Dieses Phänomen wird *Seiteneffekt* genannt. Wenn eine Funktion nicht nur lesend auf eine Instanz eines veränderlichen Datentyps zugreifen muss und Seiteneffekte nicht ausdrücklich erwünscht sind, sollten Sie innerhalb der Funktion oder bei der Parameterüber-

<sup>6</sup> Beachten Sie, dass dies nicht für unveränderliche Instanzen gilt, die veränderliche Instanzen enthalten. So können bei der Parameterübergabe eines Tupels, das eine Liste enthält, durchaus Seiteneffekte auftreten.



gabe eine Kopie der Instanz erzeugen. Das kann in Bezug auf das oben genannte Beispiel so aussehen:<sup>7</sup>

```
f(zahlen[:])
```

Neben den bisher besprochenen Referenzparametern gibt es eine weitere, seltenere Form von Seiteneffekten, die auftritt, wenn ein veränderlicher Datentyp als Default-Wert eines Parameters verwendet wird:

```
>>> def f(a=[1,2,3]):
...     a += [4,5]
...     print(a)
...
>>> f()
[1, 2, 3, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5]
>>> f()
[1, 2, 3, 4, 5, 4, 5, 4, 5, 4, 5]
```

Wir definieren im interaktiven Modus eine Funktion, die einen einzigen Parameter erwartet, der mit einer Liste vorbelegt ist. Im Funktionskörper wird diese Liste um zwei Elemente vergrößert und ausgegeben. Nach mehrmaligem Aufrufen der Funktion ist zu erkennen, dass es sich bei dem Default-Wert augenscheinlich immer um dieselbe Instanz gehandelt hat.

Das liegt daran, dass eine Instanz, die als Default-Wert genutzt wird, nur einmalig und nicht bei jedem Funktionsaufruf neu erzeugt wird. Grundsätzlich sollten Sie also darauf verzichten, Instanzen veränderlicher Datentypen als Default-Werte zu verwenden. Schreiben Sie Ihre Funktionen stattdessen folgendermaßen:

```
def f(a=None):
    if a is None:
        a = [1,2,3]
```

Selbstverständlich können Sie anstelle von `None` eine Instanz eines beliebigen anderen immutablen Datentyps verwenden, ohne dass Seiteneffekte auftreten.

<sup>7</sup> Sie erinnern sich, dass beim Slicen einer Liste stets eine Kopie derselben erzeugt wird. Im Beispiel wurde das Slicing ohne Angabe von Start- und Endindex verwendet, um eine vollständige Kopie der Liste zu erzeugen.

## 19.3 Namensräume

Bisher wurde ein Funktionskörper als abgekapselter Bereich betrachtet, der ausschließlich über Parameter bzw. den Rückgabewert Informationen mit dem Hauptprogramm austauschen kann. Das ist zunächst auch gar keine schlechte Sichtweise, denn so hält man seine Schnittstelle »sauber«. In manchen Situationen ist es aber sinnvoll, eine Funktion über ihren lokalen Namensraum hinaus wirken zu lassen, was in diesem Kapitel thematisiert werden soll.

### 19.3.1 Zugriff auf globale Variablen – global

Zunächst einmal müssen zwei Begriffe unterschieden werden. Wenn wir uns im Kontext einer Funktion, also im Funktionskörper, befinden, dann können wir dort selbstverständlich Referenzen und Instanzen erzeugen und verwenden. Diese haben jedoch nur unmittelbar in der Funktion selbst Gültigkeit. Sie existieren im *lokalen Namensraum*. Im Gegensatz dazu existieren Referenzen des Hauptprogramms im *globalen Namensraum*.

Begrifflich wird auch zwischen *globalen Referenzen* und *lokalen Referenzen* unterschieden. Dazu folgendes Beispiel:

```
def f():
    a = "lokaler String"
b = "globaler String"
```

Die Unterscheidung zwischen globalem und lokalem Namensraum wird anhand des folgenden Beispiels deutlich:

```
def f(a):
    print(a)
```

```
a = 10
f(100)
```

In diesem Beispiel existiert sowohl im globalen als auch im lokalen Namensraum eine Referenz namens `a`. Im globalen Namensraum referenziert sie die ganze Zahl 10 und im lokalen Namensraum der Funktion den übergebenen Parameter, in diesem Fall die ganze Zahl 100. Es ist wichtig, zu verstehen, dass diese beiden Referenzen nichts miteinander zu tun haben, da sie in verschiedenen Namensräumen existieren. Abbildung 19.2 fasst das Konzept der Namensräume zusammen.

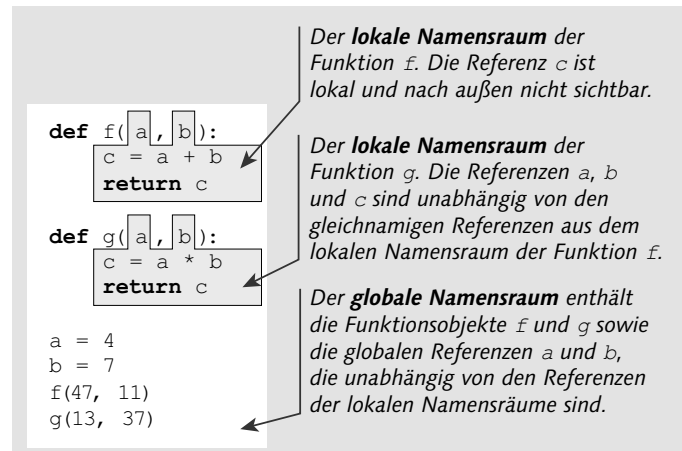


Abbildung 19.2 Abgrenzung lokaler Namensräume vom globalen Namensraum anhand eines Beispiels

### 19.3.2 Zugriff auf den globalen Namensraum

Im lokalen Namensraum eines Funktionskörpers kann jederzeit lesend auf eine globale Referenz zugegriffen werden, solange keine lokale Referenz gleichen Namens existiert:

```
def f():
    print(s)

s = "globaler String"
f()
```

Sobald versucht wird, schreibend auf eine globale Referenz zuzugreifen, wird stattdessen eine entsprechende lokale Referenz erzeugt:

```
def f():
    s = "lokaler String"
    print(s)

s = "globaler String"
f()
print(s)
```

Die Ausgabe dieses Beispiels lautet:

```
lokaler String
globaler String
```

Eine Funktion kann dennoch, mithilfe der `global`-Anweisung, schreibend auf eine globale Referenz zugreifen. Dazu muss im Funktionskörper das Schlüsselwort `global`, gefolgt von einer oder mehreren globalen Referenzen, geschrieben werden:

```
def f():
    global s
    s = "lokaler String"
    print(s)

s = "globaler String"
f()
print(s)
```

Die Ausgabe des Beispiels lautet:

```
lokaler String
lokaler String
```

Im Funktionskörper von `f` wird `s` explizit als globale Referenz gekennzeichnet und kann fortan als solche verwendet werden.

### 19.3.3 Lokale Funktionen

Es ist möglich, *lokale Funktionen* zu definieren. Das sind Funktionen, die im lokalen Namensraum einer anderen Funktion angelegt werden und nur dort gültig sind. Das folgende Beispiel zeigt eine solche Funktion:

```
def globale_funktion(n):
    def lokale_funktion(n):
        return n**2
    return lokale_funktion(n)
```

Innerhalb der globalen Funktion `globale_funktion` wurde eine lokale Funktion namens `lokale_funktion` definiert. Beachten Sie, dass der jeweilige Parameter `n` trotz des gleichen Namens nicht zwangsläufig denselben Wert referenziert. Die lokale Funktion kann im Namensraum der globalen Funktion völlig selbstverständlich wie jede andere Funktion auch aufgerufen werden.

Da sie einen eigenen Namensraum besitzt, hat die lokale Funktion keinen Zugriff auf lokale Referenzen der globalen Funktion. Um dennoch einige ausgewählte Referenzen an die lokale Funktion durchzuschleusen, bedient man sich eines Tricks mit vorbelegten Funktionsparametern:

```
def globale_funktion(n):
    def lokale_funktion(n=n):
```

```

    return n**2
return lokale_funktion()

```

Wie Sie sehen, muss der lokalen Funktion der Parameter `n` beim Aufruf nicht mehr explizit übergeben werden. Er wird vielmehr implizit in Form eines vorbelegten Parameters übergeben.

### 19.3.4 Zugriff auf übergeordnete Namensräume – `nonlocal`

Im letzten Abschnitt haben wir von den zwei existierenden Namensräumen, dem globalen und dem lokalen, gesprochen. Diese Unterteilung ist richtig, unterschlägt aber einen interessanten Fall, denn laut Abschnitt 19.3.3, »Lokale Funktionen«, dürfen auch lokale Funktionen innerhalb von Funktionen definiert werden. Lokale Funktionen bringen natürlich wieder ihren eigenen lokalen Namensraum im lokalen Namensraum der übergeordneten Funktion mit. Bei verschachtelten Funktionsdefinitionen kann man die Welt der Namensräume also nicht in eine lokale und eine globale Ebene unterteilen. Dennoch stellt sich auch hier die Frage, wie eine lokale Funktion auf Referenzen zugreifen kann, die im lokalen Namensraum der übergeordneten Funktion liegen.

Das Schlüsselwort `global` hilft dabei nicht weiter, denn es erlaubt nur den Zugriff auf den äußersten, globalen Namensraum. Für diesen Zweck existiert seit Python 3.0 das Schlüsselwort `nonlocal`.

Betrachten wir dazu einmal folgendes Beispiel:

```

def funktion1():
    def funktion2():
        nonlocal res
        res += 1
    res = 1
    funktion2()
    print(res)

```

Innerhalb der Funktion `funktion1` wurde eine lokale Funktion `funktion2` definiert, die die Referenz `res` aus dem lokalen Namensraum von `funktion1` inkrementieren soll. Dazu muss `res` innerhalb von `funktion2` als `nonlocal` gekennzeichnet werden. Die Schreibweise lehnt sich an den Zugriff auf Referenzen aus dem globalen Namensraum via `global` an.

Nachdem `funktion2` definiert wurde, wird `res` im lokalen Namensraum von `funktion1` definiert und mit dem Wert 1 verknüpft. Schließlich wird die lokale Funktion `funktion2` aufgerufen und der Wert von `res` ausgegeben. Im Beispiel gibt `funktion1` den Wert 2 aus.

Das Schlüsselwort `nonlocal` lässt sich auch bei mehreren, ineinander verschachtelten Funktionen verwenden, wie folgende Erweiterung unseres Beispiels zeigt:

```

def funktion1():
    def funktion2():
        def funktion3():
            nonlocal res
            res += 1
        nonlocal res
        funktion3()
        res += 1
    res = 1
    funktion2()
    print(res)

```

Nun wurde eine zusätzliche lokale Funktion im lokalen Namensraum von `funktion2` definiert. Auch aus dem lokalen Namensraum von `funktion3` heraus lässt sich `res` mithilfe von `nonlocal` inkrementieren. Die Funktion `funktion1` gibt in diesem Beispiel den Wert 3 aus.

Allgemein funktioniert `nonlocal` bei tieferen Funktionsverschachtelungen so, dass es in der Hierarchie der Namensräume aufsteigt und die erste Referenz mit dem angegebenen Namen in den Namensraum des `nonlocal`-Schlüsselworts einbindet.

## 19.4 Anonyme Funktionen

Beim Sortieren einer Liste mit der Built-in Function `sorted` kann eine Funktion übergeben werden, die die Ordnungsrelation der Elemente beschreibt. Auf diese Weise lassen sich die Elemente nach selbst definierten Kriterien sortieren:

```

>>> def s(x):
...     return -x
...
>>> sorted([1,4,7,3,5], key=s)
[7, 5, 4, 3, 1]

```

In diesem Fall wurde die Funktion `s` definiert, die einen übergebenen Wert negiert, um damit die Liste in absteigender Reihenfolge zu sortieren. Funktionen wie `s`, die in einem solchen oder ähnlichen Kontext verwendet werden, sind in der Regel sehr einfach und werden definiert, verwendet und dann vergessen.

Mithilfe des Schlüsselwortes `lambda` kann stattdessen eine kleine, anonyme Funktion erstellt werden:

```
s = lambda x: -x
```

Auf das Schlüsselwort `lambda` folgen eine Parameterliste und ein Doppelpunkt. Hinter dem Doppelpunkt muss ein beliebiger arithmetischer oder logischer Ausdruck stehen, dessen Ergebnis von der anonymen Funktion zurückgegeben wird. Beachten Sie, dass die Beschränkung auf einen arithmetischen Ausdruck zwar die Verwendung von Kontrollstrukturen ausschließt, nicht aber die Verwendung einer Conditional Expression.

Eine `lambda`-Form ergibt ein Funktionsobjekt und kann wie gewohnt aufgerufen werden: `s(10)`. Der Rückgabewert wäre in diesem Fall `-10`. Wie der Name schon andeutet, werden anonyme Funktionen jedoch in der Regel verwendet, ohne ihnen einen Namen zuzuweisen. Ein Beispiel dafür liefert das eingangs beschriebene Sortierproblem:

```
>>> sorted([1,4,7,3,5], key=lambda x: -x)
[7, 5, 4, 3, 1]
```

Betrachten wir noch ein etwas komplexeres Beispiel einer anonymen Funktion mit drei Parametern:

```
f = lambda x, y, z: (x - y) * z
```

Anonyme Funktionen können aufgerufen werden, ohne sie vorher zu referenzieren. Dazu muss der `lambda`-Ausdruck in Klammern gesetzt werden:

```
(lambda x, y, z: (x - y) * z)(1, 2, 3)
```

## 19.5 Annotationen

Die Elemente einer Funktionsschnittstelle, also die Funktionsparameter und der Rückgabewert, lassen sich mit Anmerkungen, sogenannten *Annotationen*, versehen:

```
def funktion(p1: Annotation1, p2: Annotation2) -> Annotation3:
    Funktionskörper
```

Bei der Definition einer Funktion kann hinter jeden Parameter ein Doppelpunkt, gefolgt von einer Annotation, geschrieben werden. Eine Annotation darf dabei ein beliebiger Python-Ausdruck sein. Die Angabe einer Annotation ist optional. Hinter der Parameterliste kann eine ebenfalls optionale Annotation für den Rückgabewert der Funktion geschrieben werden. Diese wird durch einen Pfeil (`->`) eingeleitet. Erst hinter dieser Annotation folgt der Doppelpunkt, der den Funktionskörper einleitet.

Annotationen ändern an der Ausführung einer Funktion nichts, man könnte sagen: Dem Python-Interpreter sind Annotationen egal. Das Interessante an ihnen ist, dass man sie über das Attribut `__annotations__` des Funktionsobjekts auslesen kann. Da

Annotationen beliebige Ausdrücke sein dürfen, kann der Programmierer hier also eine Information pro Parameter und Rückgabewert »speichern«, auf die er zu einem späteren Zeitpunkt – beispielsweise, wenn die Funktion mit konkreten Parameterwerten aufgerufen wird – zurückkommt.

Dabei werden die Annotationen über das Attribut `__annotations__` in Form eines Dictionarys zugänglich gemacht. Dieses Dictionary enthält die Parameternamen bzw. "return" für die Annotation des Rückgabewertes als Schlüssel und die jeweiligen Annotation-Ausdrücke als Werte. Für die oben dargestellte schematische Funktionsdefinition sieht dieses Dictionary also folgendermaßen aus:

```
funktion.__annotations__ =
{
    "p1" : Annotation1,
    "p2" : Annotation2,
    "return" : Annotation3
}
```

Mit Function Annotations könnten Sie also beispielsweise eine Typüberprüfung an der Funktionsschnittstelle durchführen. Dazu definieren wir zunächst eine Funktion samt Annotationen:

```
def strmult(s: str, n: int) -> str:
    return s*n
```

Die Funktion `strmult` hat die Aufgabe, einen String `s` `n`-mal hintereinander geschrieben zurückzugeben. Das geschieht durch Multiplikation von `s` und `n`.

Wir schreiben jetzt eine Funktion `call`, die dazu in der Lage ist, eine beliebige Funktion, deren Schnittstelle vollständig durch Annotationen beschrieben ist, aufzurufen bzw. eine Exception zu werfen, wenn einer der übergebenen Parameter einen falschen Typ hat:

```
def call(f, **kwargs):
    for arg in kwargs:
        if arg not in f.__annotations__:
            raise TypeError("Parameter '{}' "
                           " unbekannt".format(arg))
        if not isinstance(kwargs[arg], f.__annotations__[arg]):
            raise TypeError("Parameter '{}' "
                           " hat ungültigen Typ".format(arg))
    ret = f(**kwargs)
    if type(ret) != f.__annotations__["return"]:
        raise TypeError("Ungültiger Rückgabewert")
    return ret
```

Die Funktion `call` bekommt ein Funktionsobjekt und beliebig viele Schlüsselwortparameter übergeben. Dann greift sie für jeden übergebenen Schlüsselwortparameter auf das Annotation-Dictionary des Funktionsobjekts `f` zu und prüft, ob ein Parameter dieses Namens überhaupt in der Funktionsdefinition von `f` vorkommt und – wenn ja – ob die für diesen Parameter übergebene Instanz den richtigen Typ hat. Ist eines von beidem nicht der Fall, wird eine entsprechende Exception geworfen.

Wenn alle Parameter korrekt übergeben wurden, wird das Funktionsobjekt `f` aufgerufen und der Rückgabewert gespeichert. Dessen Typ wird dann mit dem Datentyp verglichen, der in der Annotation für den Rückgabewert angegeben wurde; wenn er abweicht, wird eine Exception geworfen. Ist alles gut gegangen, wird der Rückgabewert der Funktion `f` von `call` durchgereicht:

```
>>> call(strmult, s="Hallo", n=3)
'HalloHalloHallo'
>>> call(strmult, s="Hallo", n="Welt")
Traceback (most recent call last):
  [...]
TypeError: Parameter 'n' hat ungültigen Typ
>>> call(strmult, s=13, n=37)
Traceback (most recent call last):
  [...]
TypeError: Parameter 's' hat ungültigen Typ
```

Um die Überprüfung auf den Rückgabewert testen zu können, muss natürlich die Definition der Funktion `strmult` verändert werden.

19.6 Rekursion

Python erlaubt es dem Programmierer, sogenannte *rekursive Funktionen* zu schreiben. Das sind Funktionen, die sich selbst aufrufen. Die aufgerufene Funktion ruft sich so lange selbst auf, bis eine Abbruchbedingung diese – sonst endlose – Rekursion beendet. Die Anzahl der verschachtelten Funktionsaufrufe wird *Rekursionstiefe* genannt und ist von der Laufzeitumgebung auf einen bestimmten Wert begrenzt.

Im folgenden Beispiel wurde eine rekursive Funktion zur Berechnung der Fakultät einer ganzen Zahl geschrieben:

```
def fak(n):
    if n > 0:
        return fak(n - 1) * n
    else:
        return 1
```

Es soll nicht Sinn und Zweck dieses Abschnitts sein, vollständig in die Thematik der Rekursion einzuführen. Stattdessen möchten wir Ihnen hier nur einen kurzen Überblick geben. Sollten Sie das Beispiel nicht auf Anhieb verstehen, seien Sie nicht entmutigt, denn es lässt sich auch ohne Rekursion passabel in Python programmieren. Trotzdem sollten Sie nicht leichtfertig über die Rekursion hinwegsehen, denn es handelt sich dabei um einen interessanten Weg, sehr elegante Programme zu schreiben.<sup>8</sup>

19.7 Eingebaute Funktionen

Es war im Laufe des Buches schon oft von *eingebauten Funktionen* oder *Built-in Functions* die Rede. Das sind vordefinierte Funktionen, die dem Programmierer jederzeit zur Verfügung stehen. Sie kennen zum Beispiel bereits die Built-in Functions `len` und `range`. Im Folgenden werden alle bisher relevanten Built-in Functions ausführlich beschrieben. Beachten Sie, dass es noch weitere eingebaute Funktionen gibt, die an dieser Stelle nicht besprochen werden können, da sie Konzepte der objektorientierten Programmierung voraussetzen. Eine vollständige Übersicht über alle in Python eingebauten Funktionen finden Sie im Anhang dieses Buches (Abschnitt A.2).

Built-in Function	Beschreibung	Abschnitt
<code>abs(x)</code>	Berechnet den Betrag der Zahl <code>x</code> .	19.7.1
<code>all(iterable)</code>	Prüft, ob alle Elemente des iterierbaren Objekts <code>iterable</code> den Wert <code>True</code> ergeben.	19.7.2
<code>any(iterable)</code>	Prüft, ob mindestens ein Element des iterierbaren Objekts <code>iterable</code> den Wert <code>True</code> ergibt.	19.7.3
<code>ascii(object)</code>	Erzeugt einen druckbaren String, der das Objekt <code>object</code> beschreibt. Dabei werden Sonderzeichen maskiert, sodass die Ausgabe nur ASCII-Zeichen enthält.	19.7.4
<code>bin(x)</code>	Gibt einen String zurück, der die Ganzzahl <code>x</code> als Binärzahl darstellt.	19.7.5
<code>bool([x])</code>	Erzeugt einen booleschen Wert.	19.7.6

Tabelle 19.1 Built-in Functions, die in diesem Abschnitt besprochen werden

<sup>8</sup> Jede rekursive Funktion kann, unter Umständen mit viel Aufwand, in eine iterative umgeformt werden. Eine iterative Funktion ruft sich selbst nicht auf, sondern löst das Problem allein durch Einsatz von Kontrollstrukturen, speziell Schleifen. Eine rekursive Funktion ist oft eleganter und kürzer als ihr iteratives Ebenbild, in der Regel aber auch langsamer.

Built-in Function	Beschreibung	Abschnitt
<code>bytearray([source, encoding, errors])</code>	Erzeugt eine neue bytearray-Instanz.	19.7.7
<code>bytes([source, encoding, errors])</code>	Erzeugt eine neue bytes-Instanz.	19.7.8
<code>chr(i)</code>	Gibt das Zeichen mit dem Unicode-Code-point <code>i</code> zurück.	19.7.9
<code>complex([real, imag])</code>	Erzeugt eine komplexe Zahl.	19.7.10
<code>dict([arg])</code>	Erzeugt ein Dictionary.	19.7.11
<code>divmod(a, b)</code>	Gibt ein Tupel mit dem Ergebnis einer Ganzzahldivision und dem Rest zurück.  <code>divmod(a, b)</code> ist äquivalent zu <code>(a // b, a % b)</code>	19.7.12
<code>enumerate(iterable, [start])</code>	Gibt einen Aufzählungsiterator für das übergebene iterierbare Objekt zurück.	19.7.13
<code>eval(expression, [globals, locals])</code>	Wertet den Python-Ausdruck <code>expression</code> aus.	19.7.14
<code>exec(object, [globals, locals])</code>	Führt einen Python-Code aus.	19.7.15
<code>filter(function, iterable)</code>	Ermöglicht es, bestimmte Elemente eines iterierbaren Objekts herauszufiltern.	19.7.16
<code>float([x])</code>	Erzeugt eine Gleitkommazahl.	19.7.17
<code>format(value, [format_spec])</code>	Formatiert einen Wert <code>value</code> mit der Formatangabe <code>format_spec</code> .	19.7.18
<code>frozenset([iterable])</code>	Erzeugt eine unveränderliche Menge.	19.7.19
<code>globals()</code>	Gibt ein Dictionary mit allen Referenzen des globalen Namensraums zurück.	19.7.20
<code>hash(object)</code>	Gibt den Hash-Wert der Instanz <code>object</code> zurück.	19.7.21
<code>help([object])</code>	Startet die eingebaute interaktive Hilfe von Python.	19.7.22

Tabelle 19.1 Built-in Functions, die in diesem Abschnitt besprochen werden (Forts.)

Built-in Function	Beschreibung	Abschnitt
<code>hex(x)</code>	Gibt den Hexadezimalwert der ganzen Zahl <code>x</code> in Form eines Strings zurück.	19.7.23
<code>id(object)</code>	Gibt die Identität der Instanz <code>object</code> zurück.	19.7.24
<code>input([prompt])</code>	Liest einen String von der Tastatur ein.	19.7.25
<code>int(x, [base])</code>	Erzeugt eine ganze Zahl.	19.7.26
<code>len(s)</code>	Gibt die Länge einer Instanz <code>s</code> zurück.	19.7.27
<code>list([iterable])</code>	Erzeugt eine Liste.	19.7.28
<code>locals()</code>	Gibt ein Dictionary zurück, das alle Referenzen des lokalen Namensraums enthält.	19.7.29
<code>map(function, [*iterables])</code>	Wendet die Funktion <code>function</code> auf jedes Element der übergebenen iterierbaren Objekte an.	19.7.30
<code>max(iterable, {default, key})</code> <code>max(arg1, arg2, [*args], {key})</code>	Gibt das größte Element von <code>iterable</code> zurück.	19.7.31
<code>min(iterable, {default, key})</code> <code>min(arg1, arg2, [*args], {key})</code>	Gibt das kleinste Element von <code>iterable</code> zurück.	19.7.32
<code>oct(x)</code>	Gibt den Oktalwert der ganzen Zahl <code>x</code> in Form eines Strings zurück.	19.7.33
<code>open(file, [mode, buffering, encoding, errors, newline, closefd])</code>	Erzeugt ein Dateiojekt.	6.4.1
<code>ord(c)</code>	Gibt den Unicode-Code des Zeichens <code>c</code> zurück.	19.7.34
<code>pow(x, y, [z])</code>	Führt eine Potenzoperation durch.	19.7.35
<code>print([*objects], {sep, end, file})</code>	Gibt die übergebenen Objekte auf dem Bildschirm oder in andere Ausgabeströme aus.	19.7.36

Tabelle 19.1 Built-in Functions, die in diesem Abschnitt besprochen werden (Forts.)



Built-in Function	Beschreibung	Abschnitt
<code>range([start], stop, [step])</code>	Erzeugt einen Iterator über eine Zahlenfolge von <code>start</code> bis <code>stop</code> .	19.7.37
<code>repr(object)</code>	Gibt eine String-Repräsentation der Instanz <code>object</code> zurück.	19.7.38
<code>reversed(seq)</code>	Erzeugt einen Iterator, der das iterierbare Objekt <code>seq</code> rückwärts durchläuft.	19.7.39
<code>round(x, [n])</code>	Rundet die Zahl <code>x</code> auf <code>n</code> Nachkommastellen.	19.7.40
<code>set([iterable])</code>	Erzeugt eine Menge.	19.7.41
<code>sorted(iterable, [key, reverse])</code>	Sortiert das iterierbare Objekt <code>iterable</code> .	19.7.42
<code>str([object, encoding, errors])</code>	Erzeugt einen String.	19.7.43
<code>sum(iterable, [start])</code>	Gibt die Summe aller Elemente des iterierbaren Objekts <code>iterable</code> zurück.	19.7.44
<code>tuple([iterable])</code>	Erzeugt ein Tupel.	19.7.45
<code>type(object)</code>	Gibt den Datentyp einer Instanz zurück.	19.7.46
<code>zip(*iterables)</code>	Fasst mehrere Sequenzen zu Tupeln zusammen, um sie beispielsweise mit einer <code>for</code> -Schleife zu durchlaufen.	19.7.47

Tabelle 19.1 Built-in Functions, die in diesem Abschnitt besprochen werden (Forts.)

19.7.1 abs(x)

Die Funktion `abs` berechnet den Betrag von `x`. Der Parameter `x` muss dabei ein numerischer Wert sein, also eine Instanz der Datentypen `int`, `float`, `bool` oder `complex`.

```
>>> abs(1)
1
>>> abs(-12.34)
12.34
>>> abs(3 + 4j)
5.0
```

19.7.2 all(iterable)

Die Funktion `all` gibt immer dann `True` zurück, wenn alle Elemente des als Parameter übergebenen iterierbaren Objekts, also beispielsweise einer Liste oder eines Tupels, den Wahrheitswert `True` ergeben. Sie wird folgendermaßen verwendet:

```
>>> all([True, True, False])
False
>>> all([True, True, True])
True
```

Das übergebene iterierbare Objekt muss nicht zwingend nur `bool`-Instanzen durchlaufen. Instanzen anderer Datentypen werden nach den Regeln aus Abschnitt 12.6.2, »Wahrheitswerte nicht-boolescher Datentypen«, in Wahrheitswerte überführt.

19.7.3 any(iterable)

Die Funktion `any` arbeitet ähnlich wie `all`. Sie gibt immer dann `True` zurück, wenn mindestens ein Element des als Parameter übergebenen iterierbaren Objekts, also zum Beispiel einer Liste oder eines Tupels, den Wahrheitswert `True` ergibt. Sie wird folgendermaßen verwendet:

```
>>> any([True, False, False])
True
>>> any([False, False, False])
False
```

Das übergebene iterierbare Objekt muss nicht zwingend nur `bool`-Instanzen durchlaufen. Instanzen anderer Datentypen werden nach den Regeln aus Abschnitt 12.6.2, »Wahrheitswerte nicht-boolescher Datentypen«, in Wahrheitswerte überführt.

19.7.4 ascii(object)

Die Funktion `ascii` gibt eine lesbare Entsprechung der Instanz `object` in Form eines Strings zurück. Im Gegensatz zu der für denselben Zweck existierenden Built-in Function `repr` enthält der von `ascii` zurückgegebene String ausschließlich Zeichen des ASCII-Zeichensatzes:

```
>>> ascii(range(0, 10))
'range(0, 10)'
>>> ascii("Püthon")
"'P\\xfcthon'"
>>> repr("Püthon")
"'Püthon'"
```

### 19.7.5 bin(x)

Die Funktion `bin` gibt einen String zurück, der die für `x` übergebene ganze Zahl in ihrer Binärdarstellung enthält:

```
>>> bin(123)
'0b1111011'
>>> bin(-12)
'-0b1100'
>>> bin(0)
'0b0'
```

### 19.7.6 bool([x])

Hiermit wird eine Instanz des Datentyps `bool` mit dem Wahrheitswert der Instanz `x` erzeugt. Der Wahrheitswert von `x` wird nach den in Abschnitt 12.6.2, »Wahrheitswerte nicht-boolescher Datentypen«, festgelegten Regeln bestimmt.

```
>>> bool("")
False
>>> bool("Python")
True
```

Wenn kein Parameter übergeben wurde, gibt die Funktion `bool` den booleschen Wert `False` zurück.

### 19.7.7 bytearray([source, encoding, errors])

Die Funktion `bytearray` erzeugt eine Instanz des Datentyps `bytearray`, der eine Sequenz von Byte-Werten darstellt, also ganzen Zahlen im Zahlenbereich von 0 bis 255. Beachten Sie, dass `bytearray` im Gegensatz zu `bytes` ein veränderlicher Datentyp ist.

Der Parameter `source` wird zum Initialisieren des Byte-Arrays verwendet und kann verschiedene Bedeutungen haben:

Wenn für `source` ein String übergeben wird, wird dieser mithilfe der Parameter `encoding` und `errors` in eine Byte-Folge codiert und dann zur Initialisierung des Byte-Arrays verwendet. Die Parameter `encoding` und `errors` haben die gleiche Bedeutung wie bei der Built-in Function `str`.

Wenn für `source` eine ganze Zahl übergeben wird, wird ein Byte-Array der Länge `source` angelegt und mit Nullen gefüllt.

Wenn für `source` ein iterierbares Objekt, beispielsweise eine Liste, übergeben wird, wird das Byte-Array mit den Elementen gefüllt, über die `source` iteriert. Beachten Sie, dass es sich dabei um ganze Zahlen aus dem Zahlenbereich von 0 bis 255 handeln muss.

Außerdem kann für `source` eine beliebige Instanz eines Datentyps übergeben werden, der das sogenannte *Buffer-Protokoll* unterstützt. Das sind beispielsweise die Datentypen `bytes` und `bytearray` selbst.

```
>>> bytearray("äöü", "utf-8")
bytearray(b'\xc3\xa4\xc3\xb6\xc3\xbc')
>>> bytearray([1,2,3,4])
bytearray(b'\x01\x02\x03\x04')
>>> bytearray(10)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
```

Näheres zum Datentyp `bytearray` erfahren Sie in Abschnitt 13.4, »Strings – `str`, `bytes`, `bytearray`«.

### 19.7.8 bytes([source, encoding, errors])

Hiermit wird eine Instanz des Datentyps `bytes`<sup>9</sup> erzeugt, der, wie der Datentyp `bytearray`, eine Folge von Byte-Werten speichert. Im Gegensatz zu `bytearray` handelt es sich aber um einen unveränderlichen Datentyp, weswegen wir auch von einem `bytes-String` sprechen.

Die Parameter `source`, `encoding` und `errors` werden wie bei der Built-in Function `bytearray` zur Initialisierung der Byte-Folge verwendet:

```
>>> bytes(10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> bytes([1,2,3])
b'\x01\x02\x03'
>>> bytes("äöü", "utf-8")
b'\xc3\xa4\xc3\xb6\xc3\xbc'
```

### 19.7.9 chr(i)

Die Funktion `chr` gibt einen String der Länge 1 zurück, der das Zeichen mit dem Unicode-Codepoint `i` enthält:

```
>>> chr(65)
'A'
>>> chr(33)
'!'
>>> chr(8364)
'€'
```

<sup>9</sup> siehe Abschnitt 13.4.4, »Zeichensätze und Sonderzeichen«

### 19.7.10 complex([real, imag])

Hiermit wird eine Instanz des Datentyps `complex`<sup>10</sup> zur Speicherung einer komplexen Zahl erzeugt. Die erzeugte Instanz hat den komplexen Wert  $real + imag \cdot j$ . Fehlende Parameter werden als 0 angenommen.

Außerdem ist es möglich, der Funktion `complex` einen String zu übergeben, der das Literal einer komplexen Zahl enthält. In diesem Fall darf jedoch kein weiterer Parameter angegeben werden.

```
>>> complex(1, 3)
(1+3j)
>>> complex(1.2, 3.5)
(1.2+3.5j)
>>> complex("3+4j")
(3+4j)
>>> complex("3")
(3+0j)
```

Beachten Sie, dass ein eventuell übergebener String keine Leerzeichen um den `+`-Operator enthalten darf:

```
>>> complex("3 + 4j")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: complex() arg is a malformed string
```

Leerzeichen am Anfang oder Ende des Strings sind aber kein Problem.

### 19.7.11 dict([source])

Hiermit wird eine Instanz des Datentyps `dict`<sup>11</sup> erzeugt. Wenn kein Parameter übergeben wird, wird ein leeres Dictionary erstellt. Durch einen der folgenden Aufrufe ist es möglich, das Dictionary beim Erzeugen mit Werten zu füllen:

- Wenn `source` ein Dictionary ist, werden die Schlüssel und Werte dieses Dictionarys in das neue übernommen. Beachten Sie, dass dabei keine Kopien der Werte entstehen, sondern diese weiterhin dieselben Instanzen referenzieren.

```
>>> dict({"a" : 1, "b" : 2})
{'a': 1, 'b': 2}
```

<sup>10</sup> siehe Abschnitt 12.7, »Komplexe Zahlen – complex«

<sup>11</sup> siehe Abschnitt 14.1, »Dictionary – dict«

- Alternativ kann `source` ein über Tupel iterierendes Objekt sein, wobei jedes Tupel zwei Elemente enthalten muss: den Schlüssel und den damit assoziierten Wert. Die Liste muss die Struktur `[("a", 1), ("b", 2)]` haben:

```
>>> dict([("a", 1), ("b", 2)])
{'a': 1, 'b': 2}
```

- Zudem erlaubt es `dict`, Schlüssel und Werte als Keyword Arguments zu übergeben. Der Parametername wird dabei in einen String geschrieben und als Schlüssel verwendet. Beachten Sie, dass Sie damit bei der Namensgebung den Beschränkungen eines Bezeichners unterworfen sind:

```
>>> dict(a=1, b=2)
{'a': 1, 'b': 2}
```

### 19.7.12 divmod(a, b)

Die Funktion `divmod` gibt folgendes Tupel zurück:  $(a//b, a\%b)$ . Mit Ausnahme von `complex` können für `a` und `b` Instanzen beliebiger numerischer Datentypen übergeben werden:

```
>>> divmod(2.5, 1.3)
(1.0, 1.2)
>>> divmod(11, 4)
(2, 3)
```

### 19.7.13 enumerate(iterable)

Die Funktion `enumerate` erzeugt ein iterierbares Objekt, das nicht allein über die Elemente von `iterable` iteriert, sondern über Tupel der Form `(i, iterable[i])`. Dabei ist `i` ein Schleifenzähler, der bei 0 beginnt. Die Schleife wird beendet, wenn `i` den Wert `len(iterable)-1` hat. Diese Tupelstrukturen werden deutlich, wenn man das Ergebnis eines `enumerate`-Aufrufs in eine Liste konvertiert:

```
>>> list(enumerate(["a", "b", "c", "d"]))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Damit eignet sich `enumerate` besonders für `for`-Schleifen, in denen ein numerischer Schleifenzähler mitgeführt werden soll. Innerhalb einer `for`-Schleife kann `enumerate` folgendermaßen verwendet werden:

```
for i, wert in enumerate(iterable):
    print("Der Wert von iterable an", i, "ter Stelle ist:", wert)
```

Angenommen, der oben dargestellte Code wird für eine Liste `iterable = [1,2,3,4,5]` ausgeführt, kommt folgende Ausgabe zustande:

```

Der Wert von iterable an 0 ter Stelle ist: 1
Der Wert von iterable an 1 ter Stelle ist: 2
Der Wert von iterable an 2 ter Stelle ist: 3
Der Wert von iterable an 3 ter Stelle ist: 4
Der Wert von iterable an 4 ter Stelle ist: 5

```

#### 19.7.14 eval(expression, [globals, locals])

Die Funktion `eval` wertet den in Form eines Strings vorliegenden Python-Ausdruck `expression` aus und gibt dessen Ergebnis zurück:

```

>>> eval("1+1")
2

```

Beim Aufruf von `eval` können der gewünschte globale und lokale Namensraum, in denen der Ausdruck ausgewertet werden soll, über die Parameter `globals` und `locals` angegeben werden. Wenn diese Parameter nicht angegeben wurden, wird `expression` in der Umgebung ausgewertet, in der `eval` aufgerufen wurde:

```

>>> x = 12
>>> eval("x**2")
144

```



##### Hinweis

Manchmal wird `eval` dazu verwendet, Benutzereingaben als Python-Code zu interpretieren:

```

>>> eval(input("Geben Sie Python-Code ein: "))
Geben Sie Python-Code ein: 2**4
16

```

Bitte beachten Sie, dass diese Verwendung von `eval` potentiell gefährlich ist, wenn die Benutzereingaben nicht sorgfältig geprüft werden. Ein böartiger Benutzer kann hier die Programmausführung manipulieren.

#### 19.7.15 exec(object, [globals, locals])

Die Funktion `exec` führt einen als String vorliegenden Python-Code aus:

```

>>> code = """
... x = 12
... print(x**2)
... """

```

```

>>> exec(code)
144

```

Beim Aufruf von `exec` können der gewünschte globale und lokale Namensraum, in denen der Code ausgeführt werden soll, über die Parameter `globals` und `locals` angegeben werden. Wenn diese Parameter nicht angegeben wurden, wird der Code in der Umgebung ausgeführt, in der `exec` aufgerufen wurde.

##### Hinweis

Für `exec` gilt die gleiche Sicherheitswarnung wie für `eval` aus dem vorangegangenen Abschnitt: Prüfen Sie Benutzereingaben genau, bevor sie an `exec` weitergeleitet werden!



#### 19.7.16 filter(function, iterable)

Die Funktion `filter` erwartet ein Funktionsobjekt als ersten und ein iterierbares Objekt als zweiten Parameter. Der Parameter `function` muss eine Funktion oder Lambda-Form (siehe Abschnitt 19.4, »Anonyme Funktionen«) sein, die einen Parameter erwartet und einen booleschen Wert zurückgibt.

Die Funktion `filter` ruft für jedes Element des iterierbaren Objekts `iterable` die Funktion `function` auf und erzeugt ein iterierbares Objekt, das alle Elemente von `list` durchläuft, für die `function` den Wert `True` zurückgegeben hat. Dies soll an folgendem Beispiel erklärt werden, in dem `filter` dazu verwendet wird, aus einer Liste von ganzen Zahlen die ungeraden Zahlen herauszufiltern:

```

>>> filterobj = filter(lambda x: x%2 == 0, range(21))
>>> print(list(filterobj))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

```

#### 19.7.17 float([x])

Hiermit wird eine Instanz des Datentyps `float`<sup>12</sup> erzeugt. Wenn der Parameter `x` nicht angegeben wurde, wird der Wert der Instanz mit `0.0`, andernfalls mit dem übergebenen Wert initialisiert. Mit Ausnahme von `complex` können Instanzen alle numerischen Datentypen für `x` übergeben werden.

```

>>> float()
0.0
>>> float(5)
5.0

```

<sup>12</sup> siehe Abschnitt 12.5, »Gleitkommazahlen – float«

Außerdem ist es möglich, für `x` einen String zu übergeben, der eine Gleitkommazahl enthält:

```
>>> float("1e30")
1e+30
>>> float("0.5")
0.5
```

### 19.7.18 `format(value, [format_spec])`

Die Funktion `format` gibt den Wert `value` gemäß der Formatangabe `format_spec` zurück. Beispielsweise lässt sich ein Geldbetrag bei der Ausgabe folgendermaßen auf zwei Nachkommastellen runden:

```
>>> format(1.23456, ".2f") + "€"
'1.23€'
```

Ausführliche Informationen zu Formatangaben finden Sie in Abschnitt 13.4.3 über String-Formatierungen.

### 19.7.19 frozenset([iterable])

Hiermit wird eine Instanz des Datentyps `frozenset`<sup>13</sup> zum Speichern einer unveränderlichen Menge erzeugt. Wenn der Parameter `iterable` angegeben wurde, werden die Elemente der erzeugten Menge diesem iterierbaren Objekt entnommen. Wenn der Parameter `iterable` nicht angegeben wurde, erzeugt `frozenset` eine leere Menge.

Beachten Sie zum einen, dass ein `frozenset` keine veränderlichen Elemente enthalten darf, und zum anderen, dass jedes Element nur einmal in einer Menge vorkommen kann.

```
>>> frozenset()
frozenset()
>>> frozenset({1,2,3,4,5})
frozenset({1, 2, 3, 4, 5})
>>> frozenset("Pyyyyyyython")
frozenset({'h', 'o', 'n', 'P', 't', 'y'})
```

### 19.7.20 globals()

Die Built-in Function `globals` gibt ein Dictionary mit allen globalen Referenzen des aktuellen Namensraums zurück. Die Schlüssel entsprechen den Referenznamen als Strings und die Werte den jeweiligen Instanzen.

13 siehe Abschnitt 15.1, »Die Datentypen set und frozenset«

```
>>> a = 1
>>> b = {}
>>> c = [1,2,3]
>>> globals()
{'a': 1, 'c': [1, 2, 3], 'b': {}, '__builtins__': <module 'builtins'
(built-in)>, '__package__': None, '__name__': '__main__', '__doc__': None}
```

Das zurückgegebene Dictionary enthält neben den vorher angelegten noch weitere Referenzen, die im globalen Namensraum existieren. Diese vordefinierten Referenzen haben wir bisher noch nicht besprochen, lassen Sie sich davon also nicht stören.

### 19.7.21 hash(object)

Die Funktion `hash` berechnet den Hash-Wert der Instanz `object` und gibt ihn zurück. Bei einem Hash-Wert handelt es sich um eine ganze Zahl, die aus Typ und Wert der Instanz erzeugt wird. Ein solcher Wert wird verwendet, um effektiv zwei komplexere Instanzen auf Gleichheit prüfen zu können. So werden beispielsweise die Schlüssel eines Dictionarys intern durch ihre Hash-Werte verwaltet.

```
>>> hash(12345)
12345
>>> hash("Hallo Welt")
-962533610
>>> hash((1,2,3,4))
89902565
```

Beachten Sie den Unterschied zwischen veränderlichen (mutablen) und unveränderlichen (immutablen) Instanzen. Aus Letzteren kann zwar formal auch ein Hash-Wert errechnet werden, dieser wäre aber nur so lange gültig, wie die Instanz nicht verändert wurde. Aus diesem Grund ist es nicht sinnvoll, Hash-Werte von veränderlichen Instanzen zu berechnen; veränderliche Instanzen sind »unhashable«:

```
>>> hash([1,2,3,4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

## Hinweis

Seit Python 3.3 werden Hash-Werte von `str`-, `bytes`- und `datetime`-Instanzen aus Sicherheitsgründen randomisiert. Das bedeutet, dass sich Hash-Werte dieser Datentypen in zwei verschiedenen Interpreter-Prozessen unterscheiden. Innerhalb desselben Prozesses ändern sie sich aber nicht.



### 19.7.22 help([object])

Die Funktion `help` startet die interaktive Hilfe von Python. Wenn der Parameter `object` ein String ist, wird dieser im Hilfesystem nachgeschlagen. Sollte es sich um eine andere Instanz handeln, wird eine dynamische Hilfeseite zu dieser generiert.

### 19.7.23 hex(x)

Die Funktion `hex` erzeugt einen String, der die als Parameter `x` übergebene ganze Zahl in Hexadezimalschreibweise enthält. Die Zahl entspricht, wie sie im String erscheint, dem Python-Literal für Hexadezimalzahlen.

```
>>> hex(12)
'0xc'
>>> hex(0xFF)
'0xff'
>>> hex(-33)
'-0x21'
```

### 19.7.24 id(object)

Die Funktion `id` gibt die Identität einer beliebigen Instanz zurück. Bei der Identität einer Instanz handelt es sich um eine ganze Zahl, die die Instanz eindeutig identifiziert.

```
>>> id(1)
134537016
>>> id(2)
134537004
```

Näheres zu Identitäten erfahren Sie in Abschnitt 7.1.3.

### 19.7.25 input([prompt])

Die Funktion `input` liest eine Eingabe vom Benutzer ein und gibt sie in Form eines Strings zurück. Der Parameter `prompt` ist optional. Hier kann ein String angegeben werden, der vor der Eingabeaufforderung ausgegeben werden soll.

```
>>> s = input("Geben Sie einen Text ein: ")
Geben Sie einen Text ein: Python ist gut
>>> s
'Python ist gut'
```

#### Hinweis

Das Verhalten der Built-in Function `input` wurde mit Python 3.0 verändert. In früheren Versionen wurde die Eingabe des Benutzers als Python-Code vom Interpreter ausgeführt und das Ergebnis dieser Ausführung in Form eines Strings zurückgegeben. Die »alte« `input`-Funktion entsprach also folgendem Code:

```
>>> eval(input("Prompt: "))
Prompt: 2+2
4
```

Die `input`-Funktion, wie sie in aktuellen Versionen von Python existiert, hieß in früheren Versionen `raw_input`.

### 19.7.26 int([x, base])

Hiermit wird eine Instanz des Datentyps `int`<sup>14</sup> erzeugt. Die Instanz kann durch Angabe von `x` mit einem Wert initialisiert werden. Wenn kein Parameter angegeben wird, erhält die erzeugte Instanz den Wert 0.

Wenn der Parameter `x` als String übergeben wird, erwartet die Funktion `int`, dass dieser String den gewünschten Wert der Instanz enthält. Durch den optionalen Parameter `base` kann die Basis des Zahlensystems angegeben werden, in dem die Zahl geschrieben wurde.

```
>>> int(5)
5
>>> int("FF", 16)
255
>>> int(hex(12), 16)
12
```

### 19.7.27 len(s)

Die Funktion `len` gibt die Länge bzw. die Anzahl der Elemente von `s` zurück. Für `s` können Sequenzen, Mappings oder Mengen übergeben werden.

```
>>> len("Hallo Welt")
10
>>> len([1,2,3,4,5])
5
```

<sup>14</sup> siehe Abschnitt 12.4, »Ganzzahlen – int«





### 19.7.28 list([sequence])

Hiermit wird eine Instanz des Datentyps `list`<sup>15</sup> aus den Elementen von `sequence` erzeugt. Der Parameter `sequence` muss ein iterierbares Objekt sein. Wenn er weggelassen wird, wird eine leere Liste erzeugt.

```
>>> list()
[]
>>> list((1,2,3,4))
[1, 2, 3, 4]
>>> list({"a": 1, "b": 2})
['a', 'b']
```

Die Funktion `list` kann dazu verwendet werden, ein beliebiges iterierbares Objekt in eine Liste zu überführen:

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

### 19.7.29 locals()

Die Built-in Function `locals` gibt ein Dictionary mit allen lokalen Referenzen des aktuellen Namensraums zurück. Die Schlüssel entsprechen den Referenznamen als Strings und die Werte den jeweiligen Instanzen. Dies soll an folgendem Beispiel deutlich werden:

```
def f(a, b, c):
    d = a + b + c
    print(locals())
f(1, 2, 3)
```

Das bei diesem Aufruf von `locals` zurückgegebene Dictionary enthält die Referenzen `a`, `b`, `c` und `d`, die im lokalen Namensraum der Funktion existieren.

```
{'a': 1, 'c': 3, 'b': 2, 'd': 6}
```

Der Aufruf von `locals` im Namensraum des Hauptprogramms ist äquivalent zum Aufruf von `globals`.

### 19.7.30 map(function, [\*iterable])

Diese Funktion erwartet ein Funktionsobjekt als ersten und ein iterierbares Objekt als zweiten Parameter. Optional können weitere iterierbare Objekte übergeben wer-

<sup>15</sup> siehe Abschnitt 13.2, »Listen – list«

den, die aber die gleiche Länge wie das erste haben müssen. Die Funktion `function` muss genauso viele Parameter erwarten, wie iterierbare Objekte übergeben wurden, und aus den Parametern einen Rückgabewert erzeugen.

Die Funktion `map` ruft `function` für jedes Element von `iterable` auf und gibt ein iterierbares Objekt zurück, das die jeweiligen Rückgabewerte von `function` durchläuft. Sollten mehrere iterierbare Objekte übergeben werden, werden `function` die jeweils `n`-ten Elemente dieser Objekte übergeben.

Im folgenden Beispiel wird das Funktionsobjekt durch eine Lambda-Form erstellt. Es ist auch möglich, eine »echte« Funktion zu definieren und ihren Namen zu übergeben.

```
>>> f = lambda x: x**2
>>> ergebnis = map(f, [1,2,3,4])
>>> list(ergebnis)
[1, 4, 9, 16]
```

Hier wird `map` dazu verwendet, eine Liste mit den Quadraten der Elemente einer zweiten Liste zu erzeugen.

```
>>> f = lambda x, y: x+y
>>> ergebnis = map(f, [1,2,3,4], [1,2,3,4])
>>> list(ergebnis)
[2, 4, 6, 8]
```

Hier wird `map` dazu verwendet, aus zwei Listen eine zu erzeugen, die die Summen der jeweiligen Elemente beider Quelllisten enthält.

Das letzte Beispiel wird durch Abbildung 19.3 veranschaulicht. Die eingehenden und ausgehenden iterierbaren Objekte sind jeweils senkrecht dargestellt.

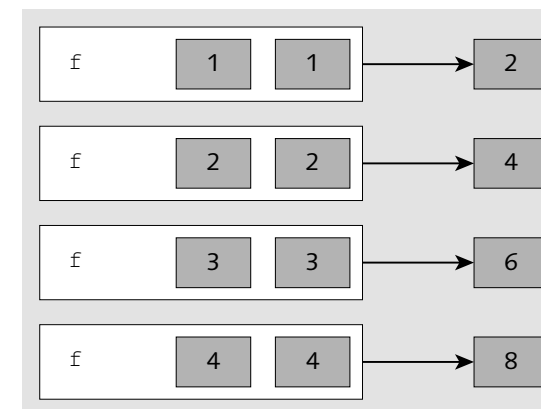


Abbildung 19.3 Arbeitsweise der Built-in Function `map`

In beiden Beispielen wurden Listen verwendet, die ausschließlich numerische Elemente enthielten. Das muss nicht unbedingt sein. Welche Elemente ein an `map` übergebenes iterierbares Objekt durchlaufen darf, hängt davon ab, welche Instanzen für `function` als Parameter verwendet werden dürfen.

### 19.7.31 `max(iterable, {default, key})` `max(arg1, arg2, [*args], {key})`

Wenn keine zusätzlichen Parameter übergeben werden, erwartet `max` ein iterierbares Objekt und gibt ihr größtes Element zurück.

```
>>> max([2,4,1,9,5])
9
>>> max("Hallo Welt")
't'
```

Wenn mehrere Parameter übergeben werden, verhält sich `max` so, dass der größte übergebene Parameter zurückgegeben wird:

```
>>> max(3, 5, 1, 99, 123, 45)
123
>>> max("Hallo", "Welt", "!")
'Welt'
```

Der Funktion `max` kann optional über den Schlüsselwortparameter `key` ein Funktionsobjekt übergeben werden. Das Maximum wird dann durch das Vergleichen der Rückgabewerte dieser Funktion bestimmt. Mit dem Parameter `key` lässt sich also eine eigene Ordnungsrelation festlegen. In folgendem Beispiel soll `key` dazu verwendet werden, die Funktion `max` für Strings *case insensitive* zu machen. Dazu zeigen wir zunächst den normalen Aufruf ohne `key`:

```
>>> max("a", "P", "q", "X")
'q'
```

Ohne eigene `key`-Funktion wird der größte Parameter unter Berücksichtigung von Groß- und Kleinbuchstaben ermittelt. Folgende `key`-Funktion konvertiert zuvor alle Buchstaben in Kleinbuchstaben:

```
>>> f = lambda x: x.lower()
>>> max("a", "P", "q", "X", key=f)
'X'
```

Durch die `key`-Funktion wird der größte Parameter anhand der durch `f` modifizierten Werte ermittelt, jedoch unmodifiziert zurückgegeben.

Über den letzten Schlüsselwortparameter `default` kann ein Wert festgelegt werden, der von `max` zurückgegeben wird, wenn `iterable` leer sein sollte.

### 19.7.32 `min(iterable, {default, key})` `min(arg1, arg2, [*args], {key})`

Die Funktion `min` verhält sich wie `max`, ermittelt jedoch das kleinste Element einer Sequenz bzw. den kleinsten übergebenen Parameter.

### 19.7.33 `oct(x)`

Die Funktion `oct` erzeugt einen String, der die übergebene ganze Zahl `x` in Oktalschreibweise enthält.

```
>>> oct(123)
'0o173'
>>> oct(0o777)
'0o777'
```

### 19.7.34 `ord(c)`

Die Funktion `ord` erwartet einen String der Länge 1 und gibt den Unicode-Codepoint des enthaltenen Zeichens zurück.

```
>>> ord("p")
80
>>> ord("€")
8364
```

Näheres zu Unicode und Codepoints erfahren Sie in Abschnitt 13.4.4, »Zeichensätze und Sonderzeichen«.

### 19.7.35 `pow(x, y, [z])`

Berechnet  $x ** y$  oder, wenn `z` angegeben wurde,  $x ** y \% z$ . Diese Berechnung ist unter Verwendung des Parameters `z` performanter als die Ausdrücke `pow(x, y) \% z` bzw.  $x ** y \% z$ .

```
>>> 7 ** 5 % 4
3
>>> pow(7, 5, 4)
3
```

### 19.7.36 print([\*objects], {sep, end, file, flush})

Die Funktion `print` schreibt die Textentsprechungen der für `objects` übergebenen Instanzen in den Datenstrom `file`. Bislang haben wir `print` nur dazu verwendet, auf den Bildschirm bzw. in die Standardausgabe zu schreiben. Hier sehen wir, dass `print` es über den Schlüsselwortparameter `file` ermöglicht, in ein beliebiges zum Schreiben geöffnetes Dateiojekt zu schreiben:

```
>>> f = open("datei.txt", "w")
>>> print("Hallo Welt", file=f)
>>> f.close()
```

Über den Schlüsselwortparameter `sep`, der mit einem Leerzeichen vorbelegt ist, wird das Trennzeichen angegeben, das zwischen zwei auszugebenden Werten stehen soll:

```
>>> print("Hallo", "Welt")
Hallo Welt
>>> print("Hallo", "du", "schöne", "Welt", sep="-")
Hallo-du-schöne-Welt
```

Über den zweiten Schlüsselwortparameter `end` wird bestimmt, welches Zeichen `print` als Letztes, also nach erfolgter Ausgabe aller übergebenen Instanzen, ausgegeben soll. Vorbelegt ist dieser Parameter mit einem Newline-Zeichen.

```
>>> print("Hallo", end=" Welt\n")
Hallo Welt
>>> print("Hallo", "Welt", end="AAAA")
Hallo WeltAAAA>>>
```

Im letzten Beispiel befindet sich der Eingabeprompt des Interpreters direkt hinter der von `print` erzeugten Ausgabe, weil im Gegensatz zum Standardverhalten von `print` am Ende kein Newline-Zeichen ausgegeben wurde.

Mithilfe des letzten Parameters `flush` können Sie ein Leeren des Datenstrompuffers nach der Ausgabe erzwingen.

### 19.7.37 range([start], stop, [step])

Die Funktion `range` erzeugt ein iterierbares Objekt über fortlaufende, numerische Werte. Dabei wird mit `start` begonnen, vor `stop` aufgehört und in jedem Schritt der vorherige Wert um `step` erhöht. Sowohl `start` als auch `step` sind optional und mit 0 bzw. 1 vorbelegt.

Beachten Sie, dass `stop` eine Grenze angibt, die nicht erreicht wird. Die Nummerierung beginnt also bei 0 und endet einen Schritt, bevor `stop` erreicht würde.

Bei dem von `range` zurückgegebenen iterierbaren Objekt handelt es sich um ein sogenanntes `range`-Objekt. Dies wird bei der Ausgabe im interaktiven Modus folgendermaßen angezeigt:

```
>>> range(10)
range(0, 10)
```

Um zu veranschaulichen, über welche Zahlen das `range`-Objekt iteriert, wurde es in den folgenden Beispielen mit `list` in eine Liste überführt:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(2, 10, 2))
[2, 4, 6, 8]
```

Es ist möglich, eine negative Schrittweite anzugeben:

```
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> list(range(10, 0, -2))
[10, 8, 6, 4, 2]
```

#### Hinweis

In Python-Versionen vor 3.0 existierten die eingebauten Funktionen `range` und `xrange`. Die alte `range`-Funktion gibt das Ergebnis in Form einer Liste zurück, während die `xrange`-Funktion so funktioniert wie die `range`-Funktion in aktuellen Python-Versionen.



### 19.7.38 repr(object)

Die Funktion `repr` gibt einen String zurück, der eine druckbare Repräsentation der Instanz `object` enthält. Für viele Instanzen versucht `repr`, den Python-Code in den String zu schreiben, der die entsprechende Instanz erzeugen würde. Für manche Instanzen ist dies jedoch nicht möglich bzw. nicht praktikabel. In einem solchen Fall gibt `repr` zumindest den Typ der Instanz aus.

```
>>> repr([1,2,3,4])
'[1, 2, 3, 4]'
>>> repr(0x34)
'52'
>>> repr(set([1,2,3,4]))
```

```
'set([1, 2, 3, 4])'
>>> repr(open("datei.txt", "w"))
"<open file 'datei.txt', mode 'w' at 0xb7bea0f8>"
```

### 19.7.39 reversed(sequence)

Mit `reversed` kann eine Instanz `sequence` eines sequenziellen Datentyps effizient rückwärts durchlaufen werden.

```
>>> for i in reversed([1, 2, 3, 4, 5, 6]):
...     print(i)
6
5
4
3
2
1
```

### 19.7.40 round(x, [n])

Die Funktion `round` rundet die Gleitkommazahl `x` auf `n` Nachkommastellen. Der Parameter `n` ist optional und mit 0 vorbelegt.

```
>>> round(-0.5)
-1.0
>>> round(0.5234234234234, 5)
0.52342
>>> round(0.5, 4)
0.5
```

### 19.7.41 set([iterable])

Hiermit wird eine Instanz des Datentyps `set`<sup>16</sup> erzeugt. Wenn angegeben, werden alle Elemente des iterierbaren Objekts `iterable` in das Set übernommen. Beachten Sie, dass ein Set keine Dubletten enthalten darf, jedes in `iterable` mehrfach vorkommende Element also nur einmal eingetragen wird.

```
>>> set()
set()
>>> set("Hallo Welt")
```

<sup>16</sup> siehe Abschnitt 15.1, »Die Datentypen `set` und `frozenset`«

```
set({'a', ' ', 'e', 'H', 'l', 'o', 't', 'W'})
>>> set({1,2,3,4})
set({1, 2, 3, 4})
```

### 19.7.42 sorted(iterable, [key, reverse])

Die Funktion `sorted` erzeugt aus den Elementen von `iterable` eine sortierte Liste:

```
>>> sorted([3,1,6,2,9,1,8])
[1, 1, 2, 3, 6, 8, 9]
>>> sorted("Hallo Welt")
[' ', 'H', 'W', 'a', 'e', 'l', 'l', 'l', 'o', 't']
```

Die Funktionsweise von `sorted` ist identisch mit der Methode `sort` der sequenziellen Datentypen, die in Abschnitt 13.2.4, »Methoden von `list`-Instanzen«, erklärt wird.

### 19.7.43 str([object, encoding, errors])

Hiermit wird ein String erzeugt,<sup>17</sup> der eine lesbare Beschreibung der Instanz `object` enthält. Wenn `object` nicht übergeben wird, erzeugt `str` einen leeren String.

```
>>> str(None)
'None'
>>> str()
''
>>> str(12345)
'12345'
>>> str(str)
"<class 'str'>"
```

Die Funktion `str` kann dazu verwendet werden, einen bytes-String oder eine `bytearray`-Instanz in einen String zu überführen. Dieser Prozess wird *Decodieren* genannt, und es muss dazu mindestens der Parameter `encoding` angegeben worden sein:

```
>>> b = bytearray([1,2,3])
>>> str(b, "utf-8")
'\x01\x02\x03'
>>> b = bytes("Hallö Wölt", "utf-8", "strict")
>>> str(b)
"b'Hall\\xc3\\xb6 W\\xc3\\xb6lt'"
>>> str(b, "utf-8")
'Hallö Wölt'
```

<sup>17</sup> siehe Abschnitt 13.4.4, »Zeichensätze und Sonderzeichen«

Dabei muss für den Parameter `encoding` ein String übergeben werden, der das Encoding enthält, mit dem der `bytes`-String codiert wurde, in diesem Fall `utf-8`. Der Parameter `errors` wurde im Beispiel oben nicht angegeben und bestimmt, wie mit Decodierungsfehlern zu verfahren ist. Die folgende Tabelle listet die möglichen Werte für `errors` und ihre Bedeutung auf:

errors	Beschreibung
"strict"	Bei einem Decodierungsfehler wird eine <code>ValueError</code> -Exception geworfen.
"ignore"	Fehler bei der Decodierung werden ignoriert.
"replace"	Ein Zeichen, das nicht decodiert werden konnte, wird durch das Unicode-Zeichen U+FFFD (◆), auch Replacement Character genannt, ersetzt.

Tabelle 19.2 Mögliche Werte des Parameters `errors`



**Hinweis**

Beachten Sie, dass der Datentyp `str` mit Python 3.0 einer Überarbeitung unterzogen wurde. Im Gegensatz zum Datentyp `str` aus Python 2.x ist er in Python 3 dazu gedacht, Unicode-Text aufzunehmen. Er ist also vergleichbar mit dem Datentyp `unicode` aus Python 2. Der dortige Datentyp `str` lässt sich vergleichen mit dem `bytes`-String aus Python 3.

Weitere Informationen über die Datentypen `str` und `bytes` sowie über Unicode finden Sie in Abschnitt 13.4.4, »Zeichensätze und Sonderzeichen«.

**19.7.44 sum(iterable, [start])**

Die Funktion `sum` berechnet die Summe aller Elemente des iterierbaren Objekts `iterable` und gibt das Ergebnis zurück. Wenn der optionale Parameter `start` angegeben wurde, fließt dieser als Startwert der Berechnung ebenfalls in die Summe mit ein.

```
>>> sum([1,2,3,4])
10
>>> sum({1,2,3,4}, 2)
12
>>> sum({4,3,2,1}, 2)
12
```

**19.7.45 tuple([iterable])**

Hiermit wird eine Instanz des Datentyps `tuple`<sup>18</sup> aus den Elementen von `iterable` erzeugt.

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

**19.7.46 type(object)**

Die Funktion `type` gibt den Datentyp der übergebenen Instanz `object` zurück.

```
>>> type(1)
<type 'int'>
>>> type("Hallo Welt") == str
True
>>> type(sum)
<class 'builtin_function_or_method'>
```

**19.7.47 zip(\*iterables)**

Die Funktion `zip` nimmt beliebig viele, gleich lange iterierbare Objekte als Parameter. Sollten nicht alle die gleiche Länge haben, werden die längeren nur bis zur Länge des kürzesten dieser Objekte betrachtet.

Als Rückgabewert wird ein iterierbares Objekt erzeugt, das über `Tupel` iteriert, die im *i*-ten Iterationsschritt die jeweils *i*-ten Elemente der übergebenen Sequenzen enthalten.

```
>>> ergebnis = zip([1,2,3,4], [5,6,7,8], [9,10,11,12])
>>> list(ergebnis)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
>>> ergebnis = zip("Hallo Welt", "HaWe")
>>> list(ergebnis)
[('H', 'H'), ('a', 'a'), ('l', 'W'), ('l', 'e')]
```

Bei der bereits besprochenen Funktion `enumerate` handelt es sich um einen Spezialfall der `zip`-Funktion:

```
>>> s = "Python"
>>> list(zip(range(len(s)), s))
[(0, 'P'), (1, 'y'), (2, 't'), (3, 'h'), (4, 'o'), (5, 'n')]
```

<sup>18</sup> siehe Abschnitt 13.3, »Unveränderliche Listen – tuple«

Damit haben wir den ersten Teil der eingebauten Funktionen besprochen. In Kapitel 21, in dem die Konzepte der objektorientierten Programmierung besprochen werden, folgt in Abschnitt 21.5, »Built-in Functions für Objektorientierung«, die Beschreibung der eingebauten Funktionen mit objektorientiertem Hintergrund.